HAL open science

Graph rewriting for model construction in modal logic

Bilal Said

To cite this version:

Bilal Said. Graph rewriting for model construction in modal logic. Computer Science [cs]. Université Paul Sabatier - Toulouse III, 2010. English. NNT : . tel-00466115

HAL Id: tel-00466115

https://theses.hal.science/tel-00466115v1

Submitted on 22 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

Doctorat de l'Université de Toulouse

Délivré par : Université Toulouse III Paul Sabatier (UPS)

Discipline ou spécialité : Informatique

Présentée et soutenue par :

Bilal Said

Le vendredi 29 Janvier 2010

Titre :

Réécriture de graphes pour la construction de modèles en logique modale

---

Directeur de thèse :

Olivier Gasquet

Professeur - Université Paul Sabatier

Rapporteurs :

Serenella Cerrito

Professeur - Université d'Evry

Didier Galmiche

Professeur - Université Henri Poincarré

Président du jury :

Jean-Paul Bodeveix

Professeur - Université Paul Sabatier

École doctorale :

Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :

Institut de recherche en Informatique de Toulouse (IRIT)

Graph Rewriting for Model Construction in Modal Logic

Thesis presented and defended by

Bilal Said

On the 29th of January 2010

In order to obtain the degree of

Doctorat de l'Université de Toulouse

Delivered by : Université Toulouse III Paul Sabatier (UPS)
Speciality : Computer Science

Advisor:
Olivier Gasquet
Professor - Université Paul Sabatier

Reviewers:

Serenella Cerrito
Professor - Université d'Evry

Didier Galmiche
Professor - Université Henri Poincarré

Examiner:
Jean-Paul Bodeveix
Professor - Université Paul Sabatier

École doctorale:
Mathématiques Informatique Télécommunications (MITT)
Unité de recherche:
Institut de recherche en Informatique de Toulouse (IRIT)

Contents

Acknowledgements **11**

Abstract **13**

Introduction **17**

I Model Construction for Normal Modal Logics **27**

Preface to Part I **29**

1 Modelling with graphs **33**
1.1 Toggle the light 33
1.1.1 A light with two switches 34
1.2 Ann is guessing the card's color 36
1.2.1 What if Ann is cheating? 37
1.3 Talking about time 38
1.4 Build a graph using LoTREC 39
1.5 Labeled Directed Graphs 40

2 Talking about models **41**
2.1 A formal language to talk about graphs 41
2.1.1 Motivation 41
2.1.2 Boolean connectors 42
2.1.3 Modal connectors 43
2.1.4 Infix versus prefix notation 43
2.1.5 Formal definition 44
2.1.6 Arity of connectors 45
2.1.7 Analyzing a formula 45
2.2 Syntax declaration in LoTREC 46
2.2.1 Defining basic symbols 46
2.2.2 Defining formulas 47
2.2.3 Prefix notation 47
2.2.4 Customized display 48
2.3 A formal way to evaluate formulas 48

2.3.1 Models . . . 48
2.3.2 Truth conditions . . . 49
2.3.3 Dual modal operators . . . 50
2.4 Reasoning problems . . . 50
2.4.1 Model checking . . . 51
2.4.2 Satisfiability . . . 51
2.4.3 Validity . . . 51
2.4.4 Model construction . . . 52

3 The model construction method 55
3.1 Model construction by hand . . . 56
3.1.1 Initialisation . . . 57
3.1.2 Classical saturation ($\wedge$) . . . 57
3.1.3 Successors creation ($\Diamond$) . . . 58
3.1.4 Propagation ($\Box$) . . . 58
3.1.5 Disjunction ($\vee$) . . . 58
3.1.6 Extracting a model . . . 59
3.2 Terminology . . . 60
3.2.1 What is a premodel? . . . 60
3.2.2 What is a rule? . . . 61
3.2.3 Other notations . . . 61
3.3 Automated model construction . . . 62
3.3.1 Language of rules . . . 62
3.3.2 Saturation with `repeat` . . . 65
3.4 The full set of rules . . . 66
3.4.1 Cut rules . . . 68
3.5 From mono to multimodal logic K_n . . . 71
3.5.1 Inclusion: a simple interaction between modalities . . . 71
3.6 Certifying a model construction method . . . 73

4 Logics with simple constraints on models 75
4.1 Reflexive models . . . 77
4.1.1 Simulating reflexivity . . . 78
4.2 Satisfiability in symmetric models . . . 78
4.2.1 Simulating symmetry by semantics . . . 79
4.3 $\mathsf{K.alt}_1$: the relation is a partial function . . . 80
4.3.1 Motivation . . . 80
4.3.2 Changing the method of K to get a method for $\mathsf{K.alt}_1$ . . . 80
4.4 Serial models for obligation and norms . . . 83
4.4.1 Changing the method of K to make it suitable . . . 83
4.4.2 Extracting a model from an open premodel . . . 85
4.5 Confluent models . . . 86
4.6 Knowledge . . . 91
4.6.1 Modelling the knowledge of one agent . . . 92
4.6.2 Rules for $\mathsf{S5}$ with implicit edges . . . 92
4.7 A general termination theorem . . . 96

5 Logics with potential cycles **99**
5.1 Model construction in K4 99
5.1.1 Adding the transitive edges 99
5.1.2 Simulating the presence of the transitive edges 100
5.1.3 Construction of transitive models may not terminate! . . 101
5.2 Model construction for knowledge may not terminate! 105
5.2.1 Rules for S5 with explicit edges 105
5.2.2 Modelling the knowledge of multiple agents 107
5.2.3 Rules for S5 with multiple agents 108
5.2.4 Adding the necessary edges 108
5.2.5 Simulating the presence of the edges 110
5.3 Hybrid Logic 113
5.3.1 Model construction for HL(@) 114
5.4 Completeness vs. termination 121
5.5 Termination by checking for loops 124

6 Model Checking **127**
6.1 Model checking on paper 127
6.1.1 The model and the formula 128
6.1.2 Top-down formula decomposition 128
6.1.3 Bottom-up satisfiability check 129
6.2 Model checking in LoTREC 130
6.2.1 Defining the model checking problem 130
6.2.2 Top-down rules 132
6.2.3 Bottom-up rules 133
6.3 Further discussions 137

7 Logics with transitive closure **139**
7.1 Linear Temporal Logic LTL 139
7.1.1 LTL models 139
7.1.2 Syntax of LTL 140
7.1.3 Semantics of LTL 140
7.1.4 Model construction for LTL in LoTREC 141
7.1.5 Termination: detecting loops by node-inclusion test . . . 144
7.1.6 Checking the fulfillment of eventualities 146
7.1.7 Termination: detecting loops by node-equality test 149
7.2 Propositional Dynamic Logic PDL 151
7.2.1 Syntax of PDL 151
7.2.2 Semantics of PDL 152
7.2.3 LoTREC rules for PDL 153
7.2.4 Termination 156
7.2.5 Fulfillment of eventualities 158

8 Layered Modal Logics **167**
8.1 Layered modal logics . . . 170
8.1.1 Layer formulas . . . 171
8.1.2 Layered frames . . . 172
8.2 Construction of simple premodels for LML . . . 173
8.2.1 The set of rules . . . 174
8.2.2 Soundness and completeness . . . 176
8.3 Dynamically filtrated premodels . . . 179
8.3.1 The dynamic filtration . . . 180
8.3.2 Soundness and completeness . . . 181
8.4 Implementing a model construction method for LML in LoTREC 182
8.4.1 Rules for layer formulas . . . 182
8.4.2 Rules for dynamic filtration . . . 182
8.5 Discussion . . . 184
8.5.1 The case of symmetry . . . 184
8.5.2 The case of permutation . . . 185

II Graph Rewriting for Model Construction in Logic 187

Preface to Part II **189**

9 Graph rewriting overview **193**
9.1 Graph structures . . . 194
9.1.1 Labeled graphs . . . 194
9.1.2 Typed attributed graphs . . . 195
9.2 Rewriting rules . . . 197
9.2.1 Pattern matching and rule applicability . . . 198
9.2.2 Rule application . . . 199
9.3 Theoretical basis . . . 200
9.3.1 Single Pushout approach . . . 201
9.3.2 Double Pushout approach . . . 201
9.3.3 Pullback approach . . . 204
9.3.4 Adhesive High-Level Replacement approach . . . 204
9.3.5 Alternative approaches . . . 205
9.4 Properties of graph rewriting systems . . . 206
9.4.1 Characterization . . . 206
9.4.2 Termination . . . 207
9.4.3 Completeness . . . 207
9.4.4 Locality . . . 208
9.4.5 Parallel and sequential independence . . . 208
9.4.6 Confluence and convergence . . . 209
9.4.7 Complexity . . . 210

10 LoTREC: rewrite graphs to construct models 213
10.1 Graph structure . . . 214
10.1.1 Formulas as attributes . . . 215
10.1.2 The type graph of LoTREC . . . 218
10.1.3 Premodels as graph instances . . . 219
10.2 LoTREC's rules as graph rewriting rules . . . 219
10.2.1 Definition . . . 220
10.2.2 A special language of conditions and actions . . . 223
10.2.3 Applicability . . . 224
10.2.4 Application . . . 228
10.3 Strategies . . . 229
10.3.1 Aimed semantics . . . 229
10.3.2 Strategy instructions . . . 230
10.4 Properties of LoTREC rewriting system . . . 231
10.4.1 Termination . . . 231
10.4.2 Completeness, locality and invertibility . . . 232
10.4.3 Parallelism, confluence and convergence . . . 233

11 Semantics of event-based rewriting 235
11.1 Modelling graph rewriting with terms . . . 236
11.1.1 Recalling terms . . . 237
11.1.2 Encoding a premodel with terms . . . 237
11.1.3 Coding rules as term rewriting rules . . . 238
11.1.4 Rewriting rule application . . . 240
11.1.5 Discussion about pattern matching process . . . 241
11.1.6 LoTREC rewriting system . . . 242
11.1.7 Equivalence between usual rewriting system and LoTREC in terms of rules . . . 244
11.2 Strategies . . . 247
11.2.1 Syntax . . . 247
11.2.2 Standard semantics . . . 247
11.2.3 Semantics of rewriting with strategies in LoTREC . . . 249
11.2.4 Equivalence between usual rewriting system and LoTREC in terms of strategies . . . 250
11.3 Related works . . . 251
11.4 Experimental results . . . 251
11.4.1 The LWB benchmark suite . . . 252
11.4.2 Evaluation of the event-based technique . . . 252

Conclusion 255

A Signatures, algebras and terms 263
A.1 Terms and term algebras . . . 265

B Technical details **267**
B.1 Graph construction in LoTREC 267
B.2 Layout and display 268

Bibliography **270**

Index **278**

Acknowledgements

My research project would not have been possible without the support of many people.

First, I would like to thank my advisor Olivier Gasquet who was very helpful and offered me assistance, support and guidance. Thank you Olivier for the exceptional friendly relationship that you allowed me to establish with you. This made me feel always comfortable in our discussions and helped me communicate much easily with you.

I would also like to thank Andreas Herzig plentifully for his patience, guide and support. Without his knowledge and invaluable assistance this work would not have been successful. Thank you Andi for your great support and help, especially in the hard and critical moments.

I would like to thank abundantly my colleague and friend François Schwarzentruber for his exceptional help and support and for his high availability. I remember now the day we first met in office 315. After that day, my journey has completely changed. The least thing that I can say to express my deepest gratitude to you is "Thank you!".

My gratitude are also due to jury members, Jean-Paul Bodeveix, Serenella Cerrito and Didier Galmiche, for their valuable advises and remarks. Special thanks are due to Serenella for the detailed, complete and helpful review which helped me in enhancing the quality of this manuscript.

Special thanks also to all my graduate colleagues and friends, especially the LILaC team members, for sharing knowledge, experiences and invaluable assistance, especially Philippe Balbiani and Yannick Chevalier. My special thanks go especially to my friends, Fahima, Nadine, Mounira and Pablo, who have always been there, to my first office mate Tiago, with whom I had interesting discussions, and to my late office mates Sihem, Elise and Srdjan.

I would like to thank Maxime Rebout for helpful discussions on graph rewriting and Nicolas Troqard for discussions on ATL and STIT.

I would like to deeply thank my beloved family, without whom I would not have had the chance to travel to France and to realize this project.

And above all, I wish to express my love and gratitude to my beloved friend, colleague, office mate and wife Marwa for her understanding and endless love, through the duration of my whole Ph.D.'s journey.

Abstract

To model the functioning of a system, to describe a situation or to represent ideas, we begin to intuitively draw bubbles and connect them by arrows as labeled graphs. Modal logics offer a formal, expressive and scalable framework to define these graphs as "models", and to express certain properties of these graphs as "formulas". They allow then to reason on these graphs and properties: wether a model satisfies a certain formula or not (model-checking), or wether there is a model satisfying a given formula or not (satisfiability / validity). For formulas and models of large sizes, these tasks become complicated, and thus, we need a tool to achieve these reasoning tasks automatically. LoTREC is an example of such tools. It allows the user to create his own proof method, through a simple and high level language without any need to a specific expertise in programming.

During my Ph.D., I revisited the work that has been already done in LoTREC and I have brought new extensions that were needed to offer the users new implementation techniques. This allowed to handle new logics (such as $\mathsf{K.alt}_1$, the universal modality, Hybrid Logic $\mathsf{HL}(@)$, Intuitionistic logic, Public Announcement Logic, ...). This achievement required the development and the expansion of the software core of LoTREC, its language and its user interface. With the new version we can experiment in a step-by-step mode or use other options to debug our method, while visualizing and analyzing the models (and / or counter-models) generated and displayed in an pretty-print layout. When we define a semi-automatic procedure, we can also start with partial models and intervene during the execution.

On the other hand, I gave myself the goal to examine the origins of LoTREC in the world of graph rewriting and to specify the semantics of its rewriting engine. This work has lead to a clear presentation of the *event-driven* mechanism that manages the pattern matching process in LoTREC in an optimized way. My work has also helped in clarifying the semantics of this mechanism and in showing its advantage in comparison with other existing techniques.

In addition, this work has established the link between the rewriting system of LoTREC and one of the well established theoretical approaches known in the community of graph rewriting. This helps in clarifying how we can inherit in our proof methods defined in LoTREC some of the already established theoretical results and properties known in the field of graph rewriting.

Résumé

Pour modéliser le fonctionnement d'un système, décrire une situation ou représenter des idées, on se met intuitivement à dessiner des bulles et les lier par des flèches sous forme de graphes étiquetés. Les logiques modales constituent un cadre formel expressif, extensible et toujours d'actualité qui permet de définir ces graphes sous forme de "modèles", et d'exprimer certaines propriétés de ces graphes sous forme de "formules" afin de pouvoir raisonner là-dessus: vérifier si un modèle satisfait une certaine formule ou non (model checking), ou bien s'il existe un modèle satisfaisant une formule donnée ou non (satisfiabilité / validité). Pour des formules et modèles de tailles importantes, ces tâches deviennent compliquées. De ce fait, un outil permettant de les réaliser automatiquement s'avère nécessaire. LoTREC en est un exemple. Il permet à son utilisateur de créer sa propre méthode de preuve, grâce à un langage simple et de haut niveau, sans avoir besoin d'aucune expertise spécifique en programmation.

Durant ma thèse, j'ai revu le travail qui était déjà accompli dans LoTREC et j'ai apporté de nouvelles extensions qui s'avéraient nécessaires pour pouvoir traiter de nouvelles logiques ($\mathsf{K.alt}_1$, universal modality, Hybrid Logic $\mathsf{HL}(@)$, Intuitionistic logic, Public Announcement Logic, ...) et offrir à l'utilisateur certaines nouvelles techniques. Cela a exigé le développement et l'extension du noyau logiciel de LoTREC, ainsi que du langage et de l'interface qu'il offre à ses utilisateurs. Avec la nouvelle version on peut expérimenter avec ses formules afin de déboguer sa méthode en l'exécutant pas-à-pas, tout en visualisant et analysant les modèles (et/ou contre-modèles) générés d'une façon ludique. Quand on définit une procédure semi-automatique, on peut aussi démarrer avec des modèles partiels et intervenir durant l'exécution.

D'autre part, je me suis donné l'objectif d'examiner les origines de LoTREC dans le monde de réécriture de graphes et de spécifier la sémantique de son moteur de réécriture. Ce travail a permis de bien présenter le mécanisme événementiel qui gère le processus de "pattern matching" d'une façon optimisée. Mon travail a permis de clarifier la sémantique de ce mécanisme, et de montrer son avantage par rapport aux techniques existantes par des résultats empiriques. En plus, ce travail a permis d'établir un lien entre le système de réécriture de LoTREC et l'une des approches théoriques bien fondées et connues chez la communauté de réécriture de graphes. Cela a permis d'éclaircir comment l'on peut hériter dans nos méthodes de preuve des résultats et des propriétés théoriques déjà bien établies dans le domaine de la réécriture de graphes.

Introduction

This thesis is divided in two parts: the first part is about automated model construction in logic and the second one is about graph rewriting. I invite the reader to follow the parts and chapters in their given order, even though the two parts are relatively independent. In any case, the preface of each part should be read first, before getting into the details of each chapter. Here we give a brief introduction to both parts.

The first part of this thesis has a pedagogical style and rhythm. It takes the reader on a long journey in modal logics via their decision procedures. I hope that this part can attract every student and researcher in logic and philosophy. This is also useful for the sake of "LoTREC" itself, since this part is the most comprehensive tutorial and documentary that have ever existed to guide LoTREC' users.

In chapter 1, we show how to model real life situations by graphs. In chapter 2 we introduce the formal language to reason about these situations, and the main reasoning problems which we are interested in solving automatically in LoTREC. Starting from chapter 3, we present our model construction method which is used to solve the satisfiability problem. It answers the question about the satisfiability of formulas, and at the same time it gives an explanation for the answer. Throughout the chapters 4 to 7, we show how to adapt or redefine model construction methods for various logics. Chapter 6, is the only exception in this list, it addresses the model checking problem and presents our method to solve it.

In the last chapter of this part, I change the pedagogical style, the reader gets into a more advanced level. I investigate by a special model construction method a family of logics called Layered Modal Logics (LML). We use the model construction method to study the complexity class of these logics, instead of giving a tractable decision procedure for them.

In the second part of this thesis, I start a different topic: graph rewriting. This part starts with a brief, but complete, overview of graph rewriting systems and their theoretical foundations as seen by the community of researchers on this subject. This allows me, in the second chapter of this part, to define and introduce the graph rewriting system of LoTREC.

In chapter 11,we give more implementation details on the *event-driven* pattern matching process used in LoTREC. The first result presented in this chapter is the formalization of this mechanism. We show its importance and impact in

enhancing the performance of the rewriting engine of LoTREC, letting the time-cost of running a model construction method be related to the sole complexity of the underlying logic. However, this optimization does not alter the usual semantics of graph rewriting used in naive systems. Proving this postulate is the second result presented in this last chapter.

To conclude my work, I present some of the existing similar tools that were developed for academic purposes. I also show some empirical results comparing LoTREC to two other satisfiability provers. I close with a discussion of the main achievements and perspectives of my work.

Contributions of this thesis

The contributions of my thesis are in the domains of automated reasoning in logic on one hand and graph rewriting on the other hand. In both cases they are theoretical and practical.

My first contribution is a "gentle" introduction to modal logics, presented in the first part of this thesis, via the *model construction* method. This work shall be completed and published in the near future as a courseware book for students in computer science logic and philosophy, in a joint work with O. Gasquet, A. Herzig, and F. Schwarzentruber. We believe that this approach enables readers that do not have knowledge of modal logics to learn what these logics are about, and allows them to implement and play with existing or new modal logics. Most importantly, they should be able to do so *without any knowledge or skills in programming.*

In automated reasoning

In automated reasoning, my main aim was to complete the work on our generic platform LoTREC which has been established since 1999. The bases go back to the paper [CFnDCGH98], where graph rewriting rules were first introduced in order to check satisfiability in modal and description logics. LoTREC's graph rewriting engine was built in the software's first version, implemented by D. Fauthoux during his Master thesis [dCFG+01]. Simple logics, such as classical logic and modal logic K were successfully implemented. The second era for LoTREC was during the Ph.D. thesis of M. Saade [GHS06a]. In this era, a rudimentary graphical user interface was developed, and methods for the basic modal logics were implemented (KT, KB, KD, K4, S5,.... In addition, there was an attempt to implement a star-free PDL, a new extension of the language (with a special `createOneSuccessor` action) allowed to tackle linear time temporal logics such as LTL, and a special tweak in the rule definitions[1] allowed to implement the K+Confluence logic.

During my Ph.D. thesis, I completed this workflow while keeping an eye on abstract and reduced extensions in LoTREC's language. For example, the

[1] Setting a Boolean to true to declare a given rule as commutative, so that the rule considers commutative graph patterns as equivalent.

ad-hoc tweak needed for K+Confluence did not succeed in tackling hybrid logics such as HL(@). However, I found that the original problem which prevents us from implementing both methods, and many others, is the same[2].

This problem can be solved for LoTREC users by using a special strategy keyword `applyOnce`. However, its implementation was a bit complicated, since its implementation in LoTREC conflicts with its event-based mechanism. Once solved, the new solution allowed us to easily implement linear logics, such as $\mathsf{K.alt}_1$ and LTL, without the need for more keywords in our language. Similarly, the implementation of K+Confluence became possible without an ad-hoc Boolean attribute in the rules, and HL(@) was successfully implemented.

On the other hand, I developed a model checking method which covers most of the implemented logics. This allowed the implementation ofPDL with iteration with a simple method that uses a model checking procedure to check the eventualities. A simpler model checking procedure was also used for LTL.

In a joint work [GS07] with O. Gasquet, we used the model construction method, introduced as a tableau-like method, to express the frame properties of a special class of logics, that we call *Layered Logics*. We also used the method to investigate the complexity of these logics. In this work, we used a special filtration technique that we call *Dynamic filtration*, to keep the size of the developed premodels within an exponential boundary. In this manuscript, I give the way to implement this technique in LoTREC.

I also revisited previously implemented methods[3]. The dozen of predefined logics needed continuous updates to keep them coherent with the newer versions of LoTREC. Some of these old methods lacked termination, some others stopped earlier with node-inclusion test. The strategies of the former methods were corrected, and in the latter methods I replaced the node-inclusion test by the convenient node-equality check, which is stricter and more appropriate termination criterion.

In graph rewriting

LoTREC has a nice rewriting engine written from scratch. It has a special event-driven optimisation based on the Java event model [Ham97]. When I decided to present this feature to people working on graph rewriting [SG08], I discovered that I needed to learn more about their terminology and tools.

This is why I made my own survey on the theoretical foundations of graph rewriting and on existing implemented tools and techniques.

Besides learning about this domain, my aim was to trace back the roots of LoTREC in graph rewriting, which had not been done before. On one hand, this work helps people working on graph rewriting to understand what is happening inside LoTREC's engine. On the other hand, it gives people interested in developing similar rewriting tools a fair amount of what they need to build their tools on solid theoretical basis.

[2] We cannot apply a rule on only one occurrence of all successful patterns in our graphs.
[3] http://www.irit.fr/ACTIVITES/LILaC/Lotrec/LotrecOld/librairy

In the end, I succeeded in introducing to both audiences the event-driven optimisation of LoTREC using a common basic mathematical formalism. In a joint work with O. Gasquet and F. Schwarzentruber, we gave the semantics of LoTREC's rewriting system using a term notation. We also proved the soundness of this optimisation w.r.t. a naive graph rewriting system.

In order to evaluate this optimisation, I made some experiments and overviewed the state of the art on the related works tackling the pattern matching problem.

In implementation

At the mid-time of my Ph.D. thesis, I wrote on my webpage:

> *"I started my thesis as a student in logic with a background in software development, but I am ending up as a software engineer with a background in logics."*

This was after more than one year of programming new parts and re-coding many existing parts in LoTREC, in order to make it more coherent, performant and ergonomic. However, when the piece of software became almost[4] clean in front of me, I got back to my theoretical research in logic and graph rewriting and all the things in-between.

I discovered later that most of this technical work cannot be valuable when presented as a Ph.D. thesis work in a domain different from software engineering, such as artificial intelligence and logic. And thus I figured out the reason behind the rarity of educative and user-friendly tools in our domain. However, I dedicated the last part of the current section to mention some of these technical details.

I invested my effort in the renovation of the software architecture and graphical user interface. The rewriting engine is now under the control of the user: it offers a debugging panel for users definitions by allowing a step-by-step commented execution. Many of the old on-a-hurry extensions had affected the correct behavior of the system, and were not commented in the code. Such leaks were repaired and sometimes entirely replaced.

I find that the major utility of my work on software development was to offer an ergonomic and simple graphical user interface, with a pretty-print display of the graphs and an automatic layouting. My main aim in this was to make this whole rich-component interface accessible via the web. In this way, LoTREC becomes easily reachable by a wide public of students and researchers, and this is the key measure of how long LoTREC will survive in our academic community.

Today, I am at the end-time of my Ph.D. thesis. So I can reformulate the sentence on my webpage differently:

> *I started by playing around a piece of software, with background in logic, but I end up with more curiosity and passion for three beautiful and interrelated sciences: logic, mathematics and computer science.*

[4] because implementation never ends!

Introduction

Cette thèse est divisée en deux parties : la première partie concerne la construction automatique de modèles dans la logique modale, alors que la seconde partie concerne la réécriture de graphes. J'invite le lecteur à suivre les parties et les chapitres dans l'ordre donné, même si les deux parties sont relativement indépendantes. Je propose aussi que la préface de chaque partie soit lue en premier avant d'entrer dans les détails de chaque chapitre. Je donne ici une brève introduction aux deux parties.

La première partie de cette thèse a un style et un objectif pédagogiques. Elle prend le lecteur dans un long voyage dans des logiques modales via leurs procédures de décision. J'espère que cette partie puisse intéresser les étudiants et les chercheurs en logique et en philosophie. Elle est aussi utile pour le bien de "LoTREC" lui-même, puisque cette partie est le tutorial et la documentation les plus complets qui ont jamais existé pour guider les utilisateurs de LoTREC.

Dans le chapitre 1, nous montrons comment modéliser des situations réelles et comment les représenter ensuite par des graphes. Dans le chapitre 2, nous introduisons un langage formel, celui des logiques modales, pour pouvoir raisonner sur ces situations. Nous introduisons également les principaux problèmes de raisonnement auxquels nous nous intéressons à résoudre automatiquement dans LoTREC. A partir de Chapitre 3, nous présentons notre méthode de construction du modèle qui est utilisée pour résoudre le problème satisfiabilité. Il répond à la question sur la satisfiabilité des formules, et en même temps, il donne une explication de la réponse. Tout au long des chapitres 4 à 7, nous montrons comment adapter ou redéfinir les méthodes de construction de modèles pour différentes logiques. Le chapitre 6 est la seule exception dans cette liste : il aborde le problème de "model-checking" et présente notre méthode qui le résout automatiquement sous LoTREC.

Dans le dernier chapitre de cette partie, je change le style pédagogique, et le lecteur entre dans un niveau plus avancé. Je mène, par une méthode spéciale de construction de modèles, une enquête d'investigation sur une famille de logiques appelées "Layered Modal Logics" (LML). Nous utilisons la méthode de construction de modèles pour étudier la classe de complexité de ces logiques, au lieu de lui donner tout simplement une procédure de décision.

Dans la seconde partie de cette thèse, je commence un sujet différent: la réécriture des graphes ("graph rewriting"). Cette partie commence par une brève, mais complète, vue d'ensemble des systèmes de réécriture de graphes et de

leurs fondements théoriques, tel que c'est vu par la communauté des chercheurs dans ce domaine. Cela me permet, dans le second chapitre de cette partie, de définir et d'introduire le système de réécriture de graphe LoTREC.

Dans le chapitre 11, je donne plus de détails sur la mise en oeuvre du modèle *événementiel* ("event-driven pattern matching") utilisé dans LoTREC. Le premier résultat présenté dans ce chapitre est la formalisation de ce mécanisme. Je montre ensuite l'importance et l'impact de ce mécanisme dans l'amélioration de la performance du moteur de réécriture de LoTREC. En effet, cette optimisation réduit les coûts en temps et en gestion de mémoire d'une méthode de construction de modèles définie pour une logique donnée, et elle laisse les coûts uniquement dépendant de la complexité de la logique en question. Toutefois, cette optimisation ne modifie pas la sémantique habituelle de réécriture de graphes telle qu'elle est définie dans un système naïf et non-optimisé. Prouver ce postulat est le deuxième résultat présenté dans ce dernier chapitre.

Pour conclure mon travail, je présente, parmi les outils de preuves qui existent actuellement, quelques outils similaires à LoTREC et qui ont été développés pour des fins académiques. Je montre également quelques résultats empiriques comparant LoTREC à deux autres provers puissant, mais qui n'étaient pas conçus pour des buts pédagogiques. Je termine par une discussion sur les principales réalisations et perspectives de mon travail.

Les contributions de cette thèse

Les contributions de ma thèse se situent dans le domaine du raisonnement automatique dans la logique modale, d'une part, et dans le domaine de réécriture de graphes, d'une autre part. Dans ces deux domaines, mes travaux portent sur des aspects théoriques ainsi que sur des applications pratiques.

Ma première contribution est une introduction ludique et pédagogique à la logique modale. Cette introduction présentée dans la première partie de cette thèse, via la méthode de construction de modèles. Ce travail est en train d'être finalisé et sera publié dans un futur proche sous forme d'un livre qui accompagnera les cours de logique donnés aux étudiants en informatique ou en philosophie. Ce livre est préparer en collaboration avec O. Gasquet, A. Herzig, et F. Schwarzentruber.

Nous croyons que cette approche permettra à nos lecteurs d'apprendre ce que sont les logiques modales, sans aucune connaissance requise au préalable sur la logique. De plus, cette approche permettra aux lecteurs de définir sur papier des méthodes de preuves pour ces logiques. Ensuite, en utilisant notre logiciel LoTREC, les lecteurs seront capables d'implémenter ces méthodes et de les exécuter automatiquement, avant tout, *sans aucune connaissance ou compétence requises en programmation.*

Dans le domaine de raisonnement automatique

Concernant le raisonnement automatique, mon objectif principal était d'accomplir les travaux qui ont été mis en place depuis 1999 sur notre plate-forme générique LoTREC. Les bases de LoTREC remontent jusqu'au papier [CFnDCGH98], où les règles de réécriture de graphes ont été introduites afin de vérifier la satisfiabilité des formules en logiques modales de base et les logiques de description. Le moteur de réécriture de graphes de LoTREC a été ainsi construit dans la première version du logiciel, mis en oeuvre par D. Fauthoux au cours de son stage de master [dCFG+01]. Des méthodes pour deux logiques simples, la logique classique et la logique modale de base K, ont été implémentées avec succès. La deuxième période de développement de LoTREC était pendant le doctorat de M. Saadé [GHS06a]. À cette époque, une interface utilisateur graphique rudimentaire a été élaborée, et les méthodes de quelques logiques modales de base ont été mises en oeuvre (KT, KB, KD, K4, S5, ...). En outre, une tentative de mettre en oeuvre une méthode pour star-free PDL a eu lieu, une nouvelle extension du langage (avec une action spéciale `createOneSuccessor`) a permis de traiter la logique temporelle linéaire LTL et une astuce spéciale introduite dans la définition des règles[5] a permis d'implémenter la logique K + Confluence.

Au cours de ma thèse de doctorat, j'ai complété ce flux de travail tout en gardant le langage de LoTREC simple et minimal, et avec des extensions le plus génériques possible. Par exemple, la solution ad-hoc nécessaire pour la logique K + Confluence n'a pas réussi à traiter les logiques hybrides, telle que HL(@). Cependant, j'ai constaté que le problème initial qui nous empêche de mettre en oeuvre une méthode pour l'une de ces deux logiques, et bien d'autres, est le même[6].

J'ai résolu ce problème par l'implémentation d'une nouvelle routine de stratégie, `applyOnce`, qui peut être appelée juste avant le nom d'une certaine règle pour permettre de l'appliquer sur une seule occurrence de graphe. Toutefois, la mise en oeuvre de cette nouvelle stratégie était compliquée, vu qu'elle a des interférences et des conflits avec le mécanisme événementiel de LoTREC. Une fois ces difficultés ont été dépassées, la nouvelle solution a résolu le problème de commutativité de motifs et bien d'autres problèmes, et nous a permis d'implémenter facilement des logiques linéaires, tels que $\mathsf{K.alt}_1$ et LTL, sans avoir besoin d'autres mots clés dans notre langage. De même, elle nous a permis d'implémenter une méthode pour la logique K + Confluence sans utiliser des solutions ad-hoc, ainsi qu'une méthode pour la logique hybride HL(@).

D'un autre côté, j'ai développé une méthode de model-checking qui couvre la plupart des logiques modales. Cela a permis l'implémentation de PDL avec itération avec une méthode simple qui utilise une procédure de vérification de modèles pour vérifier la satisfaisabilité des éventualités. Une simple procédure

[5] La déclaration d'une variable booléenne pour déclarer une règle donnée comme commutative, i.e. pour demander que la règle ne soit pas appliquée sur deux motifs de graphes considérés comme commutativement équivalents.

[6] nos règles sont appliquées en parallèles, et une règle ne peut pas être appliquée sur un seul motif graphe à la fois.

de vérification de modèles a été utilisé également pour LTL.

Dans un travail en collaboration avec O. Gasquet [GS07], nous avons utilisé la méthode de construction de modèles, présentée comme une méthode de tableau, pour exprimer les propriétés du cadre d'une classe spéciale de logiques, que nous appelons "Layered Modal Logics" (LML). Nous avons également utilisé cette méthode pour étudier la complexité de ces logiques. Dans ce travail, nous avons utilisé une technique de filtration spéciale, que nous appelons *filtration dynamique* (Dynamic Filtration), pour maintenir la taille des prémodèles développés dans une borne exponentielle. De plus, je donne, dans ce manuscrit, la façon d'implémenter cette technique de filtration sous LoTREC.

J'ai aussi revu les méthodes qui étaient développées[7] avce les versions antérieures de LoTREC. La douzaine de logiques prédéfinies nécessitaient des mises à jour régulières pour conserver leur cohérence avec les nouvelles versions de LoTREC que j'avais développées. Certaines de ces anciennes méthodes avaient des problèmes de terminaison, d'autres s'arrêtaient trop tôt avec un test d'inclusion de noeuds, alors qu'il leur fallait continuer le calcul afin de donner de bonnes réponses. Les stratégies de certaines de ces méthodes ont été corrigées, et dans d'autres j'ai remplacé le test d'inclusion par un test d'égalité, qui s'est avéré un critère plus approprié pour garantir la terminaison de ces méthodes.

Dans le domaine de réécriture de graphes

LoTREC a un moteur de réécriture de graphes performant et entièrement "fait maison". Ce moteur dispose d'une optimisation spéciale basée sur le modèle événementiel de Java [Ham97]. Quand j'ai voulu présenter ce mécanisme aux personnes qui travaillent sur la réécriture de graphes [SG08], j'ai découvert que j'avais besoin d'apprendre davantage sur leur terminologie et sur leurs outils pour mieux m'adresser à leur communauté.

Pour cela, j'ai fait ma propre enquête sur les fondements théoriques de réécriture de graphes et sur les outils et techniques déjà existant.

Outre qu'apprendre plus sur ce domaine, mon objectif était de retracer les racines de LoTREC vers ce domaine en tant qu'un système de réécriture de graphes, ce qui n'a pas été fait auparavant. D'une part, ce travail aide les personnes travaillant sur la réécriture de graphes à comprendre ce qui se passe à l'intérieur du noyau de LoTREC. D'autre part, ce travail donne aux personnes intéressés par le développement d'outils de réécrire similaires les moyens nécessaires pour construire leurs outils sur une base théorique solide.

Enfin, j'ai réussi à introduire à ces deux publics l'optimisation "event-driven" de LoTREC, en utilisant un formalisme mathématique commun et de base. Dans un travail collaboratif avec O. Gasquet et F. Schwarzentruber [GSS09], nous avons donné la sémantique du système de réécriture de LoTREC en utilisant une notation de termes. Nous avons également montré dans ce travail l'adéquation de cette optimisation vis-à-vis de la sémantique traditionnelle d'un système de réécriture naïf.

[7] http://www.irit.fr/ACTIVITES/LILaC/Lotrec/LotrecOld/librairy

Afin d'évaluer cette optimisation, J'ai bien situé notre travail par rapport aux fameux travaux et techniques utilisés pour traiter le problème du "Pattern Matching", et j'ai comparé LoTREC à d'autres outils de réécriture et de preuve automatique afin de donner des résultats empiriques plus rigoureux.

En implémentation

À mi-chemin, durant ma thèse de doctorat, j'ai écrit sur ma page Web:

> *"J'ai commencé ma thèse en tant qu'un étudiant en logique avec une certaine base en développement logiciel, mais je me retrouve aujourd'hui un ingénieur de développement logiciel avec une certaine base en logique."*

C'était après plus d'un an de programmation de nouvelles partie et après le re-codage de nombreuses parties existantes sous LoTREC, afin de rendre ce logiciel plus cohérent, performant et ergonomique. Ce n'est qu'après que le logiciel est devenu presque propre[8] en face de moi, que je suis revenu à mes recherches théoriques en logique et réécriture de graphes et sur tout ce qui est entre ces deux domaines.

J'ai découvert plus tard que la plupart de ce travail technique ne peut pas être présenté comme un travail de thèse de doctorat dans un domaine, autre que celui de la génie logiciel, comme l'intelligence artificielle et la logique. Et c'est ainsi que j'ai compris la raison derrière la rareté des outils pédagogiques et conviviaux dans notre domaine. Mais vu le temps passé là-dessus, j'ai décidé de consacrer la dernière partie de la section courante pour mentionner certains de ces détails techniques.

J'ai investi une bonne partie de mon effort dans la rénovation de l'architecture logicielle et de l'interface graphique de l'utilisateur. Une partie concerne le moteur de réécriture. Maintenant, ce moteur est sous le contrôle de l'utilisateur: il offre un panel de débogage pour les définitions des utilisateurs en permettant une exécution pas-à-pas clairement commentée. Plusieurs des anciennes extensions "vite-faites" ont affecté le bon comportement du système, et n'ont pas été commentées dans le code. Ces parties du code ont été corrigées et parfois entièrement remplacées.

Je trouve que l'utilité majeure de mon travail sur le développement logiciel de LoTREC c'était de lui offrir une interface graphique ergonomique et simple d'utilisation, avec un joli affichage et une représentation claire des graphiques et avec un "layouting" rapide et automatique. Mon objectif principal était ensuite de rendre cet ensemble de composants d'interface graphique riches entièrement accessibles via le web. De cette façon, LoTREC devient facilement accessible par un large public d'étudiants et de chercheurs. Et, à mon avis, cette simplicité d'accès est une mesure clé à prendre pour pousser le plus loin possible la durée de vie de LoTREC dans notre communauté académique et universitaire.

Aujourd'hui, je suis à la fin de ma thèse. Je peux donc reformuler la phrase sur ma page Web différemment:

[8] parce que le développement d'un logiciel ne se termine jamais!

"J'ai commencé ma thèse en jouant autour d'une pièce de logiciel avec une certaine base en logique, mais je me retrouve enfin avec plus de curiosité et plus de passion pour trois belles sciences intimement liées: la logique, les mathématiques et l'informatique."

Part I

Model Construction for Normal Modal Logics

Preface to Part I

This part of the thesis aims at introducing the most important modal logics with multiple modalities, from the perspective of the associated reasoning tasks, such as model checking, satisfiability and validity. To that end we present a tableau-like method, that we call *model construction*, for each of the introduced logics.

The central idea of this method is to try to build a Kripke model for a given input formula by breaking it down connector by connector. While usual tableaux systems build proofs that take the form of trees, our model construction method works on graphs in order to be closer to the Kripke models.

Another difference with usual tableaux calculi is that tableaux are traditionally viewed as constructing a proof (by refutation) of the validity of a formula. In contrast, we focus on the *construction of a model* for the input formula, whence the name of our procedure.

The developed methods can be implemented in our generic platform LoTREC, thus they can be automated. LoTREC provides a simple and generic language to do so, while existing systems are difficult to adapt and require diving into "geek" programming code.

At the end of this part, we use this method to investigate the frame properties and the complexity of a family of logics, that we call *Layered logics*. To bound the size of the built models, we use a special filtration technique, that we call *dynamic filtration*. We also give the implementation details of resulting method.

In chapter 1 we give a brief overview on modelling real life situations with graphs. In chapter 2 we introduce the language of modal logics as a language to talk about properties of these graphs, then we define the reasoning problems that are tackled in the next chapters.

The model construction method is introduced in chapter 3. It is given for the basic cases of modal logics K and K_n. Throughout the chapters 4, 5 and 7, many variants are given for various logics (such as KT, $\mathsf{S5}$, $\mathsf{K.alt}_1$, $\mathsf{HL}(@)$, LTL and PDL). The difficulty level of these methods is proportional to their order of appearance in their corresponding chapters. Model checking is tackled in chapter 6. Finally, layered modal logics are investigated in chapter 8.

Préface à la Partie I

Cette partie de la thèse est une introduction à la logique modale via le raisonnement automatique. Elle introduit une grande variété de logiques modales classiques, ainsi que les tâches de raisonnement qui leur sont associées, tels que le model-checking, le test de satisfiabilité et le test de validité. À cette fin, nous présentons, pour chacune de ces logiques, une procédure semblable à la fameuse méthode de tableaux, que nous appelons ici *la méthode de construction de modèles*.

L'idée centrale de cette méthode consiste à essayer de construire un modèle de Kripke pour une formule donnée en la décomposant connecteur par connecteur. Tandis que la méthode de tableaux classique construit d'habitude les preuves sous forme d'arbres, notre méthode de construction de modèles construit des graphes qui sont plus proches des modèles de Kripke.

Une autre différence avec la méthode de tableau est que les tableaux sont traditionnellement considérés comme la construction d'une preuve (par réfutation) de la validité d'une formule. En revanche, nous nous concentrons sur la *construction d'un modèle* pour la formule d'entrée, d'où le nom de notre procédure.

Les méthodes développées dans les différents chapitres de cette partie peuvent être implémentées à l'aide de notre plate-forme générique LoTREC et donc automatisées. LoTREC offre un langage simple, générique et adapté pour le faire alors que les autres systèmes existants sont plus techniques et nécessitent de se plonger dans du code et de programmer pour les étendre afin de pouvoir traiter de nouvelles logiques.

À la fin de cette partie, nous utilisons la méthode de construction de modèles pour étudier les propriétés des cadres (frames) et la complexité d'une famille de logiques, que nous appelons *Layered Modal Logics* (LML). Pour limiter la taille des modèles construits pour une logique de cette famille, nous utilisons une technique de filtration spéciale, que nous appelons *filtration dynamique* (Dynamic Filtration). Nous donnons aussi dans ce chapitre les détails techniques nécessaires à la mise en oeuvre de cette opération de filtration sous LoTREC.

Dans le chapitre 1, nous donnons un bref aperçu sur la modélisation des situations réelles avec des graphes. Dans le chapitre 2, nous introduisons le langage de la logique modale comme un langage pour parler des propriétés de ces graphes. Ensuite, nous définissons les problèmes de raisonnement (model-checking, satisfiabilité, validité et construction de modèles) qui sont abordés

dans les chapitres suivants.

La méthode de construction de modèles est introduite dans le chapitre 3. Elle est donnée pour le cas de la logique modale K, puis étendue à la logique multi-modale K_n. Au long des chapitres 4, 5 et 7, on traite une grande variété de logiques modales (telles que KT, $\mathsf{S5}$, $\mathsf{K.alt}_1$, $\mathsf{HL}(@)$, LTL et PDL). Ces logiques sont présentées par degré de difficulté croissant. La procédure de model-checking est abordée au chapitre 6. Enfin, les logiques modales LML sont étudiées au chapitre 8.

Chapter 1

Modelling with graphs

Introduction

Graphs and different ways of constructing them are central in this thesis. In this chapter, we introduce graphs, focussing on their construction. We start by explaining how to *model* some real life situations with graphs. To *model* a situation means to analyse it in more and more details until we obtain a *rigorous representation* and a better understanding of it.

We model the functioning of a light according to what *actions* are performed on the toggle switch commanding it. We model mental states of players during a simple card game, and we show how to represent their *beliefs* and their *knowledge*. We also model the functioning of a traffic light by representing the change of its light color in *time*.

After that, we show how to construct in our tool LoTREC the graphs that are used in this chapter. This introduces one part of the simple declarative language of LoTREC which describes and construct labelled graph structures. At the end we give a formal mathematical definition of graphs.

1.1 Toggle the light

Let us start with a simple example: modeling the functioning of a system consisting of a light bulb and its switch (see Figure 1.1).

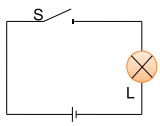


Figure 1.1: A light and a switch

The idea that first comes to our minds is: (1) "when the light is off, then

toggling the switch turns the light on". And it is true! This would well describe this lightening system.

A moment later, we start to think about other situations of this system, such as: (2) "when the light is on, then toggling the switch turns the light off". And this also describes our system.

After a while, we realize what is *generally* happening and we say that: "...actually, the *state* of the light is altered by the *action* of toggling the switch".

If we try to restrict our view of the world to this system, and if we want to represent it rigorously, then we would draw on a piece of paper the two possible states of the light: "Light_On" and "Light_Off". And we will express the alternation from the state "Light_On" to "Light_Off" with an *edge* linking the former to the latter state and labelled by the name of the action "Toggle" that makes the alternation possible. We do the same to express the alternation from "Light_Off" to "Light_On" (see Figure 1.2).

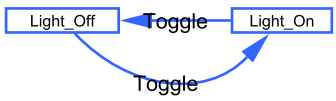


Figure 1.2: The two possible states of the light

Note that we model a system this way because we want to communicate sentences (1) and (2) to someone who reads our graph of Figure 1.2. Nevertheless, the reader may infer some other ideas about our system such as: the light can be either on or off, "Toggle" is the only action we can make, and so on .... And this is not a problem as long as these inferred properties are true in our system.

A model is said to be *rigorous* as much as it communicates well the main and correct properties of our system and as much as it does not express other false properties about our system.

Before we start to talk widely about properties of systems (as we will do later in chapter 2), we model some other examples. Then we show how to build the graph structures of these models using LoTREC. And we end by giving the formal definition of these graphs.

1.1.1 A light with two switches

Let us try to model the lightening system of Figure 1.3.

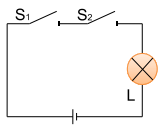


Figure 1.3: Two switches in series

As done in the first example, we should describe first the functioning of this system. Then we should try to extract a graph model that fits to represent this system as rigourously as possible.

At first sight, we notice that this system consists of a light and two switches in sequence. In this way, the light goes on if both switches S1 and S2 are toggled downward. Otherwise, if one of switches is toggled upward, the light goes off.

Since this system has two possible outputs, "Light_On" or "Light_Off", we draw two nodes with these labels to represent its *possible states*.

As for the actions that alter the system state, they consist in toggling up or down each of the switches S1 and S2. Thus, on the one hand, we link "Light_Off" to "Light_On" by an edge labeled "S1_Down_and_S2_Down". On the other hand, we link "Light_On" to "Light_Off" by three edges labeled by "S1_Up_and_S2_Down", "S1_Down_and_S2_Up" and "S1_Up_and_S2_Up".

The resulting graph is illustrated in Figure 1.4.

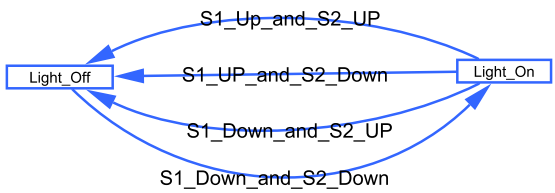


Figure 1.4: a model for a light and two switches in series

This was one possible way of modeling this system. However, in the above model, a label of an arc designates a configuration of the switches positions (S1 is Up and S2 is Down, or etc.) but it does not capture the action made on S1 or S2 in order to obtain the configuration. For example, according to this model, we can go from the state "Light_On" to "Light_Off" by "S1_Down_and_S2_Up". Than we can go back again to "Light_On" by "S1_Down_and_S2_Down". The action "Toggle_S2" executed during this last transition is not captured by its label. Moreover, if we go again from the state "Light_On" to "Light_Off" by the transition "S1_Up_and_S2_Up", the executed actions "Toggle_S1" and "Toggle_S2" are not revealed in the transition label. In addition, it is not clear if executing these two actions simultaneously is permitted or not.

Thus if we are interested in expressing more details of the system, we model it differently. For example, suppose that only atomic actions can be executed, i.e. that we can not toggle both switches simultaneously. In this case, the corresponding model should be completely different.

First, the states should be changed. They should encode, in addition to the state of the light (On or Off), the state of each switch (Up or Down). The set of possible states would consist of four states labeled as follows:

1. "Light_On", "S1_Down" and "S2_Down";

2. "Light_Off", "S1_Up" and "S2_Down";

3. “Light_Off”, “S1_Down” and “S2_Up”;
4. and “Light_Off”, “S1_Up” and “S2_UP”.

Second, a transition between two of these states should be labeled by the atomic action that is applied on S1 or on S2 in order to change the first state into the second one. For example, if the system is in the state 1, then toggling the switch S1 alters the system state to 2. The resulting model is shown in Figure 1.5.

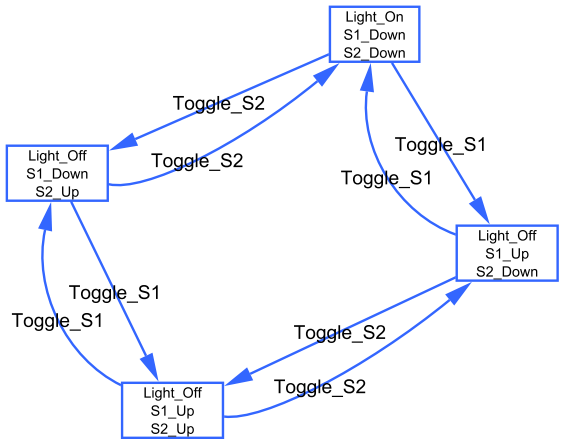


Figure 1.5: another model for a light and two switches in series

1.2 Ann is guessing the card’s color

Let us consider a simple card game. Ann and Bob have two cards: one is red and the other is black. Ann puts the cards face down on the table then closes her eyes. Bob then picks up a card and looks at it. Let us say it is red. Later, Ann has to guess the color of the card.

Suppose that we are interested in answering the following questions: what will be the answer of Ann? will she be sure of it? is Bob able to say if it is the right answer?

But to answer these questions we have to figure out first “what does Ann know about the card” and “what does Bob know about it”. Other interesting questions are “does Ann know what Bob knows” and “does Bob know what Ann knows”. In other words, we are interested in modeling the *knowledge* of Ann and Bob, i.e. their *mental states*.

Actually, Bob knows that the selected card is red. Whereas Ann does not know that. But she knows that it is either black or red. And she knows that Bob knows what it really is. Moreover, being as intelligent as us, Bob knows all that we have already said above. And Ann knows it too.

Now in order to give a graphical representation of these statements, we proceed as in Section 1.1: we try to figure out what are the possible states of this situation and what are the relations between these states.

Concerning the color of the selected card, our situation here has two possible states: "Red" where the color of the card is red, and "Black" where it is black. Thus we draw two nodes with these labels. We add the label "Actual_World" to the state "Red" to denote that it is the actual state.

It remains to add to our graph the information representing the knowledge of Ann and Bob about these states. We can use the edges to this end, as we have done in Section 1.1 to represent the actions that allow to go from a state of the system to another. Whereas in this section, we need to represent which state is *envisioned* or *known* by Ann and/or by Bob.

For instance, if Ann thinks that the situation is "Red", she still considers that "Black" is a *possible* situation. Thus we link "Red" to "Black" with an edge labeled by her name "Ann". We also link "Black" to "Red" with an edge labeled "Ann" since "Red" is a possible situation even if she thinks that it is actually "Black".

As for Bob, this does not hold, since he knows the right color. So if he thinks that it is red then the state "Black", where the color is supposed to be black, is not possible for him, and vice-versa.

Coming back to Ann, when she thinks that the actual situation is "Red" (resp. "Black"), she is considering "Red" (resp. "Black") as a possible situation too. Thus we link each state to itself with an edge labeled "Ann". The same reasoning holds for Bob. That is why we also link each state to itself with an edge labeled "Bob".

This way of modeling the mental states of Ann and Bob in this game situation gives the graph in Figure 1.6.

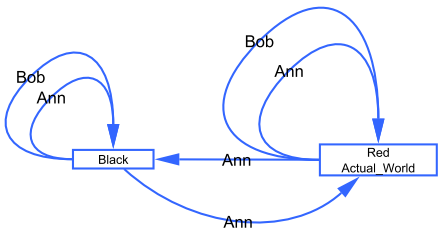


Figure 1.6: Ann guessing the color of Bob's card

1.2.1 What if Ann is cheating?

Suppose that Ann and Bob play again, but change the game rules so Ann might keep her eyes open. Bob does not know that there is a mirror behind him, so Ann can now see the card he selects. And suppose that he picked up the same red card again. How the model of Figure 1.6 would look like in this new situation? Let us try to figure it out together.

In fact, as for Bob, the situation has not changed at all: he still thinks that Ann does not know the color of the selected card, that she hesitates whether it is red or black and that he is the only one who knows that is red or black. As for Ann, she now knows what is the actual world: she is sure that the card color is red and she knows that Bob thinks that she does not know what the actual world is.

To describe this situation graphically, we should reconsider the graph model of Figure 1.6. We use this entire graph to represent the mental state of Bob. Except that we do not consider the "Red" state in it as the actual world. Hence we omit the "Actual_World" label from this state.

On the other hand, in order to represent the mental state of Ann, we draw a state labeled by "Red" and "Actual_World", and we link this state to itself by an arrow labeled "Ann", since it is the only *possible* state for Ann. Finally, in order to represent that she knows what Bob is thinking, we link this state to the "Red" state of the graph part copied from Figure 1.6.

The resulting model should look like the graph of Figure 1.7.

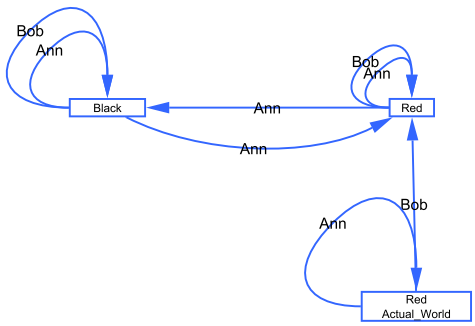


Figure 1.7: Ann is cheating

1.3 Talking about time

Let us consider a simple and common traffic lights system consisting of three colored lights: red, yellow and green. This system works endlessly over *time* in the same way. It illuminates in a precise *sequence*: the red light, then the green one, then the yellow, then red again, then green and so on. . . .

The states of the system and the relation between them are easy to be extracted from the above quick description. Obviously a graph representing a traffic light system would have three states: "Red", "Yellow" and "Green". And the relation between these states is talking about the order between these states more than anything else. The notion of time appears clearly in the periodic succession of these states.

That is why we choose to link the state "Red" to "Green", and we choose to label the edge linking them by "Then" to express the order between the *moments*

of their appearances. Similarly, we link "Green" to "Yellow" and "Yellow" to "Red", also by edges labeled "Then". The resulting graph is illustrated in Figure 1.8.

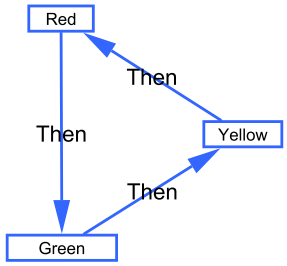


Figure 1.8: Traffic lights model

We settle for these three examples on *modeling with graphs*. Next, we show how to build such graphs in our tool LoTREC.

1.4 Build a graph using LoTREC

LoTREC offers a *declarative language* that allows to talk about graphs. One part of it is to describe graph structures by defining a set of *conditions*, and is introduced later in chapter 2. The second part is to build graphs by defining a set of *actions* of the form:

1. `createNewNode n`
2. `link n n' Label`
3. `add n Label`

The first action creates a new node in the graph and calls it `n`. The second action supposes the existence of two nodes `n` and `n'`. It creates an arc going from `n` to `n'` and labeled with `Label`. The third action supposes the existence of a node called `n` and adds to it the label `Label`.

Remark 1 (identifiers and values). In the above list of actions, `Label` is a *value*[1], whereas `n` and `n'` are node *identifiers*. The concrete values of identifiers are managed automatically by LoTREC. For example, when the action "`createNewNode n`" is executed, LoTREC creates a concrete *node-object* in the memory and assigns it to the identifier `n`. Suppose that the action "`add n Label`" is executed next, LoTREC then adds `Label` to the node-object assigned to `n` as expected.

Example 1. The graph of Figure 1.2 is constructed in LoTREC using the following set of actions:

[1]It is a *constant* value since it starts with a capital letter.

```
createNewNode state1
createNewNode state2
add state1 Light_On
add state2 Light_Off
link state1 state2 Toggle
link state2 state1 Toggle
```

The built graphs can be displayed and manipulated in LoTREC. The reader can refer to annexe B.1 to learn how to practically execute these actions under LoTREC, and to discover some other technical details.

1.5 Labeled Directed Graphs

A *directed graph* is a set of nodes and a set of directed edges, where each edge is linking a given source node to a given target node. A *labeled directed graph* is a graph where the nodes and edges are labeled by elements of a given set $\mathtt{L}$.

A node can have several labels. In contrast, an edge can have just one label, that is, an edge is identified by its source node, its target node and the label of the link connecting the source to the target. In addition, two edges with the same source and target nodes should have different labels.

Definition 1. *Given a set of labels* $\mathtt{L}$*, a labeled graph* $\mathtt{M}$ *is a tuple* $(\mathtt{W}, \mathtt{R}, \mathtt{V})$ *where:*

- $\mathtt{W}$ *is a non-empty set;*
- $\mathtt{R}$*:* $\mathtt{L} \longrightarrow 2^{\mathtt{W}\times\mathtt{W}}$*;*
- $\mathtt{V}$*:* $\mathtt{W} \longrightarrow 2^{\mathtt{L}}$*.*

$\mathtt{W}$ is the set of *nodes*, also called *states*. $\mathtt{R}$ denotes a set of labeled *edges*, also called *transitions*, over elements of $\mathtt{W}$. $\mathtt{V}$ is a *labeling function* which associates to each node in $\mathtt{W}$ a set of labels .

Conclusion

In this chapter, we modeled some real life situations with labeled graphs. We showed how to build such graphs in our tool LoTREC. Then we gave a formal definition of these graphs.

The advantage of modeling a situation or a system with a graph is not only a matter of graphical representation. As we shall see in the next chapter, we can talk about *properties* of these systems and *facts* in these situations using a special formal language.

Hence, the main use of modeling is to be able to rigourously *check* and *verify* these properties and facts on those models.

Chapter 2

Talking about models

Introduction

In the previous chapter, we saw that situations (such as knowledge about the color of cards), scenarios (such as traffic lights), machines (such as a switch-bulb system), etc. can be represented by graphs. We can also imagine how to similarly model various other systems such as a coffee machine, an automatic control system of the doors of a subway train, etc.

In this chapter we shall see how we can talk about such things without directly referring to the nodes and the edges of these graphs. To this end, we use a concise formal language that allows for some kind of quantification over edges and nodes. Our language allows the expression of properties such as "The light is on and after toggling the switch it turns out off", "Ann knows that Bob does not know the light is on", "The next state after Red is Green", "After having opened the door, the door is opened and I know that the sun is shining".

We start by introducing modal languages (section 2.1). Then we show how we can define such languages in LoTREC (section 2.2). After that, we formally state the meaning for "a formula is *true* at a node of a graph" (section 2.3.1). Finally, we define several reasoning problems consisting in verifying and evaluating the truth of formulas (section 2.4).

2.1 A formal language to talk about graphs

2.1.1 Motivation

In chapter 1, we used natural language to describe graphs. For instance, we wrote sentences in English like "The light is on and I am happy" or "Ann knows that the card is red". One may imagine that we can talk with the computer directly in english . Nevertheless, there are many drawbacks. First, english vocabulary is enormous so it is fastidious to create a computer being able to treat all english sentences. Secondly, english is ambiguous and this is tricky,

especially for computers[1]! Indeed there are syntactic issues like in "They are flying planes" [Cho02] or semantic issues like in "Most politicians are preoccupied by many problems".

Fortunately, we just want to express some precise sentences and we will be able to create suitable artificial formal languages in order to do so.

2.1.2 Boolean connectors

First we want to be able to describe physical situation. For instance, simple sentences, such as "the light is on", "the card is red" and "there is a fault", will be directly represented by keywords like $Light\_on$, $Card\_red$ and $Fault$. Those sentences can not be divided in smaller parts: we say that they are *atomic*. Those keywords are called *atomic propositions*. You can think of atomic propositions as the smallest sentences you can write in our formal language and in a given situation. Such a proposition is either true or false.

Now, we also want to describe more complicated situations like "the card is red and the light is on" or "if there is a fault then the card is red". So we need some *boolean connectors* to combine atomic propositions together. We need boolean connectors like $\wedge$ (and), $\rightarrow$ (implies[2]), $\vee$ (or), $\neg$ (not). We write:

- $Card\_red \wedge Light\_on$ for "the card is red and the light is on";
- $Fault \rightarrow Card\_red$ for "if there is a fault then the card is red";
- $\neg Card\_red$ for "the card is not red.".

Our boolean connectors can also combine non atomic sentences together:

- $Referee\_sleeping \vee (Fault \rightarrow Card\_red)$ for "either the referee is sleeping or a fault leads to a red card";
- $\neg(Fault \rightarrow Card\_red)$ for "it is false that if there is a fault then the card is red.";
- $\neg Fault \rightarrow Card\_red$ for "if there is no fault then the card is red".

With atomic propositions and boolean connectors, we are able to talk about one state of a system or a situation. Nevertheless, we also want to describe other states. For instance, we may want to talk about the *current state* of the light and its *possible states after toggling* the switch. We may also want to express the possible states that agents may imagine in a given game in addition to the current state of the game that they know, etc. That is why we introduce *modal connectors*.

[1] We want the computer to be able to understand our language, since we want it to automatically check our formulas.

[2] Logical implication can be misunderstood. We think that reading $(P \rightarrow Q)$ as (P implies Q) may infer a cause/effect relation between P and Q. Hence we prefer to read $(P \rightarrow Q)$ as (if P then Q). In this reading the focus is more on the fact that P and $\neg Q$ can not hold altogether. A good practice is to exercise these two readings with (Earth_is_flat $\rightarrow$ Santa_Claus_exists).

2.1.3 Modal connectors

We want to represent time, actions and believes of agents. We want to be able to express with our formal language sentences like "The door is closed but after having opened the door, it will be opened", and like "Ann knows that the light is on". That is why we need to introduce *modal connectors* like $[open]$ ("after having opened the door"), K_{Ann} ("Ann knows that") and X. For instance, we may write:

- $[Open]Door\_opened$ for "after having opened the door, the door is opened";
- $K_{Ann}Card\_red$ for "Ann knows the card is red";
- X Red for "next time, the light will be red".

We can mix boolean connectors and modal connectors in one sentence of our artificial language. For instance:

- $Door\_closed \wedge [Open]Door\_opened$ stands for "The door is closed and after having opened the door, it will be opened";
- "$K_{Ann}(Referee\_sleeping \vee (Fault \rightarrow Card\_red)) \wedge \neg K_{Ann}K_{referee}Fault$" stands for "Ann knows that either the referee is sleeping or a fault leads to a red card but she does not know that the referee knows that there is a fault";
- X $(Red \vee Green)$ stands for "next, the light will be either red or green".

So far, we have defined an artificial language in which we are able to express simple sentences, boolean composed ones and those representing time, knowledge of agents and actions. A sentence in our artificial language is called a *formula*. Before giving a formal definition of formulas, we discuss how they can be more clear and concise than sentences formulated in natural language.

2.1.4 Infix versus prefix notation

The formula $K_{Ann}(Referee\_sleeping \vee Fault)$ can be read just in one way: "Ann *knows* that: either the referee is sleeping or there is a fault". Whereas $K_{Ann}Referee\_sleeping \vee Fault$, in which we omit the parentheses, can be read in two different ways: as the above reading or as "*Either*: Ann knows that the referee is sleeping, *Or* there is a fault". The problem is that K_{Ann} is the *main connector* according to the first reading, whereas $\vee$ is the main connector according to the second one.

However, in the first section of this chapter we insist on the fact that we want to create this whole formal language in order to avoid the possible syntactical and/or semantical ambiguities that may rise in informal natural languages.

That is why we should disambiguate the reading of a formula. This can be done if we know the main connector in the formula. And this can be done in three possible ways:

- adding *all* the necessary *parentheses*,
- using a *prefix* form for defining formulas,
- or defining a set of *grammar rules*, as done in general to define compilers and parsers [LMB92].

The first solution is to use parentheses to change the reading of a formula when it may rise ambiguities.

The second solution consists in writing the parts of the formula in the same order in which we read them. To do so, we start by writing the main connector of the formula. Then we write, also in prefix form, each subformula successively. For example, the formula $K_{Ann}Referee\_sleeping \vee Fault$ is written $\vee K_{Ann}Referee\_sleeping\ Fault$ according to the reading "*Either*: Ann knows that the referee is sleeping, *Or* there is a fault".

The third solution consists in defining a set of rules which specify how to read the formulas. The main information coded in these rules is a predefined order of priority between the connectors. It allows to designate only one main connector for a formula, when it is defined in an infix style with missing parentheses. For example, if we set an order in which the priority of the connector K is higher than the priority of $\vee$, then the formula $K_{Ann}Referee\_sleeping \vee Fault$, in which the parentheses are missing, can be read just in one way: "*Either*: Ann knows that the referee is sleeping, *Or* there is a fault".

Nevertheless, the priority is not the whole story, especially when the rules are to be interpreted by the machine which have to recognize the formulas automatically.

2.1.5 Formal definition

We give the formal definition of the set of all formulas as follows:

Definition 2 (formula)**.** *Let $\mathcal{P}$ be a set of atomic propositions. Let $\mathcal{I}$ be a set of labels. The set $\mathcal{F}or$ is defined as the smallest set such that:*

- $\mathcal{P} \subseteq \mathcal{F}or$;
- *if* $A, B \in \mathcal{F}or$, *then* $(A \wedge B) \in \mathcal{F}or$;
- *if* $A, B \in \mathcal{F}or$, *then* $(A \vee B) \in \mathcal{F}or$;
- *if* $A, B \in \mathcal{F}or$, *then* $(A \rightarrow B) \in \mathcal{F}or$;
- *if* $A \in \mathcal{F}or$, *then* $\neg A \in \mathcal{F}or$;
- *if* $A \in \mathcal{F}or$ *and* $I \in \mathcal{I}$, *then* $[I]A \in \mathcal{F}or$.

Remark 2. Note that the parentheses are usually omitted when there is no ambiguity in the reading of a given formula.

The first point $\mathcal{P} \subseteq \mathcal{F}or$ means that each proposition is a formula. For instance, $Fault$, $Light\_on$ and $Card\_red$ are formulas. The second point means that if we connect a formula with another formula with the connector $\wedge$, we obtain a formula. For instance, since $Light\_on$ and $Card\_red$ are formulas, $Light\_on \wedge Card\_red$ is a formula too. The last point means that if A is a formula and I is a label, then $[I]A$ is a formula. For instance $Light\_on \wedge Card\_red$ is a formula, $Open$ is an label, thus $[Open](Light\_on \wedge Card\_red)$ is a formula.

As we have seen, modal connectives may have various interpretations. A generic reading of the formula $[I]A$ is: "A is *necessarily* true w.r.t. the parameter I".

Extending the language

In the next chapters we extend this language to take into account other connectors, so that other formulas become a part of the language. For example, we may wish to add to our language formulas of the form $K_I A$ or $\mathrm{X}A$, for $I \in \mathcal{I}$ and $A \in \mathcal{F}or$.

In the following subsections, we discuss more notations and more details about the syntactical definition of formulas.

2.1.6 Arity of connectors

The formula $Card\_red \wedge Light\_on$ is composed from its *two* subformulas by the mean of the $\wedge$ connector. Also, the formula $\neg Fault$ is obtained from applying the negation boolean connector $\neg$ on the formula $Fault$. In fact, the number of formulas that can be combined by a given connector is an appropriate characteristic of this connector that is called *arity*.

For example[3], $arity(\wedge) = 2$ means that the arity of $\wedge$ is 2, and $arity(\neg) = 1$ means that the arity of $\neg$ is 1. A connector of arity 1 is called *unary* and a connector of arity 2 is called *binary*.

2.1.7 Analyzing a formula

Propositions of the set $\mathcal{P}$, such as $Card\_red$, $Light\_on$, $Referee\_sleeping$ and $Fault$, are *atomic* in the sense that they can not be decomposed. In contrast, sentences like $Card\_red \wedge Light\_on$ is not atomic. It can be decomposed in $Card\_red$ and $Light\_on$. Since these two formulas are smaller than $Card\_red \wedge Light\_on$, we say that $Card\_red$ and $Light\_on$ are *strict subformulas* of $Card\_red \wedge Light\_on$.

The formula $K_{Ann}(Referee\_sleeping \vee Fault)$ consists of the connector K_{Ann} surrounding the strict subformula $Referee\_sleeping \vee Fault$. This latter has two other strict subformulas: $Referee\_sleeping$ and $Fault$, combined by the connector $\vee$. Hence, all of these *smaller* formulas are strict subformulas of $K_{Ann}(Referee\_sleeping \vee Fault)$. In addition, $K_{Ann}(Referee\_sleeping \vee$

[3]The notion of arity in logic is the same as in arithmetic. Each arithmetic operator is applied a fixed number of arguments, e.g. the arity of $+$ is two, and the arity of $\sqrt{\ }$ is 1.

Fault) is also a subformula of itself, but it is not a strict subformula since it is not strictly smaller than itself.

To compute all the subformulas of a given formula, we notice that we need a *recursive* procedure, as we state in the following definition:

Definition 3 (subformulas)**.** *Given a formula A, the set of subformulas of A denoted by $Sub(A)$, is recursively defined as:*

- $Sub(P) = \{P\}$;
- $Sub(\neg B) = \{\neg B\} \cup Sub(B)$;
- $Sub(B \wedge C) = \{B \wedge C\} \cup Sub(B) \cup Sub(C)$;
- $Sub(B \vee C) = \{B \vee C\} \cup Sub(B) \cup Sub(C)$;
- $Sub([I]B) = \{[I]B\} \cup Sub(B)$.

This definition can be extended to take into account other connectors.

2.2 Syntax declaration in LoTREC

In this subsection, we show how to define a language in LoTREC, so that we can use the formulas of this language as labels for the graphs.

We give in LoTREC an equivalent definition to enable it to recognize the formulas of this language, and *only* these formulas.

Recall that the definition of a language consists in defining the basic sets of symbols $\mathcal{P}$ and $\mathcal{I}$, a set of connectors ($\wedge$, $\vee$, ..., [.], ...) and a set of syntactical rules to make precise which formulas are acceptable in the language.

2.2.1 Defining basic symbols

In LoTREC, we fix a predefined set of constant symbols to denote the propositional symbols in $\mathcal{P}$ and labels in $\mathcal{I}$. It is the same set for all the languages. It consists of "words starting with capital letters", such as `Red`, `P`, `Ann`, `I`,... to represent respectively *Red*, *P*, *Ann*, *I*,....

To define the formulas, we let the users define their own connectors in a uniform way, then we use a unique syntactical rule to construct the formulas over the constant symbols and the connectors.

To define a given connector, we ask the user to specify:

- a *name* (`and`, `not`, ...) to represent the special symbol ($\wedge$, $\neg$ ...) of the connector; it can be any word starting with a small letter;
- a positive integer value to designate its *arity* value;

Example 2. We give in the following table the typical definition of some connectors in LoTREC:

logical connector	definition in LoTREC	
	name	arity
$\neg$	`not`	1
$\vee$	`or`	2
$\wedge$	`and`	2
$[.]$	`nec_i`	2
$K_{.}$	`knows`	2

2.2.2 Defining formulas

The formulas of the language are inductively defined in LoTREC according to the following generic rules:

- every constant symbol from $\mathcal{P}$ is a formula,
- given a connector c of arity n and formulas $A_1, \ldots, A_n$, $c\ A_1 \ldots A_n$ is a formula[4];
- there are no other formulas in this language.

Example 3. According to the connectors defined in Example 2, the formula $K_{Ann}(Referee\_sleeping \vee Fault)$ is written in LoTREC as `knows Ann or Referee_sleeping Fault`.

We can also define the following formulas:

formula on paper	definition in LoTREC
$[I]P \wedge \neg P$	`nec_i I and P not P`
$Q \vee \neg(P \wedge \neg P)$	`or Q not and P not P`

2.2.3 Prefix notation

In LoTREC, we require to use prefixed notation to define and write the formulas, i.e. to write `and P not P` instead of `P and not P`. This may burden the users at first, especially new users, so one may wonder: why not using the alternative styles with parentheses or with an infix notation? (shown in Section 2.1.4).

As for the appealing infix parentheses-free notation, it is complicated to be defined by the users. First, they have to learn how to define grammar rules. Second, they have to define a precedence order between the rules, which corresponds to the connectors priorities. Then they have to define many tweaks to disambiguate the rules themselves, such as right- and left-associativity, how to proceed in case of "Shift/Reduce" conflicts, ... (the whole chapter 8 of [LMB92]

[4]Since there is no ambiguity in the prefix notation, we omit the parentheses from the usual $c(A_1, \ldots, A_n)$ notation.

is dedicated to discuss this subject). Thus, this task is very though, even for computer-scientists.

Forcing the use of parentheses allows to avoid the need to define some of these tweaks. However, the use of parentheses all the time is not less cumbersome than the prefix notation. In addition, it is less safe, since when some parentheses are missing, misunderstandings between the machine and the user rise again.

Thus we believe that we have good reasons to settle for the unambiguous prefix notation with easy-to-define rules, especially that LoTREC's users are mainly students and researchers in logic and philosophy, and they are not necessarily computer-scientists.

2.2.4 Customized display

In LoTREC, we allow the user to choose for the prefix-defined formulas a different *output display*, such as the familiar infix notation. This makes ease the reading of the formulas, especially when they are big and numerous on the screen.

A special output display for a given connector is a string of arbitrary characters with exactly n underscores "`_`", where n is the arity value of the connector, to specify how the n parameters of the connector are to be displayed.

Example 4. In LoTREC, the typical output displays of the $\neg$, $\wedge$, $\vee$, [.] and $K_{.}$ connectors (defined in Example 2) are $\sim$ `_`, `(_ v _)`, `(_ & _)`, `[_]_` and `K(_)_` respectively.

According to these displays, the formula `knows Ann or Referee_sleeping Fault` is displayed as `K(Ann)Referee_sleeping v Fault`. The other formulas of Example 3 are displayed as follows:

definition in LoTREC	display in LoTREC
`nec_i I and P not P`	`[I](P & P)`
`or Q not and P not P`	`(Q v ~ (P & ~P))`

2.3 A formal way to evaluate formulas

In this section, we put the definition of graphs (Chapter 1) together with the definition of a property with a rigourous language to answer some interesting questions, such as: "Given a graph, given a specific state in the graph, does a given property hold at this state of this graph?".

Answering this question is one of the reasoning problems in logic, which will be introduced in the sequel. First, we give a mathematical definition of the model which formally describes a graph. Then we show how to evaluate formulas w.r.t. models.

2.3.1 Models

In chapter 1, we modelled various systems with labelled graphs. In this section, we give alternative and more mathematical definition of such structures. These

mathematical structures are named after Saul Kripke: *Kripke models*.

In a Kripke model we specify:

- the different states, also called *possible worlds*;
- the transitions between these states, also called the *accessibility relation*;
- and the *valuation* of these states.

In such a model, the transitions are labeled by elements of the set of labels $\mathcal{I}$, and the valuation specifies, for each possible world, which atomic propositions of $\mathcal{P}$ hold at that world.

Definition 4 (Kripke model)**.** *Given a set of atomic propositions $\mathcal{P}$ and a set of indexes $\mathcal{I}$, a Kripke model M is a tuple (W, R, V) where:*

- *W is a non-empty set;*
- *$R : \mathcal{I} \longrightarrow 2^{W \times W}$;*
- *$V : W \longrightarrow 2^{\mathcal{P}}$.*

Remark 3. In the sequel, we may abbreviate $R(I)$ in R_I and we may use wR_Iu to denote that $(w, u) \in R_I$.

The definition given above is generic, in the sense that it describes arbitrary models. One may add some constraints and restrictions on the accessibility relation, the valuation function or on the basic sets of symbols $\mathcal{P}$ and $\mathcal{I}$ of the underlying language. For example, we may require that the accessibility relation is *reflexive*, *symmetric* or *transitive*. We may require that some special propositions hold at a *unique* possible world. Such constraints define some special families of models, which we call *classes* of models. Throughout the chapters 4 to 7, we shall meet many of these special classes.

2.3.2 Truth conditions

Given a model M, a state w in M and a formula A, we want to define what it means that a formula A is true at the world w of the model M. For instance, if M represents a coffee machine, w represents the state "Your coin has been inserted", how can we define that the formula $[select\_coffee]Coffee$ ("after having selected a coffee, I will have a coffee") is true?

Formally, We write $M, w \Vdash A$ for "the formula A is true at the world w of the model M", and we define it as follows:

Definition 5 (truth conditions)**.** *We define $M, w \Vdash A$ by induction:*

- *$M, w \Vdash P$ iff $P \in V(w)$;*
- *$M, w \Vdash \neg A$ iff $M, w \nVdash A$;*
- *$M, w \Vdash A \wedge B$ iff $M, w \Vdash A$ and $M, w \Vdash B$;*

- $M, w \Vdash A \vee B$ *iff* $M, w \Vdash A$ *or* $M, w \Vdash B$*;*
- $M, w \Vdash A \rightarrow B$ *iff* $M, w \Vdash A$ *implies* $M, w \Vdash B$*;*
- $M, w \Vdash [I]A$ *iff for every world* u*,* wR_Iu *implies* $M, u \Vdash A$

According to this definition, a proposition P is true at a world w of a model M if, and only if the the valuation of w contains P. A formula of the type $\neg A$ is true at a world w of a model M iff the formula A is false in the world w of a model M. A formula $A \wedge B$ is true at a world w of a model M iff both formulas A and B are true at w. The last item says that formula $[I]A$ is true at a world w of a model M iff A is true at *all the possible worlds* u of *that are accessible from* w by the relation R_I in the model M.

2.3.3 Dual modal operators

According to our official reading, $M, w \Vdash [I]A$ means that A is *necessarily* true at w w.r.t. the parameter I. If $\neg[I]\neg A$ is true at a world w then, according to the above truth conditions, there exists a world u, accessible form w by the relation R_I, such that $M, u \Vdash A$. We say in this case that A is *possibly* true at w w.r.t. I, and the reason is that $\neg A$ is not necessarily true at w. Hence, the notion of possibility and the notion of necessity are dual.

However, to lighten the notation of possibility, we use a special modal connector, which is usually denoted by $\langle I \rangle A$. We define its corresponding truth condition as follows:

$M, w \Vdash \langle I \rangle A$ iff there exists a world u, such that wR_Iu and $M, u \Vdash A$

When the language is enriched with new connectors, we define their truth conditions to correctly specify their semantics, as we did here for the $\langle\rangle$ connector. For the moment, we settle for the above truth conditions, and we postpone the definition of the many other connectors, such as K and X, to the chapters where they are introduced.

Next we define the various reasoning problems in which we are interested in general, and which we intend to solve automatically using our tool LoTREC.

2.4 Reasoning problems

Many questions seem to be challenging to be answered about formulas and their truth values w.r.t. models. For example, does the truth value of a formula in a specific world of a model change when it is considered in another world of the model? can we find a formula that is true in every world of a given model? does the answer to this question change when a specific kind of models is considered? Are we able to define the formulas which are true in every model, or in every model of a certain kind? what happens if we change the truth conditions or add new ones? etc.

In this section, we reformulate some of these questions in form of a set of problems that we usually call *reasoning problems*.

2.4.1 Model checking

Given a model M and a world w, we want to check automatically whether a formula is true or not at w. For instance, given a model describing a coffee machine mechanism, we want to be able to check using the computer that the formula $[select\_coffee]Coffee$ is true at the world "Your coin has been inserted". This procedure is called *model checking*. Formally, the model checking problem has the following input and output:

- Input: a model M, a world w of the model, a formula A;
- Output: do we have $M, w \Vdash A$?

2.4.2 Satisfiability

With different inputs and outputs, we obtain different problems, such as the problem with:

- Input: a formula A;
- Output: is there a model M and a world w of M, such that $M, w \Vdash A$?

This problem is rather a *quest for a model* which makes the formula true, i.e. *which satisfies the formula*. Hence the name: *satisfiability problem*.

This general definition makes the search space very large: we look for *an arbitrary model* M, which satisfies the formula A. Typically, we add some constraints on the accessibility relation or the valuation function of the sought models (see Section 2.3.1). These constraints restrict the search space to some special classes of models. In such cases, the satisfiability problem is denoted as C-satisfiability problem, where C is some special class of models.

2.4.3 Validity

Some formulas, such as $P \vee \neg P$, seem to be true at any world of any model. Dually, it seems that its negation $\neg P \wedge P$ is *not satisfiable*, i.e. we cannot find a model which satisfies this formula. We say that the formula $P \vee \neg P$ is *valid*. While this single formula can be seen to be valid, we are still curious about checking the validity of any formula A. Hence the following problem:

- Input: a formula A;
- Output: is $M, w \Vdash A$ for every possible world w of every model M?

This is the *validity problem*. Note that the satisfiability and validity problems are dual. In fact, when the satisfiability problem is solved for the formula $\neg A$, if the answer is "No, there is no model M such that for a world w of M, $M, w \Vdash \neg A$", then the validity problem for the formula A is solved, too; the answer is: "Yes, A is valid". Conversely, when the answer is "Yes, there are a

model M and a world w such that $M, w \Vdash \neg A$", the answer for the validity problem is "No, A is not valid".

Just as the satisfiability problem, the validity problem can be also studied w.r.t. a special class of models C. In this case, we call it the C-validity problem, and we will be interested in knowing the C-valid formulas.

2.4.4 Model construction

Solving the satisfiability problem for a given formula A is to answer by "Yes" or "No". However, we may be rather interested in an explanation of these answers: why "Yes"? and why "No"?.

Especially when students are learning a new logic, or when researchers are studying a special class of models or prototyping a new one, they need to debug these answers, to understand the searched models and worlds and to visualize and analyze them. They need these explanations to better understand the behavior of special formulas, or simply because they are the authors of the decision procedure that was used to check the satisfiability and they may be misdefining a bit of code in their procedure.

For all these reasons, we are rather interested in the following problem:

- Input: a formula A;
- Output: a model M and a world w such that $M, w \Vdash A$.

This is the *model construction* problem, which consists in *computing* a model for the input formula. During the model construction process, a model, which does not satisfy the input formula A, is a *counter-model* for this formula and explains why "Yes, $\neg A$ is satisfiable" (i.e. why "No, A is not valid"). Whereas a model which satisfies A is an explanation of the answer "Yes, A is satisfiable" (i.e. "No, $\neg A$ is not valid").

Note that when the model construction method delivers *only* counter-models for the formula A (resp. $\neg A$), it answers "Yes" the question about the validity of the formula $\neg A$ (resp. A), and gives in addition all the possible explanations of this answer.

Similarly to the formerly defined C-satisfiability and C-validity problems, the model construction method may be restricted to compute only models which belong to some special class of models C.

Conclusion

In this chapter, we defined a formal language to express formulas rigourously, and we showed how to define a formal language in our tool LoTREC. Then we gave a formal framework to evaluate formulas. We also presented some challenging problems related to the evaluation of formulas. However, we did not talk about possible solutions to these problems, especially automated approaches to solve them. This is what we are going to do, starting from the next chapter.

We address the model checking problem later in Chapter 6. Among the three other problems, we choose to handle the model construction problem. We do so, since model construction subsumes the satisfiability and the validity problems, and in addition it has some interesting educative features.

Model construction methods for various logics and their implementations in our tool LoTREC are explained throughout the next chapters.

Chapter 3

The model construction method

Introduction

Usual tableaux systems are quite close to Gentzen sequent systems. The latter build proofs that take the form of trees, breaking down a given input formula connector by connector.

Such sequent systems can be defined quite straightforwardly for many basic modal logics, whose models can more or less be identified with trees. However, Kripke models are not limited to trees, and both sequent and tableaux systems for the corresponding logics are more difficult to design: somewhat contrarily to the spirit of Gentzen systems, they typically require cumbersome meta-linguistic side conditions. Contrasting with almost all existing tableaux systems that are tree-based, LoTREC works on graphs in order to get around that difficulty, and is therefore much closer to the Kripke models.

Another difference with usual tableaux calculi is that —just as sequent systems—, the latter are traditionally viewed as constructing a proof (by refutation) of the validity of a formula. In contrast, we focus on the construction of a model for the input formula. Thus for us, a tableaux calculus is rather a *model construction* procedure.

In this chapter, we show how to define such a procedure to solve problems of the form:

- Input: a formula A;
- Output: a model M and a world w such that $M, w \Vdash A$.

We explain first how to define such a method on paper (section 3.1). Then we explain the used terminology (section 3.2). After that, we show how to declare an equivalent automated method in our tool LoTREC (section 3.3), and we give its full set of rules (section 3.4). We then extend this method to deal

with two other logics: multimodal logic K_n (section 3.5), and K_n with a simple interaction between multiple modalities (section 3.5.1). Finally, we discuss how to certify the methods defined in LoTREC: i.e. how to make sure that they stick to their theoretical design on paper (section 3.6).

3.1 Model construction by hand

To introduce the model construction method, we consider the most simple case: the monomodal logic K. Its language is defined according to Definition 2 with a singleton set of indexes $\mathcal{I}$. Thus instead of having multiple relations R_I and many multimodal connectors $[I]$, we only have one relation, say R, and one modal connector, usually denoted by $\Box$. The truth conditions defined in Section 2.3.2 hold also in K, with the only difference that the truth conditions corresponding to the $[.]$ and $\langle . \rangle$ connectors become:

$M, w \Vdash \Box A$ iff for every world u, wRu implies $M, u \Vdash A$;

$M, w \Vdash \Diamond A$ iff there exists a world u, such that wRu and $M, u \Vdash A$.

Note that in K, a model is an arbitrary model. Thus there are no constraints on the accessibility relation nor on the valuation function.

Let us consider the formula $A = \Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$, and let us try to develop a model for it. First of all, we need a convenient data structure that can embed a potential model. To this end, we use a *labeled graph* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ (as defined in chapter 1). To the possible world w_0 s.t. $M, w_0 \Vdash A$ we will associate the node $\mathtt{w}_0$ in $\mathtt{M}$ s.t. $A \in \mathtt{V}(\mathtt{w}_0)$.

During the development of $\mathtt{M}$, we keep on the equivalence between the notion of "the formula A should *be true at* the world w" (i.e. $M, w \Vdash A$) and the notion of "the formula A should *belong to* the node $\mathtt{w}$" (i.e. $A \in \mathtt{V}(\mathtt{w})$).

For example, assuming that in a model M there is a world w s.t. the formula $A \wedge B$ is true at w (formally $M, w \Vdash A \wedge B$) imposes that both A and B *should be true* at w, according to the truth conditions. Thus, during the construction of the corresponding labeled graph $\mathtt{M}$ where $\mathtt{w}$ is the node associated to w, we make sure that $A \wedge B$ *belongs to* $\mathtt{w}$ (i.e. $A \wedge B \in \mathtt{V}(\mathtt{w})$). Then we extend $\mathtt{V}$ to have A and B belonging to $\mathtt{w}$ (i.e. $A, B \in \mathtt{V}(\mathtt{w})$). We say also that "we add A and B to $\mathtt{w}$".

Note that our assumption could be false and that no model exists at all. For instance, if we assume that $P \wedge \neg P$ is true at a world w of a model M, then it is required that we add both P and $\neg P$ to the associated node $\mathtt{w}$ of the corresponding labeled graph $\mathtt{M}$. However, no model can be made up from $\mathtt{M}$.

That is why, during the model construction, we call the used labeled graph a *premodel* (i.e. a quasi-model or a pseudo-model). After finishing the construction of a premodel, we check whether it can be extended to a model or not (see section 3.1.6).

In addition, we use "world" to denote a "node" from the premodel. And we state that: given a world $\mathtt{w}$ of a premodel $\mathtt{M}$, the sentence "A has been added to

$\mathtt{w}$" (i.e. $A \in \mathtt{V}(\mathtt{w})$) becomes interpreted as "A is true at $\mathtt{w}$" when, and only when, $\mathtt{M}$ is extended to a model.

Returning to our example formula $A = \Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$, we give in the sequel the detailed steps of its premodel construction.

3.1.1 Initialisation

Since we assume the existence of a model M s.t. the pointed model $M, w_0 \Vdash A$, we start with an initial premodel $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ where $\mathtt{W} = \{\mathtt{w}_0\}$, $\mathtt{R} = \emptyset$ and $\mathtt{V}(\mathtt{w}_0) = \{\Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))\}$, i.e. a single world $\mathtt{w}_0$ containing the input formula A, as shown in Figure 3.1.

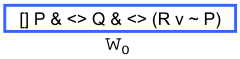


Figure 3.1: `premodel`: initial world and input formula

3.1.2 Classical saturation ($\wedge$)

Due to the truth conditions of an $\wedge$-formula (section 2.3.2), assuming that there is a model M s.t. $M, w_0 \Vdash A$, i.e. assuming $M, w_0 \Vdash \Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$, imposes both constraints:

- $M, w_0 \Vdash \Box P$ and
- $M, w_0 \Vdash \Diamond Q \wedge \Diamond(R \vee \neg P)$.

Practically, we simply add all the consequences that should be drawn from A to the world $\mathtt{w}_0$. That is why we extend the labelling function of $\mathtt{M}$ so that it becomes: $\mathtt{V}(\mathtt{w}_0) = \{\Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P)),\ \Box P,\ \Diamond Q \wedge \Diamond(R \vee \neg P)\}$.

By the same reasoning, concerning the new assumption $M, w_0 \Vdash \Diamond Q \wedge \Diamond(R \vee \neg P)$, we will have a new premodel $\mathtt{M}$ with the labelling function $\mathtt{V}(\mathtt{w}_0) = \{\Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P)),\ \Box P,\ \Diamond Q \wedge \Diamond(R \vee \neg P),\ \Diamond Q,\ \Diamond(R \vee \neg P)\}$.

These two steps are illustrated in Figure 3.2.

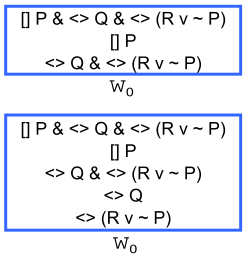


Figure 3.2: `And` rule applied successively twice on `premodel`

3.1.3 Successors creation ($\Diamond$)

According to the truth conditions, assuming that a formula $\Diamond A$ is true at a world w imposes having a possible world u accessible from w (i.e. $(w, u) \in R$) and such that A is true in u.

So considering the formula $\Diamond Q$ in the node $\mathtt{w}_0$ we should have a successor node, let it be $\mathtt{u}$, accessible from $\mathtt{w}_0$ (i.e. $(\mathtt{w}_0, \mathtt{u}) \in \mathtt{R}$), and such that $Q \in \mathtt{V}(\mathtt{u})$. Considering $\Diamond(R \vee \neg P)$ on the other hand, we should have a successor node $\mathtt{v}$ accessible from $\mathtt{w}_0$ s.t. $R \vee \neg P \in \mathtt{V}(\mathtt{v})$. Figure 3.3 sketches the premodel obtained at this computation step.

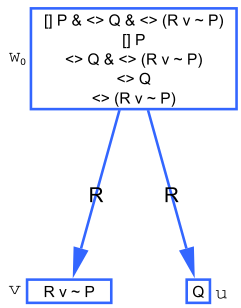


Figure 3.3: `Pos` rule applied on `premodel`

One may ask: why to create two different possible worlds $\mathtt{u}$ and $\mathtt{v}$ for each of the above $\Diamond$-formulas? The answer is simple: since we are interested in maximizing the chance of finding a model, we would look toward avoiding clashes as long as possible. Thus we chose to create a new different possible world for each $\Diamond$-formula so that their subformulas will be separated in two different valuation sets.

3.1.4 Propagation ($\Box$)

Now we consider the other new assumption: $M, w_0 \Vdash \Box P$. Referring to the corresponding truth conditions, $\Box P$ is true at w_0 iff P holds at every successor of w_0. That is why we extend the last premodel to the one where $\mathtt{V}(\mathtt{u}) = \{Q, P\}$ and $\mathtt{V}(\mathtt{v}) = \{R \vee \neg P, P\}$. This is illustrated in figure 3.4.

3.1.5 Disjunction ($\vee$)

A disjunction is satisfied if either one of its subformulas is. For instance, the formula $R \vee \neg P$ appearing in $\mathtt{V}(\mathtt{v})$ imposes that one of the formulas R or $\neg P$ belongs to the labelling function of $\mathtt{v}$.

That is why we duplicate our premodel $\mathtt{M}$ in two other premodels: $\mathtt{M}_1$ and $\mathtt{M}_2$ that are very similar to the previous premodel, except that in the first one we add R to $\mathtt{v}$, whereas in the second we add $\neg P$ to $\mathtt{v}$.

$\mathtt{M}_1$ and $\mathtt{M}_2$ are represented by `premodel.1` and `premodel.2` in figure 3.5.

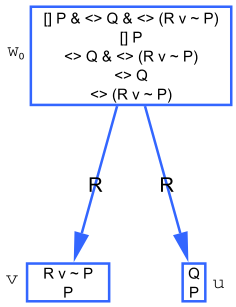


Figure 3.4: `Nec` rule applied on `premodel`

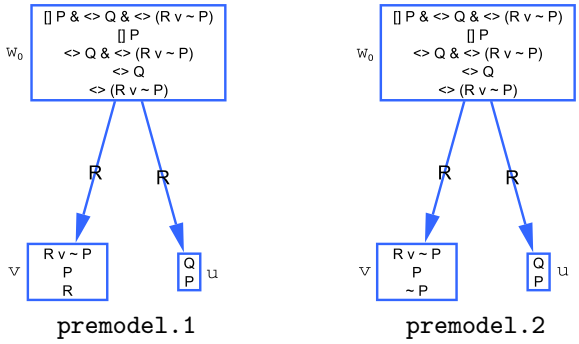


Figure 3.5: `Or` rule applied on `premodel`: creates `premodel.1` and `premodel.2`

3.1.6 Extracting a model

At this step, the satisfiability of the input formula A amounts to the satisfiability of the smaller subformulas appearing in $\mathtt{V}(\mathtt{u})$ and $\mathtt{V}(\mathtt{v})$: the atomic propositions and their negations (P, Q, R and $\neg P$).

In other words, all the consequences of the truth conditions have been drawn, and no further decomposition is needed in the obtained premodels. However, these premodels are not both extensible to a model M for A.

In fact, $\mathtt{v}$ in $\mathtt{M}_2$ (`premodel.2`) holds both an atom and its negation, P and $\neg P$. This raises a contradiction at the world corresponding to $\mathtt{v}$, and $\mathtt{M}_2$ cannot be transformed to a model. Hence, it can not be extended to a model for A (Figure 3.6).

On the other hand, the premodel $\mathtt{M}_1$ (`premodel.1`) can be extended to the model $M = (W, R, V)$ (Figure 3.7) such that:

- $W = \{\mathtt{w}_0, \mathtt{u}, \mathtt{v}\}$;
- $R = \{(\mathtt{w}_0, \mathtt{u}), (\mathtt{w}_0, \mathtt{v})\}$
- $V(\mathtt{w}_0) = \emptyset$, $V(\mathtt{u}) = \{Q, P\}$ and $V(\mathtt{v}) = \{P, R\}$.

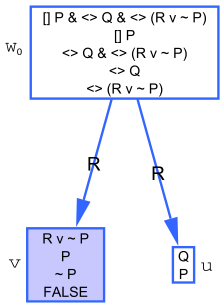


Figure 3.6: There is a clash in node `v` of `premodel.2` since it contains both P and $\neg P$.

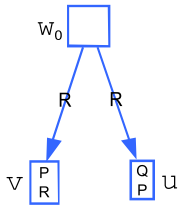


Figure 3.7: The model M extracted from $\mathtt{M}_1$.

To sum it up, we constructed in this section a semantical proof of the satisfiability of a formula A by trying to construct a model M and a world w such that A is true at w.

3.2 Terminology

In the introduction of this chapter, we discussed some of the contrasts between our model construction method and usual tableau methods. The first main difference we talked about, is that our model construction method uses graphs, which we call premodels, instead of trees. Thus first we define this graph structure.

3.2.1 What is a premodel?

To embed the constructed premodels in a convenient graph structure, we chose labelled graphs (see Definition 1). The nodes and edges are labelled by elements of the labels set $\mathtt{L} = \mathcal{I} \cup \mathcal{F}or$. However, we restrict our premodel definition to allow a node to be labeled with a subset of $\mathcal{F}or$, and an edge to be labeled with just one element of $\mathcal{I}$.

Definition 6 (premodel)**.** *A* premodel *is a labeled graph* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *where:*

- $\mathtt{W}$ *is a finite set of nodes;*

- $\mathtt{R}\colon \mathcal{I} \to 2^{\mathtt{W}\times\mathtt{W}}$ *is a family of binary relations over* $\mathtt{W}$ *and indexed by* $\mathcal{I}$*;*
- $\mathtt{V}\colon \mathtt{W} \to 2^{\mathcal{F}or}$ *is a function which maps each element of* $\mathtt{W}$ *to some set of formulas.*

3.2.2 What is a rule?

Back to the contrasts with tableaux, another main difference is that our method is not a sequent-like tree construction (e.g. [Gor99] and [Mas00]). It is rather a stepwise "pseudo-model" construction (e.g. [Bal00]). This step-by-step construction may bee seen as rewriting a graph starting from an initial node containing the input formula. At a given step, a premodel is called *partial*. In order to reach the premodel of the next step, we use an appropriate set of graph rewriting rules (or simply rules) which *only* add elements (nodes, edges and formulas) to the structure of the current premodel.

A graph rewriting rule is usually defined as a pair of left-hand side graph and right-hand side graph. When the left graph is found in the host graph where the rule is applied, it is replaced by the right hand side graph.

In our model construction method, the rules are monotonic, in the sense that when the left graph is found, only new graph elements (nodes, edges and formulas) are added to the host premodel. Hence, we define our rules by two sets:

- a set of conditions which describe the left-hand side graph (this is explained later in section 3.3);
- a set of actions which specify the set of elements (nodes, edges, formulas) to be added to the current premodel in order to obtain the next premodel.

3.2.3 Other notations

A common criteria of tableau and model construction is the notion of saturation of rules application, which leads, in case that the method is terminating, to a *complete* tableau in the classical approach, which we actually call a *complete premodel* in our approach.

Definition 7 (complete premodel)**.** *A* complete premodel *is a partial premodel on which no rule applies, more precisely a complete premodel is the least fixed point of a sequence of partial ones.*

In classical tableau, we may have different tableau *branches*, due to the non-deterministic choices in disjunctions or other special kind of rules. In our approach, we duplicate the current premodel in what we call premodel *copies* in order to consider each choice aside. A given premodel and its copy share the same structure that was developed before the fork. In addition, our rules continue in constructing all the premodels copies, exactly as tableau rules keep on exploring all tableau branches.

According to these last two similarities, saturation and branching, we define the notions of *closed* and *open* (complete) premodel as they are usually defined in the tableau method to distinguish between open and closed tableau branches.

3.3 Automated model construction

In section 3.1, we created a model for the formula $A = \Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$ by hand. In this section, we are going to see how it is possible to do this model construction using the software LoTREC.

3.3.1 Language of rules

Let us recall the example given in section 3.1.

Classical rules ($\wedge$)

In that example, we first considered the connector $\wedge$. Given $\Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$ in a world w, we had to add $\Box P$ and $(\Diamond Q \wedge \Diamond(R \vee \neg P))$ in w. Secondly given $(\Diamond Q \wedge \Diamond(R \vee \neg P))$ in a world w, we added $\Diamond Q$ and $\Diamond(R \vee \neg P))$ in w.

These two steps are very similar in the sense that they are concerning a formula of the form $A \wedge B$. In the first step, we had $A = \Box P$ and $B = (\Diamond Q \wedge \Diamond(R \vee \neg P))$. In the second step, $A = \Diamond Q$ and $B = \Diamond(R \vee \neg P))$. That is why symbols A and B are called *variables*.

To treat the connector $\wedge$, we proceed as follows: if there is a formula of the form $A \wedge B$ in a given node `w`, we have to add the formula A and the formula B to `w`. This is can be achieved by defining a *rule* in LoTREC.

As shown in Section 3.2, a rule consists of two parts: conditions and actions. Here, we have the following:

- condition: there is a formula $A \wedge B$ in a given node `w`;
- actions:
  - add formula A to node `w`;
  - and add formula B to node `w`.

In LoTREC, in order to test if there is a formula in a node, we use the primitive `hasElement`. For instance to test if a node `w` contains a formula of the form $A \wedge B$, we write: `hasElement w and variable A variable B`.

We also remark that we use here the same prefix notation introduced in section 2.2. You can also remark that we have used the keyword `variable`. When you write `variable A`, A is then stated as a variable. Otherwise, LoTREC will interpret A as a constant. Note that there is no constant symbols to denote fixed nodes, and that a node identifier `w` always denotes a variable. Hence, we shortly write in the rule `w`, as a node identifier, instead of `variable w`.

To add a formula to a given node, we simply use the primitive `add` defined in Chapter 1. For instance in order to add formula A to `w` we write

add w variable A. Note that it is the instance formula assigned to variable A that is added to the instance node assigned to w.

Here is the whole definition of the rule dealing with the connector $\wedge$ in LoTREC:

```
Rule And
 hasElement w and variable A variable B

 add w variable A
 add w variable B
End
```

Remark 4. Note that this And rule is not applicable anymore on the current premodel. Although we do not remove the formula $\Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$ which is matched to the formula $A \wedge B$, the And rule is not applicable once again on the same formula. As if LoTREC is considering treated formulas as *old elements*, while inviting the rules to be applied only on *new elements* recently added to the premodel graph. The mechanism which guarantees this finite rules application, and hence guarantees the termination of the whole premodel construction process, is explained later in Chapter 11.

Successor creation ($\Diamond$)

In the same way, the successor creation can be defined as a rule with:

- a condition: a world w containing a formula of the form $\Diamond A$;
- actions:
  - add a new node we can call u;
  - link node w to u by a relation R;
  - add formula A to the node u.

In LoTREC, we write:

```
Rule Pos
 hasElement w pos variable A

 createNewNode u
 link w node1 R
 add u variable A
End
```

For further explanation on these action keywords, you may refer to Section 1.4 or to Appendix B.1.

Propagation ($\Box$)

In the same way, the propagation can be defined as a rule. Just recall the truth condition of the $\Box$ operator: $M, w \models \Box A$ iff for all $u \in W$, such that wRu we

have $M, u \models A$. Basically, if a node `w` contains $\Box A$, we need to add A in all successors of `w`. This can be expressed not only with one condition but with two conditions in order to be able to capture the successors of `w`. More precisely, if we want to add A to a successor `u` of `w` we need not only a world `w` containing a formula of the form $\Box A$ but also a link from `w` to `u`. Here is the description of the rule:

- conditions:
  - a node `w` containing a formula of the form $\Box A$;
  - but also have a node `u` such that `w` is linked to `u`;
- action: add A to node `u`.

In LoTREC, we test if a node is linked to another one using the primitive `isLinked`. To test if `w` is linked to `u` by the relation R, we write `isLinked w u R`. The propagation rule is defined as follows:

```
Rule Nec
 hasElement w nec variable A
 isLinked w u R

 add u variable A
End
```

Disjunction ($\vee$)

The case of disjunction is a bit special. Here is the description of the rule:

- condition: a node `w` contains a formula of the form $A \vee B$;
- actions:
  - we duplicate the current premodel;
  - in one, we add A to `w`;
  - in the other one, we add B to `w`.

In LoTREC, the duplication is done with the keyword `duplicate`. The action `duplicate premodel_copy` will duplicate the current premodel where `premodel_copy` denotes the copy of the current premodel. Then we can have access to world `w` of `premodel_copy` by writing `premodel_copy.w`. Accessing `w` in the current premodel is still possible by simply writing `w` as usual. Here we can add A to `w` in the current premodel (`add w variable A`) and add B to `w` in the copy (`add premodel_copy.w variable B`).

```
Rule Or
 hasElement w or variable A variable B

 duplicate premodel_copy
```

```
 add w variable A
 add premodel_copy.w variable B
End
```

Clash rule (*False*)

In case of the presence of a formula A and its negation $\neg A$ at the same node `w`, a clash should be reported and *False* is to be add to `w`. As an option, we can ask LoTREC to stop further explorations in a clashing premodel, since it will not be extensible to a model at the end.

To do so, we define the following rule:

- conditions:
  - a node `w` contains a formula A;
  - `w` also contains the formula $\neg A$;
- actions:
  - add the falsum constant "*False*" to `w`;
  - (optionally) stop developing the current premodel.

The only new keyword needed for this rule is `stop`, which is however self-explanatory, and the rule is defined as follows:

```
Rule Stop
 hasElement w variable A
 hasElement w not variable A

 add w False
 stop
End
```

3.3.2 Saturation with repeat

In the last subsection, we have shown how to define the rewriting rules that are understood by LoTREC, and which *correctly* mimic the steps of the by-hand model construction (given in subsection 3.1) for the formula $A = \Box P \wedge (\Diamond Q \wedge \Diamond(R \vee \neg P))$. In order to apply these rules automatically in LoTREC, we need to call them in what we call a *strategy*.

For the example formula A, a possible working strategy is first to apply rule `And`, then rule `And` again, then rule `Pos`, then rule `Nec`, then rule `Or`, and finally rule `Stop`.

In LoTREC, this strategy is written like this:

```
Strategy Strategy_For_The_Example
    And
    And
```

```
    Pos
    Nec
    Or
    Stop
End
```

Remark 5 (Parallel rule application). A given rule in LoTREC is applied *wherever it is possible at once*, when it is called by the strategy. Hence, it is sufficient to call the rule `Nec` just *once* in the strategy, in order to propagate the formula P of $\Box P$ at $\mathtt{w}_0$ to *both successors* of $\mathtt{w}_0$, `u` and `v` (see Figure 3.4).

One may wonder: what if I want to solve the satisfiability problem of another formula? Let us say $P \wedge (Q \wedge (R \wedge \neg P))$. Does the same strategy fit to solve it? If we had checked out the last examples, we should have already guessed that the answer is "no".

In fact, in order to treat completely the formula $P \wedge (Q \wedge (R \wedge \neg P))$, you need to apply the rule `And` three times. If you look at the formula $P \wedge (Q \wedge (R \wedge (S \wedge \neg P)))$ you will need to apply the rule `And` four times. Generally speaking, to deal with such formulas, you need to apply the rule `And` as much as necessarily. We need to repeat the application of the rule `And` as much as it is needed. To do this, LoTREC provides the keyword `repeat` and `end`.

Here is the strategy to develop a formula with $\wedge$:

```
Strategy Strategy_For_And
  repeat
    And
  end
End
```

You can also repeat many rules application by writing name of rules to apply between the keyword `repeat` and `end`. Here is an example:

```
Strategy Example_Of_Strategy
  repeat
    And
    Or
    Pos
    Nec
  end
End
```

This strategy works right for our example, as well as many other formulas, but not for all of them, as we shall see in the next section.

3.4 The full set of rules

The set of rules, defined so far in the previous sections, is not sufficient to treat some other formulas. For instance, the formulas $\neg\neg P$, $\neg(P \wedge Q)$ and $P \rightarrow R$ cannot be handled by the rules introduced in section 3.3 alone. These formulas

rise up from combination of unary and binary connectors, from successive connectors imbrication or from complex connectors that can be rewritten in simpler basic ones according to the truth conditions. For example, verifying $\neg\neg P$ imposes verifying P. The formula $\neg(P \wedge Q)$ is true iff $\neg P$ is or $\neg Q$ is. And $P \rightarrow R$ holds iff $\neg P$ holds or R holds.

In addition to the rules given in the subsection 3.3, `Stop, And, Or, Nec` and `Pos`, we need to define the following rules :

```
Rule NotNot
 hasElement w not not variable A

 add w variable A
End

Rule NotOr
 hasElement w not or variable A variable B

 add w not variable A
 add w not variable B
End

Rule NotImp
 hasElement w not imp variable A variable B

 add w variable A
 add w not variable B
End

Rule NotAnd
 hasElement w not and variable a variable b

 add w or not variable a not variable b
End

Rule Imp
 hasElement w imp variable a variable b

 add w or not variable a variable b
End

Rule Equiv
 hasElement w equiv variable a variable b

 add w or not variable a variable b
 add w or not variable b variable a
End

Rule NotEquiv
 hasElement w not equiv variable a variable b
```

```
 add w or variable a variable b
 add w or not variable a not variable b
End
```

These rules are self-explanatory and we may verify their meanings by checking the corresponding truth conditions.

As for formulas of the form $\neg\Box A$ and $\neg\Diamond A$, we use the following two rules:

```
Rule NotPos
 hasElement w not pos variable a

 add w nec not variable a
End

Rule NotNec
 hasElement w not nec variable a

 add w pos not variable a
End
```

These rules reflect the duality of necessity and possibility $\neg\Box\neg A \leftrightarrow \Diamond A$, explained in Section 2.3.3.

All the above rules, except `Imp` and `Equiv`, are computing in fact the *negation normal form* (NNF) of the formulas. However, this is not done in a preprocessing of the input formula, but achieved all along the model construction process.

A simple strategy is to call the above rules repeatedly as follows:

```
Strategy K_Strategy
  repeat
    Stop
    NotNot
    And
    NotOr
    NotAnd
    NotImp
    NotEquiv
    Imp
    Equiv
    Or
    NotNec
    NotPos
    Pos
    Nec
  end
End
```

3.4.1 Cut rules

In addition to the previous rules, we can define some of the so called *cut* rules, which avoid unfruitful non deterministic choices when it is possible. For in-

stance, it is clear that adding $\neg P$ to the node `v` of the `premodel.2` (Figure 3.5), where we have already added $R \vee \neg P$ and P, will end up in a contradiction (Figure 3.6). Hence, it would be better, from a computational point of view[1], to directly add R to `v`, and to prevent the rule `Or` from being applied on the occurrence of $R \vee \neg P$.

This can be achieved by defining a new set of *cut* rules and calling them in the strategy before calling the `Or` rule. A cut rule lookups for a $\vee$-formula that can be simplified: it adds its consequence subformula and *marks* this $\vee$-formula as `Cut`. For example:

```
Rule CutOr_not_B_B
 hasElement w or variable A not variable B
 hasElement w variable B

 add w variable A
 markExpressions w or variable A not variable B Cut
End
```

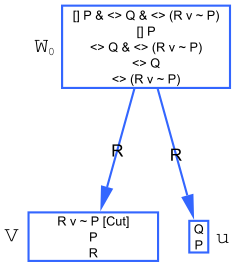


Figure 3.8: Applying a cut-rule gives only one premodel at the model construction step of Figure 3.5.

When called on the premodel of Figure 3.5, this rule adds R to the node `v` and marks the formula $R \vee \neg P$ by `Cut`. The mark of a formula is displayed in LoTREC between brackets in front of the formula, as shown in Figure 3.8.

Remark 6. Marks are harmless meta-data that can be added by the users to annotate formulas or nodes, in order to give additional information to some rules, or to the users themselves.

The `Or` rule should be changed too. When it finds an occurrence of an $\vee$-formula, it should test first for the absence of the mark `Cut` before being applied on it. Here is the right version of the `Or` rule and

```
Rule Or_With_Cut
 hasElement w or variable A variable B
 isNotMarkedExpression w or variable A variable B Cut

 duplicate premodel_copy
```

[1]Since duplicating a premodel in another copy is time and space consuming.

```
 add w variable A
 add premodel_copy.w variable B
End
```

We notice that `Or_With_Cut` has only one difference with the original `Or`: an additional condition to verify that the $A \vee B$ formula, on which the rule is going to be applied, is not marked by `Cut`.

We can define some other cut rules as follows:

```
Rule CutOr_A_A
 hasElement w or variable A variable B
 hasElement w variable A

 markExpressions w or variable A variable B Cut
End

Rule CutOr_B_B
 hasElement w or variable A variable B
 hasElement w variable B

 markExpressions w or variable A variable B Cut
End

Rule CutOr_A_not_A
 hasElement w or variable A variable B
 hasElement w not variable A

 add w variable B
 markExpressions w or variable A variable B Cut
End

Rule CutOr_B_not_B
 hasElement w or variable A variable B
 hasElement w not variable B

 add w variable A
 markExpressions w or variable A variable B Cut
End

Rule CutOr_not_A_A
 hasElement w or not variable A variable B
 hasElement w variable A

 add w variable B
 markExpressions w or not variable A variable B Cut
End
```

3.5 From mono to multimodal logic K_n

Defining model construction for the multimodal logic K_n from scratch is not necessary, and it can be easily achieved in LoTREC if we have already defined the model construction for the monomodal logic K.

Comparing to the monomodal logic K, the modal operators $\Diamond$ and $\Box$ become, in the multimodal logic K_n, binary operators of the form $\langle I \rangle$ and $[I]$, where $I \in \mathcal{I}$ is a given modality index, such as $\langle BBC \rangle It\_Rains$ and $[Obama]We\_Can$. Hence, to reuse the method of K for K_n, we have to change the definition of the modal connectors (as explained in Section 2.2).

As for the rules, we can keep on using the same set of rules defined above for the monomodal logic K, except the modal rules which have to be slightly modified. In fact, the rules `Pos`, `Nec`, `NotPos` and `NotNec` should take into account the index I of the modalities. For example, the `Pos` rule should be defined as follows:

```
Rule Pos
 hasElement w pos variable I variable A

 createNewNode u
 link w node1 variable I
 add u variable A
End
```

Note that we are considering `pos variable I varibale A`, instead of considering `pos variable A`, where `variable I` is to be substituted by any given modality index from $\mathcal{I}$. This modality will be the label of the link created by the action `link w u variable I`.

Similarly, the `Nec` rule should be modified so that the $[I]$-formulas are only propagated along the edges labeled by I:

```
Rule Nec
 hasElement w nec variable I variable A
 isLinked w u variable I

 add u variable A
End
```

The modal rules `NotNec` and `NotPos` should be modified similarly, by replacing occurrences of `R` by `variable I`.

As for the strategy, we can keep on using the same strategy defined for the monomodal logic K (Section 3.4), since we dispose of the same set of rules and the same semantics.

3.5.1 Inclusion: a simple interaction between modalities

In multimodal logics, we may have various kinds of interaction between the different modalities. In this section, we shall meet one simple kind of interaction,

called *inclusion*, and we shall explain how to adapt the above model construction method to take this interaction into account.

To this end, we consider, for simplicity and w.l.o.g., a multimodal logic with two K modalities, say $[I]$ and $[J]$, and with the axiom $[J]P \to [I]P$. This axiom changes the truth condition of the $[J]$-formulas to the following:

$$M, w \Vdash [J]A \text{ iff for every world } u, \text{ if } wR_Ju \text{ or } wR_Iu \text{ then } M, u \Vdash A$$

Which means that $[J]$-formulas are considered as if they were $[I]$-formulas. Practically, this means that, during the model construction, for every $[J]A$ formula at a given world w, the formula A should be propagated along the I-edges to all the R_I-successors of w, exactly as it is propagated along the J-edges to all the R_J-successors of w.

This equivalently means that, for a given world w, a successor by R_I is also considered as a successor for w by R_J. Indeed, there is no harm in this understanding, since the $[J]P \to [I]P$ axiom is dually equivalent to $\langle I\rangle P \to \langle J\rangle P$.

The intuition behind inclusion can be also clarified in terms of relations, by considering the relations R_I and R_J as being the relations "*is a brother of*" and "*is a sibling of*", respectively. It is clear that if we say that all the siblings of somebody are elder than her/him, then we implicitly assume that all her/his brothers are all elder than her/him. More generally, if a property P holds for all her/his siblings, then it holds also for all her/his brothers too. The reason is that the relation "is a brother of" is *included* in the relation "is a sibling of" (we denote it by $R_I \subseteq R_J$).

This is why we take this axiom as an *interaction* axiom and we call it the $\mathsf{Inclusion}$ axiom. Hence, the logic we are studying here is usually denoted as $\mathsf{K} + \mathsf{K} + \mathsf{Inclusion}$, or briefly $\mathsf{K}_2 + \mathsf{Inclusion}$.

LoTREC rules for $\mathsf{K}_2 + \mathsf{Inclusion}$

We simply give an adaptation of the model construction method, given in Section 3.5, to conveniently cope with this inclusion semantics. For instance, if we keep on using the same multimodal rules defined in Section 3.5 to deal with the unsatisfiable formula $\langle I\rangle P \wedge [J]\neg P$, then we would obtain the open premodel of Figure 3.9.

The needed adaptation is technically simple. We only need to add one rule which takes the above explained R_J-R_I interaction into account, as follows:

```
Rule Nec_J_To_I_Successors
 hasElement w nec J variable A
 isLinked w u I

 add u variable A
End
```

Running with the strategy defined in section 3.4 and calling this new rule in addition, we obtain the closed premodel of Figure 3.10.

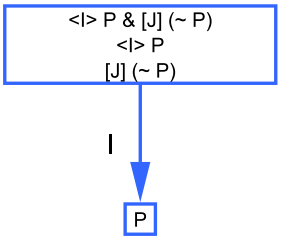


Figure 3.9: An open premodel obtained by an unadapted method for the formula $\langle I \rangle P \wedge [J] \neg P$, which is unsatisfiable K_2 + Inclusion.

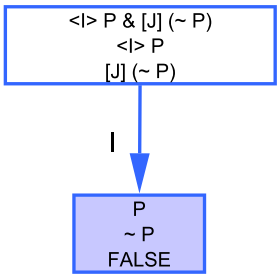


Figure 3.10: A closed premodel for the formula $\langle I \rangle P \wedge [J] \neg P$.

Note that we can deal with the inclusion of more than two relations by defining as many (similar to `Nec_J_To_I_Successors`) rules as needed (one rule by inclusion interaction).

3.6 Certifying a model construction method

Let us recall first the terminological definitions given in section 3.2 about rules and closed, open and saturated premodels. First, we formulate some properties which we would like to hold:

Definition 8 (termination)**.** *A model construction method is terminating if, and only if, at some step all premodels are either closed or saturated.*

Definition 9 (completeness)**.** *A model construction method is complete if an open saturated premodel built with the method for a formula A can be turned into a model for A.*

Definition 10 (soundness)**.** *A model construction method is sound if: for every formula A, if the method builds an open premodel for A then A is satisfiable.*

The following criteria are useful in order to guarantee the above properties:

1. We should define a *complete set of rules*, i.e. a set of rules which reflects all possible constraints on truth conditions. This is not a hard task for

the logics we have seen so far, but it becomes a tricky task for more sophisticated logics as we shall see in further chapters.

2. We have to make sure that this set of rules *only* reflects the exact needed constraints, and no other additional ones. Otherwise, it would be *unsound*.

3. We have to apply indistinctively these rules as long as possible, i.e. as long as the obtained premodels are neither closed nor saturated; otherwise one might produce premodels that are open and saturated, but which cannot be extended to models: the result would not be *sound*.

4. In case we are using a strategy to apply the rules in a given order, we have to make sure that every applicable rule will be eventually applied, otherwise one might produce premodels that are open and saturated, but which cannot be extended to models: the result would not be *sound*.

5. In case our strategy is applying the set of rules repeatedly, we have to make sure that at some step all premodels are either closed or saturated, and that the process do not run for ever[2], otherwise the method would not be *terminating*.

The above definitions and properties are to be formulated for each method as theorems, and have to be proven on a case-by-case basis for each method.

The methods that we defined in this chapter are all sound, complete and terminating. A formal proof is given later in chapter 8 for a more complex logic. In the next chapter, we give a general termination criteria (Theorem 1) that covers the methods that we have defined up to now.

Conclusion

In this chapter we introduced the model construction method. We showed that it is a tableau-like method. In contrast with usual tableaux methods, our method uses graphs instead of trees, and it is an attempt to build a Kripke model for the input formula, not a proof (by refutation) of validity of the formula.

This model construction is done in a step-by-step application of a set of graph rewriting rules. These rules are called by a strategy in order to saturate the constructed premodels.

After showing how this method works on paper, we showed how to implement an automated procedure in our software LoTREC to achieve the same method using the computer.

We considered first the case of the modal logic K, then its multimodal version K_n. After that we considered the simple $\mathsf{Inclusion}$ interaction between multiple modalities.

At the end, we discussed some of the theoretical properties of the model construction methods and how to ensure them, mainly termination, soundness and completeness.

[2] i.e. premodels never become closed nor saturated.

Chapter 4

Logics with simple constraints on models

Introduction

In chapter 1, we described how to model various systems and situations with graphs, which we call *labelled graphs*, and we talked informally about the *properties* of these systems.

In chapter 2, we showed how to express these properties in a formal language, in which a property is encoded as a *formula* and a system or a situation is represented by a *model* of this language. At the end of that chapter we gave the semantics of formulas by giving *truth conditions* for every connective, and we defined some of the interesting reasoning problems.

In chapter 3, we addressed the satisfiability problem in logics of arbitrary models, K and K_n, by defining a *model construction* procedure: we answer the question "is there a model that satisfies a given formula?" by trying to construct such a model. Then we showed how to make this model building process automatic in LoTREC.

In many applications models satisfy some *constraints* on their structures. As we will see, the method presented in chapter 3 is not appropriate any more to tackle the satisfiability problem in the models of such systems. This shall be clarified immediately.

Motivation

Let us recall the card game of Section 1.2. In the state "Red" the formula Red is obviously true, whereas the formula $K_{Bob}\neg Red$ is false. Checking this formula in the model of Figure 1.6 gives us this result, identifying $K_{Bob}\neg Red$ with $[Bob]\neg Red$. More generally, in 'appropriate' models of knowledge it should never happen that A and $K_I\neg A$ are true in the same world. In other words, $K_I A \to A$ should be valid in 'reasonable' models of knowledge. However, the

formula $K_{Bob}\neg Red \wedge Red$ is satisfiable according to the method of chapter 3: it is so in a model consisting of one node in which *Red* holds, and $K_{Bob}\neg Red$ too since that node has no successors.

How can we guarantee that $K_I A \rightarrow A$ is unsatisfiable? Well, models of knowledge should have a *reflexive*[1] accessibility relation: if the actual world is the state "Red", then this state should also be possible for Bob. This guarantees that a formula such as $K_{Bob}\neg Red$ does not hold at the same state in which *Red* holds (figure 4.1). However, the method of chapter 3 does not account for this structural constraint.

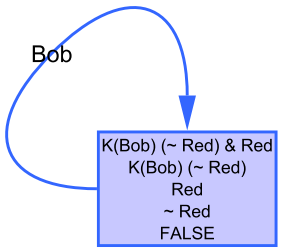


Figure 4.1: The correct closed premodel for the formula $K_{Bob}\neg Red \wedge Red$.

Let us consider another example. Suppose that we are modelling a multi-tasks system. Suppose that each state represents a task and that a state w is linked to state u if "the task represented by w can be executed in parallel while executing the task represented by u". Obviously, the statement "the task represented by u can be executed in parallel while executing the task represented by w" holds, too. Hence, for every such two states, w and u, u should be also linked to w, which means that the underlying model has to be *symmetric*.

Methodology

The above examples show that we need to extend our method given in chapter 3 in a way to capture most known structural constraints on the models of various logics. Since most of these logics are based on the classical propositional modal logic K, the method of Chapter 3 is reused as the basis of these methods.

In general, the rules dealing with classical connectors do not change in these methods. We only change the rules dealing with the modal connectors. In the literature, there are two orthogonal approaches to deal with special constraints on the accessibility relations and which are due to the validity of certain axioms:

- the accessibility relation is explicitly updated (by adding the necessary reflexive, symmetric,... edges), as the case in *labelled tableaux*,
- some reduction rules are added to take the corresponding axioms into account without affecting the accessibility relation.

[1] A relation is reflexive iff for all $w \in W$ and for all $I \in \mathcal{I}$ we have $wR_I w$.

We shall see how to implement both approaches throughout this chapter.

Moreover, we keep on using the same strategy defined in Section 3.4, except when we add new rules and when the order of the rules inside the strategy becomes important; we then state the new strategy.

What is in this chapter

In this chapter, we consider the model construction method for various logics with specific constraints on their models. We give the methods of KT (Section 4.1), KB (Section 4.2), $\mathsf{K.alt}_1$ (Section 4.3), KD (Section 4.4), K+Confluence (Section 4.5) and finally S5 (Section 4.6). In the end, we give a common termination criteria which covers the methods of all these logics.

4.1 Reflexive models

As shown in the introduction of this chapter, the method of Chapter 3 is not suitable to achieve the model construction for certain formulas whose models have to be reflexive.

In order to take into account the effect of the reflexivity of models on the satisfiability of formulas, we create for every node the necessary reflexive edge. To this end, we change the `Pos` rule, defined in Section 3.3, as follows:

```
Rule Pos
 hasElement w pos variable A

 createNewNode u
 link w u R
 add u variable A
 link u u R
End
```

Note that the only one difference is the addition of the last action `link u u R`. This action links each newly created node to itself. However, this does not include pre-created nodes, i.e. the nodes of the input partial premodel. In fact, if the user is launching the satisfiability check of a formula in a partial premodel with irreflexive nodes, the above rule does not make these nodes reflexive.

That is why a better solution would be to keep the rule `Pos` as it, and to add the following other rule:

```
Rule Reflexive_Edges
 isNewNode w

 link w w R
End
```

and to call this rule once inside the outermost `repeat` loop of the strategy of Section 3.4, as follows:

```
Strategy K_And_Reflexivity
  repeat
    Stop
    NotNot
    ..
    ..
    Nec
    Pos
    Reflexive_Edges
  end
End
```

4.1.1 Simulating reflexivity

Reflexive models are characterized by the axiom T: $\Box A \to A$, i.e. T is valid in a given logic iff the underlying models are reflexive. The reflexivity constraint on the structure of models can be simulated by changing the truth condition of the $\Box$-formulas as follows:

$$M, w \Vdash \Box A \text{ iff } M, w \Vdash A \text{ , and for all } u \text{ s.t. } wRu, M, u \Vdash A$$

So that a $\Box$-formula is true in a given world w of a model iff it is true in every successor u of w, including the world w itself.

Hence, we can deal with reflexivity using the following rule:

```
Rule Nec_To_Actual_World
 hasElement w nec variable A

 add w variable A
End
```

This rule is an alternative to the rule `Reflexive_Edges` which can be called in the strategy in order to take the reflexivity into consideration. However, we still need to call the rule `Reflexive_Edges` at the end of strategy to complete the open premodels with reflexive edges in order to transform them into reflexive models.

Remark 7. We can also handle the bimodal logic K +KT, i.e. the logic with two modalities: a K-modality $[I]$ and a KT-modality $[J]$. We can extend this method to deal with K_n +KT (see Section 3.5). Moreover, we can extend this method to deal with the inclusion axiom: $[J]P \to [I]P$ (c.f. Section 3.5.1).

4.2 Satisfiability in symmetric models

An accessibility relation is *symmetric* iff whenever it links a state w to another state u then it links u to w too. Some relations are intuitively symmetric such as the relations linking a state w of a given system to another state u iff "w

is aside of u", "w can be executed in parallel with u" or "w is equal to u" etc. These relations are intuitively *symmetric* and should link as well u to w.

We can easily check that the method of Chapter 3 is not suitable to deal with satisfiability in symmetric models, since it does not take into consideration the structural constraint of symmetry in such models.

For example, suppose that we are modelling the countries neighbourhood. Suppose that we consider states as countries, and that we define our relation over the states as: a state C is linked to a state C' iff "the country C is a neighbour of C'". It is obvious that for each such linked states C and C', C' should also be linked to C since "C' is a neighbour to C" too. Suppose that the proposition EU, when found in a state C, means that C is member of the European Union. Hence the formula $\neg EU \wedge \Diamond\Box EU$ in a state C means that the country C is not member of the European Union and that C has a neighbour, let us say C', whose neighbours are *all* members of the European Union. The reader may easily check that this formula should not be satisfiable, whereas it is satisfiable according to the method of Chapter 3.

Nonetheless, this method becomes suitable to deal with symmetry once the `Pos` rule is adjusted as follows:

```
Rule Pos
 hasElement w pos variable A

 createNewNode u
 link w u R
 add u variable A
 link u w R
End
```

This rule links each newly created node to its predecessor. However, as explained in Section 4.1, this rule does not guarantee that symmetry is respected in the input premodel. Instead of changing the rule `Pos`, it would be better to add the following rule:

```
Rule Symmetric_Edges
 isLinked w u R

 link u w R
End
```

and to call this rule once in the strategy, after calling `Pos` and before calling `Nec` for example.

Remark 8. One may thinks that we need to add the condition `isNotLinked u w R` to make sure that such a link does not exist. Nonetheless, in LoTREC two nodes can not be linked by two edges of the same label.

4.2.1 Simulating symmetry by semantics

Another way of respecting the structural constraint of symmetry, without creating symmetric arcs, is by propagating the $\Box$-formulas from children nodes to

their parent nodes, as if they were linked to them. The following rule does it:

```
Rule Nec_To_Parent_Nodes
 hasElement w nec variable A
 isLinked u w R

 add u variable A
End
```

In Section 4.1 we simulated the reflexivity by respecting the axiom T ($\Box A \to A$). However, we can not simulate the symmetry by depicting the underlying axiom B ($\Diamond\Box A \to A$) nor its dual form ($A \to \Box\Diamond A$) with a rewriting rule, i.e. a rule which adds A to each node that contains $\Diamond\Box A$, for example!! We let to the reader to verify that.

4.3 K.alt$_1$: the relation is a partial function

In this section, we are considering models where the relation is a partial function. This means that a given node can have at most one successor.

4.3.1 Motivation

In real life, there are some actions where the result is purely deterministic. After having executed the action, the result is completely determined. So the transition is a function.

4.3.2 Changing the method of K to get a method for K.alt$_1$

In the logic K, a world can have several successors. For instance the formula $\Diamond P \wedge \Diamond\neg P$ is satisfiable in the logic K (Figure 4.2). On the contrary, in the logic K.alt$_1$, a world can only have at most one successor. Hence, the formula $\Diamond P \wedge \Diamond\neg P$ is no longer satisfiable.

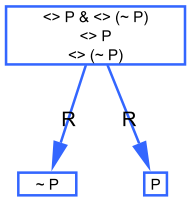


Figure 4.2: Model construction of the formula $\Diamond P \wedge \Diamond\neg P$, according to the method of K (Chapter 3).

Indeed, in K.alt$_1$, if a node w contains $\Diamond P \wedge \Diamond\neg P$, it will contain $\Diamond P$. To make this formula true, we create a new successor u linked to w and containing P. The world w also contains $\Diamond\neg P$. So there exists a successor of w containing $\neg P$. But in a model of the logic K.alt$_1$, as a world has at most one successor,

and as w has already a successor u, the successor of w containing $\neg P$ will be also the world u. Thus, u will contain both P and $\neg P$, as shown in Figure 4.3, and the formula is then reported as unsatisfiable.

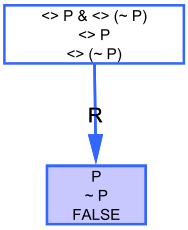


Figure 4.3: The closed premodel that we expect to have for $\Diamond P \wedge \Diamond \neg P$ in K.alt$_1$.

The following rule `Pos`, defined for dealing with the $\Diamond$ operators in K (see Chapter 3),

```
Rule Pos
 hasElement w pos variable A

 createNewNode u
 link w u R
 add u variable A
End
```

is no longer suitable for K.alt$_1$. Actually, if a world w contains a formula $\Diamond A$, we need to add a successor to w only if w has not yet any successors. That is why we should use an additional condition primitive `hasNoSuccessor` in the `Pos` rule in order to avoid creating more than one successor to any given node. In K.alt$_1$, the suitable rule should look like:

```
Rule Pos_One_Successor
 hasElement w pos variable A
 hasNoSuccessor w R

 createNewNode u
 link w u R
 add u variable A
End
```

However, this rule does not work appropriately as expected, and delivers the same premodel seen in Figure 4.2. The problem is that, according to LoTREC's philosophy, we apply a given rule at all matching patterns at once (cf. Remark 5). Which means, that in case we have two $\Diamond$-formulas in a given node with no successors, then each formula will trigger the application of the rule `Pos_One_Successor` since the node has no successors at the moment of the test. Then, the rule will be applied twice, one time by each $\Diamond$-formula, yielding two successors, exactly as the old `Pos` rule does. The `hasNoSuccessor` condition takes effect only in posterior iterations, when, for example, some other $\Diamond$-formulas are added later to the same node.

Nonetheless, this is not The End. Using the strategy keyword applyOnce just before the name of the rule Pos_One_Successor is sufficient to demand the application of the rule on only one matching pattern at a time. Encapsulating the applyOnce Pos_One_Successor call inside a repeat ... end routine applies the rule on all the possible occurrences, but after considering them one-by-one, which means that the condition hasNoSuccessor is taken into account.

Note that, using this rule, only one $\Diamond$-formula will be treated. The reason is that, once the rule is applied on one of the $\Diamond$-formulas, it creates a successor, then becomes un-applicable again (due to the presence of this successor) on whatever other $\Diamond$-formula.

It is clear, that we need two different rules:

- the first one says: if there is a $\Diamond A$ formula at a given world w with no successors then create one, say u, then add A to it,
- while the second one states that: if there is a $\Diamond A$ formula at a given world w having already a successor u, then add A to that world u.

Note that adding the formula A by the first rule is not necessary since it is achieved in the second rule. Hence the following two rules:

```
Rule Create_One_Successor
 hasElement w pos variable A
 hasNoSuccessor w R

 createNewNode u
 link w u R
End

Rule Pos
 hasElement w pos variable A
 isLinked w u R

 add u variable A
End
```

In the strategy, we may call these rules as follows:

```
Strategy Kalt1_Strategy
repeat
  ..
  ..
  applyOnce Create_One_Successor
  Pos
  ..
  ..
end
```

Considering the formula $\Diamond P \wedge \Diamond \neg P$ with this strategy should give the pre-model of Figure 4.3.

4.4 Serial models for obligation and norms

Another example we can have a look at is interpreting necessity as norm, or law and thus reading $\Box A$ as "According to the law A should be true", or "it ought to be the case that A". This is the *deontic* interpretation. As an example, think that A could be "I pays his/her taxes on time". Note that in this setting, the obligation is on the fact not on the action, usually one would say that "To pay taxes on time" is obligatory. But the related discussion is out of the scope of this thesis, and the interested reader will have to refer to the literature. In this section, we will make use of $\mathcal{O}$ instead of $\Box$, the new symbol $\mathcal{O}$ standing for "Ought to be the case that".

Now let us look at related concepts. When can we say that a fact is forbidden? For example, according to the above, we know that it is forbidden that "I does not pay taxes on time" suggesting that a fact is forbidden whenever its negation is obligatory. Thus $\mathcal{O}\neg A$ can be read indistinctively as "A is forbidden" or "$\neg A$ is obligatory". But what about things that are just not forbidden? They are simply *allowed* or *permitted*, we can denote that "A is permitted" by $\mathcal{P}A$, and we have just seen that $\mathcal{P}A$ is in fact the same as $\neg\mathcal{O}\neg A$. It may be asked why obligation is formalised by a $\Box$ and permission by a $\Diamond$. The reader may think of possible worlds of models in terms of *legal worlds*, i.e. worlds where law is respected. Now obligation means true in all legal worlds, and permission means true in at least one legal world. As such, the formula $\mathcal{P}A \wedge \mathcal{P}\neg A$ should be satisfiable (e.g. it is both permitted to eat bread and to not eat bread).

But this is not the whole story, and we must investigate whether $\mathcal{O}$ and $\mathcal{P}$ have additional properties, this is part of modelling task as the reader should have understood by now. It should be clear that if something is obligatory then it cannot be forbidden at the same time, hence the conditional $\mathcal{O}A \to \mathcal{P}A$ should be true in any case, it is called the *ideality principle* in deontic logic. Let us have a look at this.

4.4.1 Changing the method of K to make it suitable

As we said above, this amounts to saying that something like $\Box A \to \Diamond A$ is always true at any possible world of any model, or, the other way round, that its negation $\neg(\Box A \to \Diamond A)$ is always false. But if we applies the method of Chapter 3 directly, assuming that $\mathcal{P}$ is `pos` and that $\mathcal{O}$ is `nec`, we find that the latter formula is satisfiable, i.e. it can be made true!

In fact, both $\mathcal{O}A$ and $\mathcal{O}\neg A$ can be true in any world which can access no other world (remember that $\mathcal{P}A$ is defined as $\neg\mathcal{O}\neg A$). Hence, it is clear, that in order to make the ideality principle always true, deontic models should ensure that any possible world with a $\mathcal{O}A$ formula can access at least one other world.

This could be ensured by adding a rule that would add a successor to any node that contains a `nec`-formula and which has no successor by `R`, as follows:

```
Rule Create_Child_For_Leaf_Nodes_With_Nec
 hasElement w nec variable A
 hasNoSuccessor w R
```

```
 createNewNode u
 link w u R
End
```

The condition `hasElement w nec variable A` succeeds whenever `w` contains *any* `nec`-formula. The condition `hasNoSuccessor w R` succeeds only if `w` has no successors by `R`.

Let us try this rule with the simple strategy seen in Section 3.4, by simply adding it at some place inside the `repeat` loop:

```
repeat
  Stop
  NotNot
  ..
  ..
  ..
  Pos
  Nec
  Create_Child_For_Leaf_Nodes_With_Nec
end
```

If we run this strategy in a step-by-step mode in LoTREC, then we should notice that the rule `Create_Child_For_Leaf_Nodes_With_Nec` creates as many successors for a given leaf node as there are □-formulas in it. This is a side effect of the LoTREC philosophy which insists on *saturating* the application of the rules, and which is the reason behind applying the rules as much as possible on every part of the premodels.

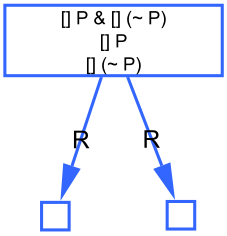


Figure 4.4: The premodel right after the call of the `Create_Child_For_Leaf¬_Nodes_With_Nec` rule. Two nodes are created: one by each □-formula.

To avoid this side effect, we may call this rule after the keyword `applyOnce`. The strategy should look like:

```
repeat
  Stop
  NotNot
  ..
  ..
  ..
```

```
    Pos
    Nec
    applyOnce Create_Child_For_Leaf_Nodes_With_Nec
  end
```

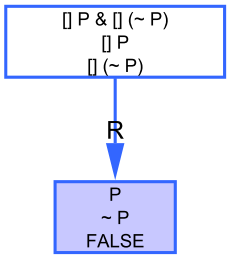


Figure 4.5: Using the appropriate method, norms semantics are respected, so that $\Box P \wedge \neg\Diamond P$ is not satisfiable for example.

4.4.2 Extracting a model from an open premodel

Thus we have obtained an open saturated premodel for $\mathcal{P}A \wedge \mathcal{P}\neg A$, but as explained in the previous chapter, we must be able, from this open saturated premodel to explicit a model of the formula, and this premodel is only almost a model since it does not satisfy the requirement of deontic models which is that all worlds must have at least one successor. Can we slightly transform this premodel into a model in a harmless way? In fact this is easy, it is sufficient to add a reflexive edge on those empty worlds, this will not change the truth value of formulas while making the premodel a real deontic model.

In LoTREC, it is possible to design a rule that does this job of adding a reflexive edge to empty nodes, BUT this must be done only on an open saturated premodel. On the one hand, it makes no sense to do it on a closed premodel since it cannot provide a model, and on the other hand, it would be false to do it on a non-saturated premodel since an empty node would not be guaranteed to remain empty in this case.

Let us call, at the end of our strategy, after the `repeat` loop, the following self-explanatory rule:

```
Rule Reflexive_Edges_On_Leaf_Nodes
 isNewNode w
 hasNoSuccessor w R

 link w w R
End
```

The condition `isNewNode w` and `hasNoSuccessor w R` ensure that the rule is applied on lastly added new nodes which have no successors by `R`.

In order to apply it on saturated premodels, we must call this rule at the end of our strategy. This way, we postpone its execution after that all the other rules have run to their ends. Thus we obtain the following strategy:

```
Strategy Strategy_For_Norms
  repeat
    Stop
    NotNot
    ..
    ..
    ..
    Pos
    Nec
    Create_Child_For_Leaf_Nodes_With_Nec
  end
 Reflexive_Edges_On_Leaf_Nodes
End
```

If we run our new strategy with the new `Reflexive_Edges_On_Leaf_Nodes` rule on the formula $\Box P \wedge \Box Q$, we would obtain the the model of Figure 4.6.

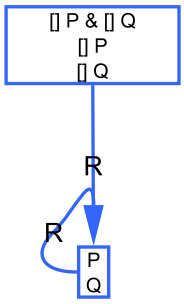


Figure 4.6: An open premodel for $\Box P \wedge \Box Q$ transformed into a model after adding a reflexive edge at the leaf node.

4.5 Confluent models

An accessibility relation R is said to be confluent if, and only if: for every worlds w, u, v such that wRu and wRv, there exists a world x such that uRx and vRx. The world x is called the *confluent world* of the tuple (w, u, v). It is well-known [Che80] that confluent models are characterized by the following axiom.

$$\mathsf{Confluence} : \Diamond\Box P \rightarrow \Box\Diamond P$$

In the following, we consider the basic modal logic with confluent models: the modal logic K+Confluence.

The simplest way to deal with formulas in this logic is to reuse the rules defined for K in Chapter 3, then to add to them the following rule:

```
Rule Confluence
 isLinked w u R
 isLinked w v R

 createNewNode x
 link u x R
 link v x R
End
```

which completes the built premodels with the necessary confluent worlds.

We can reuse the strategy seen in Section 3.4, after changing its name to `Confluence_Strategy`, by adding to it the confluence rule as follows:

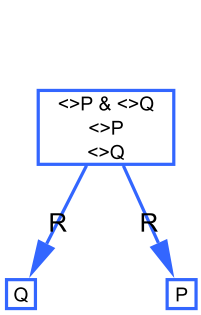
```
Strategy Confluence_Strategy
  repeat
    Stop
    NotNot
    ..
    ..
    Pos
    Nec
    Confluence
  end
End
```

Example 5. Let us run the above strategy with the formula $\Diamond P \wedge \Diamond Q$, in a step-by-step mode to check the method first.

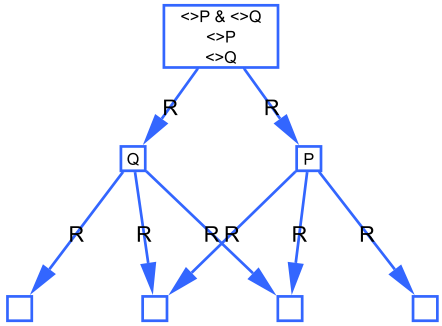
We can clearly notice from Example 5 that this version of rule for confluence is applicable on empty `u` and `v` nodes. This yields creating empty `x` nodes, since no formula will be propagated from an empty `u` or an empty `v` to their `x` child node. Hence, calling the `Confluence` rule repeatedly in the strategy does not terminate.

A remedy to this non-termination problem would be to add one of the conditions `hasElement u variable Some_Formula` or `hasElement v variable Some_Other_Formula` to guarantee that `u` or `v` is not empty. Moreover, we can add both conditions to make sure that both of them are non empty; this will not interfere with the completeness of the method. The rule becomes:

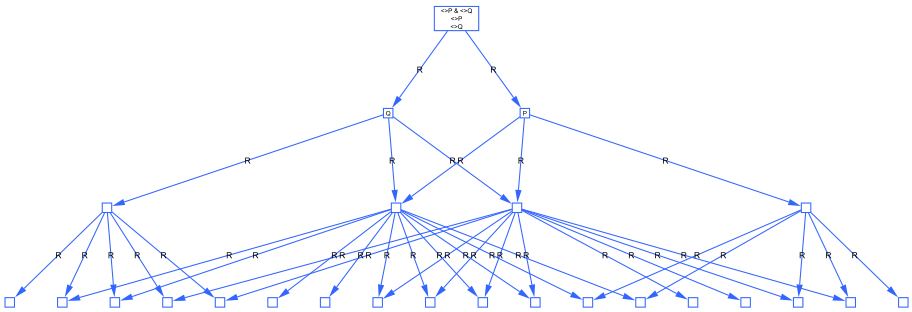
```
Rule Confluence
 isLinked w u R
 isLinked w v R
 hasElement u variable A
 hasElement v variable B

 createNewNode x
 link u x R
 link v x R
End
```



(a) step 1



(b) step 2



(c) step 3

Figure 4.7: The first three steps of running the `Confluence_Strategy` on the formula $\Diamond P \wedge \Diamond Q$.

Running the strategy with this rule of confluence on the formula of Example 5, gives end to its execution and stops at "step 2" of Figure 4.7. The termination problem is solved with this rule.

However, it seems weird, in Figure 4.7, to have four successors at "step 2" from just two successors at "step 1". In order to know what is happening, we may want to add to each `x` node created by the confluence rule the names of the `u` and `v` nodes detected by the rule and which have lead to the creation of `x`. To do so, we add two actions to the rule as follows:

```
Rule Confluence
 isLinked w u R
 isLinked w v R
 hasElement u variable A
 hasElement v variable B

 createNewNode x
 link u x R
 link v x R
 add x nodeVariable u
```

```
 add x nodeVariable v
End
```

The action `add x nodeVariable u` adds to the instance node assigned to `x` the hidden name associated (internally by LoTREC) to the instance node assigned to `u`. The second added action is similar. Running with this rule on the formula of Example 5, in a step-by-step mode again, we obtain the premodel of Figure 4.8 at the second step.

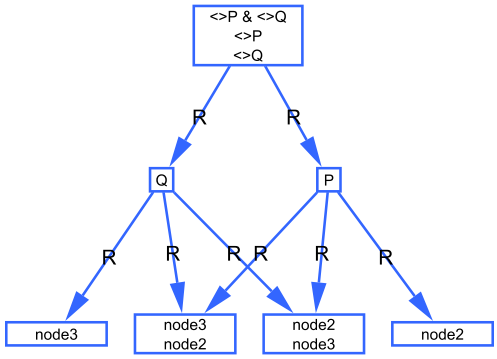


Figure 4.8: Debugging the confluence rule, by tracing which `u` and `v` nodes yield to which `x` confluent node.

In this premodel, the user may verify (by right-click on the nodes) that `node2` is the node on the right of the second row, holding the formula `P`, and that `node3` is the node on the left of the second row, holding the formula `Q`. On the third row, there are four nodes created by the rule of confluence:

1. a node labeled `node2` (as the names of `u` and `v`),
2. a node labeled `node3` (as the names of `u` and `v`),
3. a node holding `node3` and `node2` (as the names of `u` and `v`),
4. and a node holding `node2` and `node3` (as the names of `u` and `v`),

The reason behind the creation of the first two nodes of this list is that the pattern matching morphism, which is used to instantiate the variables appearing in the condition part of a rule, is not necessarily injective. So, `u` and `v` can be matched to the same node `node2`, or to the same node `node3`. This is a desirable criteria in the case of the confluence rule, since it can handle correctly the formula $\Diamond(\Box P \wedge \Box\neg P)$.

The other two nodes are obtained by considering (`node2`, `node3`) and (`node3`, `node2`) as different possible instances for the couple of variables (`u`, `v`). This is the default way of instantiating multiple variables in LoTREC, which guarantees that the rule is applied on every possible matching pattern. Whereas in this example, the creation of only one of these two nodes is sufficient to conform

to the semantical conditions of confluence. In such cases, we may prefer to be more optimal and to create only one confluent world by couple of nodes with a common parent node.

In LoTREC, it is up to the rules to specify whether two patterns should be considered as equivalent or not. In general, in order to prevent the rule from being applied on a pattern whose equivalent pattern (modulo a given criteria) was already detected, we have to:

1. use some additional actions to annotate the nodes with an information indicating which patterns were taken into account,
2. use some additional (negative application) conditions to prevent the rule from being applied on patterns which have already been taken into account,
3. and to call the rule with the `applyOnce` keyword, to guarantee that the rule is considering the patterns one by one, and does not treat all of them at once.

In our example, when a couple of instance nodes (`Instance1`, `Instance2`) instantiates the couple of variables (`u`, `v`), the confluence rule should escape the couple if it, or its equivalent couple (`Instance2`, `Instance1`), has been already taken into account.

Practically, we define a new binary connector, called `done`, and we add to the node `N0` the information about which couples of nodes were done. The confluent rule should look like the following:

```
Rule Confluence
 isLinked w u R
 isLinked w v R
 hasElement u variable A
 hasElement v variable B
 hasNotElement w done nodeVariable u nodeVariable v
 hasNotElement w done nodeVariable v nodeVariable u

 createNewNode x
 link u x R
 link v x R
 add w done nodeVariable u nodeVariable v
End
```

In order to take effect, this version of the confluence rule should be called, in the strategy seen above, preceded by the `applyOnce` keyword, as follows:

```
Strategy Confluence_Strategy
  repeat
    Stop
    NotNot
    ..
    ..
```

```
    Pos
    Nec
    applyOnce Confluence
  end
End
```

The result of calling the above strategy on the same formula of Example 5 is shown in Figure 4.9.

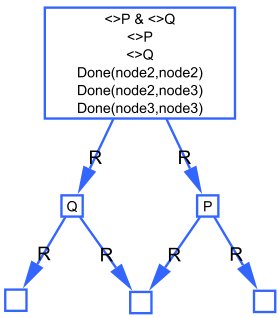


Figure 4.9: The result of the last version of the confluence rule.

4.6 Knowledge

In this section, we present the semantics of knowledge formulas, then we show how this semantics imposes some constraints on models of knowledge.

In the logic of knowledge, also known as S5, we need to write formulas of the form "I knows A" and "I does not know that A", where I is usually called an *agent* and A is a knowledge formula. The sentence "I does not know that A" means that $\neg A$ is actually possible for I. In other words, it means that I can imagine a possible world in which $\neg A$ holds. On the contrary, "I knows A" means that A holds in all worlds that I can imagine. Here, worlds that I can imagine as true are in fact all worlds that are undistinguishable from the real world. The accessibility relation R_I is defined as wR_Iu if, and only if, I can not distinguish w from u (because I has not enough information about the real world to distinguish them), i.e. both worlds can be true for I.

As we have seen in Chapter 2, a sentence of the form "I knows A" is syntactically written as K_IA. According to the above semantics, the K operator is semantically equivalent the $\Box$ modal operator. The dual behavior is captured by the sentence "I does not know that $\neg A$", which is syntactically written as $\hat{K}_IA$ (i.e. a K with a *hat*, and pronounced *k-hat*). This $\hat{K}$ operator is semantically equivalent to the $\Diamond$ modal operator.

Next, we only consider the case of one agent, since the multi-agent case involves some termination issues which are discussed later in Chapter 5.

4.6.1 Modelling the knowledge of one agent

In this section, we consider one specific agent, let us say *Bob*, and we show how to construct a model for formulas talking about his knowledge.

At the beginning of this chapter, we have seen that the formula $K_{Bob}\neg Red \wedge Red$ is reported as satisfiable by the method of Chapter 3, whereas it is not suitable (Figure 4.1). *Bob* cannot know his card is not red if his card actually is so. Agents only know true assertions. There are also other formulas that should not be satisfiable. For instance you can also notice that the formula $K_{Bob}\neg Red \wedge \neg K_{Bob}K_{Bob}\neg Red$ is satisfiable. This is also not suitable because if *Bob* knows his card is not red, he must be conscious that he knows it.

Hence, the relation must necessarily be *reflexive*. Indeed, if the relation is reflexive, formulas like $K_{Bob}\neg Red \wedge Red$ can no longer be satisfied. Moreover, the relation R_{Bob} must verify two other properties: the *symmetry* and the *transitivity*. These three properties can be intuitively viewed as follows:

- *Bob* can not distinguish a possible world w from itself;
- If *Bob* can not distinguish a world w from another world u, then obviously he can not distinguish u from w;
- Suppose that w is the real world. If w and u may both be possible real worlds for *Bob*, but v is not possible for him (because he has not enough information to distinguish w from v), then he can not neither distinguish u from v (due to the lack of the same information which would help him to distinguish w from v).

Formally, R_{Bob} must verify for every w, u and v:

- $wR_{Bob}w$; (reflexivity)
- $wR_{Bob}u$ implies $uR_{Bob}w$; (symmetry)
- $wR_{Bob}u$ and not $wR_{Bob}v$ implies not $uR_{Bob}v$, or better said $wR_{Bob}u$ and $uR_{Bob}v$ implies $wR_{Bob}v$. (transitivity)

In short, R_{Bob} is an *equivalence relation*, also known as being a *universal* relation. Notice that you can also read the relation $wR_{Bob}u$ as w and u are equal, roughly speaking, in the eyes of *Bob*, whereas equality is in fact an equivalence relation.

4.6.2 Rules for S5 with implicit edges

In this section, we show how to implement in LoTREC an appropriate set of rules and a strategy that achieve the construction of models for knowledge formulas with one agent, according to the truth conditions and the semantics discussed in the previous section.

To do so, we can reuse all the rules that deal with classical operators and which are defined in the model construction method for the logic K in Chapter 3, since the semantics of these operators does not change in the knowledge logic.

It remains, however, to define the rules which deal with the knowledge operators K and $\hat{K}$. Since we are dealing with only one agent, we need only one accessibility relation, let us say `R`, and we can deal with K and $\hat{K}$ as being the unary connectors `nec` and `pos` defined for the monomodal logic K in Chapter 3. Moreover, since the K and $\hat{K}$ operators have the same semantics of the modal operators $\Box$ and $\Diamond$ respectively, as discussed earlier in Section 4.6, we can also reuse the rules `Pos` and `Nec`, as defined in Chapter 3.

However, using only these rules, we do not guarantee that the accessibility relations, built between the worlds by the mean of the `Pos` rule, are all equivalence relations. In order to respect this semantical property, we may proceed in two different ways:

1. either, *we connect the nodes with additional edges*, representing the reflexive, symmetric and transitive arcs, in order to make sure that the accessibility relation is an equivalence relation,

2. or, *we simulate the presence of these additional edges* without creating them, by simply propagating the `nec` operator and handling the `pos` operator appropriately, as if the nodes were linked by equivalence relations and forming classes of equivalence.

The first solution seems to be easier and more straightforward than the second one. Indeed, adding the reflexive and symmetric arcs, as we have already shown in Section 4.1 and Section 4.2, is very simple. Adding the transitive arcs is also quite simple. However, in the presence of these transitive arcs, the model construction process may not terminate, as we shall see later in Chapter 5.

Thus, we present here the second solution. It consists in linking all the possible worlds to one specific node, let us say a node marked as "`Root`". The implicit meaning behind linking all the possible worlds to the same node is to express the fact that they all belong to the same equivalence class, i.e. they are implicitly all linked, each one to the other and to itself, as if all the reflexive, symmetric and transitive edges were added and linking these nodes together.

To specify such a root node, we can define a rule which looks like:

```
 isNewNode w

 mark w Root
```

Nevertheless, such rule is not only applicable on the first added node, but also on every other newly created node. We can solve this problem by adding a condition which ensures that the node `w` has no parent nodes. This can be done by adding the condition `hasNoParents w`, so the rule would become:

```
Rule Designate_The_Root
 isNewNode w
 hasNoParents w

 mark w Root
End
```

We should call this rule once at the beginning of the strategy. In this way, the first added node, i.e. the initial node with the input formula, would be designated as the root node.

Henceforth, we will replace the rules `Nec` and `Pos` by other rules.

First, each time we find a $\Diamond A$ in this root node we should create a new node and link it to this root. The following rule does it:

```
Rule Pos_At_Root
 hasElement w pos variable Formula
 isMarked w Root

 createNewNode u
 link w u R
 add u variable Formula
End
```

The newly created node becomes part of the same equivalence class.

Similarly, if a $\Diamond A$ is found in a child node (i.e. in a node different than the root node), then we would also create a new child for the root node, to guarantee that the newly created possible world will be part of the same equivalence class. This can be achieved by defining a rule which sends the $\Diamond$-formulas in child nodes to the root node, so they can be treated later by the rule `Pos_At_Root`. This rule is defined as follows:

```
Rule Pos_At_A_Child
 hasElement u pos variable Formula
 isLinked w u R

 add w pos variable Formula
End
```

Note that if a node `u` is linked to a parent node `w`, then `w` is necessarily the root node. This is why we do not need to add the condition `isMarked w Root`.

As for the $\Box$-formulas, they should be propagated to every node. To this end, we need to use four rules:

- a rule which propagates each formula $\Box A$ found at a given world w to itself,
- another rule which propagates each formula $\Box A$ found at the root node to all its children,
- one rule which propagates the formulas of the form $\Box A$ found at a child node to the root node,
- and one which propagates the formulas of the form $\Box A$ found at a child node to all the other child nodes.

The following rules accomplish those three tasks respectively:

```
Rule Nec_To_Same_World
 hasElement w nec variable Formula

 add w variable Formula
End

Rule Nec_At_Root
 hasElement w nec variable Formula
 isLinked w u R

 add u variable Formula
End

Rule Nec_At_Child_To_Root
 hasElement u nec variable Formula
 isLinked w u R

 add w variable Formula
End

Rule Nec_At_Child_To_Other_Child
 hasElement u nec variable Formula
 isLinked w u R
 isLinked w v R

 add v variable Formula
End
```

We have to define similar rules to the dual formulas of the form $\neg\Diamond A$ and $\neg\Box A$.

As for the strategy, we can reuse the strategy defined in Section 3.4, after calling the rule `Designate_The_Root` appropriately, and after replacing the rules dealing with the $\Diamond$ and $\Box$ operators by the corresponding rules listed above.

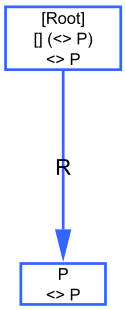


Figure 4.10: An open premodel for the formula $\Box\Diamond P$.

4.7 A general termination theorem

In this section, we give a general termination result for almost all the logics of this chapter (except S5). It will exploit the fact that the setting we are concerned with in our model construction methods is a delimited one: our rules are monotonic, and our strategies are regular expressions.

First note that when `repeat` does not occur in our strategies (i.e., they only contain the constructors `allRules` and `firstRule`) then termination is trivial since in the case of `allRules` only a finite number of rules is applied (and every applicable rule will be applied at most once), and in the case of `firstRule` only one rule will be applied (viz. the first applicable one).

In the general case, the `repeat` construction may make us loop. However, a general termination criterion was proved in [GHS06b] that suffices to cover the methods in the present chapter and that we restate now.

The key observation is that all the rules that we have introduced so far only *add strict subformulas* of the original formula. Following Fitting's terminology [Fit83] such rules might be called *strictly analytic* rules.

Let us consider any of the methods of the present chapter. Let $\mathcal{R}$ be the set of rules in this method. Let A be the initial formula on which the method is applied, and let $\mathtt{M}_0 = (\mathtt{W}_0, \mathtt{R}_0, \mathtt{V}_0)$ be the initial premodel, such that $\mathtt{W}_0 = \{\mathtt{w}_0\}$ (the root[2]), $\mathtt{R}_0 = \emptyset$ and $\mathtt{V}_0 = \{(\mathtt{w}_0, A)\}$.

To state our termination theorem we introduce a function δ which associates an integer with each node $\mathtt{w}$ of a given premodel $\mathtt{M}$. δ is said to be a ranking function if, and only if, $\delta(\mathtt{w}) < \delta(\mathtt{u})$ whenever $\mathtt{w}$ is an ancestor of $\mathtt{u}$ through the union of all edges $\bigcup_{I \in \mathcal{I}} \mathtt{R}_I$, i.e. whenever $\mathtt{w}(\bigcup_{I \in \mathcal{I}} \mathtt{R}_I)^*\mathtt{u})$. Typically, for monomodal logics, $\delta(w)$ will be the distance of w from the root node $\mathtt{w}_0$.

Theorem 1 ([GHS06b]). *Let $\delta(\mathtt{w})$ be the length of the shortest path from the root $\mathtt{w}_0$ to the node $\mathtt{w}$, and let $\mathcal{R}$ be a set of rules such that for every $\rho \in \mathcal{R}$,*

- *ρ is strictly analytic (only* strict subformulas *of the original formula are added to nodes in the action part of ρ);*
- *if the action part of ρ contains the `createNewNode`-action creating a node $\mathtt{w}$ then*
  - *the condition part of ρ checks for existence of* at least one formula *in any node among $\mathtt{w}_1, \ldots, \mathtt{w}_k$, and*
  - *$\delta(\mathtt{w}) > \max(\delta(\mathtt{w}_1), \ldots, \delta(\mathtt{w}_k))$, i.e. new nodes are strictly farther from the root $\mathtt{w}_0$ (ensuring strict decrease of their contents),*

  where $\mathtt{w}_1, \ldots, \mathtt{w}_k$ are the nodes referred to in the condition part of ρ.

Let $\mathcal{S}$ be any strategy on $\mathcal{R}$. Then the application of $\mathcal{S}$ to the input premodel $\mathtt{M}_0$ terminates, and $\mathcal{S}(\mathtt{M}_0)$ is a finite graph.

[2] A node $\mathtt{w}_0$ is called a root if $\mathtt{w}_0$ can access every other node in the transitive closure of the union of all accessibility relations.

Proof. We prove that

- application of $\mathcal{S}$ can never lead to graphs with infinite branching factor;
- application of $\mathcal{S}$ can never lead to graphs of infinite depth.

The argument is that creation of new nodes is subject to non-emptiness[3], but with our constraints, a branch cannot be of length more than the modal degree of the input formula: because of the strict sub-formula condition, nodes situated farther would be empty, and then the criterion for applying the `createNewNode`-action would no more be fulfilled.

□

Theorem 1 covers the case of logics like K, KT, KD, K+Confluence, $\mathsf{K.alt}_1$, and logics with symmetric accessibility relations such as KB, KDB, and KTB.

Conclusion

In this chapter we showed how to extend the basic model construction method given in Chapter 3 to take into account some constraints on the accessibility relation of the models of specific logics. We studied the case of reflexive, symmetric, linear, serial and confluent models. In addition we considered the case of the logic S5 whose models have an equivalence accessibility relation.

We gave at the end a general termination theorem which covers all the above presented methods except for S5 which will be revisited in Chapter 5.

[3]This refers to the condition `hasElement w pos variable A` in the rule `Pos`. One has to add that condition to the `Pos` rule of modal logic K in the preceding chapter in order to apply the present argument. This can be done without harm.

Chapter 5

Logics with potential cycles

Introduction

In the previous chapter we showed how one can implement tableaux methods for some simple modal logics such as KT. In this chapter we show more complex implementations of modal logics by introducing a blocking mechanism; in particular we show how logics with transitive accessibility relations and logics with constraints on their valuation function can be handled.

In section 5.1, we take the most simple logic with transitive models K4. In section 5.2, we revisit S5. In section 5.3, we consider the basic hybrid logic HL(@) which has a specific constraint on its valuation function. Finally, we give in section 5.4 an extension of the general termination theorem (theorem 1), and a discussion on completeness of complex model construction methods.

5.1 Model construction in K4

To implement a model construction method in LoTREC for K4, we might think of reusing the method of K developed at Chapter 3. Nevertheless, Figure 5.1 shows an example of an open premodel obtained by applying the method of Chapter 3 for the formula $\Diamond\Diamond P \wedge \Box\neg P$. However, this formula should not have any open premodel, since it is not satisfiable in transitive models.

In order to adapt the method of Chapter 3 for K4, we have to use some additional rules that take into account the transitivity property of the accessibility relation.

5.1.1 Adding the transitive edges

A first approach is to add a rule that creates the necessary edges which would make the accessibility relation transitive, as follows:

```
Rule Transitive_Edges
 isLinked w u R
```

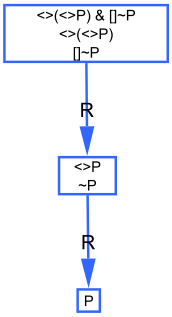


Figure 5.1: An example of an open premodel generated by method of chapter 3 for the non satisfiable formula $\Diamond\Diamond P \wedge \Box\neg P$

```
 isLinked u v R

 link w v R
End
```

To run with this rule, we call it inside the `repeat` loop of the strategy of Section 3.4. Figure 5.2 shows a closed premodel for the formula $\Diamond\Diamond P \wedge \Box\neg P$.

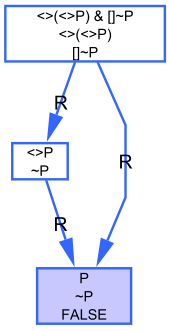


Figure 5.2: Using the rule `Transitive_Edges`, the formula $\Diamond\Diamond P \wedge \Box\neg P$ is no longer counted as satisfiable.

5.1.2 Simulating the presence of the transitive edges

We may simulate the presence of the transitive edges by propagating the $\Box$-formulas differently, as follows:

```
Rule Nec_For_Transitivity
 hasElement w nec variable Formula
 isLinked w u R

 add u variable Formula
```

```
 add u nec variable Formula
End
```

Figure 5.3 shows an example of a closed premodel for the formula $\Diamond\Diamond P \wedge \Box\neg P$, running with the above rule instead of the `Transitive_Edges` rule.

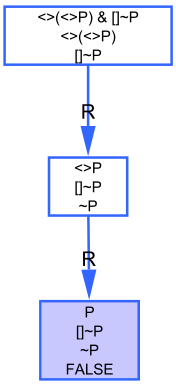


Figure 5.3: Semantics are respected using the `Nec_For_Transitivity` rule without adding the transitive edges.

5.1.3 Construction of transitive models may not terminate!

Using one of the above rules for transitivity, the model construction of some formulas does not terminate. Let us look at the conditions of theorem 1 in chapter 4. The first rule `Transitive_Edges` is strictly analytic, but it violates the second condition of the general termination theorem (precisely, due to the addition of transitive edges every node either is the root, or has distance one to the root). The second rule `Nec_For_Transitivity` is *not* strictly analytic, so it violates the first condition of the theorem.

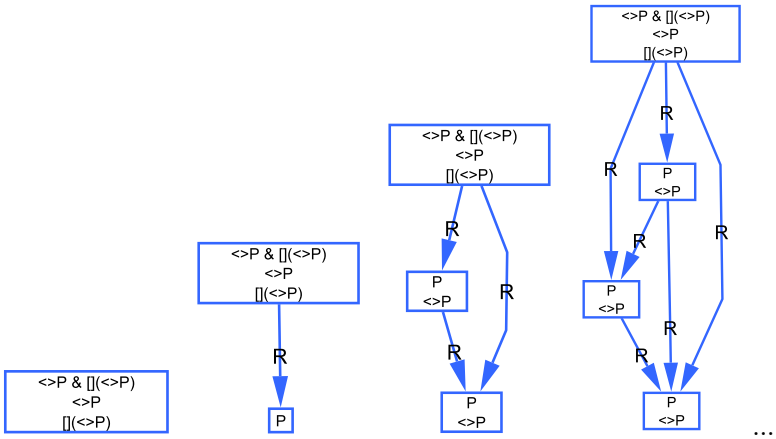
In fact, as we shall see in the sequel, we need to add some special tweaks to these rules and/or to the strategy in order to control the way they are applied.

First, let us consider an example of model construction, using the first presented method adding edges, for the formula $\Diamond P \wedge \Box\Diamond P$.

Figure 5.4 shows the premodel obtained in a step-by-step run with a breakpoint on the `Pos` and `NotNec` rules, which are the only rules creating new successors. We should be aware that between two successive steps, all the other rules are also called, including the `And`, `Nec` and `Transitive_Edges` rules.

At step 1, the initial formula is reduced to its conjuncted subformulas, $\Diamond P$ and $\Box\Diamond P$. From step 1 to step 2 the rule `Pos` is applied on the formula $\Diamond P$ which yields to the first successor, linked to the initial node by R and containing P. At step 2, the rule `Nec` is applied and sends $\Diamond P$ forward to the new child node, which makes the rule `Pos` applicable once again and leads to step 3. At

step 3, the transitive edge from the first node to the third is added, making the rule `Nec` applicable. This causes the propagation of the formula $\Diamond P$ once again to the new child and makes the rule `Pos` applicable once again, and the cycle continues like that.



(a) step 1 (b) step 2 (c) step 3 (d) step 4

Figure 5.4: A non-terminating model construction for the formula $\Diamond P \wedge \Box\Diamond P$.

Practically, the reason of this ad infinitum run is "the development of $\Diamond$-formulas which have been already treated with no new $\Box$-formulas appearing in between" [HSZ96]. For instance, imagine we have at a given world w a $\Diamond$-formula, say $\Diamond A$, with the set of $\Box$-formulas $\{\Box A_1, \ldots, \Box A_n\}$[1]. This leads to creating a new R-successor node u of w with the following set of formulas $\{A, A_1, \ldots, A_n, \Box A_1, \ldots, \Box A_n\}$. Suppose that one of the formulas A_i is in fact the formula $\Diamond A$. This will lead to the creation of a new R-successor node v of u with exactly the same set of formulas in u.

Hence, a combinatorial argument limits the number of possible nodes that can be created in an infinite sequence of successors, where we necessarily get a repeated node, i.e. a *cycle* or a *loop* [Gor99]. A terminating method should detect such loop and stop the construction instead of continuing in an infinite path. Loosely speaking, we should never treat a set of formulae if we have already met this set beforehand.

The process of detecting repeated nodes is called "*loop-check*". This detection is only the first stage of the "*loop-blocking*" process, which consists, in addition to the loop-check, in blocking the detected looping nodes.

The loop-check process can be sometimes expensive. Thus, a great deal was accorded to enhancing its efficiency, yielding various techniques used to achieve this test.

[1] In the version with explicit transitive edges, this set is the union of all $\Box$-formulas cumulated from all the ancestor nodes of w.

One way to achieve it is to build a premodel with all the *super-set of sub-formulas* of the input formula à la Fisher-Ladner [FL77]. In such premodels, before applying a rule on a given pattern, we check wether we have not applied it already on that same pattern. However, this test is still expensive and the pre-processing of the input formula leads to premodels of exponential size, which is adequate for EXPTIME logics such as PDL, but it does not make sense in PSPACE logics.

Another way is to keep the usual way of constructing the premodels in an ascending minimalist way (such as in LoTREC) then make a test of *node-inclusion* or *node-equality* of new added nodes and their ancestor nodes, in terms of their sets of formulas. To achieve this test, we compare the whole sets of formulas in leaf nodes and their ancestors in every path and at each iteration of the model construction process

An alternative to this test is to simply check if all the couples $\Diamond A, \Box B$ in a given node have been already treated in ancestor nodes, by testing whether they belong or not to a *history* of cumulated treated $(\Diamond A, \Box B)$-couples [HSZ96]. In this alternative, we are trying to reduce the amount of information related to previous achieved computations in a clever way so only minimal relevant information are stored in memory. However, this does not drastically reduce the time-cost of the loop-check process.

This is why S. Cerrito and M. Cialdea proposed in [CM97] a third way to achieve the loop-check. This approach was used by F. Massacci in [Mas00], where he defines, for each logic, an upper bound on the depth of the premodels (i.e. on the length of the sequence of successor nodes without loop) that we may construct for a given formula of that logic, where this bound is polynomial in the size of the input formula. He stops automatically the construction of a premodel once this depth is reached. However, this was only defined for few modal logics (from K to S4) and it is not always applicable if we cannot calculate a bound on the depth of the models in a certain logic.

Next, we give the implementation in LoTREC of the second solution with "node-inclusion" test.

Detecting loops by testing for nodes inclusion

The implementation of loop-check in LoTREC via node-inclusion test is very simple. First, we define the rule:

```
Rule Mark_Looping_Nodes
 isNewNode u
 isAncestor w u
 contains w u

 mark u Loop_Node
End
```

which is self-explanatory. Then we change the rule `Pos` as follows:

```
Rule Pos
```

```
 hasElement w pos variable A
 isNotMarked w Loop_Node

 createNewNode u
 link w u R
 add u variable A
End
```

The mark `Loop_Node` is not predefined keyword at all, and we may use instead any other word as a tag. The most important is to use the same mark in both rules `Mark_Looping_Nodes` and `Pos`. With the first rule, we effectively achieve the *loop-check* process, and we declare which nodes are looping by marking each one of them as `Loop_Node`. Then we achieve the *loop-blocking* process by preventing the `Pos` rule from being applied on $\Diamond$-formulas of a node reported as `Loop_Node`.

Note that we have to call these two rules in an appropriate order: `Mark_Looping_Nodes` should be called first, before `Pos`, otherwise the loop-check does not take effect.

Figure 5.5 is a remake of the example of Figure 5.4, using the `Mark_Looping_Nodes` rule with the corresponding new version of the `Pos` rule.

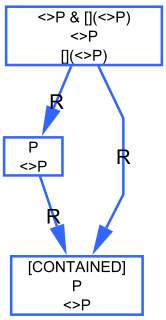


Figure 5.5: Blocking loops by performing nodes inclusion test.

Remark 9. For more optimality, we may also block rules dealing with other formulas (especially the `Or` rule dealing with disjunctions) in order to make sure that contained nodes are completely blocked, and that formulas inside them are not treated by the corresponding rules anymore.

Remark 10. Terminating methods for other transitive logics, such as S4 , KB4 and KD4 are combinations of the method developed here for K4 and those developed so far KT, KB and KD. We can also define a terminating method for the logic K + K4 + Inclusion by combining their corresponding methods (c.f. Section 3.5.1, and Remark 7 page 78).

5.2 Model construction for knowledge may not terminate!

It is normal since transitivity is embedded in S5. Moreover, in the method given in Section 4.6, all the ◇-formulas are sent backward to the same node; the common father node marked as `Root`. Since a given formula can be added only once to the *set of formulas* of a given node, we guarantee in this method that each ◇-formula is treated only once, and we avoid loops caused by repeated nodes.

Nevertheless, other methods may not terminate, as we show in the next sections, without using some tweaks in the rules and/or the strategy.

5.2.1 Rules for S5 with explicit edges

The idea of this method is to create all the necessary (reflexive, symmetric and transitive) edges, in order to make the accessibility relation an equivalence relation. Rules for creating these edges are presented in Sections 4.1, 4.2 and 5.1, and can be defined as follows:

```
Rule Reflexive_Edges
 isNewNode w

 link w w R
End

Rule Symmetric_Edges
 isLinked w u R

 link u w R
End

Rule Transitive_Edges
 isLinked w u R
 isLinked u v R

 link w v R
End
```

In addition to the above rules, the simple `Nec` rule, defined in the method of modal logic K in Chapter 3, is sufficient to propagate the □-formulas in accordance with the truth conditions:

```
Rule Nec
 hasElement w nec variable Formula
 isLinked w u R

 add u variable Formula
End
```

As for the rule dealing with the ◇-formulas, if we use the standard version:

```
Rule Pos
 hasElement w pos variable Formula

 createNewNode u
 link w u R
 add u variable Formula
End
```

then our method may not terminate. Figure 5.6 shows an example of endless premodel construction for the formula $\Box\Diamond P$.

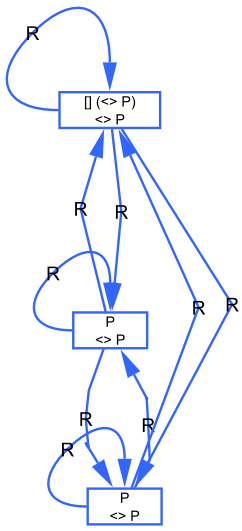


Figure 5.6: A method for S5, which creates reflexive, symmetric and transitive arcs and which uses a naive rule to deal with $\Diamond$-formulas, does not terminate with some formulas, such as the formula $\Box\Diamond P$.

The problem is already explained in Section 5.1 (Figure 5.4), where the same formula is considered in the logic K4.

To avoid such kind of loops, we have to guarantee that a $\Diamond A$ does not yield to the creation of a new node with A when such a node has been already created. To this end, we use a special rule which marks such $\Diamond$-formulas as `Fulfilled`, as follows:

```
Rule Mark_Fulfilled_Pos
 hasElement w pos variable A
 isLinked w u R
 hasElement u variable A

 markExpressions w pos variable A Fulfilled
End
```

then we change the `Pos` rule, in a way that it does create new successors for already `Fulfilled` ◇-formulas:

```
Rule Pos
 hasElement w pos variable Formula
 isNotMarkedExpression w pos variable Formula Fulfilled

 createNewNode u
 link w u R
 add u variable Formula
End
```

The `Mark_Fulfilled_Pos` rule should be called in the strategy right before the `Pos` rule, to guarantee that ◇-formulas, which exist in the premodel at a given iteration, are all marked first as fulfilled, if they really are, before being treated by the `Pos` rule.

Running the same formula of Figure 5.6 with the new rules, leads to a terminating premodel, as shown in Figure 5.7.

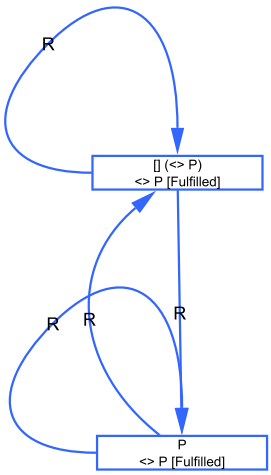


Figure 5.7: Fulfilled ◇-formulas does not lead to create new nodes.

5.2.2 Modelling the knowledge of multiple agents

If we have many agents, we will have an equivalence relation per agent.

Syntax definition in LoTREC

In order to define the syntax of knowledge formulas, we may reuse the definition of the classical logic operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$, as shown in Chapter 2. In addition, we have to define the knowledge operators K and $\hat{K}$. This can be done as follows:

Name	Arity	Display
`knows`	2	`K(_)_`
`knowsHat`	2	`K^(_)_`

According to these operator definitions, the formula $K_{Bob}A \rightarrow \neg\hat{K}_{Bob}\neg A$ should be defined in LoTREC as `imp knows Bob A not knowsHat Bob not A`, which will be displayed as `K(Bob)A -> ~K^(Bob)~A`.

5.2.3 Rules for S5 with multiple agents

Since the K and $\hat{K}$ have the same semantics as the modal operators $\Box$ and $\Diamond$, formulas of the form $K_I A$ and $\hat{K}_I A$ can be treated by rules similar to those defined for the multimodal logic K_n in Section 3.5.

However, using these only rules, we do not guarantee that the accessibility relations, built between the worlds by the mean of the `Pos` rule, are all *equivalence* relations. We have discussed this fact earlier in Section 4.6, with the case of monomodal logic S5, and we have proposed two solutions, which we abord in the sequel: one consists in building explicit additional edges, and the other simulates their presence.

5.2.4 Adding the necessary edges

Symmetric and transitive edges can be added by means of the following rules:

```
Rule Symmetric_Edges
 isLinked w u variable AgentI

 link u w variable AgentI
End

Rule Transitive_Edges
 isLinked w u variable AgentI
 isLinked u v variable AgentI

 link w v variable AgentI
End
```

As for reflexive edges, there is no direct way to add them by means of rules. Suppose that we only have one agent `A1` in the modal operators of the input formula, then we would create the following rule:

```
Rule Reflexive_Edges_For_A1
 isNewNode w

 link w w A1
End
```

If we want to deal with another formula with two agents, `A1` and `A2`, then we should add to this rule another action: `link w w A2`. If we have three agents, then we should add a third action and so on...

What we expect to have to make this construction simpler is to be able to write just one rule as follows:

```
Rule Reflexive_Edges
 isNewNode w
 isAgent variable A

 link w w variable A
End
```

as if all the names of agents can be identified with the condition `isAgent variable A`. However, this is not possible without preprocessing the input formula and extracting every agent name found inside the modal connectors K and $\hat{K}$. This would be tough!

Instead, we can use the following rule:

```
Rule Reflexive_Edges
 hasElement w knows variable AgentI variable Formula

 link w w variable AgentI
End
```

which creates the strictly necessary reflexive edges, needed to propagate the K-formulas appropriately.

Using the above `Reflexive_Edges`, `Symmetric_Edges` and `Transitive_Edges` rules, all that we need to propagate the K-formulas appropriately is the following rule:

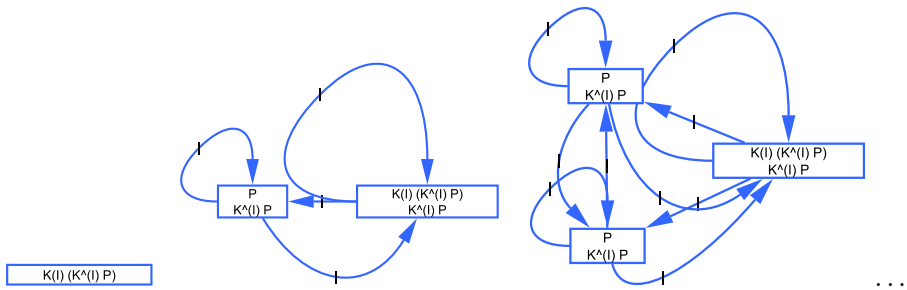
```
Rule Knows
 hasElement w knows variable AgentI variable Formula
 isLinked w u variable AgentI

 add u variable Formula
End
```

As for the $\hat{K}$-formulas, we may lean first toward using the usual rule:

```
Rule KnowsHat
 hasElement w knowsHat variable AgentI variable Formula

 createNewNode u
 link w u variable AgentI
 add u variable Formula
End
```

Nevertheless, the resulting method may not terminate. For example, when running with the formula $K_I\hat{K}_IP$, as seen in Figure 5.8, the model construction does not terminate due to the transitivity of the accessibility relation, as discussed in Section 5.1.

The problem lies in developing new successors with already treated $\hat{K}$-formulas, i.e. the ones which have already been made *fulfilled*. That is why we add the following rule:



(a) step 1 (b) step 2 (c) step 3

Figure 5.8: Non-terminating model construction for multi-S5.

```
Rule Mark_Fulfilled_KnowsHat
 hasElement w knowsHat variable AgentI variable Formula
 isLinked w u variable AgentI
 hasElement u variable Formula

 markExpressions w knowsHat variable AgentI variable Formula Fulfilled
End
```

which mark the already fulfilled $\hat{K}$-formulas. Then we change the `KnowsHat` rule as follows:

```
Rule KnowsHat
 hasElement w knowsHat variable AgentI variable Formula
 isNotMarkedExpression w knowsHat variable AgentI variable Formula Fulfilled

 createNewNode u
 link w u variable AgentI
 add u variable Formula
End
```

so it escapes already fulfilled $\hat{K}$-formulas.

Note that `Mark_Fulfilled_KnowsHat` and `KnowsHat` should be called in this order in the strategy, otherwise `Mark_Fulfilled_KnowsHat` would have no effect. Running with these new rules on the same formula of Figure 5.8 we should obtain a terminating construction which leads to the premodel of Figure 5.9.

5.2.5 Simulating the presence of the edges

In this section, we give the alternative of the method shown in the last section. The method addressed here is similar to the method given in Section 4.6 for monomodal S5.

It consists in linking created (child) nodes to the same (parent) node to reflect the fact that they to the same equivalence class w.r.t. a given accessibility relation. Doing so, there is no need to create explicit reflexive, symmetric an transitive edges between these nodes to link all of them together, provided that

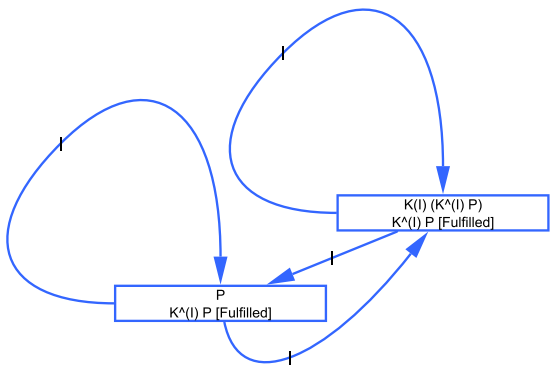


Figure 5.9: Terminating model construction for multi-S5.

the method propagates the K-formulas appropriately, as if these edges were concretely created.

To this end, we define the following self-explanatory rules:

```
Rule Knows_To_Same_World
 hasElement w knows variable AgentI variable Formula

 add w variable Formula
End

Rule Knows_I_To_I_Children
 hasElement w knows variable AgentI variable Formula
 isLinked w u variable AgentI

 add u variable Formula
End

Rule Knows_I_To_I_Parent
 hasElement w knows variable AgentI variable Formula
 isLinked u w variable AgentI

 add u variable Formula
End

Rule Knows_I_To_I_Sibling
 hasElement w knows variable AgentI variable Formula
 isLinked u w variable AgentI
 isLinked u v variable AgentI

 add v variable Formula
End
```

The rules dealing with $\hat{K}$-formulas should create new successors to fulfill these formulas, and at the same time, they should keep the nodes of the same

equivalence class linked to the same parent node.

When a $\hat{K}$-formula is found in the first node, containing the input formula, it should create new successors without any other considerations:

```
Rule KnowsHat_With_No_Parents
 hasElement w knowsHat variable AgentI variable Formula
 hasNoParents w

 createNewNode u
 link w u variable AgentI
 add u variable Formula
End
```

Suppose now that we have a child node u created and linked to the parent node w by the relation I. If a formula $\hat{K}_J A$ is found in u then we should create a node v to hold the formula A, i.e. to fulfill the formula $\hat{K}_J A$. Two cases may take place:

- If $J \neq I$, then v should be linked to u by J; as if a new equivalence class by J has just started, including u and v for the moment.
- Otherwise, i.e. $J = I$, the new node v should belong to the same equivalence class of u and w by the relation I. Hence, according to our design, it should be linked by I to the same parent node w. Which means that the formula $\hat{K}_I A$ should be treated as if it was found in the parent w.

These two cases are taken into account by the following two rules:

```
Rule KnowsHat_I_With_J_Parent
 hasElement u knowsHat variable AgentI variable Formula
 isLinked w u variable Agent_J
 areNotEqual variable AgentI variable Agent_J

 createNewNode v
 link u v variable AgentI
 add v variable Formula
End

Rule KnowsHat_I_With_I_Parent
 hasElement u knowsHat variable AgentI variable Formula
 isLinked w u variable AgentI

 add w knowsHat variable AgentI variable Formula
End
```

These rules can be called in any order inside the strategy.

Remark 11. Methods for logics with axioms 4 or 5 or both, such as KD45, are definable using the already presented methods for K4 and S5. The Universal logic can be tackled exactly as monomodal S5. The logic K +Universal is the logic with two modalities: a K-modality $\Box$, and an S5-modality [U], with the inclusion axiom $[\mathrm{U}]P \to \Box P$. The model construction method for this logic is also definable by combining the methods for S5 and for Inclusion.

5.3 Hybrid Logic

Along the previous sections of the last two chapters, we discovered many logics with special properties or constraints on their accessibility relations. Their model construction methods may vary a lot according to these constraints. In this section, we discover a new family of logics, called *hybrid logics* [ABM01], with a special constraint on their *valuation function.*

In a hybrid logic, we can use special propositional symbols $N, M, \ldots$, called *nominals*, to refer to specific worlds in the Kripke model: the world called N, the world called M,... Together with a special operator @, called *satisfaction operator*, we can write $@_N A$ to say that *at* the world referred by N, A is true.

To build the language of hybrid logic, we add new symbols to the the basic language of the modal logic K. In addition to the set of atomic propositions $\mathcal{P}$, the classical operators $\neg$,$\wedge$,... and the modal operators $\Diamond$ and $\Box$, we add a new set of propositional letters $\mathcal{N}$, such that $\mathcal{N}$ and $\mathcal{P}$ are disjoint ($\mathcal{N} \cap \mathcal{P} = \emptyset$), together with a unary operator @. A formula in hybrid logic is then defined as being any modal formula defined over $\mathcal{N} \cup \mathcal{P}$, or as a formula of the form $@_N A$, where $N \in \mathcal{N}$ and A is another hybrid formula. The logic defined above is the basic hybrid logic called $\mathsf{HL}(@)$.

Models of $\mathsf{HL}(@)$ have a special definition, in contrast with basic models defined in Section 2.3.1:

Definition 11 ($\mathsf{HL}(@)$ model)**.** *Given a set of atomic propositions $\mathcal{P}$ and a set of nominals $\mathcal{N}$, a $\mathsf{HL}(@)$ model M is a tuple (W, R, V) where:*

- *W is a non-empty set;*
- *$R \subseteq W \times W$;*
- *$V\colon W \to 2^{\mathcal{N} \cup \mathcal{P}}$, where for all $N \in \mathcal{N}$, there exists a* unique *$w \in W$ such that $N \in V(w)$.*

The third item of this definition clarifies what we said at the beginning of this section about hybrid logics, as being logics with constraints on the valuation function.

As for the semantics of $\mathsf{HL}(@)$, modal operators are interpreted in the same way as in K. We only define the truth conditions corresponding to nominals and the @ operator:

$M, w \Vdash N$ if, and only if, $N \in V(w)$;

$M, w \Vdash @_N A$ if, and only if, at the unique world $u \in W$ such that $N \in V(u)$ we have $M, u \Vdash A$.

The second condition clarifies the semantics of the @ operator. In order to evaluate a formula $@_N A$ at world w, we *jump* to the unique world u where N holds, and check that A is true at u.

These logics are very expressive in comparison with modal logics introduced so far till now. For instance, we can represent with hybrid formulas some frame

properties which are not definable with formulas of modal logics, such as irreflexivity, asymetry and antisymetry (for proofs see [BdRV01]). For example, the formula $@_N \neg \Diamond N$ says that the world named N is not reachable from itself by the accessibility relation. Thus this formula is valid on precisely those frames which are irreflexive and which are in fact characterized by the axiom schema $@_N \neg \Diamond N$.

After this short introduction, we show now how to implement HL(@) in LoTREC.

5.3.1 Model construction for HL(@)

The main ideas of the method that we propose are:

1. we treat classical and modal connectors as usual by the rules given in Section 3.4;
2. for each formula of the form $@_N A$, we add A to the world named N, and if such a world does not exist, then we create it and we add A to it;
3. for every worlds w,u, if there exists a nominal N such that $N \in V(w)$ and $N \in V(u)$, then we identify w and u as being the same world.

The details are given in the sequel.

Defining the necessary connectors

We start by defining, in addition to the language of the modal logic K, a new binary connector `at`, with a pretty display such as @(_,_), where the first _ denotes a nominal and the second _ denotes a formula. In this way, `at N P` is displayed as @(N,P) and it is used to denote the formula $@_N P$.

Distinguishing nominals from atomic propositions

Suppose we have constructed in LoTREC a node `w` containing P and $\neg Q$, and a different node `u` containing P and Q. This is not enough to produce a clash. In contrast, if these two nodes contain in addition the same nominal N, then we should detect a clash: since N is a nominal the two nodes must be identical, i.e. we have Q and $\neg Q$ at the same world, which closes the premodel.

Since in LoTREC we only have one set of constant symbols (words starting with capital letters), these symbols will be shared by the atomic propositions $\mathcal{P}$ and the nominals $\mathcal{N}$. Hence, we should find a special way to distinguish between atoms and nominals. To do so, we propose to add a special unary connector `nominal` to surround the nominals, such that a formula of the form `N` is taken to be an atomic proposition, while `nominal N` is taken to be a nominal. Therefore the formula $@_N \neg \Diamond N$ is written in LoTREC as `at nominal N not pos nominal N`.[2]

[2]Note that there is no ambiguity when the symbol is the first argument of the @ operator:

Using a common root

When a formula of the form $@_N A$ is added to a given node, we should check every world to detect whether it contains N, in which case we add the formula A to it. However in LoTREC, we cannot browse the whole set of nodes in a given premodel using the simple conditions of LoTREC's language. We would rather proceed as in S5 (see Section 4.6): we create a *common parent* node, let us call it `root`, and then we link this `root` to every node in the premodel by a special link `Root`. We also mark this `root` node by a special mark `Root`, in order to find it easily in our rules.

We define the following rules:

```
Rule Init
 isNewNode w
 isNotMarked w Root

 createNewNode root
 link root w Root
 mark root Root
End

Rule Link_Root_To_New_Nodes
 isNewNode w
 isNotMarked w Root
 isAncestor root w
 isMarked root Root

 link root w Root
End
```

The rule `Init` is to be called at the beginning of the strategy. It creates a new node marked by `Root` and linked to the node containing the input formula. The rule `Link_Root_To_New_Nodes` should be called after the rules creating new nodes (such as the rule `Pos`). It links the `root` node to every other node. Running with these two rules and the formula $\Diamond P$ we obtain the premodel of Figure 5.10.

Checking for the existence of nominals

Let us consider now formulas of the form $@_N A$, where N is a nominal and A is an arbitrary formula of HL(@). As we have said, when a formula of this form is found to a given node, we should check:

- every world to detect the world named by N in order to add the formula A to it;
- in case there is no such world, we should create it.

To cover both cases we define three rules. The first one is:

then the first parameter must be be a nominal. We may chose then to not use the connector `nominal` to surround the first parameter of @.

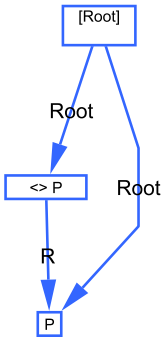


Figure 5.10: Running with `Init` and `Link_Root_To_New_Nodes` rules.

```
Rule Check_For_Existence_Of_Nominals
 hasElement w nominal variable N
 isLinked root w Root

 add root nominal variable N
End
```

The rule `Check_For_Existence_Of_Nominals` adds every nominal N to the `root` node, when this nominal is already assigned to a given node. Hence, after applying this rule we can be sure that the node `root` contains all the nominals which have been already added to the nodes of the premodel before the rule `At_N_When_N_Is_New` is applied.

```
Rule At_N_When_N_Is_New
 hasElement w at nominal variable N variable A
 isLinked root w Root
 hasNotElement root nominal variable N

 createNewNode u
 add u nominal variable N
 link root u Root
End
```

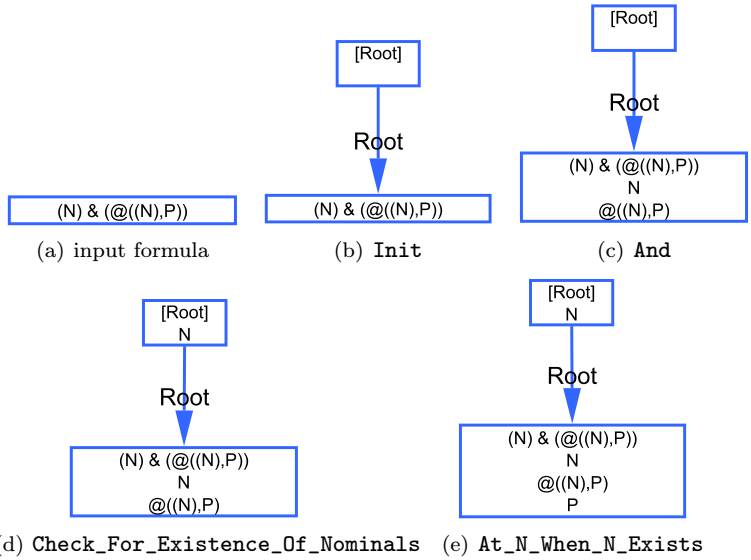
This rule creates a new node and assigns a nominal N to it, if such a nominal N has not been yet assigned to any node (i.e. it does not belong to the node `root`). Once the above rules have been applied for all occurrences of nominals we can apply the following rule, which reflects the truth condition for @-formulas:

```
Rule At_N_When_N_Exists
 hasElement w at nominal variable N variable A
 isLinked root w Root
 isLinked root u Root
 hasElement u nominal variable N

 add u variable A
End
```

In the strategy, these rules should be called as follows:

```
Hybrid_Logic_Strategy
  ...
  ...
  repeat
    Check_For_Existence_Of_Nominals
    applyOnce At_N_When_N_Is_New
  end
  At_N_When_N_Exists
  ...
  ...
End
```

Running with the formula $N \wedge @_N P$, this strategy gives the premodel of Figure 5.11.



(a) input formula (b) `Init` (c) `And`

(d) `Check_For_Existence_Of_Nominals` (e) `At_N_When_N_Exists`

Figure 5.11: Results of applying the rules dealing with @-formulas on $N \wedge @_N P$.

Identifying nodes with the same nominal

Let us consider the formula $\neg P \wedge N \wedge \Diamond(P \wedge N)$. Using the set of rules and the the strategy defined till now, we obtain the open premodel of Figure 5.12.[3]

However, this formula is not satisfiable! The reason is that our current rules do not identify the two nodes which have the same nominal N. Once this is taken into account we shall obtain P and $\neg P$ at the same node; and this

[3]Note that, since there is a display option in LoTREC which allows to hide nodes marked by a specific mark, the `root` node is hidden in Figure 5.12. We did this in order to make the figures more readable. We use this option throughout the remaining examples.

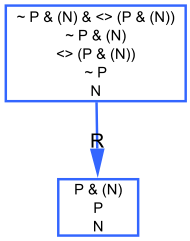


Figure 5.12: Without node identification, an open premodel for the unsatisfiable formula $\neg P \wedge N \wedge \Diamond(P \wedge N)$.

leads to a closed premodel, and in consequence the formula will be reported as unsatisfiable.

According to the rules we have seen up to now, in a given premodel we may have two or more nodes containing the same nominal N. However, according to the HL(@) model definition, there is a *unique* world in which a given nominal N dwells. Hence, when two or more nodes contain the same nominal N, they should be considered as *equal*. We say that these nodes form a N-*equivalence class*, where only one node is sufficient to represent this class.

Practically, we proceed as follows:

1. we choose one of these nodes to represent its equivalence class, we call it the N-*node*;
2. we link each node in the class to its representative N-node by a special link, say `Equal_To`;
3. we identify each of these nodes with the N-node by:
   - copying all its formulas to the N-node,
   - redirecting all its in-edges to the N-node,
   - and redirecting all its out-edges to the N-node.

The first and second items in the above list are ensured by the following two rules:

```
Rule Chose_A_Representative
 hasElement w nominal variable N
 isLinked root w Root
 hasNoSuccessor root nominal variable N

 link root w nominal variable N
End

Rule Detect_Non_Representative_Nodes
 hasElement w nominal variable N
 isLinked root w Root
```

```
 isLinked root u nominal variable N
 areNotIdentical w u

 link w u Equal_To
End
```

The first rule `Chose_A_Representative` is to be *applied only once* to chose a representative node for just *one* nominal, say N. To designate this N-node, we link it to the root node by the nominal N. In this way, we can check the next time we apply this rule whether an N-node is already created for the nominal N or not.

This link also allows to ease the access to this N-node from the root, as we notice in the rule `Detect_Non_Representative_Nodes`. This rule links every node with a nominal N to the corresponding N-node. This link allows the next rules to *copy* the contents and the context of the nodes to their N-nodes, as follows:

```
Rule Copy_Formulas
 hasElement w variable A
 isLinked w u Equal_To

 add u variable A
End

Rule Copy_In_Edges
 isLinked w u Equal_To
 isLinked parent w R

 link parent u R
End

Rule Copy_Out_Edges
 isLinked w u Equal_To
 isLinked w child R

 link u child R
End
```

The above rules are to be called in the strategy as follows:

```
Hybrid_Logic_Strategy
  ...
  ...
  repeat
    applyOnce Chose_A_Representative
    Detect_Non_Representative_Nodes
  end
  repeat
    Copy_Formulas
    Copy_In_Edges
    Copy_Out_Edges
  end
```

```
   ...
   ...
End
```

Running these new rules with the modified strategy we obtain a conveniently closed premodel for the formula $\neg P \wedge N \wedge \Diamond(P \wedge N)$, as shown in Figure 5.13.

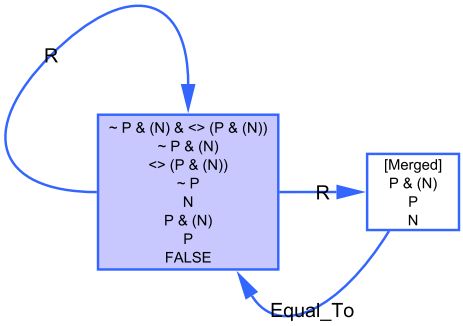


Figure 5.13: A closed premodel for the formula $\neg P \wedge N \wedge \Diamond(P \wedge N)$.

Termination is almost there...

Let us consider now the formula $@_N \Diamond N$. Running our last strategy in the step-by-step mode yields a non-terminating execution, whose first steps are shown in Figure 5.14.

Nevertheless, it does not require a big effort to terminate. In fact, if we had merged all the nodes of a N-equivalence class, we should not have this non-termination problem, since merged nodes would have been physically deleted.

To simulate this deletion, we simply ask our rules to ignore the identified nodes. To do so, we add to our rules the condition: `hasNoSuccessor w Equal_To`, especially the rule `Pos`.

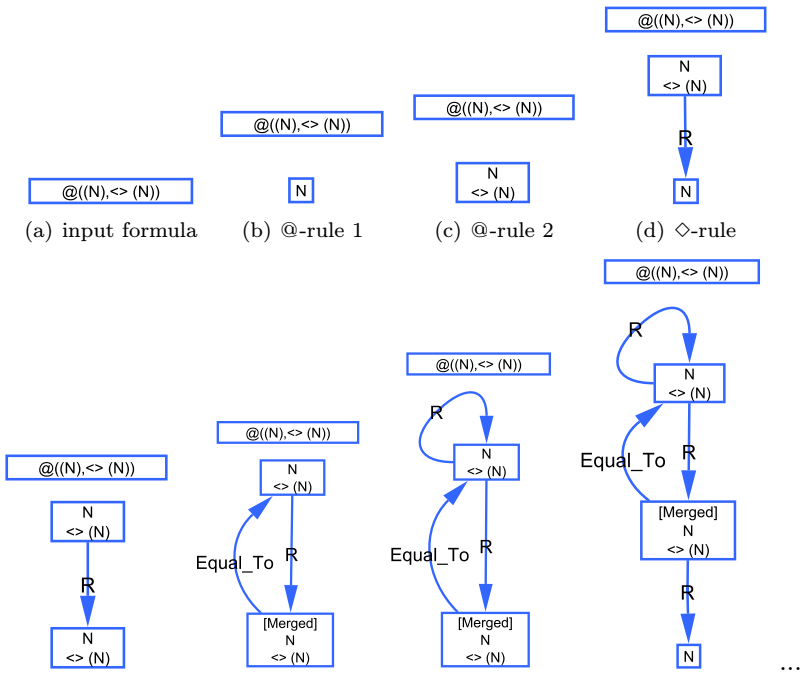
Optionally, we can also add this condition to the other rules, especially the the `Or` rule, to prevent them from being applicable on merged nodes.

We have said that "termination is almost there" since we do not achieve an explicit loop-check solely for the sake of termination. The work done to identify the nodes is already needed, and above all, to satisfy the truth conditions corresponding to the nominals.

Running on the same formula with these new settings, the method terminates and delivers the open premodel of Figure 5.15.

The order of rules in the strategy matters

As shown in previous methods of this chapter, the order of the rules is important to ensure termination. The golden rule is to call the rules which check and mark the nodes to be blocked *before* calling the rules which should not be applied on blocked nodes.



(a) input formula (b) @-rule 1 (c) @-rule 2 (d) ◇-rule

(e) @-rule 2 (f) identification (g) merge (h) ◇-rule again..

Figure 5.14: An example formula shows that our method is not yet terminating.

A last example should clarify this point. Let us consider the formula $@_N\Diamond(N \wedge \Diamond P) \wedge @_N\Box\Diamond(N \wedge \Diamond P)$.

Constructing a premodel for this formula should be terminating, and should deliver an open premodel as in Figure 5.16. This result is only possible if in the strategy, the node identification rule `Detect_Non_Representative_Nodes` is called *before* the `Pos` rule.

The reader may verify in a step-by-step mode that otherwise, the ◇-formulas existing in the identified nodes (marked as `Merged` in the figure) will be treated first by the `Pos` rule. Which creates new *similar* nodes with the same set of formulas, where the ◇-formulas are also treated first, and so on...

5.4 Completeness vs. termination

While soundness proofs are generally straightforward, completeness proofs are strictly more complex. The underlying algorithms are usually very procedural, and in consequence the proofs are not so formal. But there exists a standard methodology as exposed e.g. in [Fit83]. It is usually based on a *fair strategy* giving equal rights to all the rules (which corresponds more or less to our basic fair algorithm): it is assumed that every applicable rule $\rho \in \mathcal{R}$ will eventually

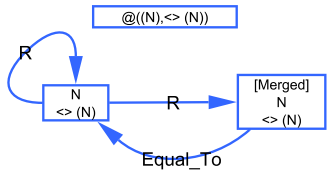


Figure 5.15: An open premodel for the formula $@_N \Diamond N$. It is obtained after a terminating run.

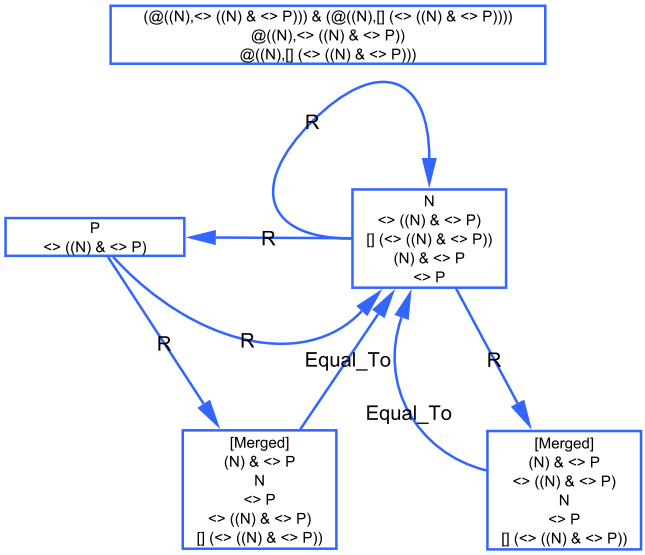


Figure 5.16: An example where $\Diamond$-formulas should be treated after the identification of nodes with the same nominal.

be applied. This strategy ensures that a model can be associated to an open premodel.

Such an algorithm does not always terminate for modal logics beyond the simple logics; for example the fair application of the standard K4 tableau rules to the formula $\Box\Diamond P \wedge \Diamond P$ runs forever (see Section 5.1).

This is because the application of the π-rules (in the Smullyan-Fitting terminology) to formulas of the form $\Diamond A$ at a node `w` creates new nodes with non-strict subformulas.

Therefore the last part of the standard presentations in the literature ([Fit83, Gor99, Mas00]) contains an algorithm that combines rule applications in a way such that termination is guaranteed. Typically:

- The application of 'many' rules is blocked if they have already been applied. This is the case for the α-, β-, and π-rules, but not for the ν-rules[4].

[4]Mainly, ν-rules are $\Box$-like rules, π-rules are $\Diamond$-like rules, β-rules are branching rules and

- The application of the π-rules to node `w` is blocked if `w` is included in (or identical to) some ancestor node.

Then a combinatorial argument limits the number of possible nodes that can be created in a graph, and the number of possible graphs.

As pointed out in [FdCGHS05], it is important to observe that *completeness has to be re-proved for such a terminating algorithm.* Usually informal arguments are employed here. They basically say that the restriction on rule applicability imposed by the terminating algorithm is *harmless* in what concerns completeness.

Clearly, while proving completeness for the fair strategy is already a hard task, to prove completeness for the terminating strategy is even harder. As said above, when working with complex modal logics one often would like to modify the interplay between the modalities. Moreover, even for a given logic one would often like to fine-tune the strategy in order to improve performance of the algorithm, shorten proofs, decrease the size of the resulting model, etc. For all these reasons it seems to be too hard a task to prove both completeness and termination of the decision procedures for these logics.

We believe that for such complex logics formal termination proofs are more important in practice than formal completeness proofs, and that it is sufficient to only *conjecture completeness.*

To support our claim, suppose we have at our disposal a tableau algorithm that is both terminating and sound for logic L, together with an algorithm that allows us to build a model from an open premodel (usually consisting in closing the accessibility relation under some property). Suppose we want to know whether a given formula A is L-satisfiable or not.

Let us proceed as follows:

- If the algorithm returns a set of premodels all of which are closed, soundness of the algorithm ensures that A is L-unsatisfiable.

- If the tableau algorithm returns at least one open premodel `M` then our completeness conjecture says that it can be turned into an L-model of A. Now this can be verified in the following way:

  1. Apply the model building algorithm to build a model M, with actual world w_0.
  2. Check whether M is a L-model.
  3. Model check whether $M, w_0 \Vdash A$.
  4. If $M, w_0 \Vdash A$ then A is indeed L-satisfiable; else we have discovered that our conjecture was erroneous, and that the algorithm has to be modified to "get more complete" w.r.t. L-unsatisfiability.

Thus we propose to postpone the work on completeness until the good logic has been found together with a satisfactory (terminating) strategy.

α-rules are simple classical and reduction rules.

5.5 Termination by checking for loops

The second criterion guaranteeing termination of theorem 1 does not apply to logics with transitive accessibility relations. In these logics we use a loop test to detect every node that is included in some ancestor[5]. Such nodes are reported as loop-nodes and they are to be blocked. We block a loop-node by preventing the the `Pos`-rule from being applied on it. The following theorem (that was proved in [GHS06b]) states the conditions under which this ensures termination.

Theorem 2 ([GHS06b]). *Let $\mathcal{R}$ be a set of rules such that for every $\rho \in \mathcal{R}$,*

- *ρ is analytic (only* subformulas *of the original formula are added to nodes in the action part of ρ);*
- *The `Pos`-rule does not apply in a node which is included in some ancestor.*

Let $\mathcal{S}$ be a strategy that is built from $\mathcal{R}$. Then the model construction terminates, in other words apply $\mathcal{S}$ to any formula leads to a finite premodel.

Proof. First, observe that due to the subformula condition:

- every node only contains a finite number of formulas, and
- there can only be a finite number of nodes with differing associated set of formulas.

We prove that (1) application of $\mathcal{S}$ can never lead to premodels of infinite depth, and (2) application of $\mathcal{S}$ can never lead to premodels with infinite branching factor.

The argument is that creation of new nodes is subject to non-inclusion in an ancestor node, which due to the condition on $\mathcal{S}$ is always tested before node creation. Therefore a branch of infinite length would contain an infinite number of nodes having different associated formula sets. This cannot be the case because our rules are monotonic: no formulas are erased. This even gives us the classical upper bound: length of branches is bounded by an exponential in the length of the input formula.

Infinite branching in node `w` could only be produced by introducing infinitely many new nodes `u` for each $\Diamond A$ formula in `w`. But given a node `w`, the number of $\Diamond$-formulas being bounded by the (linear) number of sub-formulae of the input formula, each node may only have a linear number of successors. Thus the branching factor is bounded. This ends the proof. □

This theorem covers the case of logics build over the standard axioms K, T, B, 4, 5, Confluence. Moreover, it applies to hybrid logic HL(@) as presented in this chapter.

To sum it up, the above theorem and theorem 1, page 96, give some general termination criteria for our strategies. Together, these criteria cover all standard

[5] `w` is included in $\mathtt{w}_0$ if for every formula A appearing in `w`, A also appears in $\mathtt{w}_0$.

modal logics, including Linear Temporal Logic LTL and Propositional Dynamic Logic PDL, as we shall see in the next chapter.

For logics that do not fit into the above framework we have to prove termination on a case-by-case basis.

Conclusion

In this chapter we gave the model construction method for logics with transitive accessibility relations, such as K4 and S5, and for a hybrid logic HL(@) with a specific constraint on the valuation function of its models.

In the methods of these logics, it is required to guarantee the termination by blocking loop-nodes. In the case of K4 and S5 a loop-node is detected by performing a node inclusion test. In the case of HL(@), a loop-node is avoided by identifying each pair of nodes named by the same nominal.

Chapter 6

Model Checking

Introduction

In chapter 2, we defined one of the reasoning problem as:

- Input: a formula A, a model M and a world w in M;
- Output: does A hold at w?

This chapter presents a procedure to answer this question for formulas of modal logics. First, we show the main flow of this procedure when it is achieved on paper. Then we show how to implement a set of rules to automate the run of this procedure using LoTREC.

6.1 Model checking on paper

Given a formula A, a model M and a specific world w of M, the corresponding model checking problem can be formulated as "is A true at the world w of the model M?", or also "does A hold at the world w of the model M?".

Answering such a question is what we call a *model checking* procedure, and it is usually achieved in three steps:

1. model construction,
2. analysis of the input formula,
3. synthesisation of the answer about its truth.

In the following subsections we detail each step, as it is usually done manually on paper. Then in the next section, we show how to automatize these steps in LoTREC.

6.1.1 The model and the formula

Formally, a model is defined as a triple $M = (W, R, V)$ over a set of labels $\mathcal{I}$ and a set atomic propositions $\mathcal{P}$, as shown in Definition 1, page 40.

Given $\mathcal{I} = \{ \text{ R } \}$ and $\mathcal{P} = \{P, Q\}$, let us consider the following example model $M = (W, R, V)$ where:

- $W = \{w, u, v, x\}$,
- $R : \mathcal{I} \to 2^{W \times W}$ such that: $R(\text{ R }) = \{(w, u), (w, v), (v, u), (v, x)\}$,
- $V : W \to 2^{\mathcal{P}}$ such that: $V(w) = \{P\}$, $V(u) = \emptyset$, $V(v) = \{P, Q\}$ and $V(x) = \{Q\}$.

Practically, a model is represented as a graph. Figure 6.1 shows a graphical representation of the above model M, where w is the node at the top of the graph, u is at the right hand side of the reader, x is at the bottom and v is the fourth other node.

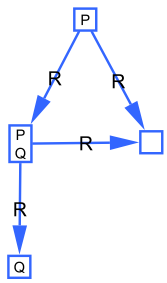


Figure 6.1: Example of a model.

Let us also consider the formula $A = \Diamond\Box(P \vee Q)$, and let us formulate the following model checking problem: does $M, w \Vdash A$?

To solve this problem, we add the formula A to the world w, as shown in Figure 6.2, then we try to answer the question "is A true at w?" according to the truth conditions of modal logic.

If A is an atomic formula (i.e. an element of $\mathcal{P}$) the we would be able to answer this question. Otherwise, we should analyse A and its subformulas until being able to evaluate their truth, as shown in the next subsections.

6.1.2 Top-down formula decomposition

This step consists in decomposing the formula into subformulas, and transforming the question about its truth into smaller questions about the truth of its subformulas, according to the truth conditions.

For example, according to the truth conditions, the formula $\Diamond\Box(P \vee Q)$ is true at w if, and only if, its subformula $\Box(P \vee Q)$ is true at least at one of the successors of w, i.e. at u or at v. That is why in "step 1" of Figure 6.3, we add

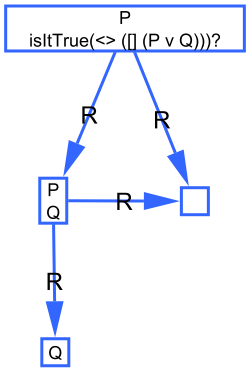


Figure 6.2: Formulating the problem of model checking.

the formula $\Box(P \vee Q)$ to both u and v, in order to be tested whether it is true or not at these worlds.

Now $\Box(P \vee Q)$ is true at v if, and only if, for every world w' linked to v, $P \vee Q$ is true at w'. That is why we add $P \vee Q$ to all the successors of v, i.e. u and x, in order to be tested in there, as shown in "step 2" of Figure 6.3.

This new added formula, $P \vee Q$, is true at u (resp. x) if, and only if, one of its disjuncts P or Q is true at the same world u (resp. x). This is why we add these disjuncts to the corresponding worlds u and x in "step 3" of Figure 6.3.

Since these disjuncts are atomic formulas, no further decomposition is needed anymore, and the answers about the truth values of these analysed formulas can be synthesised from now on.

We also halt this decomposition procedure when we obtain $\Diamond$-formulas or $\Box$-formulas in leaf nodes (nodes with no successors). For example, the formula $\Box(P \vee Q)$ can not be decomposed furthermore at the world u.

6.1.3 Bottom-up satisfiability check

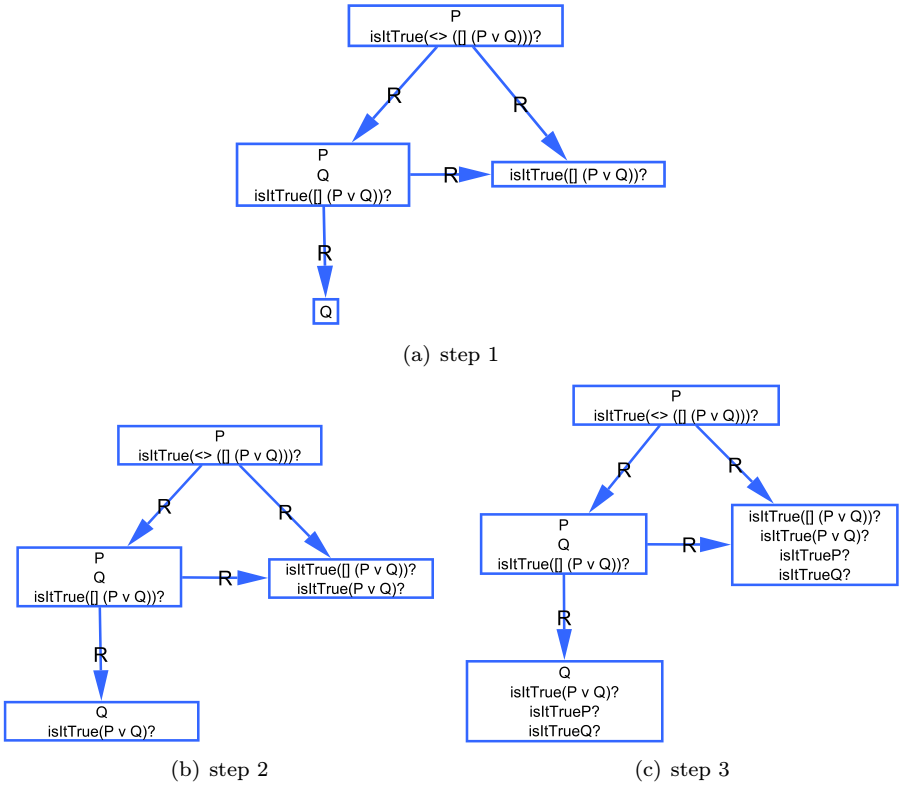
At the end of the above decomposition phase, we start being able to answer the questions about the truth of the formulas step-by-step.

First, we can answer the questions about the atomic formulas (i.e. elements of $\mathcal{P}$). For example, Q is true at x but false at u, whereas P is false in both of them, as shown in "step 1" of Figure 6.4.

Other formulas can be then recursively checked according to the truth conditions and the truth of their subformulas. For example, $P \vee Q$ is true at x since one of its disjunct, Q, is true at x, but it is not in u since its both disjuncts are not ("step 2" of Figure 6.4).

Hence, $\Box(P \vee Q)$ is not true at v, since $P \vee Q$ is not true in all its successors, namely, it is false at u ("step 3" of Figure 6.4).

At leaf nodes, $\Diamond$-formulas are evaluated to false, since there is no successor at all, i.e. for a formula $\Diamond B$ at a leaf node, it is not possible to have a successor



(a) step 1

(b) step 2

(c) step 3

Figure 6.3: Top-down formula analysis.

where B holds. Dually, $\Box$-formulas are evaluated to true. In our example, the formula $\Box(P \vee Q)$ is true at u, as shown in "step 3" of Figure 6.4.

Finally, the question about the truth of the input formula $A = \Diamond\Box(P \vee Q)$ at the world w can be answered by "yes, it is", since its subformula $\Box(P \vee Q)$ is true at, at least, one of w's successors, namely u ("step 4" of Figure 6.4).

6.2 Model checking in LoTREC

In this section, we show how to implement the above described model checking procedure in LoTREC. Our aim is to achieve model checking automatically.

6.2.1 Defining the model checking problem

The construction of a model consists in creating a graph, according to Definition 1, page 40. In LoTREC, this can be done, as shown in Appendix B.1, by three different ways:

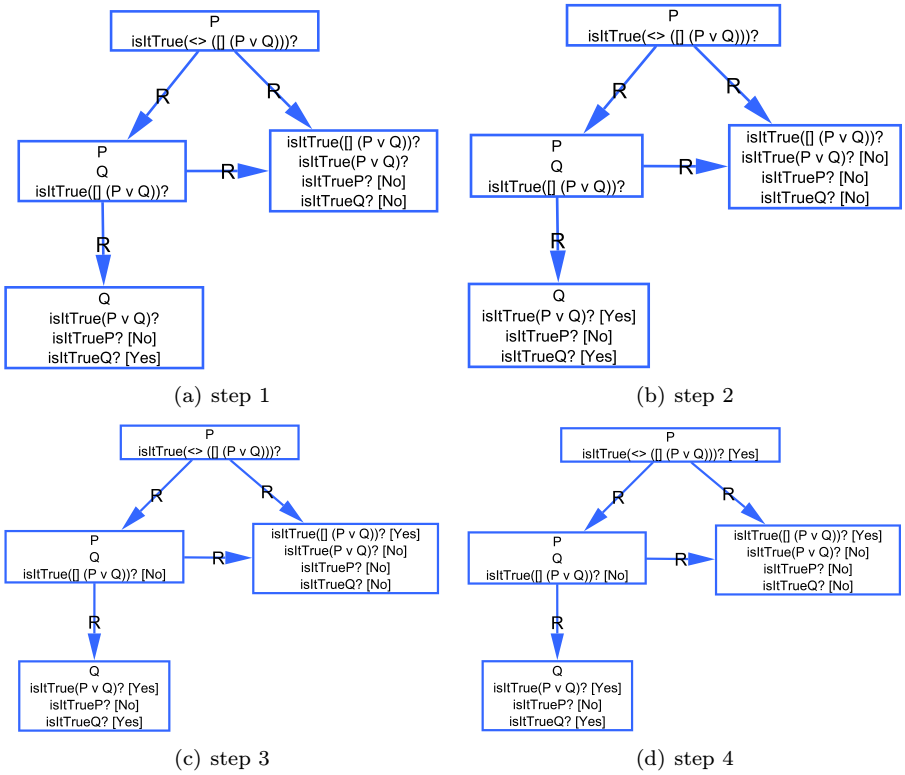


Figure 6.4: Bottom-up satisfiability check.

1. using a rule,
2. loading a (saved) model file, or
3. adding nodes, edges and formulas using the GUI.

The following rule can take care of building the model of Figure 6.1, if called once at the beginning of the model checking strategy:

```
Rule Build_Model
 createNewNode w
 createNewNode u
 createNewNode v
 createNewNode x
 link w u R
 link w v R
 link v u R
 link v x R
 add w P
 add v P
```

```
 add v Q
 add x Q
End
```

You may refer to Appendix B.1 for more details.

Once the model is built, we can define a problem of model checking of the form "does $M, w \Vdash A$?" by adding the formula A to the world w. This can be done inside the rule defining the model, or after loading the model from a file or drawing it by hand. However, the formula can not be simply added *as it*, as we may notice from the following example.

Suppose A is the atomic formula P, and that we want to check its satisfiability at the world w of the model M of Figure 6.1. Adding the formula A directly would be confusing, since we may confound henceforth the added atom P and the other atom P which has been already figuring in the world w by definition of M.

In order to distinguish between the added formulas and the atoms preexisting in the model by definition, we encapsulate the added formulas in a special unary connector[1]. This connector is defined as follows:

Name	Arity	Display
`isItTrue`	1	`isItTrue(_)?`

Thus in the above example we would add `isItTrue(`P`)?` to w instead of P.

To add the formula $\Diamond\Box(P \vee Q)$ to the model of Figure 6.1 using the rule `Build_Model`, we only have to add the action `add w isItTrue pos nec or P Q`. We obtain the same result as in Figure 6.2.

6.2.2 Top-down rules

The top-down steps presented in section 6.1.2 can be achieved automatically by defining an appropriate set of rules, which transform the question about the truth value of the input formula into smaller questions about the truth values of its subformulas.

For example, to check the truth of a formula of the form $\neg A$ at a world w, the truth of A should be checked at w. To check formulas of the form $A \wedge B$ or of the form $A \vee B$, both formulas A and B should be checked. This is done by the following rules:

```
Rule Not_Top_Down
 hasElement w isItTrue not variable A

 add w isItTrue variable A
End

Rule And_Top_Down
 hasElement w isItTrue and variable A variable B
```

[1] We could have used to the same end some other techniques, such as marking the formulas with special marks.

```
 add w isItTrue variable A
 add w isItTrue variable B
End

Rule Or_Top_Down
 hasElement w isItTrue or variable A variable B

 add w isItTrue variable A
 add w isItTrue variable B
End
```

Rules dealing with formulas obtained with other Boolean connectors, such as $\rightarrow$ and $\leftrightarrow$, can be similarly defined.

As for modal formulas, their check consists in checking their subformulas in successor nodes, as follows:

```
Rule Nec_Top_Down
 hasElement w isItTrue nec variable A
 isLinked w u R

 add u isItTrue variable A
End

Rule Pos_Top_Down
 hasElement w isItTrue pos variable A
 isLinked w u R

 add u isItTrue variable A
End
```

To guarantee that the formulas are recursively decomposed, we shall call the above rules repeatedly, as in the following strategy, for example:

```
Strategy Top_Down
  repeat
    Not_Top_Down
    And_Top_Down
    Or_Top_Down
    Nec_Top_Down
    Pos_Top_Down
  end
End
```

6.2.3 Bottom-up rules

After calling the top-down rules, we can start to evaluate the truth of some of the subformulas. The truth value of a formula A can be given by marking the formula `isItTrue(`A`)?` with `Yes` to say that it is true, or with `No` otherwise.

The first answers can be formulated for atoms, as done by the following self-explanatory rules:

```
Rule Atom_True_Bottom_Up
 hasElement w isItTrue variable A
 isAtomic variable A
 hasElement w variable A

 markExpressions w isItTrue variable A Yes
End

Rule Atom_Not_True_Bottom_Up
 hasElement w isItTrue variable A
 isAtomic variable A
 hasNotElement w variable A

 markExpressions w isItTrue variable A No
End
```

After that the first answers are formulated, we can synthesis these answer recursively up toward the input formula, as shown in the bottom-up steps of section 6.1.3.

For negation formulas, we use the following rule:

```
Rule Not_True_Bottom_Up
 hasElement w isItTrue not variable A
 isMarkedExpression w isItTrue variable A No

 markExpressions w isItTrue not variable A Yes
End

Rule Not_Not_True_Bottom_Up
 hasElement w isItTrue not variable A
 isMarkedExpression w isItTrue variable A Yes

 markExpressions w isItTrue not variable A No
End
```

For conjunctive Boolean formulas, we define the following rules:

```
Rule And_True_Bottom_Up
 hasElement w isItTrue and variable A variable B
 isMarkedExpression w isItTrue variable A Yes
 isMarkedExpression w isItTrue variable B Yes

 markExpressions w isItTrue and variable A variable B Yes
End

Rule And_Left_Not_True_Bottom_Up
 hasElement w isItTrue and variable A variable B
 isMarkedExpression w isItTrue variable A No

 markExpressions w isItTrue and variable A variable B No
End
```

```
Rule And_Right_Not_True_Bottom_Up
 hasElement w isItTrue and variable A variable B
 isMarkedExpression w isItTrue variable B No

 markExpressions w isItTrue and variable A variable B No
End
```

The dual of the above rules is used to treat disjunctive formulas:

```
Rule Or_Not_True_Bottom_Up
 hasElement w isItTrue or variable A variable B
 isMarkedExpression w isItTrue variable A No
 isMarkedExpression w isItTrue variable B No

 markExpressions w isItTrue or variable A variable B No
End

Rule Or_Left_True_Bottom_Up
 hasElement w isItTrue or variable A variable B
 isMarkedExpression w isItTrue variable A Yes

 markExpressions w isItTrue or variable A variable B Yes
End

Rule Or_Right_True_Bottom_Up
 hasElement w isItTrue or variable A variable B
 isMarkedExpression w isItTrue variable B Yes

 markExpressions w isItTrue or variable A variable B Yes
End
```

As for modal formulas of the form $\Box A$ and found at a world w, we use a special condition keyword `isMarkedExpressionInAllChildren`, to check if the formula A is marked by `Yes` in all the successors of w. If it is the case, then `isItTrue(`$\Box A$`)?` is marked by `Yes`[2]. Otherwise, it is sufficient to test if A is marked by `No` in one of w's successors, to mark `isItTrue(`$\Box A$`)?` by `No`. These two cases are taken into account by the following rules:

```
Rule Nec_True_Bottom_Up
 hasElement w isItTrue nec variable A
 isMarkedExpressionInAllChildren w isItTrue variable A R Yes

 markExpressions w isItTrue nec variable A Yes
End

Rule Nec_Not_True_Bottom_Up
 hasElement w isItTrue nec variable A
 isLinked w u R
```

[2]Note that if w has no successors at all, then `isMarkedExpressionInAllChildren` holds too, and `isItTrue(`$\Box A$`)?` is conveniently marked by `Yes`.

```
 isMarkedExpression u isItTrue variable A No

 markExpressions w isItTrue nec variable A No
End
```

Dually, ◇-formulas are handled by the following rules:

```
Rule Pos_Not_True_Bottom_Up
 hasElement w isItTrue pos variable A
 isMarkedExpressionInAllChildren w isItTrue variable A R No

 markExpressions w isItTrue pos variable A No
End

Rule Pos_True_Bottom_Up
 hasElement w isItTrue pos variable A
 isLinked w u R
 isMarkedExpression u isItTrue variable A Yes

 markExpressions w isItTrue pos variable A Yes
End
```

The set of bottom-up rules can be called in one strategy:

```
Strategy Bottom_Up
  repeat
    Atom_True_Bottom_Up
    Atom_Not_True_Bottom_Up
    Not_True_Bottom_Up
    Not_Not_True_Bottom_Up
    And_True_Bottom_Up
    And_Left_Not_True_Bottom_Up
    And_Right_Not_True_Bottom_Up
    Or_Not_True_Bottom_Up
    Or_Left_True_Bottom_Up
    Or_Right_True_Bottom_Up
    Nec_True_Bottom_Up
    Nec_Not_True_Bottom_Up
    Pos_Not_True_Bottom_Up
    Pos_True_Bottom_Up
  end
End
```

Putting all together

To apply all the above rules we define a new strategy calling the above smaller strategies and rules as follows:

```
Strategy Model_Checking_Strategy
  Build_Model
  Top_Down
```

```
  Bottom_Up
End
```

Note that if the model is built without using the rule `Build_Model`, then this rule can be omitted from the strategy.

6.3 Further discussions

The method given in this chapter is suitable for solving model checking problems in all the monomodal logics presented in former chapters.

In order to deal with multimodal versions of these logics, our method should be extended. However, the extension is simple and consists in replacing the formulas `nec variable A`, `pos variable A` and the occurrences of `R` in the modal rules by the formulas `nec variable R variable A`, `pos variable R variable A` and by occurrences of `variable R` respectively, exactly as done in Chapter 3 to extend the model construction method of K to K_n.

Dealing with formulas from other logics, having other sets of special connectors, or having different semantics, may necessitate more effort to adapt the above method.

A special case of model checking is when the formula must be checked in an already constructed premodel resulting from the application of one of the model construction methods implemented in LoTREC. For example, after building a premodel for an LTL or PDL formula, some subformulas (namely eventualities) should be checked in the premodel (see Section 7.1.6).

Recall that a model construction method creates premodels, and that a premodel has to be extended in order to obtain a model which conforms to Definition 1. The reason is that during the premodel construction:

- some atoms may not be added at some worlds, without being necessarily false at these worlds;
- and some (reflexive, symmetric, transitive...) edges may not be created.

Hence, using the model checking method presented in this chapter, the evaluation of atomic and modal formulas, including the input formula itself, could be erroneous.

In most cases, we can still use this method to check formulas in premodels after turning them into models (i.e. after generating the lacking edges and atoms). In some special cases, the method is to be changed and adapted to take into account the specificities of the constructed premodels.

Chapter 7

Logics with transitive closure

Introduction

In chapter 5 we saw how tableaux methods can be implemented for logics such as K4 and S4. In the last chapter we learned how to define a model checking procedure. Now we show how to put these techniques altogether to handle logics whose accessibility relations are *transitively closed*: LTL and PDL.

7.1 Linear Temporal Logic LTL

In a temporal logic the set of possible worlds correspond to *moments in time*. Models of time have a *temporal accessibility relation* between the worlds. It defines an order on the moments. A moment u which is accessible from a moment w is "in the future" of w. The accessibility relation between these worlds depends on our view of time.

7.1.1 LTL models

In LTL we have a linear discrete time line that is isomorphic to the set of natural numbers $\mathbb{N}$ together with the successor function. It follows that the LTL accessibility relation is serial, linear and discrete. When wRw' then w' is the *next* moment after w. (It is often considered that the set of possible worlds is the set of natural numbers $\mathbb{N}$, but it is not necessary to do so.) Hence, given a moment w we can write $\langle w_0, w_1, \ldots \rangle$ to designate the sequence of moments that are in the future of w, where w_0 is w.

7.1.2 Syntax of LTL

Temporal logics extend classical propositional logic with a set of *temporal operators* that navigate between moments using the accessibility relation. The basic operators are:

- $\mathrm{X}A$: read as "next time A";
- $\mathrm{F}A$: read as "finally A" or "eventually A";
- $\mathrm{G}A$: read as "globally A" or "always A";
- $A\mathrm{U}B$: read as "A until B".

Sometimes, the alternative notations OA, $\Box A$ and $\Diamond A$ are used for the first three operators.

7.1.3 Semantics of LTL

The truth conditions for the Boolean connectives are as usual (as in Definition 5). The truth conditions of the temporal operators are given with respect to a linear time model $M = (W, R, V)$ as follows, where $\langle w_0, w_1, \ldots \rangle$ is the sequence of moments that are in the future of w, and $w_0 = w$:

$M, w \Vdash \mathrm{X}A$ iff $M, w_1 \Vdash A$;

$M, w \Vdash \mathrm{F}A$ iff there exist $i \in \mathbb{N}$ such that $M, w_i \Vdash A$;

$M, w \Vdash \mathrm{G}A$ iff for all $i \in \mathbb{N}$ we have $M, w_i \Vdash A$;

$M, w \Vdash A\mathrm{U}B$ iff there exists $i \in \mathbb{N}$ such that $M, w_i \Vdash B$ and for all $0 \leq j < i$ we have $M, w_j \Vdash A$.

According to the above truth conditions, we can verify that the following equivalences are valid in LTL:

1. $\neg\mathrm{X}A \leftrightarrow \mathrm{X}\neg A$
2. $\neg\mathrm{G}A \leftrightarrow \mathrm{F}\neg A$ (alternatively, $\neg\mathrm{F}A \leftrightarrow \mathrm{G}\neg A$)
3. $\mathrm{G}A \leftrightarrow A \wedge \mathrm{XG}A$
4. $\mathrm{F}A \leftrightarrow A \vee \mathrm{XF}A$
5. $A\mathrm{U}B \leftrightarrow B \vee (A \wedge \mathrm{X}(A\mathrm{U}B))$
6. $\top\mathrm{U}A \leftrightarrow \mathrm{F}A$
7. $\neg(A\mathrm{U}B) \leftrightarrow (\neg B \wedge \neg A) \vee (\neg B \wedge \mathrm{X}\neg(A\mathrm{U}B))$

The last axiom is equivalent to $(\neg B\mathrm{U}(\neg B \wedge \neg A)) \vee \mathrm{G}\neg B$.

The language of LTL allows us to express a bunch of interesting properties:

- Safety: "Something bad will not happen",
  e.g. $\mathrm{G}\neg(Metro\_Door\_Closing \wedge Pass\_Sensor\_On)$;
- Liveness "Something good will happen",
  e.g. $\mathrm{F}Rich$,
  $\mathrm{G}(Start\_Thesis \rightarrow \mathrm{F}\ Finish\_Thesis)$;
- Fairness "If something is attempted/requested infinitely often, then it will be successful/allocated infinitely often",
  e.g. $\mathrm{GF}\ Ready \rightarrow \mathrm{GF}\ Run$.

In what follows we show how to implement a model construction method for LTL which takes into consideration the above semantics.

7.1.4 Model construction for LTL in LoTREC

First, LTL connectors should be defined in LoTREC. This can be done as explained in chapter 2, section 2.2.

Since classical propositional formulas in LTL have the same semantics as in classical propositional logic, the rules dealing with classical connectors are exactly the same rules defined in Chapter 3 for modal logic K. In the sequel, we give the rules dealing with the temporal operators.

Rules for $\mathrm{X}A$ and $\neg\mathrm{X}A$ formulas

The rule dealing with the X operator is similar to the rule dealing with the $\Diamond$ operator, since: for every formula $\mathrm{X}A$ found in a world w, it has to create a new successor, say world u, and link it to w by the accessibility relation R, then add the formula A to it.

However, in order to ensure the *linearity* of the resulting (pre)models, this rule should make sure that each node has *at most one successor* node. This can be achieved as explained in Section 4.3 for the rule dealing with the $\Diamond$ operator in $\mathsf{K.alt}_1$. Hence, we define the following two rules:

```
Rule Create_One_Successor
 hasElement w next variable A
 hasNoSuccessor w R

 createNewNode u
 link w u R
End

Rule Next
 hasElement w next variable A
 isLinked w u R

 add u variable A
End
```

The rule `Create_One_Successor` should be called in the strategy after the `applyOnce` keyword, as explained in Section 4.3. The rule `Next` propagates the subformula A of every XA formula appropriately to the created successor.

Another rule is needed to replace every formula of the form ¬XA by its equivalent formula X¬A, as follows:

```
Rule NotNext
 hasElement w not next variable A

 add w next not variable A
End
```

This replacement formula is then treated by the X-rules.

Rules for GA and ¬FA formulas

According to the truth conditions, a formula GA is true at a given world w if it holds at w and at every world in the future of w.

Hence, we define a rule which, for every formula GA found at a world w, adds both A and XGA to w. This ensures that in the next state (i.e. the next possible world linked to w) GA will be true, i.e. that A and XGA will be also true, and so on.

```
Rule Globally
 hasElement w globally variable A

 add w variable A
 add w next globally variable A
End
```

A formula ¬FA is true if, and only if, the formula G¬A is. Hence, the following rule:

```
Rule NotFinally
 hasElement w not finally variable A

 add w globally not variable A
End
```

Using this rule, we rewrite ¬FA formulas by G¬A formulas, which are handled then by the rule `Globally`.

Rules for FA and ¬GA formulas

A formula FA is true if, and only if, the formula A is or the formula XFA is. Hence the following rule:

```
Rule Finally
 hasElement w finally variable A

 duplicate premodel_copy
 add w variable A
```

```
 add premodel_copy.w next finally variable A
End
```

This rule makes a copy of the actual premodel, where a formula FA is found at a world w, and calls it `premodel_copy` (See detailed explanation Section 3.3.1). In the world w of the current premodel, it adds A, which *directly* fulfills the formula FA at w. In the world w of `premodel_copy`, it adds instead XFA. Later, when the `Next` rule is applied, the formula FA is added to the next successor of w in `premodel_copy`, if there is any. In this way, the fulfillment of FA is *postponed* in the `premodel_copy` to the next step of the model construction process.

The `Finally` rule also deals with formulas obtained from the following rule, when it is applied on $\neg$GA formulas:

```
Rule NotGlobally
 hasElement w not globally variable A

 add w finally not variable A
End
```

Rules for AUB and $\neg(A$U$B)$ formulas

The U connector is handled with the following rule:

```
Rule Until
 hasElement w until variable A variable B

 duplicate premodel_copy
 add w variable B
 add premodel_copy.w variable A
 add premodel_copy.w next until variable A variable B
End
```

This rule makes a copy of the actual premodel, where a formula AUB is found at a world w, and calls it `premodel_copy` (exactly as the rule `Finally` does). In the world w of the current premodel, it adds B, which *directly* fulfills the formula AUB. In the world w of `premodel_copy`, it adds instead A and X(AUB). Later, when the `Next` rule is applied, the formula AUB is added to the next successor of w in `premodel_copy`, if there is any. This way, the fulfillment of the formula AUB (by having the formula B) is *postponed* in the world w of `premodel_copy` to the next step of the model construction process.

Negated U-formulas are handled by the following rule:

```
Rule NotUntil
 hasElement w not until variable A variable B

 duplicate premodel_copy
 add w and not variable B not variable A
 add premodel_copy and not variable B next not until variable A variable B
End
```

For every $\neg(A\mathrm{U}B)$ formula found at a world w, the above rule intends to add either $\neg B \wedge \neg A$ or $\neg B \wedge \mathrm{X}\neg(A\mathrm{U}B)$ to w.

7.1.5 Termination: detecting loops by node-inclusion test

Despite the fact that the above rules are non-analytic, the accessibility relation in LTL models is transitive, as discussed earlier in this chapter. Hence, the same non-termination problem faced when dealing with K4 would appear when dealing with LTL. Thus, the above method may not always terminate.

For example, constructing the premodels for the formula $\mathrm{GX}P$, using this method, does not terminate.

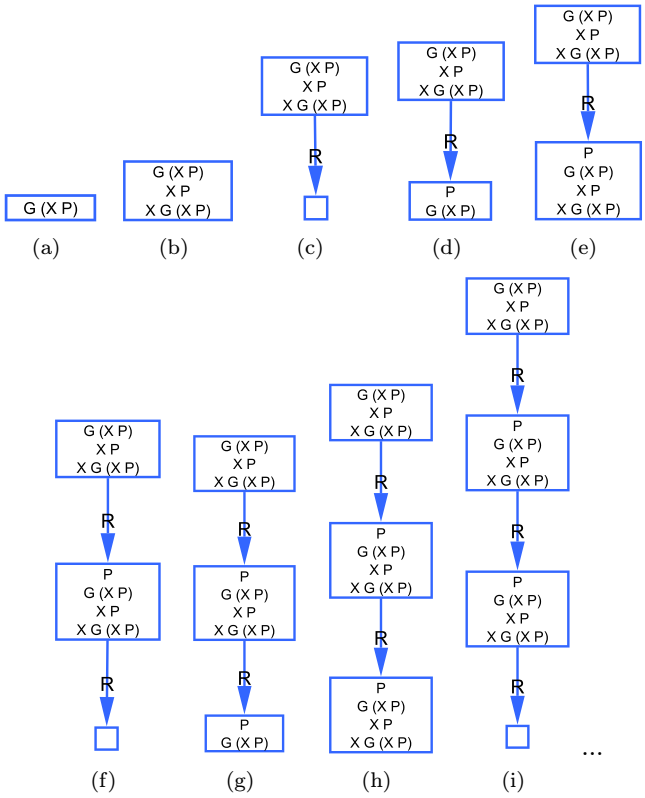


Figure 7.1: Without performing a loop-test, the model construction method for LTL may not terminate.

As we may notice in Figure 7.1, the formula $\mathrm{GX}P$ (a) yields a successor with P and the same original formula $\mathrm{GX}P$ (d), which leads to a new successor with P and the same initial formula $\mathrm{GX}P$ (g), and so on...

To avoid such infinite loops, we add to the above method an inclusion-test rule:

```
Rule Mark_Node_Included_In_An_Ancestor_Node
 isNewNode u
 isAncestor w u
 contains w u

 mark u Loop_Node
End
```

and we block every node included in an ancestor node by changing the rule `Create_One_Successor` as follows:

```
Rule Create_One_Successor
 hasElement w next variable A
 hasNoSuccessor w R
 isNotMarked w Loop_Node

 createNewNode u
 link w u R
End
```

The rule `Create_One_Successor` should be called in the strategy right after the rule `Mark_Node_Included_In_An_Ancestor_Node`, otherwise, this latter would not take effect.

Running the obtained method on the same formula GXP terminates, as shown in Figure 7.2.

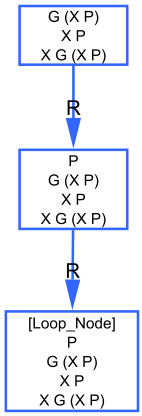


Figure 7.2: Loop-blocking by checking for node-inclusion.

Note that we may change similarly any other rule, so it becomes unapplicable on nodes marked as `Loop_Node`, since such nodes should be blocked (i.e. not explored by the rules anymore). Especially, we may change the branching rules if we are interested in enhancing the performance of our method, such as the rules `Or`, `Finally`, `Until` and `NotUntil`. However, these rules should be always called before the rule `Mark_Node_Included_In_An_Ancestor_Node`. Otherwise, we

may never mark equal nodes at the convenient moment before creating new successors. Here is a sketch of the needed strategy:

```
Strategy LTL_Strategy
   repeat
    repeat
      CPL_Strategy
      NotFinally
      NotGlobally
      NotNext
      Globally
      Finally
      Until
      NotUntil
    end
    Mark_Node_Included_In_An_Ancestor_Node
    applyOnce Create_One_Successor
    Next
  end
End
```

This loop-blocking technique is exactly the same used in Section 5.1 to guarantee the termination of the model construction method for K4. Although, it also guarantees the termination in LTL, the resulting model construction method would be not suitable, as we may see in Section 7.1.7.

In the next section, we deal with another problem which appears once we use a loop-blocking technique in LTL method. As we shall see, when some premodels are blocked due to loop-check, they remain open, however, they are not "*extensible*" to a model.

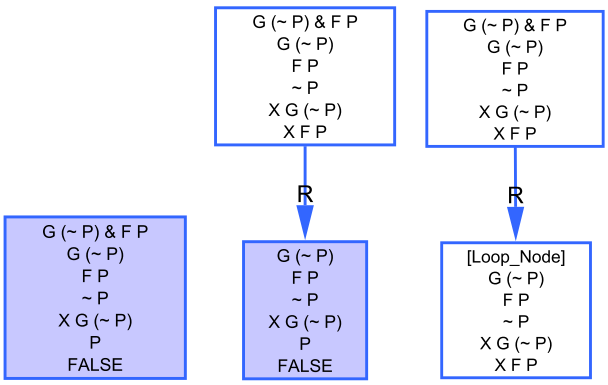
7.1.6 Checking the fulfillment of eventualities

There is a specificity for the method of LTL with loop-check in comparison with K4. In K4, any open premodel is extensible to a model, whether it is loop-free or not. Whereas in LTL, it is not always the case.

For instance, the formula $\mathrm{G}\neg P \wedge \mathrm{F}P$ is not satisfiable. Which means that our premodel construction method should only give closed premodels for this formula. However, Figure 7.3 shows that an open premodel is obtained for this formula (premodel.2.2).

Note that this open premodel is blocked since a loop is detected after checking for node-inclusion. Note also that the formula $\mathrm{F}P$ is not fulfilled in this open premodel, since P does not hold at any world of this premodel, and that, instead, $\neg P$ holds at every world. Hence, this premodel is not extensible to a model of the input formula (and its subformulas). Such a premodel is called an "*inextensible*" premodel.

We conclude that, in LTL, we have to check the fulfillment of the eventualities (i.e. $\mathrm{F}A$ and $A\mathrm{U}B$ formulas) in open premodels in order to distinguish between "*extensible*" and "*inextensible*" open premodels.



(a) premodel.1 (b) premodel.2.1 (c) premodel.2.2

Figure 7.3: An example of an open (blocked) premodel for an unsatisfiable formula.

An FA (resp. AUB) formula is fulfilled at a given world w if, and only if, the formula A (resp. B) is added to w or to one of the successor nodes of w. If w is a loop node included in an ancestor node u, then a formula FA (or AUB) is fulfilled in w if, and only if, it is fulfilled in its loop-ancestor-node u.

To catch the loop ancestor easily, we may change the `Mark_Node_Included_In_An_Ancestor_Node` rule as follows:

```
Rule Mark_Node_Included_In_An_Ancestor_Node
 isNewNode u
 isAncestor w u
 contains w u

 mark u Loop_Node
 link u w Loop
End
```

Henceforth, every loop node would be linked to its containing ancestor node by a special link labeled `Loop`.

Now we can check the fulfillment of eventualities using a model-checking-like technique (see Chapter 6). To mark every fulfilled FA eventuality, we call repeatedly the following rules at the end of the LTL strategy:

```
Rule Finally_Fulfilled_At_The_Same_World
 hasElement w finally variable A
 hasElement w variable A

 markExpressions w finally variable A Fulfilled
End

Rule Finally_Fulfilled_At_A_Successor
 hasElement w finally variable A
```

```
 isAncestor w u
 hasElement u variable A

 markExpressions w finally variable A Fulfilled
End

Rule Finally_Fulfilled_At_Ancestor_Parent_Node
 hasElement u finally variable A
 isLinked u w Loop
 isMarkedExpression w finally variable A Fulfilled

 markExpressions u finally variable A Fulfilled
End
```

Similar rules should be defined (by replacing A by B) to mark fulfilled $A\mathrm{U}B$ eventualities.

After calling these rules, unfulfilled eventuality would be left without being marked as `Fulfilled`. Hence, a node containing an eventuality not marked as fulfilled should be marked as `Inextensible_Premodel`, as follows:

```
Rule Unfulfilled_Finally_Means_Inextensible_Premodel
 hasElement w finally variable A
 isNotMarkedExpression w finally variable A Fulfilled

 mark w Inextensible_Premodel
End

Rule Unfulfilled_Until_Means_Inextensible_Premodel
 hasElement w until variable A variable B
 isNotMarkedExpression w until variable A variable B Fulfilled

 mark w Inextensible_Premodel
End
```

The following two rules propagates the `Inextensible_Premodel` mark to every node of a given "*inextensible*" premodel, including the node containing the input formula, so the user may notice this information easily:

```
Rule Propagate_Inextensible_Premodel_Mark_Up
 isMarked w Inextensible_Premodel
 isLinked u w R

 mark u Inextensible_Premodel
End

Rule Propagate_Inextensible_Premodel_Mark_Down
 isMarked w Inextensible_Premodel
 isLinked w u R

 mark u Inextensible_Premodel
End
```

Running the obtained method with the same example formula $\mathrm{G}\neg P \wedge \mathrm{F}P$ gives the same two closed premodels (premodel.1 and premodel.2.1) of Figure 7.3, whereas the third open premodel (premodel.2.2) is reported as `Inextensible_Premodel` as shown in Figure 7.4.

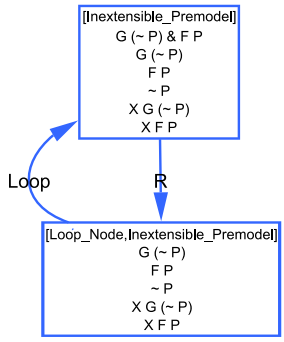
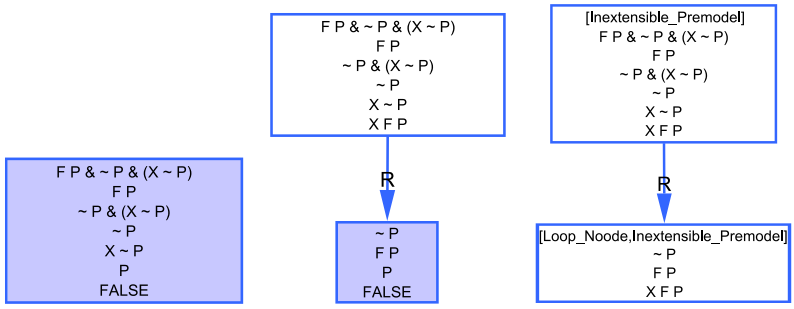


Figure 7.4: The only open premodel for the formula $\mathrm{G}\neg P \wedge \mathrm{F}P$ is reported as a `Inextensible_Premodel`, since the eventuality formula $\mathrm{F}P$ is not fulfilled in it.

7.1.7 Termination: detecting loops by node-equality test

There is another specificity for the method of LTL with loop-check in comparison with K4. Checking for node inclusion is sufficient in the case of K4, as shown in Section 5.1, while it is not suitable in the case of LTL anymore.

To see this, let us consider, for instance, the formula $\mathrm{F}P \wedge \neg P \wedge \mathrm{X}\neg P$.



(a) premodel.1 (b) premodel.2.1 (c) premodel.2.2

Figure 7.5: Blocking a node included in an ancestor node guarantees the termination in LTL, but it is not suitable for dealing with some formulas, for instance $\mathrm{F}P \wedge \neg P \wedge \mathrm{X}\neg P$.

Figure 7.5 shows the result of running our method (with node-inclusion loop test and eventualities fulfillment check). It yields three premodels: two of them are closed (premodel.1 and premodel.2.1), and the third one (premodel.2.2) is

open, but stopped due to node-inclusion and then reported as an inextensible premodel since the eventuality formula FP is not fulfilled in it.

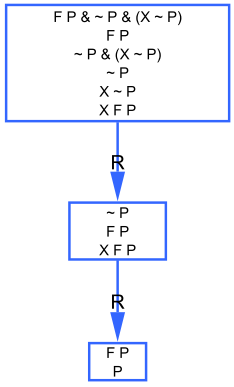


Figure 7.6: The premodel of the formula F$P \wedge \neg P \wedge$ X$\neg P$, which is missed by the method with node inclusion test, and which could have been obtained by developing furthermore the premodel 2.2 of Figure 7.5.

Nevertheless, this formula is satisfiable as shown in Figure 7.6. In this figure, the premodel could have been obtained by our method if it has continued developing the premodel.2.2 blocked by node-inclusion just one further step.

Hence, to guarantee the termination of the LTL method, we have to judge the presence of a loop by check for the node-*equality*, instead of node-inclusion. This can be done by the following rule:

```
Rule Mark_Node_Equal_To_An_Ancestor_Node
 isNewNode u
 isAncestor w u
 contains w u
 contains u w

 mark u Loop_Node
End
```

We may also add to this rule the action `link u w Loop` as explained in Section 7.1.6. This rule is to be called instead of the rule `Mark_Node_Included_In_An_Ancestor_Node`, right before the rule `Create_One_Successor` which has to be changed, as shown in Section 7.1.5.

Using node-equality loop-blocking, the premodel shown in Figure 7.6 can be found by our method for the formula F$P \wedge \neg P \wedge$ X$\neg P$.

To test furthermore the obtained method, we show the result of running it on the satisfiable formula GXP (Figure 7.7), and on the unsatisfiable formula G(F$P \wedge \neg P$) (Figure 7.8). Both runs terminate due to node-equality check. The open premodel of the former formula is "extensible" to a model. Whereas the only open premodel of the second formula is not extensible to a model, hence called "inextensible".

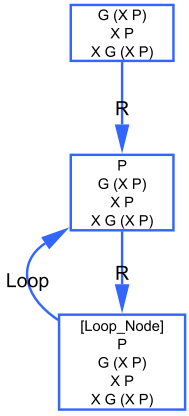


Figure 7.7: Model construction for the formula GXP.

7.2 Propositional Dynamic Logic PDL

Propositional Dynamic Logic (PDL) was proposed in theoretical computer science as a logic to reason about programs, and has also been used by philosophers in order to reason about events and actions. Here we chose to use the term 'program', and not 'action'.

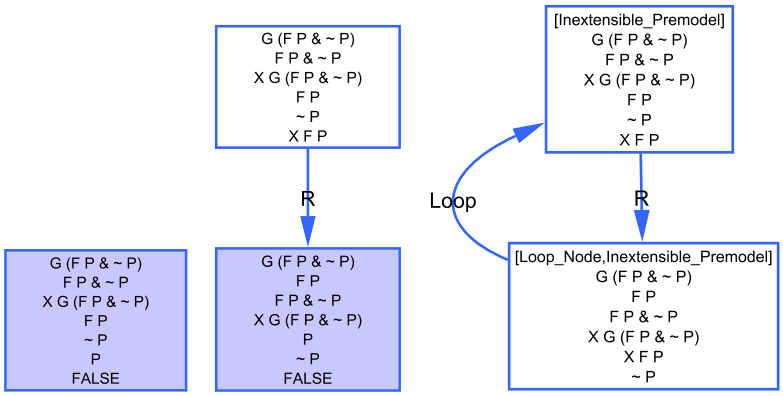
7.2.1 Syntax of PDL

Suppose given countable sets $\mathcal{P}$ of atomic formulas and $\mathcal{I}$ of atomic programs. Complex programs are then built from atomic programs by using connectives that are familiar from programming languages. We denote complex programs by $X, Y, \ldots$ The intended meaning of such complex programs is given in the following table:

Operator	Reading
$X;Y$	X then Y
$X \cup Y$	nondeterministic choice between X and Y
$A?$	test of the formula A: if A is true then continue, else fail
X^*	execute X an arbitrary number of times

Note that tests have formulas as arguments. Formally we therefore have to define the set of programs $\mathcal{P}rog$ and the set of formulas $\mathcal{F}or$ by mutual recursion, as the smallest set such that:

- $I \subseteq \mathcal{P}rog$
- if $X, Y \in \mathcal{P}rog$, then $X;Y \in \mathcal{P}rog$;
- if $X, Y \in \mathcal{P}rog$, then $X \cup Y \in \mathcal{P}rog$;



(a) premodel.1 (b) premodel.2.1 (c) premodel.2.2

Figure 7.8: Model construction for the formula $\mathrm{G}(\mathrm{F}P \wedge \neg P)$.

- if $A \in \mathcal{F}or$, then $A? \in \mathcal{P}rog$;
- if $X \in \mathcal{P}rog$, then $X^* \in \mathcal{P}rog$;
- $\mathcal{P} \subseteq \mathcal{F}or$;
- if $A, B \in \mathcal{F}or$, then $A \wedge B \in \mathcal{F}or$;
- if $A, B \in \mathcal{F}or$, then $A \vee B \in \mathcal{F}or$;
- if $A, B \in \mathcal{F}or$, then $A \rightarrow B \in \mathcal{F}or$;
- if $A \in \mathcal{F}or$, then $\neg A \in \mathcal{F}or$;
- if $A \in \mathcal{F}or$ and $X \in \mathcal{P}rog$, then $[X]A \in \mathcal{F}or$.

The formula $[X]A$ is read "after every possible execution of program X, A holds".

There exist extensions of that basic language with more program operators, such as converse and intersection.

7.2.2 Semantics of PDL

In PDL, programs are interpreted as transitions between worlds.

A PDL model is of the form $M = (W, R, V)$, whose ingredients have the same definition as in multimodal logics (cf. Chapter 2). In particular R maps every program $X \in \mathcal{P}rog$ to an accessibility relation R_X.

The truth conditions for the Boolean connectives are as usual (cf. in Definition 5). Moreover:

$$M, w \Vdash [X]A \text{ iff } M, w_i \Vdash A \text{ for every } w' \in R_X(w).$$

However, not every model of the above kind is acceptable. The reason is that up to now we have no guarantee that complex programs are interpreted as they should be. A *standard model* has to satisfy the following constraints:

- $R_{X;Y} = R_X \circ R_Y$;
- $R_{X \cup Y} = R_X \cup R_Y$;
- $R_{A?} = \{\langle w, w\rangle \mid w \in W \text{ and } M, w \Vdash A\}$;
- $R_{X^*} = (R_X)^*$.

The following equivalences are valid in standard models.

1. $\neg[X]A \leftrightarrow \langle X\rangle\neg A$
2. $[X;Y]A \leftrightarrow [X][Y]A$
3. $[X \cup Y]A \leftrightarrow [X]A \wedge [Y]A$
4. $[A?]B \leftrightarrow \neg A \vee B$
5. $[X^*]A \leftrightarrow A \wedge [X][X^*]A$

Dually, we have the following equivalences:

1. $\neg\langle X\rangle A \leftrightarrow [X]\neg A$
2. $\langle X;Y\rangle A \leftrightarrow \langle X\rangle\langle Y\rangle A$
3. $\langle X \cup Y\rangle A \leftrightarrow \langle X\rangle A \vee \langle Y\rangle A$
4. $\langle A?\rangle B \leftrightarrow A \wedge B$
5. $\langle X^*\rangle A \leftrightarrow A \vee \langle X\rangle\langle X^*\rangle A$

7.2.3 LoTREC rules for PDL

We start with the outline of the method. We are going to have the following tableaux rules:

- rules dealing with classical formulas are those of Section 3.4;
- rules for formulas of the form $\langle X\rangle A$ and $[X]A$ where X is a complex, non-atomic program: they rewrite these formulas according to the equivalences given in Section 7.2.2,
- rules for formulas of the form $\langle I\rangle A$ and $[I]A$ formulas, where I is an *atomic* program: they create new I-successors and propagate $[I]A$ formulas as done in the usual $\Diamond$- and $\Box$-rules.

We give in the sequel some of the rules of the second and third bullets.

Rules for sequence

To deal with $\langle X;Y\rangle$- and $[X;Y]$-formulas we use the following two rules:

```
Rule Pos_Seq
 hasElement w pos seq variable X variable Y variable A

 add w pos variable X pos variable Y variable A
End

Rule Nec_Seq
 hasElement w nec seq variable X variable Y variable A

 add w nec variable X nec variable Y variable A
End
```

The first rule reduces every $\langle X;Y\rangle A$ formula to the formula $\langle X\rangle\langle Y\rangle A$, whereas the second reduces every $[X;Y]A$ to $[X][Y]A$.

Rules for choice

According to the equivalences $\langle X\cup Y\rangle A \leftrightarrow \langle X\rangle A \vee \langle Y\rangle A$ and $[X\cup Y]A \leftrightarrow [X]A \wedge [Y]A$, we define the following two rules:

```
Rule Pos_Union
 hasElement w pos union variable X variable Y variable A

 add w or pos variable X variable A pos variable Y variable A
End

Rule Nec_Union
 hasElement w nec union variable X variable Y variable A

 add w and nec variable X variable A nec variable Y variable A
End
```

They are self-explanatory.

Rules for test

Rules for the test are inspired from the fact that $\langle A?\rangle B \leftrightarrow A\wedge B$ and $[A?]B \leftrightarrow \neg A \vee B$. We define them as follows:

```
Rule Pos_Test
 hasElement w pos test variable A variable B

 add w and variable A variable B
End

Rule Nec_Test
 hasElement w nec test variable A variable B
```

```
 add w or not variable A variable B
End
```

Rules for transitive closure operators

The $\langle^*\rangle$ and $[^*]$ operators are interpreted similarly to the G and F operators of LTL. A formula $\langle X^*\rangle A$ is true at a world w if, and only if, either A is true at w or $\langle X\rangle\langle X^*\rangle A$ is. In the second case, we are postponing the fulfillment of $\langle X^*\rangle A$ to an X-successor of w. Dually, a formula $[X^*]A$ is true at a world w if, and only if, A and $[X][X^*]A$ are true at w. This guarantees that in any X-successor of w $[X^*]A$ will be true, i.e. A will be true and $[X][X^*]A$ and so on...

In LoTREC, we define the following two rules:

```
Rule Pos_Star
 hasElement w pos star variable X variable A

 add w or variable A pos variable X pos star variable X variable A
End

Rule Nec_Star
 hasElement w nec star variable X variable A

 add w and variable A nec variable X nec star variable X variable A
End
```

Note that for the negation of the modal operators we use the usual two rules:

```
Rule Not_Pos
 hasElement w not pos variable X variable A

 add w nec variable X not variable A
End

Rule Not_Nec
 hasElement w not nec variable X variable A

 add w pos variable X not variable A
End
```

Rules for modal operators with atomic programs

When the above rules are called repeatedly, every complex should have been reduced to either a formula of the form $\langle I\rangle A$ or $[I]A$, where I is an atomic program. At this stage, we call the following two rules:

```
Rule Pos_Atomic_Program
 hasElement w pos variable I variable A
 isAtomic variable I

 createNewNode u
```

```
 link w u variable I
 add u variable A
End

Rule Nec_Atomic_Program
 hasElement w nec variable I variable A
 isLinked w u variable I

 add u variable A
End
```

They are exactly the same usual multimodal ◇- and □-rules.

7.2.4 Termination

However, the above method does not terminate when dealing with a $\langle^*\rangle$-formula. For instance, let us consider the model construction of the formula $\langle I^*\rangle P$, as shown in Figure 7.9.

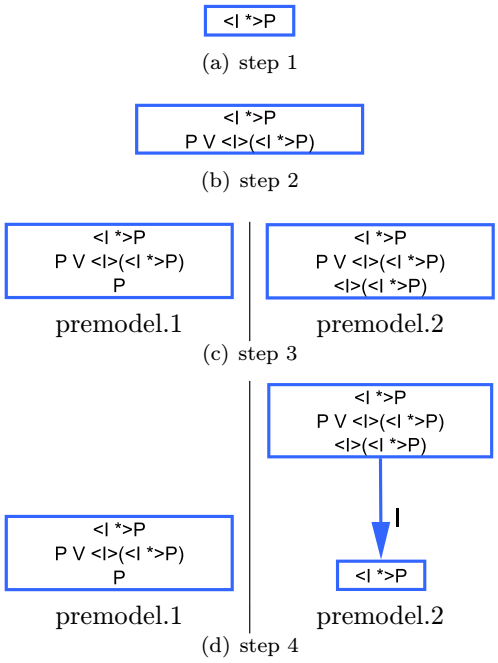
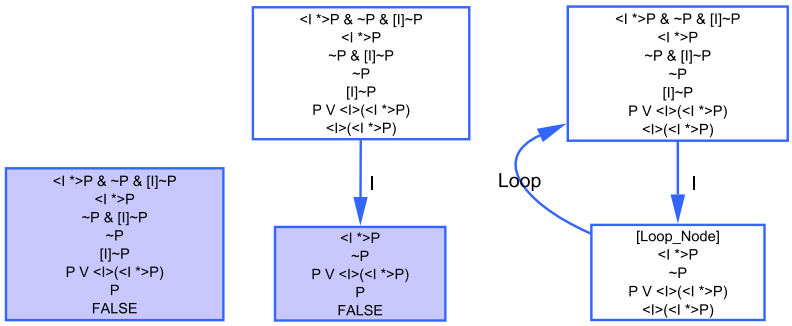


Figure 7.9: Non terminating premodel construction of $\langle I^*\rangle P$.

We notice that at step 4, the same initial node with the input formula is reappearing once again in premodel.2, which means that the first steps will be repeated again and again.

If we block the nodes after a node-inclusion test, we may fall in the same problem discussed in Section 7.1.7 in LTL. It is sufficient to check that the

model construction of the satisfiable formula $\langle I^* \rangle P \wedge \neg P \wedge [I]\neg P$ stops earlier without giving any open premodel that is extensible to a model, i.e. it only gives *inextensible* open premodels (see Figure 7.10).



(a) premodel.1 (b) premodel.2.1 (c) premodel.2.2

Figure 7.10: Loop-blocking using node-inclusion test stops earlier, and does not give an extensible open premodel for the formula $\langle I^* \rangle P \wedge \neg P \wedge [I]\neg P$.

Hence, it is clear that we will guarantee the termination by blocking equal nodes. These nodes are marked first by the rule `Mark_Node_Equal_To_An_Ancestor_Node` defined in Section 7.1.7. Then, we avoid developing these nodes by adding to the rules the (negative) condition `isNotMarked w Loop_Node`, as done in the following rule for example:

```
Rule Pos_Atomic_Program
 hasElement w pos variable I variable A
 isAtomic variable I
 isNotMarked w Loop_Node

 createNewNode u
 link w u variable I
 add u variable A
End
```

Note that, to guarantee the termination of our method, it is sufficient to block this rule. Nevertheless, we may also block other rules, especially the rule `Or` (since it duplicates the whole premodel when applied).

As for the strategy, we should call as long as possible all the classical rules and the rules for test, choice, sequence and iteration, then we call the rule `Mark_Node_Equal_To_An_Ancestor_Node` before calling the rule `Pos_Atomic`. A sketch of such a strategy is:

```
Strategy PDL_Strategy
  repeat
    repeat
      CPLStrategy
      Not_Nec
```

```
      Not_Pos
      Pos_Test
      Nec_Test
      Pos_Star
      Nec_Star
      Pos_Union
      Nec_Union
      Pos_Seq
      Nec_Seq
    end
    Mark_Node_Equal_To_An_Ancestor_Node
    Pos_Atomic_Program
    Nec_Atomic_Program
  end
End
```

Running with this loop-free method, we can find an open premodel for the formula $\langle I^* \rangle P \wedge \neg P \wedge [I] \neg P$, as shown in Figure 7.11.

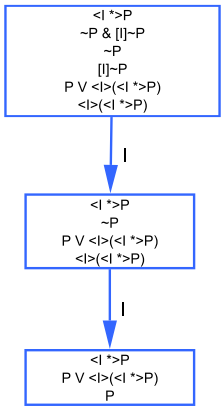
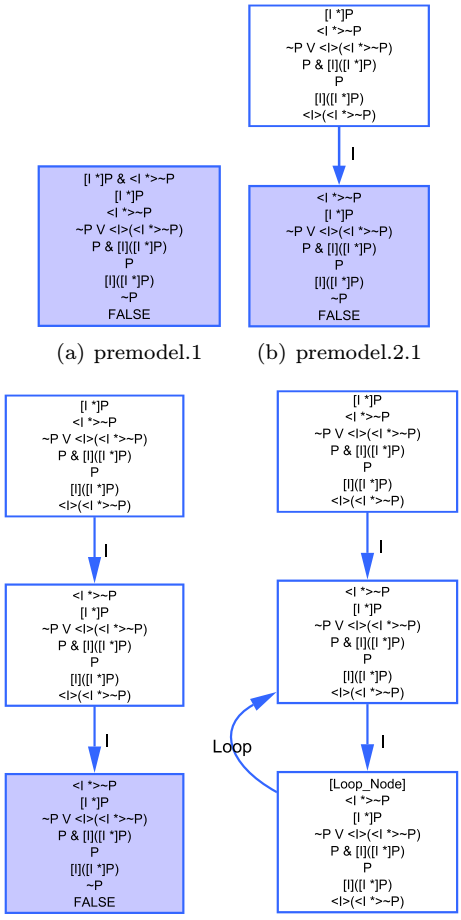


Figure 7.11: An extensible open premodel for the formula $\langle I^* \rangle P \wedge \neg P \wedge [I] \neg P$.

7.2.5 Fulfillment of eventualities

Blocking loop nodes may lead to open premodels with un fulfilled $\langle^* \rangle$-formulas. For instance, the model construction of the formula $[I^*]P \wedge \langle I^* \rangle P$ yields three closed premodels and one open premodel, which should be reported as an *inextensible* premodel (non-extensible to a model) since $\langle I^* \rangle P$ is not fulfilled in it (see Figure 7.12).

We may think that we may decide whether an eventuality $\langle X^* \rangle A$ is fulfilled at a world w by simply checking for the presence of the formula A in one of its successors, as it is the case in LTL (or other temporal logics). However, this is not sufficient. In PDL, we must verify in addition that this successor is connected to w by X-steps. Nevertheless, X could be a complex program.



(a) premodel.1 (b) premodel.2.1

(c) premodel.2.2.1 (d) premodel.2.2.2

Figure 7.12: An inextensible open premodel for the formula $[I^*]P \wedge \langle I^* \rangle P$.

Hence, verifying that two nodes are linked by a specific program may have the same complexity as the original problem.

A first solution is to proceed à la Pratt [Pra80], by keeping track, for every eventuality $\langle X^* \rangle A$, of the information about its *postponement* (i.e. unfulfillment) along the X-paths built while treating the subformulas of this eventuality.

A second solution is to proceed à la de Giacomo&Massacci [dGM00], who are inspired from model checking techniques used for μ-calculus, and consists in renaming each eventuality $\langle X^* \rangle A$ in a given node w by an auxiliary variable, let us say E, and then reducing the $\langle X^* \rangle A$ formula to $\langle X \rangle E$. When E is found later in a successor node u of w then we know that u is connected to w through X-steps.

However, we propose in LoTREC an alternative simpler solution: to check

every formula, including the eventualities, in a one-way bottom-up verification. This solution keeps the premodels clearer by avoiding additional cumbersome edges or information in the graph structure of the premodels, even if it is in some cases suboptimal w.r.t. the other two solutions mentioned above.

This model checking process is not exactly the same method presented in Chapter 6, since the premodels are not fully specified and the analysis of the input formula was done on a different basis. In addition, we shall see that in our model checking here, we only mark `True` formulas. Nevertheless, both methods share the same marking technique.

Checking literals formulas

When both literals P and $\neg P$ coexist at the same node, the premodel is reported as closed, then it is discarded due to the rule `Stop`, and hence, it is not checked by the rules that we are defining here.

Otherwise, an atomic formula P, or its negation $\neg P$, is supposed to be true when it is added by the rules of our method.

Hence, we mark every positive literal P and negative literal $\neg P$ as `True`:

```
Rule Mark_Positive_Literal
 hasElement w variable P
 isAtomic variable P

 markExpressions w variable P True
End

Rule Mark_Negative_Literal
 hasElement w not variable P
 isAtomic variable P

 markExpressions w not variable P True
End
```

Checking classical formulas

As for classical rules, we define a rule to mark double negated formulas ($\neg\neg$), a rule to mark the conjunctions and two rules for the disjunctions, as follows:

```
Rule Mark_Not_Not
 isMarkedExpression w variable A True
 hasElement w not not variable A

 markExpressions w not not variable A True
End

Rule Mark_And
 isMarkedExpression w variable A True
 isMarkedExpression w variable B True
 hasElement w and variable A variable B
```

```
 markExpressions w and variable A variable B True
End

Rule Mark_Or_Left
 isMarkedExpression w variable A True
 hasElement w or variable A variable B

 markExpressions w or variable A variable B True
End

Rule Mark_Or_Right
 isMarkedExpression w variable B True
 hasElement w or variable A variable B

 markExpressions w or variable A variable B True
End
```

Rules concerning other classical connectors (such as $\rightarrow$, ...) are treated in the next set of rules, since they belong to the family of "reduced formulas".

Checking reduced formulas

PDL $\Diamond$- and $\Box$-formulas with complex programs are decomposed according to the reduction rules defined in Section 7.2.3. For example, the rule `Pos_Seq` reduces every $\langle X;Y\rangle A$ formula to the formula $\langle X\rangle\langle Y\rangle A$. According to this decomposition, we mark $\langle X;Y\rangle A$ as `True` in the nodes where the formula $\langle X\rangle\langle Y\rangle A$ is marked as `True`. Whence the rule:

```
Rule Mark_Pos_Seq
 hasElement w pos seq variable X variable Y variable A
 isMarkedExpression w pos variable X pos variable Y variable A True

 markExpressions w pos seq variable X variable Y variable A True
End
```

The rules marking other kinds of reduced formulas, including classical formulas that are reduced to other formulas, are all defined the same way as `Mark_Pos_Seq`.

Remark 12. Reduced formulas are checked w.r.t. their decomposition. For example, suppose that we want to define the rule `Mark_Not_Imp` which checks if a formula $\neg(A \rightarrow B)$ should be marked `True` at a given world w. Then this rule could be defined in two different ways, depending on how the formula $\neg(A \rightarrow B)$ was reduced by the rule `Not_Imp` during the model construction:

- if it was treated by adding the formulas A and $\neg B$ to w, then `Mark_Not_Imp` should mark it `True` whenever both formulas A and $\neg B$ are marked so,
- otherwise, if it was reduced to the formula $A \wedge \neg B$, then the rule `Mark_Not_Imp` should mark the formula $\neg(A \rightarrow B)$ as `True` whenever the formula $A \wedge \neg B$ is marked so.

The first version of the rules is:

```
Rule Mark_Not_Imp
 hasElement w not imp variable A variable B
 isMarkedExpression w variable A True
 isMarkedExpression w not variable B True

 markExpressions w not imp variable A variable B True
End
```

Whereas the second version is:

```
Rule Mark_Not_Imp
 hasElement w not imp variable A variable B
 isMarkedExpression w and variable A not variable B True

 markExpressions w not imp variable A variable B True
End
```

This is why the user should take into consideration how her/his *own* reduction rules are defined, then he/she defines the corresponding checking rules accordingly.

Checking $\langle I \rangle A$ and $[I]A$ formulas

Formulas of the form $[I]A$ are checked exactly as the $\Box$-formulas in the standard model checking method given in Chapter 6, by checking if A is marked `True` in all children nodes at once.

```
Rule Mark_Nec_Atomic_Program
 hasElement w nec variable I variable A
 isMarkedExpressionInAllChildren w variable A variable I True

 markExpressions w nec variable I variable A True
End
```

Since we are interested in marking true formulas only, $\langle I \rangle A$ formulas are easily checked by the following rule:

```
Rule Mark_Pos_Atomic_Program
 hasElement w pos variable I variable A
 isMarkedExpression u variable A True
 isLinked w u variable I

 markExpressions w pos variable I variable A True
End
```

Inheriting marks from loop-parent nodes

Some formulas, especially $\langle X \rangle$- and $\langle X^* \rangle$-formulas, will not be marked at a loop node, since such nodes are blocked and have no further successors. Thus, such

$\langle\rangle$-formulas may not be marked `True` in such nodes, even if they were in their equal nodes. In order to avoid confusion between what is marked in the parent-equal node and what is not, we copy the `True` mark of every formula in a parent node to its equal loop-node using the following rule:

```
Rule Mark_Formulas_In_Loop_Nodes
 hasElement w variable A
 isLinked w u Loop
 isMarkedExpression u variable A True

 markExpressions w variable A True
End
```

With this rule, we come to the end of our model checking rules. These rules should be called repeatedly at the end of the PDL strategy that constructs the premodels.

Interpreting the results of the model checking process

Once the application of the above rules is finished, we are sure that only unfulfilled formulas are not marked as `True` in our open premodels. Hence, we can report inextensible premodels by performing the following simple check:

```
Rule Not_True_Eventuality_Means_Inextensible_Premodel
 hasElement w pos star variable X variable A
 isNotMarkedExpression w pos star variable X variable A True

 mark w Inextensible_Premodel
End
```

If we want that this information becomes displayed in every node of `Inextensible_Premodel` we may use the rules defined at the end of Section 7.1.6 in order to propagate this mark to all the nodes of inextensible premodels.

The result of running this method with the formula $[I^*]P \wedge \langle I^*\rangle P$ changes, and the open `premodel.2.2.2` of Figure 7.12 is reported as an inextensible premodel as shown in Figure 7.13.

However, it is not the case that every open premodel with a loop is an inextensible one. For example, running our method with the formula $[I^*](\langle I^*\rangle P \wedge \langle I\rangle\neg P)$ ends up with (some closed premodels, inextensible open premodels and) an open premodel stopped due to the loop-test but it is still a extensible open premodel as shown in Figure 7.14.

If we remove the complex formulas from this premodel, and we keep only on the atomic formulas, we find that this premodel is in fact the pretty model of Figure 7.15.

Conclusion

In this chapter we handled two logics with transitive closure: LTL and PDL. Their semantics are quite different, but share the same "eventually"-kind of for-

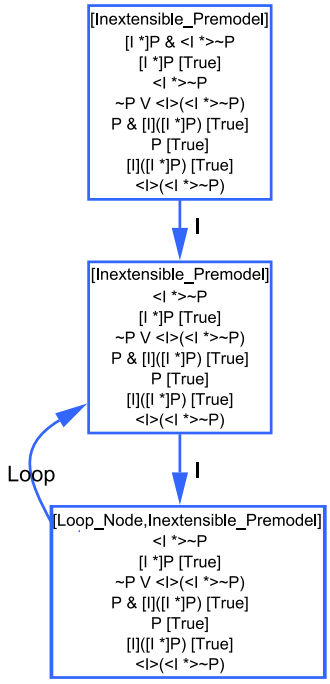


Figure 7.13: Inextensible open premodels are designated due to model checking.

mulas: FA in LTL and $\langle I^* \rangle A$ in PDL. Hence, their model construction methods share the same rule of eventuality "postponement": if A is eventually true then A is true at this step, otherwise at the next step we have to check whether A will be eventually true.

In addition, both methods need a node-equality loop check, otherwise the method may stop before giving the chance to some eventualities to become satisfiable. When the method halts, we need to perform a model checking like procedure in order to check for the fulfillment of all the postponed eventualities.

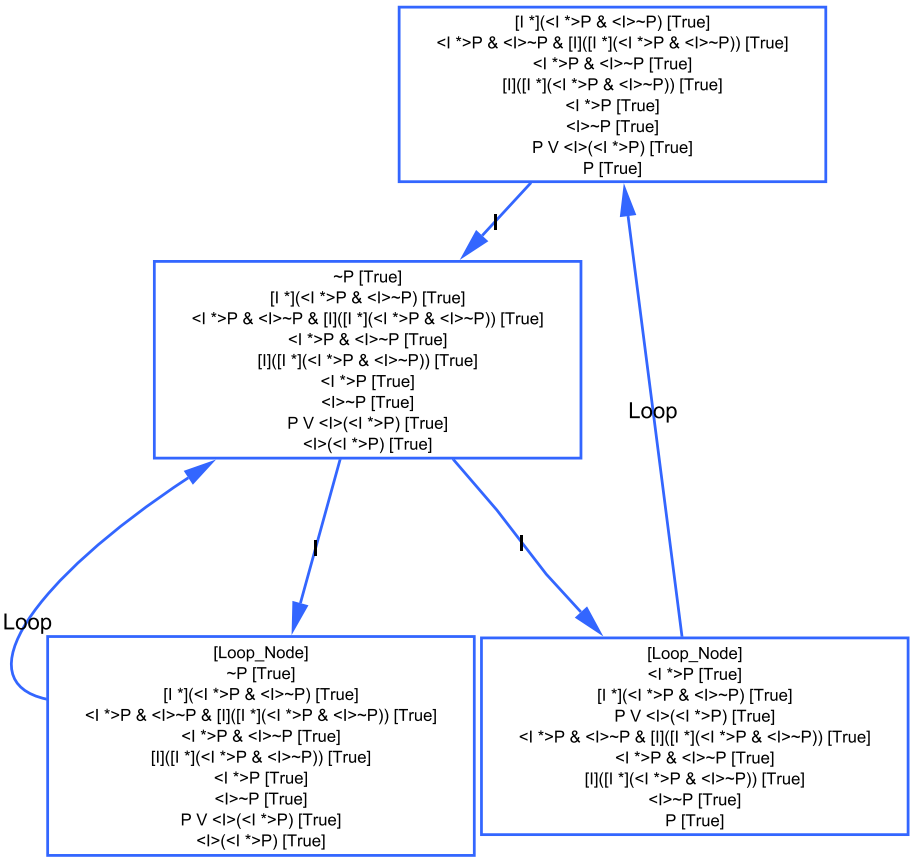


Figure 7.14: Open premodel involving loops could be a good premodel.

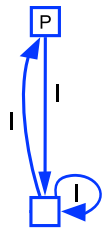


Figure 7.15: The model of the formula $[I^*](\langle I^* \rangle P \wedge \langle I \rangle \neg P)$ which is obtained from the open premodel of Figure 7.14 (Note: the empty world has a $\neg P$).

Chapter 8

Layered Modal Logics

Introduction

Some AI problems, like formalizing interaction of rational agents in the BDI (Belief-Desire-Intention) framework, or like complex ontologies require modal or description logics whose models possess complex properties. A typical example is the property associated with the "no forgetting" axiom ($[I][J]P \to [J][I]P$). Such properties can not be handled by tree-like structures (as it is the case for logics such as K, S4, S5, PDL, ...) Hence, general results about the decidability and complexity of such logics are of high interest, as well as theoretical tools that allow to reason about these logics.

Concerning theoretical tools, standard *filtration* [LS77] is one of them and it is powerful enough to prove decidability - and even finite model property (f.m.p.) - of many standard modal logics (K, K4, K5, ...). However, for complex logics, it often fails to establish decidability and even when it does, it does not give tight upper bounds on the complexity. Further refinements like *selective filtration* [Gab70] permits to state about various logics (e.g. in [Sha04] PSPACE-completeness of many extensions of K4 is proved), and *filtration via bisimulation* was defined and used in [She04] to establish by a complex proof the f.m.p. of a wide class of multi-modal logics (mainly full and weak products) but without stating explicit upper bounds to the satisfiability problem.

Tableaux are a good tool so achieve such a task. In [Lad77], tableaux were used to show that the complexity of S4 is in PSPACE (hardness being proved by a reduction to QBF). In our work here, tableaux are used as a theoretical tool for investigating the complexity of a family of modal logics, more than as a way of designing tractable decision procedures.

Still, there are logics that are not in the scope of the methods mentioned above. For instance, K+Confluence (corresponding to the axiom $\Diamond\Box P \to \Box\Diamond P$), for which standard filtration techniques fail. Tableaux may be easy to design for such logics, but they may be non-terminating and even if not, they may overestimate the complexity.

In this chapter we investigate a class of what we call Layered Modal Logics (LML). Roughly speaking, LML are logics characterized by semantical properties only stating the existence of possible worlds that are in some sense "further" than the others. They can be seen as non-transitive confluent logics, such as star-free PDL with confluent programs for example. Typically, they include various confluence-like properties (mono and multi modal). Such properties are of interest for formalizing the interaction between dynamic and epistemic modalities for rational agents, as mentioned at the beginning of this introduction.

We also address the complexity of LML by means of the tableau method. We use however our own notation for tableaux and tableau method, introduced in Chapter 3 as *premodels* and *model construction*.

It is not surprising that these logics are decidable. However, we show that they are all in NEXPTIME, by marrying model construction with a stepwise filtration-like technique that we call *dynamic filtration*. This operation allows to filtrate the nodes of a premodel one layer at a time, keeping the size of the constructed premodels within an exponential of the length of the input formula. This bound is the best possible for the class of layered logics since one of them is known to be NEXPTIME-complete.

With some adjustments, we show that our method can cope with symmetry and converse, and with permutation. We believe that this technique may be extended to cope with other modal logics that are characterized by some class of frames which are directed acyclic graphs, as we shall discuss at the end of this chapter.

This work is strongly inspired by what has been done in [GS07], except w.r.t. the implementation.

In section 8, we give the necessary backgrounds, in section 8.1, we define layered modal logics, then in section 8.2 we design simple premodels for these logics that we improve in section 8.3 to dynamically filtrated premodels in order to prove the NEXPTIME-membership of the satisfaction problem for layered logics by means of the model construction method. In section 8.4, we give an insight of the implementation of such method in LoTREC. We conclude with some discussion about the extension of the range of dynamic filtration.

Preliminaries

The language of the logics that we study here is the usual multi-modal language defined in Chapter 2, Definition 2. We recall it here in the following definition:

Definition 12 (Language)**.** *The language of a multi-modal logic is defined by the following: let $\mathcal{P}$ be a set of atomic propositional symbols, $\mathcal{I}$ be a set of indexes, and as usual let $\bot$ denotes falsity. The set $\mathcal{F}or$ of formulas (we will only consider* negative normal form *or NNF) is given by the BNF:*

$$A ::= \bot|P|\neg P|(A \wedge A)|(A \vee A)|\langle I\rangle A|[I]A \text{ (where } P \in \mathcal{P} \text{ and } I \in \mathcal{I})$$

and as usual, $(A \to B)$ *abbreviates* $(\neg A \vee B)$*,* $[I]^0 A$ *is* A *and* $[I]^{n+1} A$ *is* $[I][I]^n A$*. Given a formula* A*, we denote by* $|A|$ *the length of* A*.*

Definition 13 (Modal degree)**.** *The modal degree of a formula* A *is denoted by* $d(A)$ *and is inductively defined as usual by:*

- $d(P) = d(\neg P) = d(\bot) = 0$ *(for any atomic proposition* P*),*
- $d([I]A) = d(\langle I \rangle A) = d(A) + 1$*,*
- $d(A \wedge B) = d(A \vee B) = \max(d(A), d(B))$*.*

  The modal degree of a finite set Γ *of formulas is denoted by* $d(\Gamma)$ *and is equal to* $\max_{A \in \Gamma}(d(A))$*.*

Definition 14 (Relations)**.** *Given a relation* R_I *over a set* W*, we denote by* ${R_I}^*$ *its reflexive and transitive closure, by* ${R_I}^+$ *its transitive closure, by* ${R_I}^-$ *its converse (i.e.* $(w, u) \in {R_I}^-$ *iff* $(u, w) \in R_I$*), and by* $(R_I \cup {R_I}^-)$ *the symmetric closure of* R_I*. Finally, given a family (i.e. a set) of relations* $R = \{R_I : I \in \mathcal{I}\}$*, we will also denote by* R *the relation consisting of the union of the relations of* R*, i.e.* $R = (\bigcup_{I \in \mathcal{I}} R_I)$*.*

Remark 13. We will also use the fact that a connected graph without isolated points (see below) may be represented by the set of its edges, i.e. by a conjunction of literals of the form $R_I(w, u)$.

In the following definition, we recall the definition of frame, model and satisfaction, which have been (partially) defined in Chapter 2:

Definition 15 (Semantics: frames, models and satisfaction)**.**

- *(Multi-relational Kripke)* frames *are graphs*[1] (W, R)*, where* R *is a family of binary relations indexed by* $\mathcal{I}$*, and with a root* w_0*, such that: any* $w \in W$ *is accessible from* w_0 *via* $(R \cup R^-)^*$*,*
- *(Kripke)* models *are pairs* (F, V) *where* F *is a frame and* V *is a valuation function (*$V : W \longrightarrow 2^{\mathcal{P}}$*), such a model is said to be based on* F*.*
- Pointed models *are pairs* M, w *where* M *is a model* (W, R, V) *and* $w \in W$*.*
- *That a formula* A *is* satisfied *by some pointed model (in symbols* $M, w \Vdash A$*) is defined recursively as follows (we only give the clauses concerning modal connectors):*
  - $M, w \Vdash \langle I \rangle A$ *iff there exists* $u \in W$ *such that* $R_I(w, u)$ *and* $M, u \Vdash A$*;*
  - $M, w \Vdash [I]A$ *iff for every* $u \in W$*, if* $R_I(w, u)$ *then* $M, u \Vdash A$*;*

[1] More precisely, Rooted Directed Acyclic Graphs (RDAGs).

Definition 16 (Frame formula)**.** *A* frame formula $\Phi(x_1, \ldots, x_n)$ *is a first-order formula (the x_i's are its free variables) which is a conjunction of literals of the form $R_I(x, y)$ (with $I \in \mathcal{I}$), or equivalently a finite set of such literals.*

Now we recall the definition of the satisfiability problem (which has been already defined in Chapter 2):

Definition 17 (Satisfiability Problem)**.** *The satisfiability problem w.r.t. a class C of frames: given a formula A does there exist some pointed model M, w based on some frame of C and such that $M, w \Vdash A$? This problem is referred to as C-satisfiability problem and the set of C-satisfiable formulas will be denoted by $\mathsf{Sat}(\mathsf{C})$.*

Definition 18 (Vector notation)**.** *For the sake of brevity, we introduce here a vector notation which is as follows: (given the variables $x_1, \ldots, x_n$, the variable x, the functions $H_1, \ldots, H_m$ and the function H)*

- *The sequence $(x_1, \ldots, x_n)$ will be abbreviated by $\overrightarrow{x}$, with $Card(\overrightarrow{x}) = n$;*
- *$H(x_1, \ldots, x_n)$ will be abbreviated by $H(\overrightarrow{x})$;*
- *the sequence $(H(x_1), \ldots, H(x_n))$ will be abbreviated by $H.\overrightarrow{x}$; (note the dot)*
- *the sequence $(H_1(x), \ldots, H_m(x))$ will be abbreviated by $\overrightarrow{H}.x$;*
- *and the sequence $(H_1(\overrightarrow{x}), \ldots, H_m(\overrightarrow{x}))$ will be abbreviated by $\overrightarrow{H}.\overrightarrow{x}$.*

Definition 19 (Subframe/subgraph)**.** *Given a frame $F = (W, R)$, a first-order formula $\Phi(\overrightarrow{x})$ whose free variables are $x_1, \ldots, x_n$ and given an assignment $\sigma\colon \{x_1, \ldots, x_n\} \mapsto W$, we consider that $\sigma(F)$ denotes the subframe $\mathbf{f} = (\mathbf{w}, \mathbf{r})$ of F where $\mathbf{w} = \{\sigma(x_1), \ldots, \sigma(x_n)\}$ and $\mathbf{r}_I = (R_I)_{|\mathbf{w}}$ (i.e. the restriction of R_I to $\mathbf{w}$). We say that the subframe $\mathbf{f}$ satisfies Φ (in symbols $\mathbf{f} \models \Phi(\sigma.\overrightarrow{x})$) iff $\Phi(\sigma(x_1), \ldots, \sigma(x_n))$ is true in F.*

Definition 20 (Depth)**.** *Given a frame $F = (W, R)$ with root w_0, and represented by a set of literals $R_I(w, u)$, i.e. by a frame formula $\Phi(\overrightarrow{x})$ whose variable are assigned to elements of W by an assignment σ, given $w \in W$, we define the* depth *of w in F (or in Φ), denoted by $\delta_F(w)$ (or $\delta_\Phi(w)$), as the length of the shortest path from w_0 to w. Inductively:*

- $\delta_F(w_0) = 0$;
- $\delta_F(w) = \min_{R_I(u,w) \in R}(\delta_F(u) + 1)$

8.1 Layered modal logics

In this section, we define the class of logics that we intend to explore.

8.1.1 Layer formulas

First, we formalize the properties characterizing the class of frames of these logics in what we call *layer formulas*.

Definition 21 (layer formula)**.** *A* layer *formula is a first-order formula of the form:*

$$\forall \overrightarrow{x} : \exists \overrightarrow{y} : \phi(\overrightarrow{x}) \rightarrow \psi(\overrightarrow{x}, \overrightarrow{y})$$

where ϕ and ψ are frame formulas and with the constraints given below:

(i) $\phi(\overrightarrow{x})$ is a conjunction (that we will identify with a set) of literals $R_I(x_i, x_j)$ (where $x_i, x_j \in \overrightarrow{x}$ and $I \in \mathcal{I}$);

(ii) $\psi(\overrightarrow{x}, \overrightarrow{y})$ is a conjunction of literals $R_I(x_i, y_j)$ or $R_I(y_j, y_k)$ (where $x_i \in \overrightarrow{x}$, $y_j, y_k \in \overrightarrow{y}$, $j < k$ and $I \in \mathcal{I}$);

(iii) $\forall y_j \in \overrightarrow{y} : \exists x_i \in \overrightarrow{x} : \exists I \in \mathcal{I} : (R_I(x_i, y_j))$ is a conjunct of ψ; we exclude properties stating the existence of isolated nodes;

(iv) $\delta_{\phi \wedge \psi}(y_k) > \delta_{\phi \wedge \psi}(x_j)$ for all $y_k \in \overrightarrow{y}$ and all $x_j \in \overrightarrow{x}$: the depth of existential nodes will always be strictly greater than that of their parent nodes, hence, their modal degree will be strictly smaller.

Notice that such a formula may be seen as a rule which rewrites a graph by adding nodes and edges to it, ϕ describing the left-hand side of the rule (the subgraph to be rewritten) while $\phi \wedge \psi$ describes the right-hand side (the result of the rewriting) (see Example 7).

Examples of layer formulas are:

Example 6. Seriality $\forall x : \exists y R_I(x, y)$.

Example 7. Bimodal confluence (Figure 8.1):

$$\forall x, y, z : \exists u : (R_I(x, y) \wedge R_J(x, z)) \rightarrow (R_J(y, u) \wedge R_I(z, u))$$

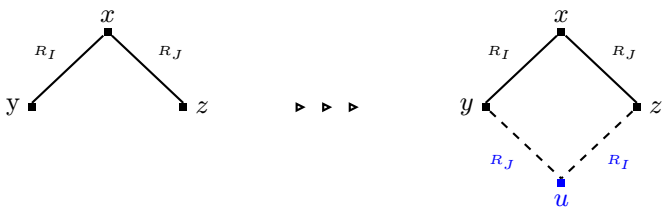


Figure 8.1: Bimodal confluence property viewed as a layer formula, i.e. as a *rewriting rule* describing a graph rewriting step.

8.1.2 Layered frames

In this subsection, we define the *layered frames*, and hence, the class of frames characterizing layered logics.

Definition 22 (layered frame)**.** *An LF-*layered *frame $F = (W, R)$ is a finite frame (of root w_0) which verifies a finite set LF of* layer formulas*. We denote by C_{LF} the class of all LF-layered frames.*

Figure 8.2 shows a graphical representation of a layered frame.

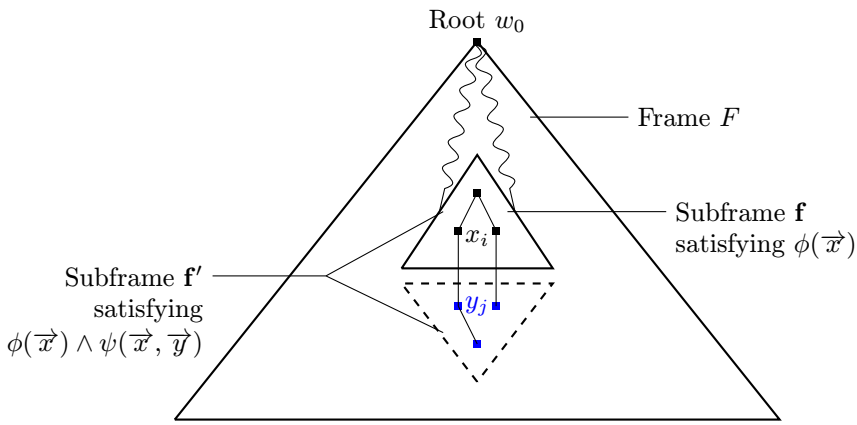


Figure 8.2: A graphical representation of how a layer formula describes certain properties of (or, constraints on) a layered frame.

Definition 23 (skolemized layer formula)**.** *A skolemized layer formula is the result of skolemization of a layered formula $\chi = \forall \overrightarrow{x} : \exists \overrightarrow{y} : \phi(\overrightarrow{x}) \rightarrow \psi(\overrightarrow{x}, \overrightarrow{y})$, denoted as $skol(\chi)$, and defined as:*

$$skol(\chi) = \forall \overrightarrow{x} : \phi(\overrightarrow{x}) \rightarrow \psi(\overrightarrow{x}, \overrightarrow{H}.\overrightarrow{x})$$

with $\overrightarrow{H}.\overrightarrow{x} = (H_1(\overrightarrow{x}), \ldots, H_k(\overrightarrow{x}))$ (and $k = Card(\overrightarrow{y})$).

Since ψ is a conjunction, $skol(\chi)$ is equivalent to the conjunction of formulas of the form:

(i) either $\forall \overrightarrow{x} : \phi(\overrightarrow{x}) \rightarrow R_I(x_i, H_j(\overrightarrow{x}))$

(ii) or $\forall \overrightarrow{x} : \phi(\overrightarrow{x}) \rightarrow R_I(H_i(\overrightarrow{x}), H_j(\overrightarrow{x}))$.

Henceforth, we consider skolemized formulas as being of one of the forms (i) and (ii). By extension, the set of skolemized layered formulas of type (i) and type (ii), and corresponding to a set LF of layered formulas, is denoted by SLF. The corresponding class of frames is denoted by C_{SLF}.

In the sequel, we shall see that $\Diamond$-formulas are also treated by the introduction of Skolem functions.

Example 8. The skolemized layered formula for Seriality is $\forall x\colon R_I(x, H(x))$.

Example 9. The skolemized layered formulas for Bimodal Confluence are:

- $\forall x, y, z\colon R_I(x, y) \wedge R_J(x, z) \to R_J(y, H(x, y, z))$, and
- $\forall x, y, z\colon R_I(x, y) \wedge R_J(x, z) \to R_J(z, H(x, y, z))$.

Remark 14. It is well-known that $A \in \mathsf{Sat}(\mathsf{C}_{LF})$ if, and only if, $A \in \mathsf{Sat}(\mathsf{C}_{SLF})$, since skolemization preserves the satisfiability.

Definition 24 (layered modal logics)**.** Layered *modal logics (LML) are those characterized by a class C_{SLF} of frames, i.e. by the class of layered frames which verify a set SLF of skolemized layer formulas of type (i) and (ii).*

It is a natural question to ask whether a given logic is a layered logic or not (note that layered logics are semantically defined). For example, we know that modal logic K+Confluence is characterized by the class of confluent frames. Since the first-order formula expressing confluence is a layer formula, K+Confluence is clearly a layered logic. However, in general, given a set of properties, checking if they are equivalent with some set of layer formulas is likely to be undecidable, although we did not formally prove it.

The main result of the next sections is that the C_{SLF}-satisfiability problem for any SLF-layered logic is in NEXPTIME, whence decidable. More precisely, it is decidable by a non-deterministic Turing machine in time $O(2^{c.|A|})$ where c is a constant.

8.2 Construction of simple premodels for LML

The definition given here is slightly different than Definition 6, page 60.

Definition 25 (partial premodel)**.** *A* partial premodel *is a labeled graph* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *where:*

- $\mathtt{W}$ *is a finite set of nodes;*
- $\mathtt{R}\colon \mathcal{I} \to 2^{\mathtt{W}\times\mathtt{W}}$ *is a family of binary relations over* $\mathtt{W}$ *and indexed by* $\mathcal{I}$ *(hence* $(\mathtt{W}, \mathtt{R})$ *is a finite frame);*
- *with a root* $\mathtt{w}_0 \in \mathtt{W}$*, such that: any* $\mathtt{w} \in \mathtt{W}$ *is accessible from* $\mathtt{w}_0$ *via* $(\mathtt{R} \cup \mathtt{R}^-)^*$*;*
- $\mathtt{V}\colon \mathtt{W} \to 2^{\mathcal{F}or}$ *is a function which maps each element of* $\mathtt{W}$ *to some set of formulas.*

Henceforth, for a node $\mathtt{w} \in \mathtt{W}$, we abbreviate $d(\mathtt{V}(\mathtt{w}))$ in $d(\mathtt{w})$. We may also represent a function $\mathtt{V}\colon \mathtt{W} \to 2^{\mathcal{F}or}$ by its extension, i.e. by the set $\{(\mathtt{w}, A) | A \in \mathtt{V}(\mathtt{w})\}$.

Definition 26. *Given two such functions* $\mathtt{V}_1$ *and* $\mathtt{V}_2$*, we define* $\mathtt{V}_1 \cup \mathtt{V}_2$ *by:* $\mathtt{V}_1 \cup \mathtt{V}_2(\mathtt{w}) =$

- $\mathtt{V}_1(\mathtt{w}) \cup \mathtt{V}_2(\mathtt{w})$ *if* $\mathtt{w} \in \mathrm{dom}(\mathtt{V}_1) \cap \mathrm{dom}(\mathtt{V}_2)$*;*
- $\mathtt{V}_1(\mathtt{w})$ *if* $\mathtt{w} \in \mathrm{dom}(\mathtt{V}_1)$ *and* $\mathtt{w} \notin \mathrm{dom}(\mathtt{V}_2)$*;*
- $\mathtt{V}_2(\mathtt{w})$ *if* $\mathtt{w} \in \mathrm{dom}(\mathtt{V}_2)$ *and* $\mathtt{w} \notin \mathrm{dom}(\mathtt{V}_1)$*.*

Definition 27 (Skolemizing $\Diamond$)**.** *Let* $(\mathtt{W}, \mathtt{R}, \mathtt{V})$ *be a partial premodel for the input formula* A*, and let* $Sub(A)$ *denote the set of subformulas of* A *(Definition 3). For each triple* $(I, \mathtt{w}, B) \in \mathcal{I} \times \mathtt{W} \times Sub(A)$*, we associate a Skolem term* $\langle I \rangle(B, \mathtt{w})$ *(intuitively this term will denote one world accessible from* $\mathtt{w}$ *and making* B *true, thus making* $\langle I \rangle B$ *true at* $\mathtt{w}$*).*

8.2.1 The set of rules

As already stated in Section 3.2, a rule defines how to rewrite a premodel by specifying which graph elements (nodes, edges and formulas) are to be added to it. A rule may be seen then as a function ρ applied to a partial premodel $\mathtt{M}$ and computing what new elements are to be added to $\mathtt{M}$.

Definition 28 (Rule application)**.** *Let* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *be a partial premodel, and let* ρ *be a rule, to denote that* $\mathtt{M}'$ *is obtained from* $\mathtt{M}$ *by applying rule* ρ*, we write:*

$$\mathtt{M}' = \rho(\mathtt{M}) = \mathtt{M} \cup \nu_\rho(\mathtt{M})$$

where $\nu_\rho(\mathtt{M}) = (\mathbf{w}, \mathbf{r}, \mathbf{v})$ *denotes respectively the sets of new nodes, new* I*-edges (for each* $I \in \mathcal{I}$*) and new pairs node-formula that are to be added to* $\mathtt{M}$ *in order to obtain* $\mathtt{M}'$*.*

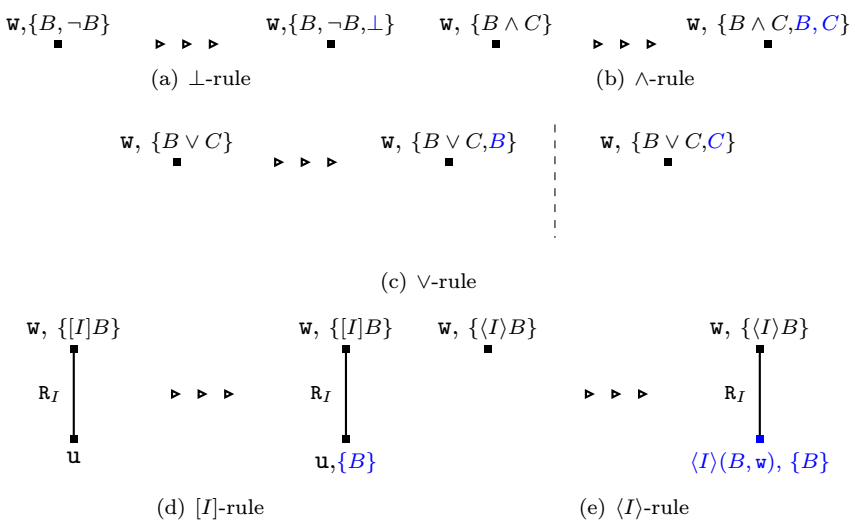
Definition 29 (Set of rules)**.** *For each rule* ρ*, we indicate the result of its application on a premodel* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *as a triple* $\nu_\rho(\mathtt{M}) = (\mathbf{w}, \mathbf{r}, \mathbf{v})$*. Variables* $\mathtt{w}$ *and* $\mathtt{u}$ *are implicitly universally quantified over* $\mathtt{W}$*. Moreover, we use both notations* $B \in \mathtt{V}(\mathtt{w})$ *and* $(\mathtt{w}, B) \in \mathtt{V}$ *without distinction.*

- $\nu_\bot(\mathtt{M}) = \Big(\emptyset, \emptyset, \{(\mathtt{w}, \bot)\}\Big)$ *for all* $B, \neg B \in \mathtt{V}(\mathtt{w})$*,*
- $\nu_\wedge(\mathtt{M}) = \Big(\emptyset, \emptyset, \{(\mathtt{w}, B), (\mathtt{w}, C)\}\Big)$ *for all* $(B \wedge C) \in \mathtt{V}(\mathtt{w})$*,*
- $\nu_\vee(\mathtt{M}) = \Big(\emptyset, \emptyset, \{(\mathtt{w}, D_{\mathtt{w}, B \vee C})\}\Big)$ *for all* $(B \vee C) \in \mathtt{V}(\mathtt{w})$
  if $B, C \notin \mathtt{V}(\mathtt{w})$ *then* $D_{\mathtt{w}, B \vee C}$ *is non-deterministically chosen among* B *and* C*,*
  else $D_{\mathtt{w}, B \vee C}$ *is any of* B *and* C *which is already in* $\mathtt{V}(\mathtt{w})$*,*
- $\nu_{[I]}(\mathtt{M}) = \Big(\emptyset, \emptyset, \{(\mathtt{u}, B)\}\Big)$ *for each* $I \in \mathcal{I}$*, for all* $\mathtt{w}, \mathtt{u}$ *such that* $\mathtt{R}_I(\mathtt{w}, \mathtt{u})$ *and* $[I]B \in \mathtt{V}(\mathtt{w})$*,*

- $\nu_{\langle I\rangle}(\mathtt{M}) = \Big(\{\langle I\rangle(B,\mathtt{w})\}, \{(\mathtt{w}, \langle I\rangle(B,\mathtt{w}))\}, \{(\langle I\rangle(B,\mathtt{w}), B)\}\Big)$ *for each* $I \in \mathcal{I}$ *and for all* $\langle I\rangle B \in \mathtt{V}(\mathtt{w})$*; Note that:* $\delta(\langle I\rangle(B,\mathtt{w})) > \delta(\mathtt{w})$,
- *If the formula* $\chi = \forall\overrightarrow{x}\colon \phi(\overrightarrow{x}) \to R_I(x_i, H_j(\overrightarrow{x}))$ *is in SLF, and if for some assignment* σ *of* $\overrightarrow{x}$ *over* $\mathtt{W}$ *we have* $(\mathtt{W},\mathtt{R}) \models \phi(\sigma.\overrightarrow{x})$*, then*

  if $\exists x_k \in \overrightarrow{x}\colon d(\mathtt{V}(\sigma(x_k))) > 0$ [2]
  then $\nu_\chi(\mathtt{M}) = \Big(\{H_j(\sigma.\overrightarrow{x})\}, \{(\sigma(x_i), H_j(\sigma.\overrightarrow{x}))\}, \emptyset\Big)$
  NB: $\delta(H_j(\sigma.\overrightarrow{x})) > \delta(\sigma(x_k))$*;*
  else $= \Big(\emptyset, \{(\sigma(x_l), \sigma(x_m))\}, \emptyset\Big)$ *for each* $x_l, x_m \in \overrightarrow{x}$*;*
- *If the formula* $\chi = \forall\overrightarrow{x}\colon \phi(\overrightarrow{x}) \to R_I(H_i(\overrightarrow{x}), H_j(\overrightarrow{x}))$ *is in SLF, and if for some assignment* σ *of* $\overrightarrow{x}$ *over* $\mathtt{W}$ *we have* $(\mathtt{W},\mathtt{R}) \models \phi(\sigma.\overrightarrow{x})$*, then*

  if $\exists x_k \in \overrightarrow{x}\colon d(\mathtt{V}(\sigma(x_k))) > 0$
  then $\nu_\chi(\mathtt{M}) = \Big(\{H_i(\sigma.\overrightarrow{x}), H_j(\sigma.\overrightarrow{x})\}, \{(H_i(\sigma.\overrightarrow{x}), H_j(\sigma.\overrightarrow{x}))\}, \emptyset\Big)$
  NB: $\delta(H_j(\sigma.\overrightarrow{x})) > \delta(H_i(\sigma.\overrightarrow{x})) > \delta(x_k)$*;*
  else $= \Big(\emptyset, \{(\sigma(x_l), \sigma(x_m))\}, \emptyset\Big)$ *for each* $x_l, x_m \in \overrightarrow{x}$.

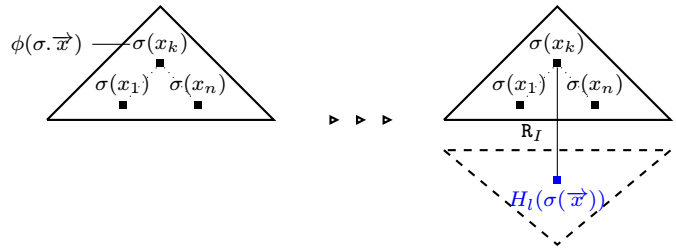


(a) ⊥-rule (b) ∧-rule (c) ∨-rule (d) [I]-rule (e) ⟨I⟩-rule

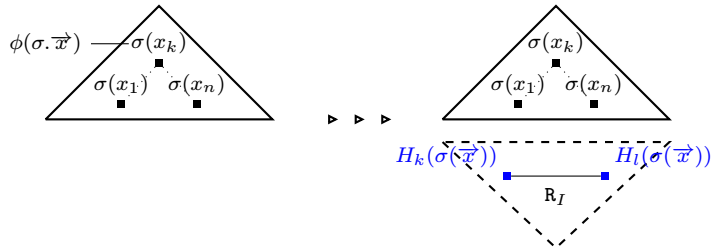
Figure 8.3: Graphical representation of classical and modal rules.

Figures 8.3 and 8.4 present the above rules graphically.

[2]This condition stops the computation when nodes only contains non modal formulas.



(a) rule for type(i) SLF-formulas



(b) rule for type(ii) SLF-formulas

Figure 8.4: Graphical representation of rules for layered formulas.

Remark 15.

- If A is the input formula, $\forall \mathtt{w} \in \mathtt{W}\colon \mathtt{V}(\mathtt{w}) \subseteq Sub(A)$.
- Since the set SLF is finite, so are the sets of added nodes $\mathbf{w}$ of each rule.
- In the last two "else" parts, no new nodes are added ($\mathbf{w}$ is empty).

8.2.2 Soundness and completeness

A direct examination of the rules above shows that all rules ρ except the $\langle I\rangle$-rule and SLF-rules are terminating on partial (and hence finite) premodels, i.e. there exists an integer n such that $\rho^n(\mathtt{M}) = \rho^{n+1}(\mathtt{M})$ (where $\rho^{n+1}(\mathtt{M}) = \rho(\rho^n(\mathtt{M}))$), and we denote by ρ^* the iteration of ρ up to the least fixed point in these cases. Note that we use $(\rho \circ \rho')(\mathtt{M})$ to denote $\rho(\rho'(\mathtt{M}))$.

Definition 30 (Meta-rules)**.** *Let us define the following meta-rules called* Sat$\Box$, $\Diamond$ *and* Slf*:*

- $\nu_{\mathrm{Sat}\Box} = (\nu_\bot \cup \nu_\wedge \cup \nu_\vee \cup (\bigcup_{I\in\mathcal{I}} \nu_{[I]}))^*$ *(classical saturation and* $[I]$ *propagation: terminates since frames and formulas are of finite size),*
- $\nu_\Diamond = (\bigcup_{I\in\mathcal{I}} \nu_{\langle I\rangle})$ *(without star!),*

- $\nu_{\mathrm{Slf}} = (\bigcup_{\chi \in SLF} \nu_\chi)$.

Definition 31 ((partial) simple premodel)**.** *A simple premodel* $\mathtt{M}$ *for an input formula* A *is a* least fixed point *of a sequence* $\mathtt{M}_0 = (\mathtt{W}_0, \mathtt{R}_0, \mathtt{V}_0)$, $\mathtt{M}_1, \ldots$[3] *where*[4]*:*

- $\mathtt{W}_0 = \{\mathtt{w}_0\}$ *(the root),* $\mathtt{R}_0 = \emptyset$ *and* $\mathtt{V}_0 = \{(\mathtt{w}_0, A)\}$*;*
- $\mathtt{M}_{i+1} = \mathrm{Sat}\Box(\mathrm{Slf}(\Diamond(\mathtt{M}_i)))$*, where* $\mathrm{Sat}\Box$, Slf *and* $\Diamond$ *are the meta-rules given above.*

We have then $\mathtt{M} = (\mathrm{Sat}\Box \circ \mathrm{Slf} \circ \Diamond)^*(\mathtt{M}_0)$*. Any other premodel in this sequence is a* partial *simple premodel.*

Figure 8.5 gives a screen shot of a one-step construction of a simple premodel.

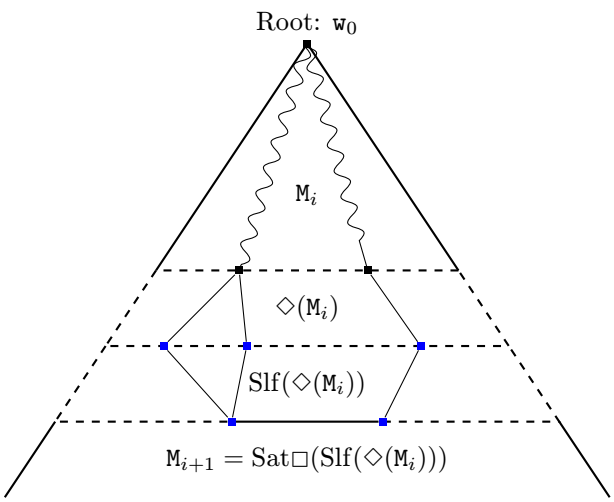


Figure 8.5: During the construction of simple premodels, new added nodes belong to layers of higher depth.

Definition 32 (closed and open simple premodel)**.** *A simple premodel is* closed *if some node in it contains* $\bot$*; it is* open *otherwise.*

Definition 33 (First occurrence)**.** *Given a (possibly partial) simple premodel* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *(Definition 31), and a node* $\mathtt{w} \in \mathtt{W}$*, the* first occurrence *of* $\mathtt{w}$ *in* $\mathtt{W}$ *is the index of introduction of* $\mathtt{w}$ *in* $\mathtt{W}$*:* $\mathbf{fst}(\mathtt{w}) = \min_j(\mathtt{w} \in \mathtt{W}_j))$*. Hence* $\mathtt{w} \in \mathtt{W}_i$ *iff* $i \geq \mathbf{fst}(\mathtt{w})$*.*

[3]We will prove later that this sequence is finite, hence it has a fixed point.
[4]Due to rule $\vee$, the are several such sequences

Proposition 1. *Let A be an input formula ($d(A)$ being its modal degree), and let $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ be a (possibly partial) simple premodel from a sequence $\mathtt{M}_0, \ldots, \mathtt{M}_n, \ldots$ with root $\mathtt{w}_0$. Then we have the following:*

1. *For all $j > i \geq \mathbf{fst}(y)$ we have $\delta_{\mathtt{M}_i}(y) = \delta_{\mathtt{M}_j}(y)$: once set, the depth of a node do not change at further iteration, we will then denote it by $\delta(y)$;*

2. *for all $j > i \geq \mathbf{fst}(y)$ we have $d_{\mathtt{M}_i}(y) = d_{\mathtt{M}_j}(y)$: idem to 1. but for the modal degree of nodes;*

3. *if $\delta(x) = \min_{x_k \in \overrightarrow{x}}(\delta(x_k))$ and $y = H(\overrightarrow{x})$, or $y = \langle I \rangle(B, x)$ then $d(y) \leq d(x) - 1$: new added nodes are of smaller modal degrees;*

4. *for all y we have $\mathbf{fst}(y) \leq \delta(y)$;*

5. *for all y we have $0 \leq d(y) + \delta(y) \leq d(A)$ and as a consequence $\mathbf{fst}(y) \leq d(A)$: the algorithm stops after at most $d(A)$ iterations.*

Proof. 1. Since shortest path cannot shrink due to constraints on rules.

2. Since classical saturation and propagation of boxed formulas are performed at each iteration, and since nodes introduced later are of strictly greater depth.

3. Since rules that add formulas to new nodes ($\Diamond$ and $\Box$ rules) strictly decrease the modal degree of formulas; thus along a path $(\mathtt{w}_0, y)$ of length p, a node y will only receive formulas of degree at most $d(A) - p$. Hence, formulas of highest degree in y will come from a shortest path, i.e. from x.

4. By induction on $\mathbf{fst}(y)$: If $\mathbf{fst}(y) = 0$ then $y = r$ and $\delta(r) = 0$; if $\mathbf{fst}(y) = p+1$ then there exist $x_1, \ldots, x_n \in \mathtt{W}_p$ and $x_k \notin \mathtt{W}_{p-1}$ (otherwise y would be in $\mathtt{W}_p$) with $y = H(\overrightarrow{x})$. Then since $\delta(y) > \delta(x_k)$ and $\delta(x_k) \geq \mathbf{fst}(x_k) = p$ (by IH) hence $\delta(y) \geq p + 1$. The same holds if we consider $y = \langle I \rangle(B, x)$ with $x \in \mathtt{W}_p$ (and $x \notin \mathtt{W}_{p-1}$).

5. Again by induction on $\mathbf{fst}(y)$. If $\mathbf{fst}(\mathtt{w}_0) = 0$ then $y = \mathtt{w}_0$, $d(\mathtt{w}_0) = d(A)$ and $\delta(\mathtt{w}_0) = 0$: done. Suppose $\mathbf{fst}(y) = p + 1$ then we must have either $y = H(\overrightarrow{x})$ for some $x_1, \ldots, x_n \in \mathtt{W}_p$, or $y = \langle I \rangle(B, x)$ for some $x \in \mathtt{W}_p$. For the first case, let $\delta(x) = \min_{x_k \in \overrightarrow{x}}(\delta(x_k))$, rules ensure that $\delta(y) = \delta(x)+1$ and since $d(y) \leq d(x) - 1$ we have $d(y) + \delta(y) \leq d(x) + \delta(x) \leq d(A)$ (by IH). For the case where $y = \langle I \rangle(B, x)$ the same proof holds.

□

Lemma 1 (Completeness and soundness of simple premodels)**.** *Let A be a formula and SLF a set of layered formulas, then A is C_{SLF}-satisfiable if, and only if, there exists an open simple premodel $\mathtt{M}$ (as in Definition 31, with* Slf *being the set of rules corresponding to the set SLF).*

Proof. Completeness is immediate by comparison with naive premodels (those with purely non-deterministic but fair strategy where all rules are applied only *non-deterministically* but *eventually at some iteration*) which are trivially complete since they reduce to model construction. Now, it can be checked that the strategy $\mathrm{Sat}\Box(\mathrm{Slf}(\Diamond))^*$ is fair hence if some simple premodel is open then so is some naive one and completeness follows from that of naive tableaux.
Soundness is easily proved by induction on iterations and with the help of proposition 1. It remains to prove that frame of an open simple premodel is an SLF-frame: we have to prove that $(\mathtt{W}, \mathtt{R}) \models SLF$.

Let $\chi = \forall \overrightarrow{x} : \phi(\overrightarrow{x}) \to R_I(x_i, H_j(\overrightarrow{x}))$ be in SLF, and suppose that for some assignment σ over $\mathtt{W}$, we have $(\mathtt{W}, \mathtt{R}) \models \phi(\sigma.\overrightarrow{x})$ and let $m = \max_{x_k \in \overrightarrow{x}}(\mathbf{fst}(\sigma(x_k)))$ (m is the first iteration at which the all elements of $\sigma.\overrightarrow{x}$ have been introduced to $\mathtt{W}$). There are two cases:

(a) If $\exists x_k \in \overrightarrow{x} : \sigma(x_k) \in \mathtt{W}_m$ and $d(\mathtt{V}_m(\sigma(x_k))) > 0$ then $(\sigma(x_i), H_j(\sigma.\overrightarrow{x})) \in (\mathtt{R}_I)_{m+1}$ by then-part of rule ν_χ, hence $(\sigma(x_i), H_j(\sigma.\overrightarrow{x})) \in \mathtt{R}_I$;

(b) If $\forall x_k \in \overrightarrow{x} : \sigma(x_k) \in \mathtt{W}_m$ but $d(\mathtt{V}_m(\sigma(x_k))) = 0$
then we are done since $(\sigma(x_i), \sigma(x_i)) \in \mathtt{R}_I$ by else-part of the same rule.

A similar reasoning holds in the case of SLF formulas of type (ii) since:
in case (a) $(H_i(\sigma.\overrightarrow{x}), H_j(\sigma.\overrightarrow{x})) \in \mathtt{R}_I$ by then-part,
and in case (b) $(\sigma(x_i), \sigma(x_j)) \in \mathtt{R}_I$ by else-part. □

Lemma 2. *Layered modal logics are decidable by Proposition 1 and by Lemma 1 of completeness and soundness.*

Proof. Since only finitely many nodes are added at each iteration and since there are at most $d(A)$ iterations. □

At least one layered modal logic (e.g. K+confluence, see [Gas06]) is known to have an NEXPTIME-hard satisfiability problem. Hence the best we can do for this class of logics is to prove their membership in NEXPTIME. In order to do so, we develop and apply, in the sequel, a premodel construction method with dynamic filtration.

8.3 Dynamically filtrated premodels

When applied on a partial premodel $\mathtt{M}$, rules of Definition 29 add new nodes (set $\mathbf{w}$) and new edges (set $\mathbf{r}$) in order to fulfill the semantics of $\Diamond$ connectors and existential properties stated by SLF formulas.

However, some of these existential properties may lead to an explosion in the number of nodes added at each iteration, i.e. in the size of the resulting premodels. For instance, a property such as: $\forall x, y, z : \exists u : x{R_I}^* y \wedge x{R_I}^* z \to yR_I u \wedge zR_I u$ may lead to premodels of double exponential in the size of the input formula. Knowing that the number of different subsets of $Sub(A)$ is bound by $2^{|A|}$, such huge premodels would have multiple copies of the same node (i.e. of its associated set of formulas).

Hence, for proving our complexity result, we need to make sure that each added node is *unique*. This is why, we identify, at each iteration, every new node when it is equivalent to (i.e. labeled by the same set of formulas labeling) another new node. Note that we do not identify new nodes with old ones. This layer-by-layer node identification justify the name of *dynamic filtration*.

8.3.1 The dynamic filtration

We define the dynamic filtration as a rule-like operation that we call Df. This operation is applied at each iteration after the rules $\Diamond$, Slf and Sat$\Box$ to identify equivalent new nodes. Thus, at first, we define this equivalence relation, then we define our filtration by defining the effect of its application on a simple premodel.

Definition 34 (dynamic filtration)**.** *Let* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *be a partial simple premodel, let* $\nu_{\mathrm{Sat}\Box\circ\mathrm{Slf}\circ\Diamond} = (\mathbf{w}, \mathbf{r}, \mathbf{v})$*, so that* $\mathrm{Sat}\Box(\mathrm{Slf}(\Diamond(\mathtt{M}))) = (\mathtt{W}\cup\mathbf{w}, \mathtt{R}\cup\mathbf{r}, \mathtt{V}\cup\mathbf{v})$*. Let* $\mathtt{M}' = (\mathtt{W}', \mathtt{R}', \mathtt{V}') = \mathrm{Df}(\mathrm{Sat}\Box(\mathrm{Slf}(\Diamond(\mathtt{M}))))$*, where* Df *is the* dynamic filtration *operation defined w.r.t. to the following total equivalence relation over* $\mathtt{W}\cup\mathbf{w}$*:*

- $\mathtt{w} \equiv \mathtt{u}$ *if, and only if:*
  - $\mathtt{V}(\mathtt{w}) = \mathtt{V}(\mathtt{u})$ *for* $\mathtt{w}, \mathtt{u} \in \mathbf{w}$*,*
  - $\mathtt{w} = \mathtt{u}$ *for* $\mathtt{w} \in \mathtt{W}$ *(just to make* $\equiv$ *total over* $\mathtt{W}\cup\mathbf{w}$*)*
- $\bar{\mathtt{w}}$ *denotes the equivalence class of* $\mathtt{w}$*,*
- $\bar{\mathbf{w}} = \{\bar{\mathtt{w}} : \mathtt{w} \in \mathbf{w}\}$*.*

Then we define the dynamically filtrated model $\mathtt{M}'$ *as:*

- $\mathtt{W}' = \mathtt{W} \cup \bar{\mathbf{w}}$*;*
- $\mathtt{R}' = \mathtt{R} \cup (\equiv \circ\ \mathbf{r}\ \circ \equiv)$[5]*, i.e.* $\mathtt{R} \subseteq \mathtt{R}'$*, and for* $(\mathtt{w}, \mathtt{u}) \in \mathbf{r}$*,* $(\bar{\mathtt{w}}, \bar{\mathtt{u}}) \in \mathtt{R}'$*;*
- $\mathtt{V}' = \mathtt{V} \cup (\mathbf{v})_{|\bar{\mathbf{w}}}$*.*

Definition 35 (filtrated premodel)**.** *A* filtrated *premodel* $\mathtt{M}$ *for an input formula* A *is a least fixed point of a sequence* $\mathtt{M}_0 = (\mathtt{W}_0, \mathtt{R}_0, \mathtt{V}_0), \mathtt{M}_1, \ldots$[6] *where*[7]*:*

- $\mathtt{W}_0 = \{\mathtt{w}_0\}$ *(the root),* $\mathtt{R}_0 = \emptyset$ *and* $\mathtt{V}_0 = \{(\mathtt{w}_0, A)\}$*;*
- $\mathtt{M}_{i+1} = \mathrm{Df}(\mathrm{Sat}\Box(\mathrm{Slf}(\Diamond(\mathtt{M}_i))))$*, where* Sat$\Box$*,* Slf *and* $\Diamond$ *are the meta-rules given above and* Df *is the dynamic filtration operation.*

[5] Where $\circ$ denotes composition: $R_I \circ R_J(w, u)$ iff there exists v s.t. $R_I(w, v)$ and $R_J(v, u)$.
[6] We will prove later that this sequence is finite, hence it has a fixed point.
[7] Due to rule $\vee$, the are several such sequences.

8.3.2 Soundness and completeness

First of all, we can easily check the following proposition:

Proposition 2. *Proposition 1 holds for filtrated premodels.*

Proposition 3. *As a direct consequence of the definitions above, at each iteration, only an exponential number of nodes is added.*

In fact, $Card(\{\mathtt{w}\colon \mathtt{w} \in \mathtt{W}_i \text{ and } \mathbf{fst}(\mathtt{w}) = k \text{ for some } k \leq i\}) \leq Card(\equiv^i) \leq 2^{Card(Sub(A))} = 2^{c.|A|}$ for some constant c, where $\equiv^i$ is the equivalence relation defined (as in Definition 34) at iteration i.

Lemma 3. *Let* $\mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ *be an open filtrated premodel for* A *then* $\mathtt{M}$ *contains at most exponentially many nodes each of size bounded by* $|A|$.

Proof. From Proposition 1, $\forall \mathtt{w} \in \mathtt{W}\colon 0 \leq \mathbf{fst}(\mathtt{w}) \leq d(A) \leq |A|$. From Proposition 3, there are at most $2^{c.|A|}$ nodes for each iteration, hence $\mathtt{M}$ contains at most $|A|.2^{c.|A|}$ nodes which is bounded by $2^{c'.|A|}$ for some constant c'. Note that nodes are of maximal size $|A|$ since $\mathtt{V}(\mathtt{w}) \subseteq Sub(A)$. □

Lemma 4 (Soundness and completeness)**.** *The model construction method with the dynamic filtration is sound and complete for layered modal logics.*

Proof. Let $\mathtt{M}_0, \ldots, \mathtt{M} = (\mathtt{W}, \mathtt{R}, \mathtt{V})$ be a sequence of simple premodels, with $\mathtt{M}$ as the fixed point, and let $\mathtt{M}'_0, \ldots, \mathtt{M}' = (\mathtt{W}', \mathtt{R}', \mathtt{V}')$ be the sequence of filtrated premodels $\mathrm{Df}(\mathtt{M}_0), \ldots, \mathrm{Df}(\mathtt{M})$.

Let $M = (W, R, V)$ be the Kripke model defined over $\mathtt{M}$, where $W = \mathtt{W}$, $R = \mathtt{R}$ and $V = \mathtt{V}|_{\mathcal{P}}$, and let $M' = (W', R', V')$ be the model defined similarly over $\mathtt{M}'$.

Let $\bar{w}$ be the equivalence class of w (as defined in Definition 34) w.r.t. the equivalence relation $\equiv^{\mathbf{fst}(w)}$, i.e. the relation defined at the $\mathbf{fst}(w)$ iteration.

Let us consider then the function $m : W \to W'$ defined as $m(w) = \bar{w}$. It is straightforward to check that m is a p-morphism (i.e. a pseudo-epimorphism [Seg68]) from M to M': $w \in W$ and $m(w) \in W'$ satisfy the same set of atomic propositions, and for $w, u \in W$, $(w, u) \in R$ if, and only if, $(m(w), m(u)) \in R'$, thus, w and $m(w)$ satisfy the same formulas.

Hence the method constructing filtrated premodels is sound and complete for layered logics, since the method constructing simple premodels is so. □

As a direct consequence of lemmas 4 and 3, we have:

Theorem 3. *SLF-satisfiability is decidable in non-deterministic exponential time.*

Proof. The following non-deterministic algorithm does the job: guess a premodel for A and check whether it is open. This can be done in linear time in the size of the premodel, hence in time bounded by $O(2^{c.|A|})$. □

8.4 Implementing a model construction method for LML in LoTREC

In what follows, we show how to implement in LoTREC a model construction method for an LML logic characterized by a set SLF of layer formulas.

At first, we reuse the rules of the method for multi-modal K_n, which are defined in Chapter 3. They cover all the classical and modal rules. We still need to define however two sets of rules: a set that deals with the layer formulas of SLF, and a set of rules which filtrate the simple premodels.

8.4.1 Rules for layer formulas

Given a layer formula $\chi = \forall \overrightarrow{x} : \exists \overrightarrow{y} : \phi(\overrightarrow{x}) \rightarrow \psi(\overrightarrow{x}, \overrightarrow{y})$, it can be read as "if ϕ is true, then ψ should be so". As said before in Definition 21, χ describes a kind of graph pattern rewriting. Hence, it should be easily transcripted as a rule in LoTREC.

Recall that ϕ (rep. ψ) is a conjunctions of literals of the form $R_I(x_i, x_j)$ (resp. $R_I(x_i, y_j)$ or $R_I(y_i, y_j)$). So practically, the rule corresponding to χ would say that if the structure of interrelated nodes described by ϕ exists in the current premodel, then add to it the nodes and structure described by ψ. And this is easy to define in LoTREC by means of conditions and actions.

Suppose that `Chi` is the rule defined in LoTREC for χ. Then:

- for each literal in ϕ of the form $R_I(x_i, x_j)$ we add to `Chi` the condition `isLinked xi xj R_I`,
- for every y_i appearing in ψ we add to `Chi` the action `createNewNode yi`,
- and for each literal in ψ of the form $R_I(x_i, y_j)$ we add to `Chi` the action `link xi yj R_I` (similarly for $R_I(y_i, y_j)$ literals).

It is clear that we need as many rules as there are frame formulas in SLF.

8.4.2 Rules for dynamic filtration

According to Definition 34, the dynamic filtration operation consists of two steps: (1) detection of equivalent new nodes and (2) then identification of multiple equivalent nodes with one of them. This can be achieved in LoTREC by defining a set of rules for each step of the operation.

First, remember that equivalent new nodes are those added recently (at the last iteration) and yet having the same labels, i.e. the same set of formulas. To detect newly added nodes we use the condition `isNewNode x`, and to test if two nodes `x` and `y` are equivalent, we double check wether `contains x y` and `contains y x`. Hence, to detect couples of equivalent new nodes we can define a rule of the form:

```
Rule Detect_Equivalent_Nodes
  isNewNode x
```

```
  isNewnode y
  contains x y
  contains y x

  link y x Equivalent_To
End
```

This rule links the detected equivalent nodes by a special link labeled `Equivalent_To`. However, we may to chose to mark both of them by `Equivalent_To_Another_Node`, or to do whatever else to register this information in the graph.

The next step is to filtrate the premodel by merging equivalent nodes, then changing the relations and the valuation function accordingly.

Well, since there is no `merge node1 node2` action predefined in LoTREC, we may proceed in two steps as follows:

1. for a given class of equivalent nodes we chose a representative node,
2. all the other nodes of this class will be disabled, i.e. no rule would apply on them, as if they were merged in the representative node, and since, they do not exist in the premodel (at least for the rules),
3. all the in- and ou-edges of these nodes should be redirected toward their representative nodes.

The first step can be achieved by the following rule:

```
Rule Chose_Representative
  isLinked x y Equivalent_To
  isNotMarked x Has_Representative

  mark x Representative
End
```

One may ask "How a node could be marked as `Has_Representative`?" the answer is in the following rule:

```
Rule Mark_Nodes_Having_A_Representative
  isLinked y x Equivalent_To
  isMarked x Representative

  mark y Has_Representative
End
```

In order to mark only one node by class of equivalence, the rule `Chose_Representative` should be called after the `applyOnce` strategy keyword, right before the rule `Mark_Nodes_Having_A_Representative`. If called inside a `repeat`...`end`, in this way:

```
repeat
  applyOnce Chose_Representative
  Mark_Nodes_Having_A_Representative
end
```

we guarantee that all representatives of all equivalence classes will be designated, and only one representative by class.

The second step consisting on disabling the nodes which have a representative is much simpler. It consists on adding to every defined rule the condition `isNotMarked x Has_Representative`, to prevent the rules form being applied on such (virtually merged) nodes.

The last step is feasible via two rules. The first is:

```
Rule Copy_In_Edges
  isLinked y x Equivalent_To
  isMarked x Representative
  isLinked z y variable Some_R

  link z x variable Some_R
End
```

Note that there is no need to test whether `y` is marked as `Has_Representative` since this is subsumed by the fact that `x` is marked as `Representative`.

The other rule, we call `Copy_Out_Edges`, is defined similarly, with `z y` and `z x` being replaced by `y z` and `x z`.

8.5 Discussion

Some properties cannot directly be considered as layered ones, we discuss some of them and show how they can still be handled.

8.5.1 The case of symmetry

In order to treat such a property, the rule $\langle I \rangle$ must be modified as follows:

$$\nu_{\langle I \rangle}(\mathtt{M}) = \Big(\{\langle I \rangle (B, u)\}, \{(u, \langle I \rangle (B, u)), (\langle I \rangle (B, u), u))\}, \{(\langle I \rangle (B, u), B)\} \Big)$$

for each $I \in \mathcal{I}$ and for all $\langle I \rangle B \in \mathtt{V}(u)$: i.e. add the symmetric edge at the same time a new node is introduced (Figure 8.6).

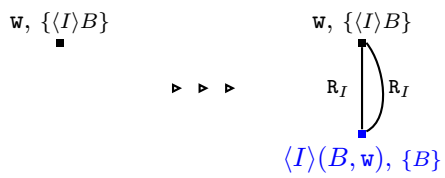


Figure 8.6: Modified $\langle I \rangle$ rule for symmetry.

This respect the strict increase of node depth (condition (iv) of definition

22)[8]. The same kind of adjustment can be used for tense forms of layered logics (i.e. where converse modalities, $[I^-]$ and $\langle I^- \rangle$ for $I \in \mathcal{I}$, are allowed).

8.5.2 The case of permutation

The case of one permutation property (Figure 8.7) like e.g.

$$\forall x, y, z \colon \exists u \colon (R_I(x, y) \wedge R_J(y, z)) \to (R_J(x, u) \wedge R_I(u, z))$$

corresponding to the axiom $[J][I]P \to [I][J]P$ can be handled by the same kind of adjustment of the $\langle J \rangle$ rule.

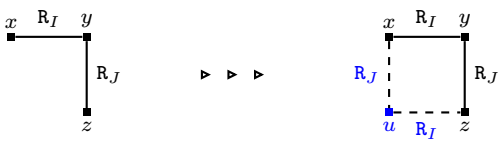
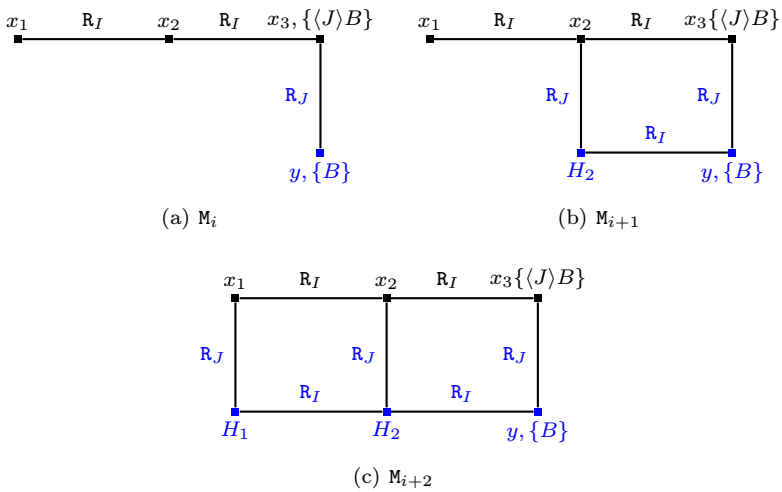


Figure 8.7: Graphical representation of the permutation layer formula.

In fact, without adjustment, the model construction may seem as in Figure 8.8. Which rises questions about conformity with Proposition 1, especially whether $\mathbf{fst}(H_i) \leq \delta(H_i)$ (hence wether $\mathbf{fst}(H_i) \leq d(A)$) and whether $\delta(H_i) > \delta(y)$, for $i \in \{1, 2\}$.



(a) $\mathtt{M}_i$ (b) $\mathtt{M}_{i+1}$

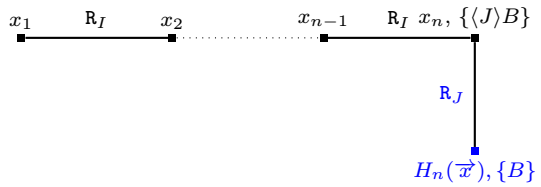
(c) $\mathtt{M}_{i+2}$

Figure 8.8: Permutation without modification of the rules.

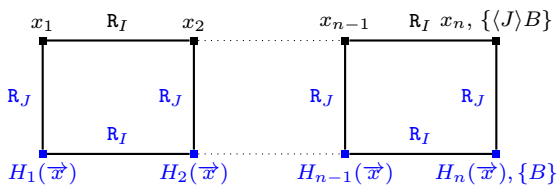
Instead, we use the rule below (Figure 8.9) which is essentially (we omit the case of empty nodes):

[8]Of course, for logic KB on its own, our present method is not interesting since it is known to have a PSPACE satisfiability problem.

If $(x_j, x_{j+1}) \in \mathtt{R}_I$ for $0 \leq j < n$ and $\langle J \rangle B \in \mathtt{V}(x_n)$ then
$\nu_{perm}(\mathtt{M}) = \{$
$\{H_j(\overrightarrow{x}) \colon 0 \leq j < n\}$,
$\{(x_j, H_j(\overrightarrow{x})) \colon 0 \leq j \leq n\} \cup \{(H_j(\overrightarrow{x}), H_{j+1}(\overrightarrow{x})) \colon 0 \leq j < n\}$,
$\{(H_n(\overrightarrow{x}), B)\}$ $\}$
with a modified definition of δ such that depth along $\mathtt{R}_J$ is "heavier" than along $\mathtt{R}_I$, this ensures that in the rule, all $H_j(\overrightarrow{x})$ are deeper than x_j's and is harmless w.r.t. Proposition 1.



(a) usual $\langle I \rangle$-rule



(b) adapted $\langle I \rangle$-rule

Figure 8.9: All nodes due to the permutation are created and linked at once.

Formulated in this way, the above rule handled permutation while respecting the increase of nodes depth which is the main point on which rely our complexity result.

Conclusion

We have defined a new class of modal logics called *layered* because their models can be constructed layer by layer. These logics are specified by properties of their models which are thus called layered.

We proved they are decidable and that their satisfiability problem lies in NEXPTIME. This is the best possible lower bound since some of them are known to be complete for this complexity class.

To achieve this, we designed a sound, complete tableau calculus that permitted us to ensure the small model property of layered logics. With the help of various possibilities (modification of the $\Diamond$ rules, of the depth definition,...) we believe that this technical tool may extend to other logics and even to some transitive layered logics by introducing loop tests over whole subgraphs. This is subject of ongoing work.

Part II

Graph Rewriting for Model Construction in Logic

Preface to Part II

As shown in Chapter 3, a premodel can be embedded in a graph structure. Consequently, model construction for a given formula in modal logic can be done, as shown throughout the Chapters 3 to 7, by applying an appropriate set of graph rewriting rules on an initial graph, where the initial graph is composed of a single node labeled by the considered formula. This was first presented in the work of Castilho et. al. in [CFnDCGH98], which gave birth to our generic theorem prover LoTREC in 1999 [dCFG$^+$01].

LoTREC is conceived to be a generic platform, in the sense that it can cover a wide spectrum of various logics having standard Kripke's semantics. Indeed, LoTREC allows the user to define a new logic syntax, and then offers a simple way to define this logic semantics via rewriting rules. These rules can be defined in a high level textual language that can be easily learned by non computer-scientists, such as students and researchers in logic and philosophy.

However, LoTREC is a special purpose graph rewriting tool for two reasons. First, it can only deal with a special structure of graphs which encodes Kripke's models: mainly graphs with nodes and edges labeled by formulas. So it does not allow the definition of other labels or attribute types such as reals, integers or strings, for example. Second, the definition of rewriting rules in LoTREC is restricted by some constraints, due to the application domain of LoTREC in logic, and for matters of efficiency in the rewriting process. One of these restrictions is that we can only define connected patterns on the left-hand side of a given rule. Another restriction is that the right-hand side of a rule is always adding elements (hence, the rules are said to be *monotonic*).

Due to these restrictions, the graph rewriting kernel of LoTREC is built from scratch and written in native imperative Java code without using any external library. On the one hand, this allows LoTREC to have its own way of compiling the rules, of performing the graph pattern matching and of applying the rules in an optimal way. This helps in achieving the graph rewriting task in the most possible efficient way, letting the complexity to be only related to the underlying logic. On the other hand, this raises some questions about the soundness and the semantics of graph rewriting in LoTREC, and about its theoretical properties in contrast to well-established graph rewriting approaches and tools such as PROGRES [Sch97] and AGG [Tae99].

In fact, in some graph rewriting approaches there are some well known results showing that such and such theoretical properties are taken for granted or can

be defined in this or that way (see Section 9.4). Thus by knowing that the rules defining a model construction procedure in LoTREC are based on one of those rewriting approaches, many theoretical properties that we wish to have for that procedure can be proved easily or inherited directly from those already established results.

Nevertheless, tracing back the links of LoTREC to its theoretical roots in the literature of graph rewriting has never been done. For all the above reasons, I decided to trace back these roots during my Ph.D. thesis. Furthermore, my attempt in this second part is to establish the bridge between model construction in modal logics and the domain of graph rewriting by studying the special case of LoTREC.

In the first chapter, I give an overview of graph rewriting. In the second chapter I establish the link between LoTREC and its roots in graph rewriting. In the last chapter I present the event-driven optimisation of the pattern matching process in LoTREC.

I hope that this part makes the semantics of graph rewriting in LoTREC accessible to everyone who is familiar with the usual graph rewriting notations. People with background in logic who are not so familiar with graph rewriting notations but interested in developing similar graph-based computation tools, can find a fair amount of what they need to fund their tools on solid theoretical basis. This work should also help in reducing the gap that exists between the graph rewriting theory and its Java implementation in LoTREC.

Préface à la Partie II

Comme déjà montré au chapitre 3, un prémodèle est un graphe où les noeuds et les arrêtes sont étiquetés. Par conséquent, la construction de modèles pour une formule donnée peut être accomplie, comme indiqué dans les chapitres 3 à 7, en appliquant un ensemble de règles de réécriture de graphes sur un graphe initial à un seul noeud étiqueté par la formule considérée. Cette approche fut d'abord présentée dans les travaux de Castilho et. al. [CFnDCGH98], puis elle a donné naissance à notre démonstrateur générique LoTREC en 1999 [dCFG+01].

LoTREC est conçu pour être une plate-forme générique : il couvre un large spectre de différentes logiques ayant une sémantique standard de Kripke. En effet, LoTREC permet à l'utilisateur de définir la syntaxe d'une nouvelle logique, puis offre un moyen simple pour définir des règles de réécriture adaptées à la sémantique de cette logique. Ces règles sont définies dans un langage textuel de haut niveau qui peut être facilement maîtrisé par des non-informaticiens, comme par exemple des étudiants et des chercheurs en logique et/ou en philosophie.

Toutefois, LoTREC est un outil de réécriture de graphes à usage spécial pour deux raisons. Premièrement, il ne peut traiter que des graphes ayant une structure particulière : celle qui encode un modèle de Kripke. Principalement, il traite des graphes avec des noeuds et des arcs étiquetés par des formules. En outre, il ne permet pas la définition d'étiquettes complexes ou d'autres types d'attributs tels que les réels, les entiers ou les chaînes des caractères par exemple. Deuxièmement, la définition de règles de réécriture sous LoTREC est limitée par certaines contraintes, à cause de la spécificité du domaine d'application de LoTREC en logique, et pour des raisons d'efficacité dans le processus de réécriture. Une de ces restrictions est que nous ne pouvons pas définir des motifs de graphes déconnectés comme membres gauches d'une règle donnée. Une autre restriction est que les règles sont *monotones* : on ne supprime pas des noeuds, d'arcs ni des formules.

Grâce à ces restrictions, le noyau de réécriture de graphes de LoTREC est construit à partir de zéro et il est entièrement écrit en code impératif et natif Java, sans l'utilisation d'aucune librairie externe. En effet, cela permet à LoTREC d'avoir sa propre façon de compiler les règles, d'effectuer le "pattern matching" des graphes et d'appliquer les règles d'une manière optimale. Cela aide à accomplir la tâche de réécriture de graphes de la manière la plus efficace, en laissant la complexité uniquement liée à la logique en question. D'autre part, cela soulève quelques questions sur la sémantique de réécriture de graphes

sous LoTREC, et sur ses propriétés théoriques en contraste avec les approches déjà bien établies et les outils disponibles tels que PROGRES [Sch97] et AGG [Tae99].

En effet, dans certaines approches de réécriture de graphes on a des résultats bien connus démontrant que telle ou telle propriété théorique est garantie ou peut être définie de telle ou telle façon (voir la section 9.4). Ainsi, en sachant que les règles définissant une certaine procédure de construction de modèles dans LoTREC sont fondées sur l'une de ces approches de réécriture, de nombreuses propriétés théoriques que nous souhaitons avoir pour cette procédure peuvent être montrées facilement ou héritées directement de ces résultats déjà établis.

Néanmoins, remonter les liens de LoTREC jusqu'à ses racines théoriques dans la littérature de réécriture de graphes n'a jamais été fait. Pour toutes ces raisons, j'ai décidé de retracer ces racines au cours de ma thèse de doctorat. En outre, ma tentative dans cette seconde partie du manuscrit est d'établir les ponts entre la construction de modèles pour les logiques modales et le domaine de réécriture de graphes, en étudiant le cas particulier de LoTREC.

Dans le premier chapitre, je présente une vue d'ensemble du domaine de la réécriture de graphes. Dans le deuxième chapitre, j'établis le lien entre LoTREC et ses racines dans ce domaine. Dans le dernier chapitre, je présente l'optimisation du processus de "pattern matching" dans LoTREC, qui est basée sur un modèle événementiel.

J'espère que cette partie rende la sémantique de réécriture de graphes sous LoTREC accessible à tous ceux qui sont familiers avec les notations habituelles utilisées dans le monde de réécriture de graphes. Les logiciens, qui ne sont pas forcément familiers avec ces notations, mais qui sont intéressés par le développement d'un outil similaire à LoTREC, peuvent trouver dans cette partie des bases théoriques solides pour créer leurs propres outils. Ce travail devrait également contribuer à réduire l'écart entre la théorie de réécriture de graphes et sa version Java implémentée sous LoTREC.

Chapter 9

Graph rewriting overview

Introduction

The main idea of *graph rewriting*, also known as *graph transformation*, is the rule-based modification of graphs. A *graph rewriting rule* describes how to *derive* a graph from another. To this end, a rule describes the changes to be made on a given graph and the part of the graph where to apply the changes.

Graph rewriting has been influenced by the mathematical tradition to use axioms and inference rules for reasoning. It is also similar to Chomsky grammars in formal language theory where productions are used to describe and construct the words. Another closely related theoretical root is *term rewriting*. Married together, term rewriting and graph rewriting gave birth to *term graph rewriting* [Plu99], the approach that uses the graphs to encode terms and the graph rewriting techniques to efficiently perform computations on terms.

Research on graph transformation dates back to the 1970s and still has a growing popularity nowadays. Graphs and graph transformation have been studied and applied in many fields of computer science: in database design, software engineering [BH02], image recognition, concurrent and distributed systems, and fields of other domains such as Origami folding, chemical reactor simulation and many others (see for example [BFG96, EP06, AIK06, IT09]). A detailed presentation of various graph transformation approaches and their application areas is given in [Roz97, EEKR99].

The simple reason behind the popularity of graphs is that often when we think about a new project, system, map, procedure, etc., we intuitively draw some bubbles and arcs to write down and organize our ideas in form of a graph. This makes our descriptions more concise and easily accessible to others. An immediate consequence is the interest in graph transformation, due to the need to define the dynamic and computational aspects on the graphs, and then due to the need to automatize these computations.

9.1 Graph structures

Graphs are a common and intuitive way to model and represent a project, a system, a situation or a procedure in a convenient level of abstraction. This abstraction allows us to work on such concepts more easily. The basic definition of a graph is the one considering a graph as a *relational structure* $G = (V, E)$ where V is a set of vertices (also called nodes), $E \subseteq V \times V$ is a set of edges (also called arcs) defining a relation between the vertices. According to this definition, there exists at most one edge between a given pair of vertices. To avoid this restriction, we suppose E and V as disjoint sets of elements ($V \cap E = \emptyset$) and we define two total mappings $source, target : E \to V$ such that each edge e links a given source node $source(e)$ to a target node $target(e)$.

9.1.1 Labeled graphs

Such a graph structure remains insufficient in practice to reason about some complex situations. For instance, suppose that we want to represent by a graph a simple traffic light system in which the `Current_State` is `Red`, and will turn to `Green`, then `Yellow`, then `Red` again and so on..., as can be seen in Figure 9.1. The basic graph structure defined above would not be suitable, that is why we define what we call *labeled* graphs.

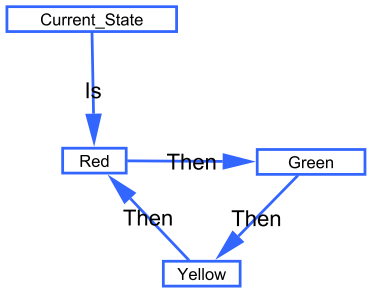


Figure 9.1: A labeled graph

A labeled graph is a graph structure $G = (V, E, source, target, \ell_V, \ell_E)$ defined over a set of labels $\mathcal{L} = \mathcal{L}_V \cup \mathcal{L}_E$ using two labeling functions $\ell_V : V \to \mathcal{L}_V$ and $\ell_E : E \to \mathcal{L}_E$. We may use $\ell : V \cup E \to \mathcal{L}$ instead of ℓ_V and ℓ_E for the sake of simplicity. Some restrictions can be added to the definition so that the labels of the vertices have to be different than the labels of the edges ($\mathcal{L}_V \cap \mathcal{L}_E = \emptyset$), or that a pair of nodes cannot be linked by two edges having both the same label ($E \subseteq V \times \mathcal{L}_E \times V$).

The number of vertices of a graph G is denoted by $|G|$ and it is called *size* of G. For example, the size of the graph of Figure 9.1 is 4.

To avoid confusion when talking about two graphs, we use V_G, E_G, $source_G$, $target_G$ and ℓ_G to denote the sets of elements and the functions of the graph G. If a graph H contains a graph G, i.e. $V_G \subseteq V_H$, $E_G \subseteq E_H$ and $\ell_G(V_G \cup E_G) \subseteq$

$\ell_H(V_H \cup E_H)$ then G is a *subgraph* of H, denoted by $G \subseteq H$. Union, intersection and complement of graphs are also defined in terms of union, intersection and complement of their sets of vertices, edges and labels.

In order to talk about resemblance of two graphs G and H and the correspondence between their elements, we define what we call a graph *morphism*.

Definition 36 (Graph morphism)**.** *A total graph morphism $m : G \to H$ between the graphs G and H is a pair of mapping functions $m = (m_V : V_G \to V_H, m_E : E_G \to E_H)$ that preserve the structure and labeling functions of G and H, i.e.*

- $source_H \circ m_E = m_V \circ source_G$,
- $target_H \circ m_E = m_V \circ target_G$,
- $\ell_H \circ m_E = \ell_G$,
- *and* $\ell_H \circ m_V = \ell_G$.

A *partial* morphism from G to H is a total morphism from some subgraph $D \subseteq G$ to the graph H. The graph morphism is *injective* if both m_V and m_E are injective and *surjective* if both m_V and m_E are surjective. A graph morphism is an *isomorphism* if both m_V and m_E are bijections.

9.1.2 Typed attributed graphs

A labeled graph is still not sufficient to represent rich data structures. For example, if we want to add a label representing a counter, it would be better to have a label of *type* integer on which we are able to perform some computations using the usual arithmetic *operators*. This cannot be done with labeled graphs, and *typed attributed graphs* (TAG) are used instead [EPT04, EEPT06]. In this thesis, we are interested in attributed graphs since their structure is rich enough to encode models of modal logics with formulas as attributes, as we will see in Chapter 10.

The idea of TAGs is simply to add to the graph structure nodes that represent the attributes, we call them *data vertices*, and edges to link the graph nodes and graph edges to those data vertices, we call them *data edges*. This is done by defining a special kind of graphs called E-graphs used as a support for attributed graphs. E-graphs are sketched in Figures 9.2(a), 9.2(b) and are defined as follows:

Definition 37 (E-graph)**.** *An E-graph is defined as $G = (V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i \in [1,3]})$ where*

- *V_1 and V_2 are respectively the sets of graph vertices and* data *vertices,*
- *E_1,E_2 and E_3 are respectively the sets of graph edges,* vertex data *edges and* edge data *edges,*
- *$source_1 : E_1 \to V_1$, $source_2 : E_2 \to V_1$ and $source_3 : E_3 \to E_1$,*

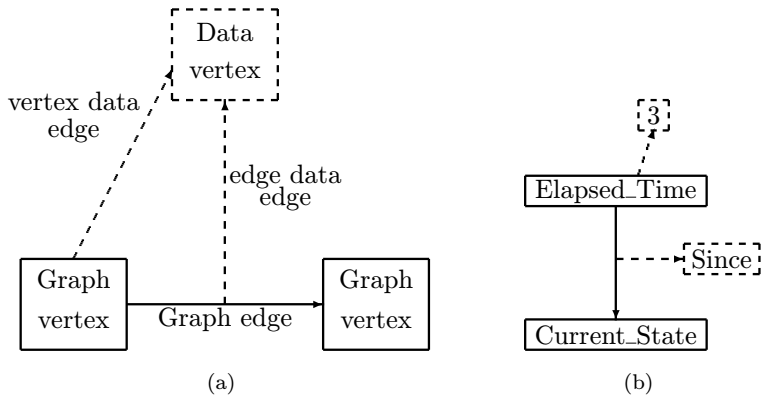


Figure 9.2: (a) a schematic representation of attributed graphs, (b) an example of an attributed graph

- $target_1 : E_1 \to V_1$, $target_2 : E_2 \to V_2$ *and* $target_3 : E_3 \to V_2$.

An E-graph becomes an attributed graph when adding an algebraic signature[1] for the attributes, as follows:

Definition 38 (attributed graph)**.** *Given a signature SIG =(S, OP), an attributed graph, with attributes of sorts $S' \subset S$, is a pair (G, D) of an E-graph G and a SIG-algebra D such that $\bigcup_{s \in S'} D_s = V_2$.*

As in the case of labeled graphs, we can talk about similarities between attributed graphs by defining morphisms. Attributed graph morphism is necessary to define next "typed attributed graphs", and is used later in Section 9.3.4 to define their transformations. First, we define morphisms on E-graphs.

Definition 39 (E-graphs morphism)**.** *A morphism $m : G \to G'$ between two attributed graphs G and G' is defined by a tuple of five mappings $m = (m_{V_1}, m_{V_2}, m_{E_1}, m_{E_2}, m_{E_3})$ where*

- $m_{V_i} : V_i \to V'_i$ *for* $i \in [1, 2]$*;*
- $m_{E_i} : E_i \to E'_i$ *for* $i \in [1, 3]$*,*

and such that m commutes with all the source and target functions.

A morphism between two attributed graphs is then defined in terms of a pair of morphisms: a graph morphism and an algebra homomorphism[2] with a specific interaction condition.

[1]Algebraic signatures were broadly introduced in [EM85]. However we sketch a basic introduction needed to the understanding of the sequel in Appendix A.

[2]Signature morphisms and algebra homomorphisms are defined in Appendix A.

Definition 40 (attributed graph morphism)**.** *A morphism $m : G \to G'$ between two attributed graphs is a pair, $m = (m_G, m_D)$, of an E-graph morphism m_G and an algebra homomorphism m_D such that the following diagram is a pullback for all $s \in S'$*

$$\begin{tikzcd} D_s \arrow[r, "m_{D,s}"] \arrow[d, hook] & D'_s \arrow[d, hook] \\ V_2 \arrow[r, "m_{G,s}"] & V'_2 \end{tikzcd}$$

This diagram reflects a condition on the interaction between the graph and data morphisms. This condition practically means that the data attributes in G' can only be calculated from computations made on the data attributes of G.

Typed attributed graphs are attributed graphs of a specific type, i.e. attributed graphs dotted with a graph morphism to a *type graph*. Thus, we need first to define what is a type graph.

Definition 41 (type graph)**.** *A type graph is a distinguished attributed graph (G_0, Z), where Z is the SIG-final algebra*[3]*.*

Definition 42 (typed attributed graph)**.** *A typed attributed graph $((G, D), t)$ over a type graph (G_0, Z) is defined by the attributed graph (G, D) and the attributed graph morphism $t : G \to G_0$.*

We can finally define morphisms on TAGs as follows:

Definition 43 (typed attributed graphs morphism)**.** *A typed attributed graph morphism $m : G \to G'$, between two attributed graphs typed over the same type graph (G_0, Z), is an attributed graph morphism verifying the following additional condition*

$$t' = m \circ t$$

Other definitions given in Section 9.1.1 for labeled graphs can be also easily extended to the case of typed attributed graphs. Moreover, we can show that labeled graphs are special cases of TAGs (see [EEPT06] Chapter 2).

9.2 Rewriting rules

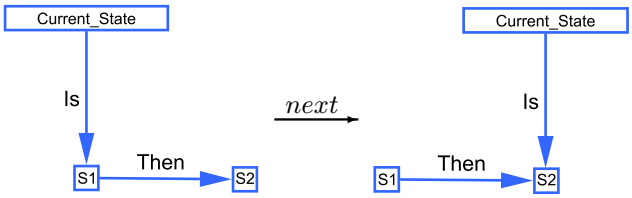
Regardless from the kind of the graph we are dealing with, a graph transformation is described by a graph rewriting rule which consists of a pair of graphs $\rho = (L, R)$, where L denotes the left-hand side graph and R denotes the right-hand side graph. The application of a given rule ρ on a given graph G yields another graph, call it H. It is denoted by $G \overset{\rho}{\Rightarrow} H$, and is called a *derivation*, *production* or a *rewriting step*. Roughly speaking, ρ describes the changes that need to be made on G in order to obtain H, and specifies on what parts of G

[3]see Appendix A.

these changes should be performed. To this end, it defines a mapping between the graph elements (i.e. nodes, edges, labels...) of L and R, often as a graph morphism. This mapping describes which elements are preserved, deleted or created when applying the rule ρ. Note that since this description is abstract and has to be matched with instance graphs, we call L and R *pattern* graphs.

$$L \xrightarrow{\rho} R$$

(a)



(b)

Figure 9.3: (a) the definition of a graph rewriting rule, (b) an example of a rule called "*next*".

Example 10. In Figure 9.3(b), we give a graphical representation of a rule called *next*. The nodes are preserved in both its left and right hand side graphs L and R. The arc labeled `Is` that links the node labeled `Current_State` to the node labeled `S1` in L is deleted in R, then another arc with the same label (`Is`) is added to R, in order to link the node labelled `Current_State` to the node labeled `S2`.

A derivation $G \overset{\rho}{\Rightarrow} H$, denoting the application of a rule $\rho = (L, R)$ on a graph G, consists of two steps:

1. finding a match of L in G,
2. constructing H from G by replacing the found occurrence of L by R.

These two steps are explained and formalized in the following subsections.

9.2.1 Pattern matching and rule applicability

The first step in a derivation $G \overset{\rho}{\Rightarrow} H$ consists in checking whether the rule is applicable on the host graph G. Practically, a match of L in G has to be identified, using a total morphism $m : L \to G$ (in the sense of Definition 36). It consists in finding a subgraph $D \subseteq G$ such that D matches L w.r.t. m. This process is known as the *pattern matching process*, and D is called the *match*, the

occurrence of ρ or also the *redex*. As we discuss in Section 9.4, this step is the most expensive with respect to time cost in the graph transformation process.

Some rewriting systems have additional default conditions that are verified during the pattern matching process. For example, in some systems, a rule called many times is not applied twice on the same matching subgraph. In other systems, if the effect of applying a rule on a given matching redex (i.e. the outcome of the rule application on it) is already present in the host graph, then it is said to be not applicable on that redex, and another match, if any, has to be considered. Such conditions constitute what is usually called the *rule applicability*.

9.2.2 Rule application

The second step of a derivation $G \overset{\rho}{\Rightarrow} H$ is quite straightforward and consists in applying the set of changes described by ρ to the graph G, which usually yields a new graph, call it H. Practically, after finding a match and verifying that the rule is applicable on it, H is obtained from G by deleting the elements of G appearing in L and not appearing in R, then adding the elements of R not appearing in L. Elements shared by L and R are preserved during the derivation from G to H. Consequently, H is constructed as $G \setminus (L \setminus R) \cup (R \setminus L)$, with respect to a morphism $\rho' : G \to H$. G is called the *host* graph and H the *replacement* graph.

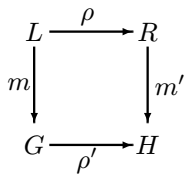
Example 11. Figure 9.4 gives in (a) a schematic representation of a rule application, and in (b) an example of the application of the rule *next* of Figure 9.3(b) on the graph representing the traffic light system in Figure 9.1.

The main technical problems are how to delete L and how to connect R in its place in a way where the resulting graph H remains a well-defined graph. Figure 9.5 shows two examples of problematic rules.

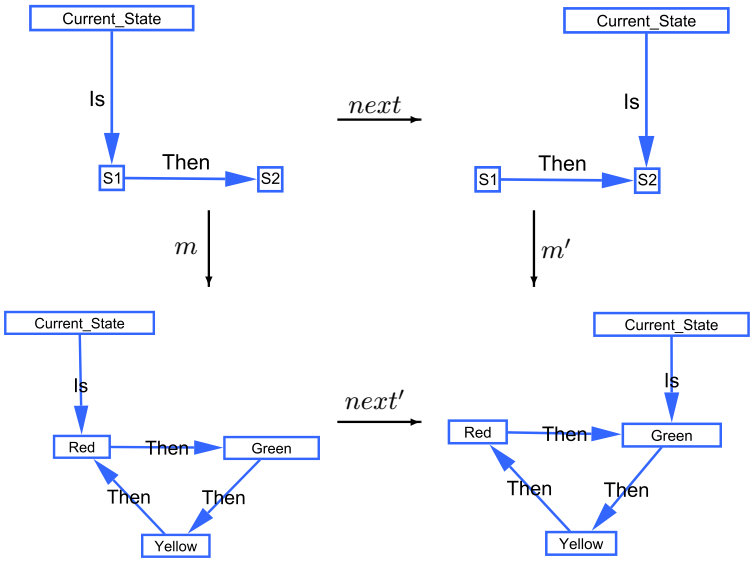
Example 12. The first rule (Figure 9.5(a)) deletes the vertices `S1` and `S2` and leaves behind the vertex `S3` with a dangling edge, i.e. its application on a given graph does not yield a well-defined graph. The resolution of such a problem is to either systematically delete such dangling edges or forbid this kind of rule.

The second rule (Figure 9.5(b)) consists in merging two nodes `S1` and `S2` by deleting `S2` and preserving `S1`. If the underlying rewriting approach allows non-injective morphisms, then its left-hand side graph can be mapped to a graph with a single node `S`, by mapping both nodes `S1` and `S2` to `S`. Thus, applying the rule becomes confusing: does it delete `S` or preserve it? The three possible ways to proceed in this case are to systematically preserve the node, to systematically delete it or to forbid this kind of rules.

In the literature, there are several different theoretical approaches to define graph transformations, which lead to different solutions to the problems discussed in Example 12. In Section 9.3, we introduce some of the most known theoretical approaches used to define rewriting rules.



(a)



(b)

Figure 9.4: Schematic representation of rules application (a), and an example of the application of the *next* rule of Figure 9.3(b) on the graph of Figure 9.1.

9.3 Theoretical basis

Funding graph transformations on a theoretical and mathematical basis is essential to formally describe, better understand and express the properties of graph rewriting systems. Many approaches were used in the literature. Among them, we are presenting the algebraic approaches that were established since the 70's [Ehr79] and are still evolving [EEPT06, RFS08].

The algebraic approach adopts the *category theory* as the mathematical framework and tool to describe graph transformations. For example, according to the category theory, sets and functions define a category in which we can describe the construction of a set from other sets using a *gluing construction* i.e. by deleting some elements, adding by union some others, transforming other elements by applying the functions and so on... Such construction is called a *pushout* in categorical terms, and it is very similar to what a graph rewriting rule is performing on graphs during its application. The category theory has

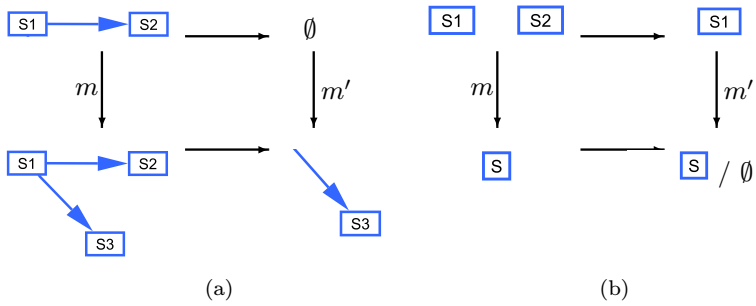


Figure 9.5: Problematic rules

also a dual construction called *pullback* that describes the reverse engineering of a pushout.

All these notions are well known in the literature, and will be presented in the sequel for the case of graph transformations. We will also see in Section 9.4 that using the categorical approach we can easily express many theoretical properties of graph rewriting systems.

9.3.1 Single Pushout approach

Graphs defined in Section 9.1 as $G = (V, E, source, target)$ form a special *algebra* with two base sets V and E, and two operations *source* and *target*. Graph morphisms $m : G \to H$ are special cases of algebra homomorphisms since they are compatible with the operations *source* and *target* as stated in Definition 36. Hence, these graphs and their morphisms constitute a special category in which the diagram of Figure 9.4(a) is a *pushout*. The underlying theoretical approach of this diagram is called the *Single Push Out* (SPO), in contrast to the *Double Push Out* (DPO) which we introduce and compare to the SPO in the next subsection. This algebraic view of graph transformation is presented in depth in [EEPT06].

9.3.2 Double Pushout approach

In the DPO approach, a rule is a triplet $\rho = (L \xleftarrow{l} K \xrightarrow{r} R)$, where L and R are the left and right hand side graphs and K is the common interface of L and R, i.e. their intersection, and l and r are graph morphisms as defined in Definition 36. Figure 9.6(a) and 9.6(b) show the schema and an example of a DPO rule definition.

According to the DPO, applying the rule ρ on a graph G consists of the following steps:

1. a match m of L in G is to be found, such that m is structure preserving, i.e. a graph morphism;

2. all vertices and edges matched by $L \setminus K$ are removed from G, which becomes the graph D;

3. the graph D is glued together with $R \setminus K$ to obtain the graph H.

The removed part at step 2. is not a graph, in general, but the remaining structure D resulting from $(G \setminus m(L)) \cup m(K)$ still has to be a legal graph, i.e. no edges should be left dangling. This means that m must satisfy a suitable condition, called the *gluing condition*, which makes sure the gluing of $L \setminus K$ and D is equal to G.

Definition 44 (gluing condition)**.** *The DPO definition of a rule* $\rho = (L \xleftarrow{l} K \xrightarrow{r} R)$ *satisfies the gluing condition iff the following two conditions hold:*

- *no edge* $e \in E_G \setminus m_E(E_L)$ *is incident to any of the vertices in* $m_V(V_L \setminus l_V(V_K))$*, which is called* dangling edge condition*;*

- *there are no* $a, b \in V_L \cup E_L$ *such that* $m(a) = m(b)$ *and* $a \notin l(V_K \cup E_K)$*, which is known as the* identification condition*.*

The dangling edge condition requires that if a rule ρ defines the deletion of a vertex, then it must also define the deletion of all the edges that are incident to this vertex in order to ensure that D has no dangling edges, and the identification condition requires that each element that will be deleted by the application of ρ has only one pre-image in L.

Under the gluing condition, the match of the overlapping part of R and L in K is not deleted from G, the graph D exists and is unique[4] up to isomorphism. Also, gluing the graphs D and $R \setminus K$ yields a well-defined graph H. This *gluing construction* of graphs can be considered as an algebraic quotient construction in the algebra of graphs and graph morphisms.

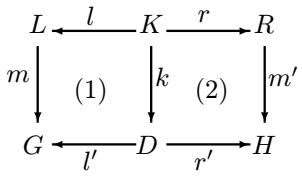
Comparison of DPO and SPO

The main feature of the DPO approach is that a rule must satisfy the *gluing condition*. In the SPO approach, this condition is not necessarily satisfied, and this may result into problematic rules as shown in Example 12. However, this can be avoided by deleting systematically all the incident edges of a deleted vertex, and solving vertex deletion/preservation conflicts in favor of deletion, for example, leading to a well-defined graph H.

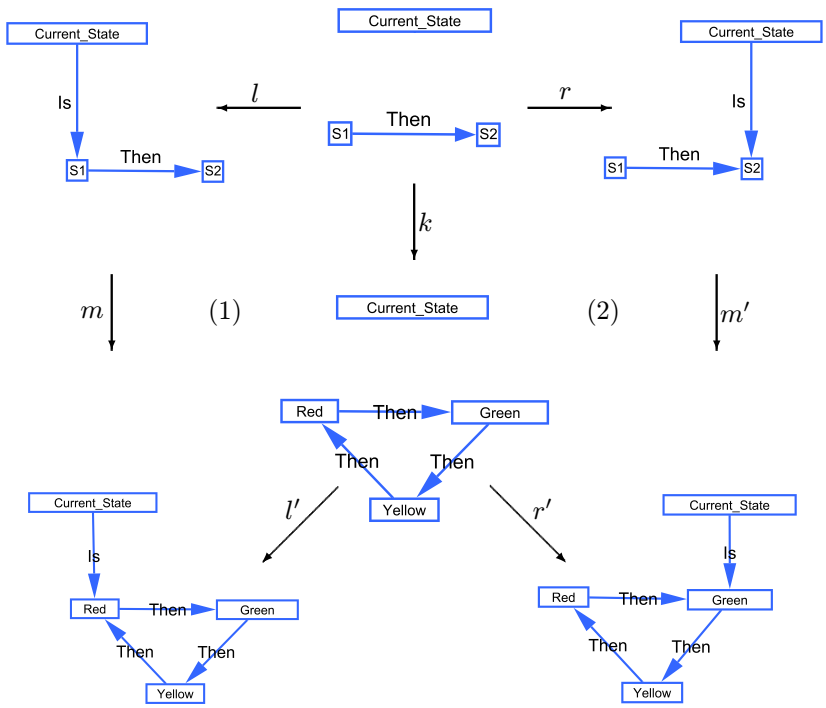
Moreover, the SPO is said to be more expressive than DPO, in the sense that it may model effects and special transformations that cannot be modeled in the *restrictive* DPO approach.

Nonetheless, the DPO rules are characterized by the *invertibility* property that is a desirable feature when graph rewriting is used for software engineering and database systems. An invertible rule allows the reconstruction of previous states by reversing the procedure of deleting and creating the vertices and edges.

[4] Chapter 3 of [EEPT06], Fact 3.11 page 45.



(a)



(b)

Figure 9.6: (a) Double pushout rule definition, (b) an example: the application of the rule *next* of Figure 9.3(b) w.r.t. the double pushout approach.

This is possible in DPO since the rules specify symmetrically added and deleted elements. Whereas in a SPO rule, it is not possible to specify all the modifications that has been carried out during the rule application. Thus, an ad-hoc solution is to save some additional information during the rule application at run-time.

9.3.3 Pullback approach

The SPO and DPO are the most famous algebraic approaches. Others propose yet another categorical approach called *pullback* based on the pullback construction which is the dual construction of the pushout in the category of graphs and graph morphisms. It has many advantages discussed in [BJ01], such as the possibility of defining subgraph replication and copying (i.e. cloning) any arbitrary graph with a single rule.

9.3.4 Adhesive High-Level Replacement approach

The algebraic approaches mentioned above are mostly used to describe the graph *structural* transformations. However, when dealing with complex graph structures, such as attributed graphs, we need to describe the computations made on the graph attributes during the transformations, as well as the changes of the graph structure.

Many approaches have been developed for typed attributed graphs so far in the literature ([HKT02, BFK00]). However, we adopt the "adhesive HLR category" algebraic approach introduced by Ehrig, Prange and Taentzer in [EPT04], then matured and broadly discussed by the authors in [EEPT06]. We give more credentials to this approach since it provides as fundamental results the Local Church-Rosser, Parallelism, Concurrency, Embedding and Extension Theorem and a Local Confluence Theorem known as Critical Pair Lemma in the literature.

This approach inspired Rebout, Féraud and Soloviev, the authors of the "Double Push-out Pullback" (DPoPb) approach, in [RFS08]. This newer approach preserves all the theoretical results of its elder one mentioned above. Moreover, it uses a homogeneous theoretical framework to define graphs structural transformations as well as their attributes computations. It also takes into account recursive typing (to define meta-meta-models for example), and it avoids having infinite instances of a given type.

These two approaches are the most suitable to define rewriting rules in our platform LoTREC, since they capture typed attributed graph transformations. However, the definitions of the graph rewriting rules in LoTREC (Section 10.2) follow the adhesive HLR categorical approach to a large extent, rather than the DPoPb approach due to the way the attributes are defined and used in LoTREC. The benefit of defining the rules according to the adhesive HLR approach is to be able to apply all its theoretical properties and results on our system. For example, we can find out easily under which conditions two rules are applicable

in parallel or whether or not the Church-Rosser property holds for a given set of rules (see Section 9.4).

A special class of monomorphisms

It is shown in [EPT04] that typed attributed graphs and a special class of monomorphisms form an adhesive HLR category. The proof given in the first paper[EPT04] is a short version of the proof sketched in the book [EEPT06], where the authors introduce step-by-step, through out the chapters, many categorical constructions for various families of graphs, until they end up with the categorical construction for typed attributed graphs.

To capture the main flow of these constructions, we dispose already of all the necessary ingredients in Section 9.1.2. In fact, it is easy to show that E-graphs and E-graph morphisms form the category $\mathsf{E\text{-}Graphs}$. The categories of signatures and algebras, Sig and $\mathsf{Alg}(SIG)$, are defined in Appendix A. The category $\mathsf{A\text{-}Graphs}$ is formed by attributed graphs and their morphisms. And finally, the attributed graphs typed over an attributed type graph $ATG = (G_0, Z)$ and their morphisms form the category $\mathsf{A\text{-}Graphs}_{ATG}$.

However, to be an adhesive HLR category, and thus inherit all its properties, the category $\mathsf{A\text{-}Graphs}_{ATG}$ has to restrict its class of morphisms.

Definition 45 (distinguished typed attributed graphs monomorphism)**.** *The class of distinguished monomorphisms $\mathcal{M}$ for the category $\mathsf{A\text{-}Graphs}_{ATG}$ is defined by $m = (m_G, m_D) \in \mathcal{M}$ if m_G is injective and m_D is an isomorphism.*

Theorem 4. *The category $(\mathsf{A\text{-}Graphs}_{ATG}, \mathcal{M})$ of typed attributed graphs over ATG is an adhesive HLR category.*

Proof. see [EPT04]. □

The definition of rewriting rules on typed attributed graphs remains similar to that in the DPO approach for labeled graphs, as follows:

Definition 46 (TAGs rewriting rule)**.** *A typed attributed graph rewriting rule $\rho = (L \xleftarrow{l} K \xrightarrow{r} R)$ is defined by three typed attributed graphs L,K and R attributed over the term algebra TA*[5] *of their data signatures, and two morphisms $l, r \in \mathcal{M}$.*

The definition and schema of rules application would look exactly like the DPO rules application definition given in Section 9.3.2 and sketched in Figures 9.6(a) and 9.6(b).

9.3.5 Alternative approaches

Many other approaches were developed on other theoretical basis. The *node label replacement* [ER91] allows a single node (in the left hand side graph L) to be replaced by an arbitrary graph R. The idea is very close to the *context*

[5]Terms and term algebras are defined in Appendix A.

free grammars. A logical approach were also developed in [Cou97], in which graph transformation and properties of graphs are defined in *monadic second-order logic* MSO. Others have developed more powerful but complex approaches, such as the *programmed graph replacement* approach of PROGRES [Sch91] that aims to allow the definition of complex left-hand side patterns and the use of programs to control the non deterministic choices in rules application. These approaches and many others were presented in details in [Roz97].

9.4 Properties of graph rewriting systems

Usually, a graph rewriting system consists of a set of graph rewriting rules $\mathcal{R}$. The transformation of a given graph G by a graph rewriting system is defined by a finite, or possibly infinite, sequence of derivations of the form $G \overset{\rho_1}{\Rightarrow} G_1 \ldots \overset{\rho_i}{\Rightarrow} G_i \overset{\rho_{i+1}}{\Rightarrow} \ldots$ where $\rho_i \in \mathcal{R}$ for all i. However in general, the goal of a rewriting system is to be able to compute for any given graph G the *final* graph G_f that is carried out by applying *repeatedly* the set of its rules $\mathcal{R}$. If there is no $\rho \in \mathcal{R}$ that would apply on G_f furthermore, then G_f is a *normal form* of G. The derivation sequence leading to G_f from G is denoted by $G \overset{*}{\Rightarrow} G_f$ when there is no need to explicit the rules applied at each step.

9.4.1 Characterization

Practically, in order to compute the normal form of a graph, a graph rewriting system has to make two non-deterministic choices at each rewriting step:

1. which rule to apply?
2. if a rule is applicable on many different subgraphs of the host graph, then on which subgraph this rule should be applied?

The first choice is decided by defining what we call a *control structure* or a *strategy*. It consists in defining an *ordering* on the set of rules, so that the rules are sequentially applied according to that ordering. Whereas the second choice is embedded in the policies of the *pattern matching process* and the notion of *rule applicability*.

Hence, a graph rewriting system is not simply specified by a set of rules, but it is also specified by the underlying choices concerning:

- the approach used to define the rules,
- the set of (implicit) constraints on rules applicability, i.e. the way the pattern matching process is achieved,
- the way the rules are applied,
- and the strategy that defines the order according to which the rules are applied.

According to these choices, a graph rewriting system is characterized by a set of theoretical properties, such as the properties defined in the sequel.

9.4.2 Termination

As stated at the beginning of Section 9.4, a rewriting system is usually conceived to perform finite computations, and that is why we are interested in the *termination* property of rewriting systems. The termination is studied in general w.r.t. to a set of rules, but when a strategy is defined on the rules, then it should also be taken into account.

Definition 47 (termination)**.** *A graph rewriting system with a set of rules $\mathcal{R}$ is terminating iff for every graph G there is no infinite derivation sequence $G \overset{\rho_1}{\Rightarrow} G_1 \ldots \overset{\rho_i}{\Rightarrow} G_i \overset{\rho_{i+1}}{\Rightarrow} \ldots$ where $\rho_i \in \mathcal{R}$ for all i.*

The question wether or not a rewriting system is terminating can be reduced to a halting problem of a Turing machine, i.e. does a rewriting system halt for a particular set of rules. Thus, in general, termination of rewriting is unsolvable. However, there are many restrictions that can be made on the rules and their application in order to guarantee the termination.

The most obvious sufficient condition for termination is that each rewrite rule in $\mathcal{R}$ must reduce the size of the initial transformed graph G. Otherwise, other restrictions should hold. For example, D. Plum showed in [Plu95] that termination of a graph rewriting system is guaranteed when the application of its rewriting rules consists of finite minimal derivations in which each step depends on previous steps, what he called *forward closures*. One of the constraints that the author wants is that each of these rules has to remove, during its application, at least one element of the redex matching its left-hand side graph L. Otherwise, a single rule would be infinitely applicable. However, this can be avoided practically by tuning the notion of the *applicability* of a rule, so that a rule is not applicable on the same graph twice at the same place.

The authors of [EEdL$^+$05] show that a special family of systems called *layered* graph rewriting systems is terminating[6]. The idea is to stratify the rules in different layers. Rules at each layer are applied as long as possible before the rules in the next layers. Moreover, they distinguish between *deletion* and *non-deletion* layers. Rules in deletion layers should delete at least one item. Rules in non-deletion layers do not delete any item, but they have *negative application conditions* to prohibit an infinite number of applications of the same rule.

9.4.3 Completeness

Definition 48 (completeness)**.** *A graph rewriting system is complete iff for every graph G and every rule $\rho \in \mathcal{R}$ there is a well-defined graph H such that $G \overset{\rho}{\Rightarrow} H$, i.e. the direct derivation by a rule ρ is computable for every graph G.*

[6]Note that *layered grammars* [EEdL$^+$05] has no relation with the layered modal logics introduced and studied in Chapter 8.

This property seems hard to be fulfilled since usually a rule is not applicable on some graphs for example those not having a redex matching the left-hand side of the rule. However, this property can be fulfilled if the rule application is relaxed, so that if a rule ρ is not applicable to a graph G, then we can anyway achieve the derivation $G \overset{\rho}{\Rightarrow} G$. It remains to make sure that the rules are defined in a way that guarantees their application, i.e. to make sure during the compile-time that every rule satisfies the gluing condition if it is defined in DPO, or that the system knows how to proceed in case of problematic rules when they are defined in SPO.

9.4.4 Locality

The rule-base nature of graph rewriting systems ensures a certain degree of locality as the application of a rule manipulates locally the occurrence of the rule in the host graph. The locality leads to low time-cost for rules application. Nonetheless, the definition of global states or variables, left-hand conditions looking for an ancestor of a node or describing disconnected subgraph patterns may weaken this property.

9.4.5 Parallel and sequential independence

Two rules are *sequentially independent* if they can be interleaved when applied sequentially leading to the same result. They are *parallel independent* or *concurrent* if they can be applied in arbitrary order on the same graph.

Definition 49 (sequential independence)**.** *Two rules ρ_1 and ρ_2 are sequentially independent iff for very derivation $G \overset{\rho_1}{\Rightarrow} G_1 \overset{\rho_2}{\Rightarrow} H$ there is an equivalent derivation $G \overset{\rho_2}{\Rightarrow} G_2 \overset{\rho_1}{\Rightarrow} H$.*

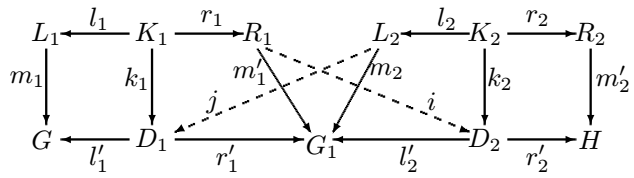
Remark 16. $G_1 \overset{\rho_1}{\Rightarrow} G_2 \overset{\rho_2}{\Rightarrow} G_3$ are sequentially independent iff $G_1 \overset{\rho_1^{-1}}{\Leftarrow} G_2 \overset{\rho_2}{\Rightarrow} G_3$ are parallel independent, where ρ_1^{-1} is the inverse rule of ρ_1. Thus specifying parallel independence is done once specifying the sequential independence is.

Characterizing the parallel or sequential independence amounts to specifying the conditions under which these properties hold.

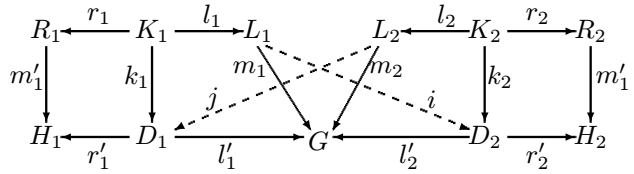
Informally, two rules are sequentially independent if the match of the latter one does not depend on elements generated by the former one, and the latter one does not delete items that have been accessed by the former one.

To formalize this condition, let us consider two rules $\rho_1 = (L_1 \overset{l_1}{\leftarrow}_1 K_1 \overset{r_1}{\rightarrow} R_1)$ and $\rho_2 = (L_2 \overset{l_2}{\leftarrow}_2 K_2 \overset{r_2}{\rightarrow} R_2)$, as usually defined in the DPO or the HLR adhesive categorical approaches (Figure 9.7(a)). The two derivations $G \overset{\rho_1}{\Rightarrow} G_1 \overset{\rho_2}{\Rightarrow} H$ are sequentially independent if all nodes and edges in the intersection of the comatch $m'_1 : R_1 \rightarrow G_1$ and the match $m_2 : L_2 \rightarrow G_1$ are gluing items w.r.t. both rules, i.e.

$$m'_1(R_1) \cap m_2(L_2) \subseteq m'_1(r_1(K_1)) \cap m_2(l_2(K_2)).$$



(a) sequential independence



(b) parallel independence

Figure 9.7: Parallel and sequential independence.

On the other hand, the derivations $G \overset{\rho_1}{\Rightarrow} H_1$ and $G \overset{\rho_2}{\Rightarrow} H_2$ (Figure 9.7(b)) are parallel if all nodes in the matches are gluing items with respect to both rules, i.e.

$$m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2)).$$

Hence the following characterization (proved in [EEPT06] Fact 3.18):

Definition 50 (characterization of parallel and sequential independence)**.** *Two derivations $G \overset{\rho_1}{\Rightarrow} H_1$ and $G \overset{\rho_2}{\Rightarrow} H_2$ are parallel independent iff there exist morphisms $i : L_1 \to D_2$ and $j : L_2 \to D_1$ such that $l'_2 \circ i = m_1$ and $l'_1 \circ j = m_2$ (see Figure 9.7(a)).*

Two derivations $G \overset{\rho_1}{\Rightarrow} G_1 \overset{\rho_2}{\Rightarrow} H$ are sequentially independent iff there exist morphisms $i : R_1 \to D_2$ and $j : L_2 \to D_1$ such that $l'_2 \circ i = m'_1$ and $r'_1 \circ j = m_2$ (see Figure 9.7(b)).

This shows how such properties are easy to be defined in the categorical approaches and justifies their use as a theoretical basis for defining graph rewriting, for instance, our graph rewriting system LoTREC.

9.4.6 Confluence and convergence

If two rules ρ_1 and ρ_2 can be applied in parallel on a given graph G, then the application of both rules (in any order) results in a unique (up to isomorphism) graph H. This means that there exists a rule, denoted by $\rho_1 + \rho_2$, that represents the composition of ρ_1 and ρ_2, consisting essentially of their disjoint union and denoting the *parallel* application of ρ_1 and ρ_2 (see Figure 9.8). To formalize this we use the notation $G \overset{\rho;m}{\Rightarrow} H$ to explicit the morphism used to compute the match during the rule application.

Theorem 5 (Local Church-Rosser)**.** *Given two parallel independent derivations* $G \overset{\rho_1,m_1}{\Rightarrow} H_1$ *and* $G \overset{\rho_2,m_2}{\Rightarrow} H_2$*, there is a graph* H *together with derivations* $H_1 \overset{\rho_2,n_2}{\Rightarrow} H$ *and* $H_2 \overset{\rho_1,n_1}{\Rightarrow} H$ *such that* $G \overset{\rho_1,m_1}{\Rightarrow} H_1 \overset{\rho_2,n_2}{\Rightarrow} H$ *and* $G \overset{\rho_2,m_2}{\Rightarrow} H_2 \overset{\rho_1,n_1}{\Rightarrow} H$ *are sequentially independent.*

Proof. see [EEPT06], Theorem 5.1. □

$$\begin{tikzcd}
 & G \arrow[dl, "\rho_1, m_1"'] \arrow[dr, "\rho_2, m_2"] \arrow[dd, "\rho_1 + \rho_2"] & \\
H_1 \arrow[dr, "\rho_2, n_2"'] & & H_2 \arrow[dl, "\rho_1, n_1"] \\
 & H &
\end{tikzcd}$$

Figure 9.8: Parallelism and confluence.

In such case, we say that the rewriting system is *locally confluent.* It is said to be (globally) *confluent* if every pair of rules are independent.

Definition 51 (Confluence)**.** *A graph rewriting system is confluent iff for every derivations* $G \overset{*}{\Rightarrow} G_1$ *and* $G \overset{*}{\Rightarrow} G_2$ *there is a graph* H *such that* $G_1 \overset{*}{\Rightarrow} H$ *and* $G_2 \overset{*}{\Rightarrow} H$*.*

This property dictates that no matter how we diverge from a given graph, there is always a way of joining derivations at a common result graph. Hence, a confluent rewrite system delivers a *unique* normal form for all sequences of rewrite rules application [DJ90].

However, in practice, the confluence of a system is studied according to a specific strategy and/or constraints on rules application. Note also that the problem of determining wether a graph rewriting system is confluent is an undecidable problem [DJ90].

Definition 52 (Convergence)**.** *A graph rewriting system is convergent iff it is terminating and confluent.*

9.4.7 Complexity

Studying the complexity of a graph rewriting system is reducible to studying the complexity of the two steps (Sections 9.2.1 and 9.2.2) of a rule derivation in that system.

Since applying a rule consists in adding, deleting or modifying elements that are as many as the size of the right-hand side graph of the rule (i.e. at most $|R|$), this step is achieved in general in polynomial time.

However, the pattern matching process is known to be NP-complete. In fact, it consists in matching the $|L|$ elements of the left-hand side graph of the rule with the $|G|$ elements of the host graph. Thus the time cost is $O(|G|^{|L|})$.

The effect of this time cost is proportional to the number of established unsuccessful pattern matching processes. Thus misdealing with the additional

applicability conditions may increase in practice the influence of this time cost on the overall performance of the underlying rewriting system.

Conclusion

In this chapter we gave an overview of graph rewriting. We showed various graph structures that differ in their ability to express richer data. We introduced the categorical approach to define graph rewriting rules. Then we showed how rule definitions may lead to non well-defined graph and how to avoid this.

In the end we gave some of the theoretical properties that can be inherited from some graph rewriting approaches, which have already established results certified by the graph rewriting community.

Chapter 10

LoTREC: a graph rewriting tool for model construction in modal logic

Introduction

In the last chapter we gave a survey on the various approaches used to define graphs and graph transformations, we establish in this chapter the link between these approaches and LoTREC.

We do this by discussing some of the specificities of our tool, namely:

1. the kind of graphs it can deal with,
2. the theoretical approach on which rules definition lie,
3. and the way the rules are compiled and applied.

These specificities of LoTREC are successively addressed in sections 10.1, 10.2 and 10.3. First, we specify the structure used to embed the premodels as attributed graphs. Then we show how the computations performed on the premodels are defined as graph rewriting rules, and how these definitions conform to the HLR approach (Section 9.3.4). We also show some of the restrictions that we impose on the definition of the rules, and we discuss the underlying reasons for these constraints.

After that, we present the language of strategies definable and used in LoTREC, in contrast with the strategy languages of other tools.

Finally, according to these specificities, we discuss in Section 10.4 some of the properties that are acquired by a model construction method defined in LoTREC, once seen as a graph rewriting system.

10.1 Graph structure

Graphs are usually defined in the graph rewriting literature as two (disjoint) sets of elements (or objects). One contains the nodes and the other contains the edges of the graph. As shown in Section 9.1, we may add some other sets of elements to this basic graph structure in order to enrich it. For example, we may annotate the nodes and/or edges with labels to make them more significative (see Section 9.1.1). We may also annotate them by attributes, so we can perform some computations on these attributes during the transformation of the graphs (see Section 9.1.2).

In LoTREC, graphs are meant to represent Kripke's models (see Definition 4), although the users may have their own interpretations of the graphs as sequent trees, automaton or other data structures.

Thus in general, nodes represent the possible worlds, and edges represent the accessibility relations between these worlds. Hence, we should be able to label a node representing a possible world with a set of formulas that are supposed to be true in that world. Similarly, we should be able to label an edge representing an accessibility relation with a formula to express the name of that relation (see Figure 10.1).

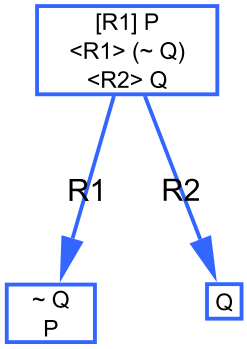
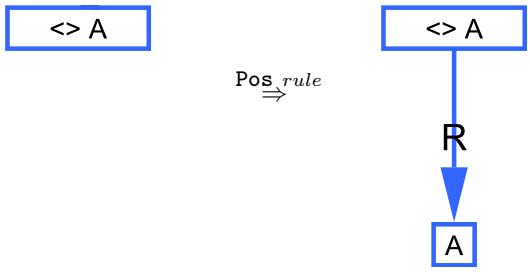


Figure 10.1: A premodel constructed in LoTREC.

Furthermore, we would like to be able to perform some computations on these formulas during the model construction process. We can see this, for example, when defining a rule that, given a node labeled with the formula `<>A`, creates a new node labeled with the *subformula* `A` (see Figure 10.2); or when defining a rule that, when the formulas `A` and `B` are found in a given node, adds to the same node the formula `A & B` obtained by applying the operator `&` to the formulas `A` and `B`. Thus formulas should not be treated as simple labels, but rather as *attributes* of *type* "formula", on which we can define a set of *operations* within the definitions of the rules.

Consequently, it seems that *attributed graphs*, defined in Section 9.1.2, are the most suitable structure to define graphs of LoTREC.



(a) left-hand side (LHS) (b) right-hand side (RHS)

Figure 10.2: During the application of the Pos rule, the formula A is calculated from the formula $\Diamond A$ using the subformula operator Sub.

10.1.1 Formulas as attributes

As stated in Chapter 2, formulas of a given logic L can be defined inductively on its set of atomic propositions ($\mathcal{P}$) and its set of connectors ($Conn$) (also called *operators*). This definition says that:

- any atomic proposition $P \in \mathcal{P}$ is a formula,
- for any connector $c \in Conn$, given n formulas $A_1, \ldots, A_n$, where $n = arity(c)$, $c(A_1, \ldots, A_n)$ is also a formula,
- and that there are no other formulas in the logic L.

To use the formulas as attributes (such as in [EEPT06]), they should be defined as terms upon an algebra and an algebraic signature (Signatures and algebras are introduced in Appendix A). Nevertheless, we can show that when a set of connectors is defined in LoTREC, an algebraic signature can be automatically generated. Hence, an algebra can be built upon this signature, whose terms are the formulas of the underlying logic.

We call these terms *expressions* to give the intuition that they can be used to represent more general types of attributes, other than formulas. In the sequel, we use both "formula" and "expression" without making any distinction.

Indeed, defining in LoTREC a set of connectors $Conn$ (as explained in Chapter 2) is equivalent to defining a special signature EXP (which stands for expressions) with:

- only one sort exp,
- a set of constant symbols K, which is meant to be the set of atomic propositions $\mathcal{P}$,
- and a set of operation symbols OP, which is effectively the set of user-defined connectors $Conn$.

Definition 53 (expression signature)**.** *Given a logic* L *with the set of atomic propositions* $\mathcal{P}$ *and the set of connectors* $Conn$*, we define the corresponding* expression signature EXP *as follows:*

$$\begin{array}{llll} EXP & = & & \\ sorts & : & exp & \\ operators & : & e & : \rightarrow exp, \text{ for all } e \in \mathcal{P} \\ & & op & : \underbrace{exp, \ldots, exp}_{arity(op) \text{ times}} \rightarrow exp, \text{ for each } op \in Conn \end{array}$$

Example 13. Suppose that we are defining in LoTREC the logic CPL with the set of atomic propositions $\mathcal{P}$ and the set of (boolean) connectors $Conn = \{\texttt{not}, \texttt{and}\}$, where $arity(\texttt{not}) = 1$ and $arity(\texttt{and}) = 2$. The following signature of expressions becomes the default one in LoTREC:

$$\begin{array}{llll} EXP & = & & \\ \text{sorts} & : & exp & \\ \text{operators} & : & e & : \rightarrow exp, \text{ for all } e \in \mathcal{P} \\ & & \texttt{not} & : exp \rightarrow exp \\ & & \texttt{and} & : exp, exp \rightarrow exp \end{array}$$

Beside the auto-generation of a signature given a set of connectors, a default algebra on these expressions is also automatically generated.

Definition 54 (expression algebra)**.** *Let* EXP *be the expression signature of a logic* L*, and let* $\mathcal{F}or$ *be the set of formulas of* L*. The corresponding* expression algebra *is defined as follows:*

$$\begin{array}{lll} A_{exp} & = & \mathcal{F}or \\ e_A & = & e \in A_{exp}, \text{ for all } e \in \mathcal{P} \\ op_A & : & \underbrace{A_{exp} \times \ldots \times A_{exp}}_{arity(op) \text{ times}} \rightarrow A_{exp}, \text{ for each } op \in Conn \\ & & (e_1, \ldots, e_n) \rightarrow op(e_1, \ldots, e_n), \text{ where } n = arity(op) \end{array}$$

The auto-generated algebra does not infer any semantics. In LoTREC, the semantics are embedded in the rewriting rules defined by the user, and there is no effective computation associated to the operators by interpreting their syntactical definitions as data signatures or algebras.

Example 14. If connectors `not` and `and` are defined as in Example 13, the default algebra automatically defined in LoTREC is the following:

$$
\begin{array}{lcl}
A_{exp} & = & \mathcal{F}or \\
e_A & = & e \in A_{exp} \\
\texttt{not}_A & : & A_{exp} \to A_{exp} \\
 & & e \to \texttt{not}(e) \\
\texttt{and}_A & : & A_{exp} \times A_{exp} \to A_{exp} \\
 & & (e_1, e_2) \to and(e_1, e_2)
\end{array}
$$

Formulas are then defined as terms upon this algebra, exactly as indicated in Section A.1 of Appendix A.

Example 15. According to the signature and algebra defined in the above examples, and for $P, Q \in K$, the expression $\texttt{and}(P, \texttt{not}(Q))$ becomes a recognizable term in LoTREC (i.e. the formula $P \wedge \neg Q$ becomes recognizable in LoTREC).

Other types of attributes

In some cases, we need to mark special nodes with additional meta-data, called *marks*, during the transformations. This is the case for example if, in a given rule, we want to annotate a node by a special mark (such as the `Loop_Node` mark used to indicate a given node is equal to an ancestor node in Chapter 5 Section 5.1).

We have also the possibility to annotate formulas with marks as well. For example, we can mark a given formula as `True` or `False` (Chapter 6), or as `active` or `inactive` (Chapter 4 Section 4.6).

Consequently, in addition to expressions, we also have to deal with different *types* of attributes, for instance the marks presented above. However, since they only consist of constant values with no operations on them, we consider them as simple labels not as attributes, and thus we do not include them in our definitions here.

In future versions of LoTREC, it is possible that we will need to use other types of attributes in order to define model construction for some logics. Namely, we need to use the integers to define the *cardinalities* of the accessibility relations for some description logics, which have to be considered as *integers*. Such extensions can be easily done in the future since:

1. practically, LoTREC can deal with any type of attributes defined as objects of any Java class (which is already done in the general purpose rewriting system AGG [Tae99]),

2. and theoretically, after such extensions, the rewriting system of LoTREC will still be captured by the general attributed graph transformation framework, as long as these new attribute types can be defiend via algebraic signatures and algebras.

To conclude,recall that defining a new logical language in LoTREC amounts to the definition of a new syntax of formulas. As such we can say that it is easy to define a new logical language in LoTREC since this can done by simply

defining the appropriate set of connectors and their arities. As a consequence of this definition, formulas, defined as terms upon these connectors, become the only recognizable attributes in LoTREC, and graphs that are attributed by these formulas become the only recognizable graphs in LoTREC. Also LoTREC can always be extended to take into account new types of attributes.

10.1.2 The type graph of LoTREC

In a given rewriting system, a type graph (Definition 41) is usually fixed, thus every graph instance defined or handled by the system should be dotted with a graph morphism to this type graph. This ensures that graph instances are well defined and satisfy the constraints specified in the underlying type graph.

Unlike general purpose rewriting systems which accept various user-defined type graphs, LoTREC has a fixed type graph which has only one type of nodes, one type of edges and one type of attributes which is *Expression*.

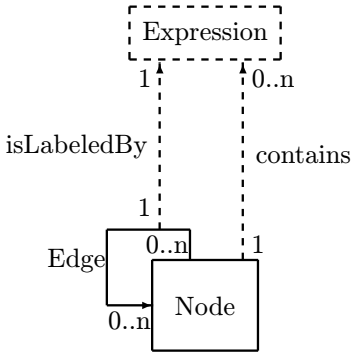


Figure 10.3: The basic type graph of LoTREC, shown as an E-graph.

The type graph of LoTREC is given in Figure 10.3. According to this type graph:

- nodes, edges and formulas (i.e. *expressions*) of an instance graph should be separated in three disjoint sets;
- a node may have no or many successor and/or parent nodes, and it may have no or many formulas;
- an edge links exactly one *source* node to one *target* node[1] and it is labeled by exactly one formula;
- thus a formula is either labeling exactly one node or exactly one edge;
- two (not necessarily different) nodes can be linked by two or more edges, provided that the edges are labeled with different formulas;

[1]This can be considered as a multiplicity constraint on the *Edge* type.

- and, there are no additional constraints on the graph structures in LoTREC.

This type graph can be extended to take into account other attribute types. For instance, we show in Figure 10.4 how to add a new type called *Mark* to this type graph, in order to take into account the marks (of nodes and expressions) and their constraints.

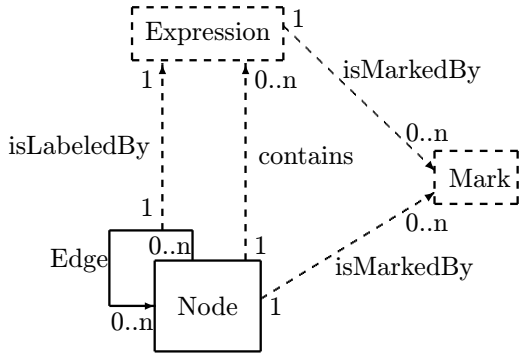


Figure 10.4: The complete type graph of LoTREC.

According to this complete type graph, an instance graph should also satisfy two additional conditions, namely, a node or a formula can be marked with one or more marks, whereas a mark is only associated with exactly one node or exactly one formula.

10.1.3 Premodels as graph instances

The graph instances in LoTREC are the premodels. They are defined as typed attributed graphs according to Definition 42. Thus, they are graphs attributed by formulas (resp. and marks) w.r.t. the specification described by the type graph of Figure 10.3 (resp. 10.4).

Example 16. Figure 10.5 shows how the premodel of Figure 10.1 is defined as an attributed graph typed over the type graph of LoTREC.

10.2 LoTREC's rules as graph rewriting rules

Once the syntax of a given logic is defined via a set of connectors, the rules in LoTREC are meant to give the semantics of this logic. In fact, the semantics can be encoded in graph transformation rules as long as the underlying logic has a standard Kripke semantics (as shown throughout the Chapters 3 to 7).

In this section, we show how these rules are defined in LoTREC, and how this definition fits into the theoretical framework of the adhesive HLR approach (Section 9.3.4). Then we give some of the constraints imposed on their definitions and the special way in which they are applied.

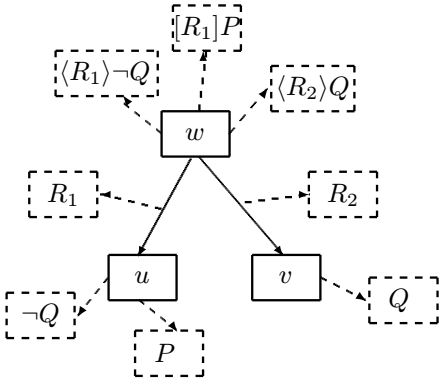


Figure 10.5: The premodel of Figure 10.1 represented as a node- and edge-attributed graph.

10.2.1 Definition

The definition of rules in LoTREC follows the single-pushout approach (see Section 9.3). Thus a rule $\rho = (L, R)$ is represented by a left-hand side (LHS) graph L, a right-hand side (RHS) graph R, together with a partial graph morphism $\rho : L \to R$. However, this morphism can be considered as a span of two total graph morphisms, with the domain of the morphism as the gluing graph, i.e. $L \xleftarrow{l} dom(\rho) \xrightarrow{r} R$, the way DPO rules are denoted (see Section 9.3.2). Both r and l belong to the special class of monomorphisms $\mathcal{M}$ (Definition 45), thus graph transformations in LoTREC become special constructions from the adhesive HLR category of typed attributed graphs and their morphisms.

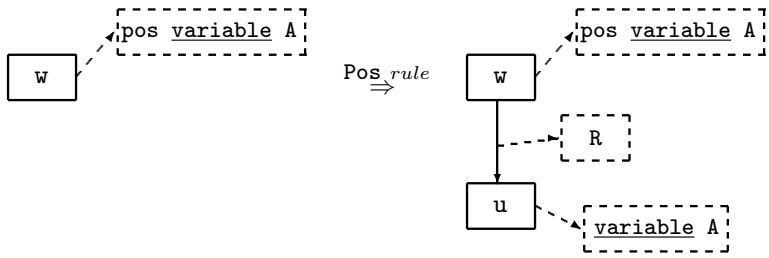
This approach forms a suitable framework for our rules, since it allows us to define computations on the attributes of nodes and edges , as well as the transformations to be made on the graph structure.

Example 17. Let us consider the modal logic K. Let the connector `pos` be the mono-modal operator $\Diamond$, with $arity(\texttt{pos}) = 1$. The rule `Pos`, defined in Figure 10.6, is applied on any node `w` attributed by a formula of the form $\Diamond A$ (since `variable A` is to be instantiated by a variable-free formula during the rule applicability check).

The application of `Pos` consists in:

- creating a new node `u`,
- linking `u` to `w`,
- adding to `u` the formula A assigned to the `variable A` (computed as a subformula of $\Diamond A$).

Note that `R` is a constant value, and that `w` and `u` are variables to be instantiated. However, we omit the `variable` keyword in front of the nodes, since it is not possible to use constant node symbols in LoTREC's rules.



(a) left-hand side (LHS) (b) right-hand side (RHS)

Figure 10.6: The LHS and RHS graphs of the rule in Figure 10.2 represented as node- and edge-attributed graphs.

Negative application conditions (NACs)

As for the other graph rewriting tools, we can add to the rules in LoTREC as many Negative Application Conditions (NACs) as needed. These conditions specify, for a given rule, which graph objects (nodes, edges, ...) should not be present in the host graph in order to apply the rule.

Example 18. We may add to the rule `Pos` of Example 17 a NAC stating that the rule is applicable on every node `w` if it does not have any successor node `u` by the relation `R` (see the gray part of the graph in Figure 10.7).

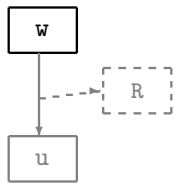


Figure 10.7: Negative Application Condition (NAC).

Constants and variables

The LHS and NACs of a rule may contain constant or variable attribute values (such as the `w` node and `variable A` in Example 17). NACs are allowed to contain the LHS only partially. The scope of a variable is its rule, i.e. each variable is globally known in its rule. For example, the RHS may contain variables declared in the LHS. Multiple usage of the same variable is allowed and can be used to express equality of values.

New variables figuring only in the RHS are allowed if they are designating node identifiers (such as the `u` node in Example 17). This means that new nodes have to be created by the rule. Whereas new variable attributes are not allowed in the RHS. Only attributes with constants can be added to the RHS (such as `R`

in Example 17), and only attributes with variables that are already figuring in the LHS (such as `variable A` in Example 17), thus instantiated during the rule applicability check, can be used in the RHS.

Empty LHS and empty RHS

The LHS and RHS could be empty set graphs. A rule with an empty RHS does not alter the host graph when applied, i.e. it has no effect on the host graph regardless of the fact that it is applicable on it or not.

A rule with an empty LHS is applicable on any host graph (hence, it is applicable infinitely often when called repeatedly). Such a rule is usually defined to be applied just once on the premodels. For example, the rule `Build_Premodel` is called at the beginning of the model checking method to build and initialize the checked models (Chapter 6). (See also the rule `Build_Greeting_Graph` in Appendix B.1).

LHS connectedness

In LoTREC, the LHS graph should be connected: two nodes of the LHS graph are necessarily connected either by being directly linked by an edge, or indirectly linked to other connected nodes.

Example 19. Figure 10.8 shows an example of an ivalid rule definition where the LHS graph is disconnected.

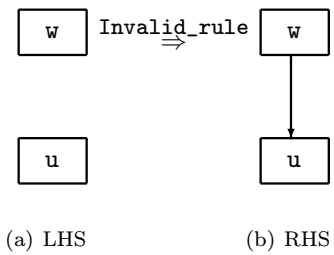


Figure 10.8: An invalid rule definition, since the LHS is disconnected.

This restriction follows from the nature of the application domain of LoTREC in logic, where the constructed models are usually connected graphs. This also allows to drastically enhance the performance when achieving the pattern matching process.

However when needed, we can simulate connectedness between all the nodes of the graph by creating a universal relation linking them together or linking them to a common specific node (such as done for S5 in Chapter 4).

Non-determinism and multiple RHSs

A rule may have one, two or more RHS graphs. This allows the users to define different alternative sets of transformations that have to be made on the host graph where the LHS graph of the rule is matched.

Having alternative transformations is vital to encode *non-deterministic choices* in logic (ex. disjunctions). In fact, since in LoTREC semantics are encoded in the rules, the non-determinism is also embedded in the rules definition.

However, this changes nothing to the formal definition given above. In fact, if a rule is defined by a left-hand side graph L and a set of graphs on the right-hand side $R_1, \ldots, R_n$, then it can be viewed as a set of transformations $L \to R_1$, $\ldots$, $L \to R_n$, where each transformation is exactly defined as a rule with only one RHS, without any further constraints or restrictions.

Example 20. For example, a disjunction $A \vee B$ is true iff one of its disjuncts, A or B, is true. Suppose that we have a graph node with the formula $P \vee Q$. To implement the semantics of the $\vee$ operator with a single rule, we define a rule that has two RHS graphs: in the first one we add A to the node, in the other one we add B (see Chapter 3, Section 3.1). The result will be two graphs, which are both explored in the rest of the rewriting process.

10.2.2 A special language of conditions and actions

LoTREC does not have a graphical interface to define the rules by drawing their LHS, NACs and RHS graphs. Instead, LoTREC offers a high-level textual language of *conditions* and *actions* to describe these graphs. This language is simple and easily accessible to non-computer scientists, such as students and researchers in philosophy and logic.

Example 21. The `Pos` rule of Example 17 (Figures 10.2 and 10.6), is defined in LoTREC as follows:

```
Rule Pos
 hasElement w pos variable A

 createNewNode u
 link w u R
 add u variable A
End
```

We notice that we use the keyword `variable` to indicate that `A` is a variable that has to be instantiated, whereas `R` is a constant symbol. Nodes identifiers, such as `w` and `u`, are always considered as variables.

This definition says that wherever a node `w` holds a formula of the form `pos variable A` (i.e. $\Diamond A$ where A is an arbitrary formula), then the rule should create a new node `u`, link it to that `w` by an edge labeled by `R` and add to it the subformula A (assigned to `variable A`). Thus, this definition reflects exactly what the graphical definitions of the `Pos` rule (Figures 10.2 and 10.6) are meant to say.

In LoTREC, conditions are used to represent graph patterns and are meant to describe the LHS of the rules. For example, `hasElement node formula` states the existence of a formula in a given node and `isLinked n n' R` states that two nodes are linked. It is also possible to define NACs in this language, such as `hasNotElement node formula` and `isNotLinked n n' R`.

Actions are used to describe the RHS graph of rules. However, they define the RHS only partially, by specifying how it is calculated from the LHS graph. In example 21, the three given actions define the changes to be made on the LHS graph of the rule `Pos` in order to obtain its RHS graph.

Throughout the chapters 3 to 7 we use most of the conditions and actions that are predefined in LoTREC[2].

In the following two sections, we show how these conditions are checked and verified, and how these actions are then applied.

10.2.3 Applicability

As shown in Section 9.2.1, the check of rule applicability, also known as the pattern matching process, consists in finding a subgraph of the host graph G that matches the LHS of the rule, i.e. finding a total morphism $m : L \to G$.

Matching the structure and the attributes

In LoTREC, the LHS graph is to be instantiated first, then NACs are simply checked on the found graph instance. Besides matching the graph structure, the attributes annotating this structure are also matched. The attributes of the LHS are matched by comparing constant values or by instantiating variables.

Instantiating variables consists in finding a *substitution* for these variables that assigns constant values from the instance host graph. When a substitution fails, another substitution should be considered. Thus, the instantiation of variables is only a subprocess recursively called in the whole pattern matching process.

Instantiation of complex attributes

The rewriting engine of LoTREC does not use existing tools and was built from scratch. The main reason behind this is the integration of the attributes matching within the whole matching process. In fact, in general purpose rewriting tools, such as AGG [Tae99], the instantiation of the variables appearing in the attributes is only done for basic data structures, such as integers, reals, strings and some other Java types. Hence, it is not possible to define or integrate the instantiation of complex attributes (such as the expressions used in LoTREC to encode logical formulas) in the whole pattern matching process of these generic rewriting tools.

[2] A complete (but out-of-date) list of conditions and actions is available in [Sah04].

A special notion of applicability

In general, we say that a rule is applicable if we succeed in finding a morphism m and a subgraph $D \subseteq G$ such that $m(L) = D$. However, when considered in the global context of repeated applications, it is often desired not to apply the same rule on the same subgraph D twice, by considering the same morphism. Otherwise, it is clear that the rewriting process will not terminate.

To avoid rules infinite reapplication, the users have to exhaustively add an appropriate set of NACs to the definition of every rule. Nonetheless, we avoid such inconvenience in LoTREC by taking these NACs into consideration by default. This is automatically achieved by the *event-driven pattern matching* technique, introduced and formalized in the next chapter. In addition, this technique helps in reducing the time-cost of the whole pattern matching process.

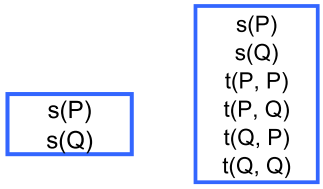
Non-injectivity of matching morphisms

It is possible to consider a non-injective matching morphism $m : L \to G$ in LoTREC, i.e. we may instantiate two variables of the LHS with the same value. This allows to captures all possible matches of a given pattern.

Example 22. Let us consider a logic with two connectors s and t of arities 1 and 2 respectively, and where for every formulas A, B, if $s(A)$ and $s(B)$, then we have $t(A, B)$.

To define the semantics of this logic, we use the following rule:

```
Rule Combinatory
 hasElement node s variable A
 hasElement node s variable B

 add node t variable A variable B
End
```



(a) host graph (b) result graph

Figure 10.9: The application of the rule `Combinatory`, using the default non-injective morphism to match its LHS with the host graph.

Let us consider a graph with a single node containing two formulas, $s(P)$ and $s(Q)$, where $P, Q \in \mathcal{P}$ (i.e. are atoms, or constant formulas), as shown in Figure 10.9(a). The application of the rule `Combinatory` on this graph results in the graph of Figure 10.9(b).

Since matching morphism is non-injective, the variables `A` and `B` can be mapped to the same value P. That is why the rule adds $t(P, P)$ and $t(Q, Q)$ to the node of the result graph.

However, we may restrict our mapping to be injective if needed. This can be done by adding a convenient set of NACs to the rule.

Example 23. To avoid non-injective matching of the variables `A` and `B` such as done in Example 22, we may add to the rule `Combinatory` the NAC "`areNotEqual variable A variable B`", and we obtain the following rule:

```
Rule Combinatory_Injective
 hasElement node s variable A
 hasElement node s variable B
 areNotEqual variable A variable B

 add node t variable A variable B
End
```

The application of this rule on the same host graph (Figure 10.9(a)) yields to the graph of Figure 10.10.

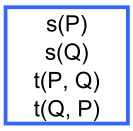


Figure 10.10: The result of the application of `Combinatory_Injective` on the host graph of Figure 10.9(a).

Detection of commutative patterns

It is sometimes required to consider two graph patterns as equivalent according to a certain permutation in the value assignment of some variables appearing in these patterns.

Example 24. Suppose that for all formulas A, B of the logic defined in Example 22, we have $t(A, B)$ if and only if $t(B, A)$. And suppose that we are interested in deducing the minimum set of equivalent formulas of the form $t(A, B)$. Then we have to change the rule of Example 23 to avoid capturing the patterns that are equivalent modulo commutativity of `A` and `B`.

In the previous version of LoTREC, it was possible to declare a rule as commutative by setting a Boolean flag to true or false. By default, this flag is set to false. When it is set to true, two graph patterns are judged by the rule to be equivalent modulo commutativity if there is a permutation in the values assigned to *all* their variables. Whereas in some cases, two graph patterns are judged to be equivalent if *only some* of their variables values are permutative.

Hence, in our current version of LoTREC, we adopt a new solution, consisting in defining in the rule itself wether or not it should consider two patterns equivalent. We do this by adding NACs that prevent the rule from being applied twice when an equivalent pattern is found.

Example 25. We give here a variant of the rule of Example 23, which takes into consideration the equivalence criteria of Example 24:

```
Rule Combinatory_Injective_Commutative
 hasElement node s variable A
 hasElement node s variable B
 areNotEqual variable A variable B
 hasNotElement node t variable A variable B
 hasNotElement node t variable B variable A

 add node t variable A variable B
End
```

Nevertheless, applying the rule `Combinatory_Injective_Commutative` of Example 25 on the graph of Figure 10.9(a) would give the graph of Figure 10.10, i.e. the same result as applying the rule `Combinatory_Injective` of Example 23. The reason and the solution of this problem are explained in the next subsection.

Applying a rule on one occurrence

In case the LHS of a given rule can be matched with two or more subgraphs of the host graphs (also called *occurrences*), LoTREC applies the rule on all these occurrences at once and in parallel. To predict this behavior, the user has to imagine that there is implicitly a kind of universal quantifier on the variables figuring in the conditions of the defined rules.

However, we may force LoTREC to apply a given rule on only one occurrence of its LHS by adding the keyword "`applyOnce`" before the name of the rule where it is called in the strategy (as shown later in Section 10.3).

Example 26. Applying the rule of Example 25 on the host graph of Figure 10.9(a), by calling it in the strategy after the `applyOnce` keyword, results in the graph of Figure 10.11.

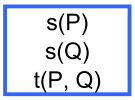


Figure 10.11: The graph resulting from the application of the rule of Example 23 on the host graph of Figure 10.9(a), called after the strategy keyword `applyOnce`.

10.2.4 Application

After finding a matching subgraph for the LHS of a rule in a host graph, the rule application consists in taking out this subgraph and replacing it by the RHS of the rule. In fact, since a match $m : L \to G$ is a total morphism, any graph element e (a node or an edge) in L has a proper image element $m(e)$ in the host graph G. Now, if e also has an image $\rho(e)$ in the RHS graph R, its corresponding element $m(e)$ in the host graph is *preserved* during the application of ρ; otherwise it is *removed*. Elements appearing exclusively in the RHS graph without an original element in the LHS graph are *newly created* during the rule application. Finally, the elements of the host graph which are not covered by the match are not affected by the rule application at all; they form the *context* in which the graph transformation is done.

In addition to manipulating the nodes and edges of a graph, a rule may also perform attributes computations. During the rule application, expressions are evaluated with respect to the instantiation of variables induced by the current match.

On the DPO gluing condition

All the DPO conditions (Definition 44), discussed in Section 9.3.2, are respected in LoTREC. In fact, there are no dangling edges since nodes cannot be removed and edges are only created between already existing nodes. This guarantees that the result of the application of any rule will still be a well-defined graph.

However, in future versions, we should always keep an eye on preserving these conditions when extending the language of conditions and actions. For example, if a new remove-node action is to be implemented, than all dangling edges should be automatically deleted. Similarly, if we implement a new action `merge n1 n2` which merges two nodes, than LoTREC should implicitly avoid the eventual side effects of such procedure: it has to make sure that the matching $m : L \to G$ is injective, i.e. `n` $\neq$ `n'`, and it should make sure that all the incident edges are systematically and appropriately deleted and/or redirected.

Deletion actions and monotonic rules

If the RHS of a rule in LoTREC consists only in *adding* nodes, edges, expressions and marks, then we call it *monotonic*. Rules in LoTREC are often monotonic. This is due to the application domain of LoTREC in logic. In fact, we never needed to remove a node, an edge or an expression in the rules of all our predefined logics. Thus, there are no such actions available in the language of LoTREC. Moreover, we do not need to define such actions, since we may use our marking technique to annotate a given element as deleted, so the rules just ignore it, as if it were physically deleted.

However, there are only two non-monotonic actions, corresponding to the `unmark` and `unmarkExpressions` keywords, which can be used to remove a mark of a node or a mark of an expression. Nonetheless, these actions were only used in a special model checking algorithm, where some expressions are marked as

`True` at a given step of the rewriting process, then they become marked as `False` at later steps.

Summary

To summarize, the effect of a rule application in LoTREC is

- complete: any change specified in the rule is actually performed during its application;
- minimal: nothing more is done that what is specified in the rule;
- local: only the subgraph of the host graph that was matched to the LHS of the rule is affected.

By respecting the gluing condition and using distinguished attributed graph monomorphisms in the definitions of the rules, LoTREC graph transformations correspond strongly to the typed attributed graph transformations defined in Section 9.3.4.

10.3 Strategies

Graph rewriting systems use what is called *control structures* in the rewriting literature, or *strategies* in LoTREC, to create an imperative ordering between a set of rules. This ordering is used to control and choose between the possible sequences of their successive applications on a given host graph.

In some systems, strategies are defined with simple structures. For example, in AGG[Tae99], defining a strategy amounts to stratifying a set of rules in an ordered set of different subsets called *layers*. The rules of a given layer are applied as much as possible, before starting the application of the rules of the next layer, and without being able to re-apply the rules of a previous layer. In AGG, we cannot define sub-layers, neither repeat the application of some layers.

In some other systems, such as PROGRES [Sch97], strategies are more expressive, but get more and more complex. In [ZS92], the authors of PROGRES define a powerful but complicated language of what they call *control structures*. In this language, we can repeat the application of a set of rules, we can define If-Then-Else structures and we can even backtrack on the choices of already applied rules, by defining a notion of *successful choice* according to some criteria.

In LoTREC we have our own strategy language as shown in what follows.

10.3.1 Aimed semantics

Since non-determinism is taken into account in the rules by defining multiple RHSs, we do not need to have complex strategies in LoTREC, such as those of PROGRES. However, they should be slightly more elaborated than the simple layering strategies of AGG.

In fact, we would like to be able to:

1. apply a rule by default on all possible occurrences of its LHS in the host graph,

2. apply a rule on only one occurrence when needed,

3. stratify the rules in different ordered sets, in order to be able to:

   (a) apply the first applicable rule in the set, then escape the rest,

   (b) apply all the rules of the set according to their order,

   (c) or apply the rules as much as possible (which is known as the *saturation* of rules application).

4. call a set in another and imbricate these calls as much as needed.

10.3.2 Strategy instructions

That is why, we adopt in LoTREC a simple, but sufficiently expressive, language of strategies. This language is inspired from *regular expressions* languages, and it has many common semantics with the tactic language ANGEL [MGW96], which is the strategy language used by the TWB theorem prover [AG03].

Practically, defining a strategy consists in associating, to a *strategy name*, a set of *instructions*. An instruction, is defined inductively as follows:

1. a rule name is an instruction,

2. a rule name preceded by the keyword `applyOnce` is an instruction,

3. a set of instructions surrounded by one of the following pairs of keywords

   (a) `firstRule` and `end`,

   (b) `allRules` and `end`,

   (c) `repeat` and `end`,

   is also an instruction,

4. another strategy name is an instruction,

5. and there are no other ways to define instructions.

Each instruction in this enumerated list ensures the functionality of the corresponding item in the list given in section 10.3.1.

The semantics of LoTREC's strategies will be formalized in the next chapter.

10.4 Properties of LoTREC rewriting system

In Section 9.4 of Chapter 9, we show some of the theoretical properties that a given rewriting system acquires according to its characterization.

In the previous sections of this chapter, we have presented the specificities of LoTREC which characterize its rewriting system. Mainly, we explained how the rules and strategies are defined in it, i.e. how the choice of the rule to be applied is made and how the choice of the graph pattern where it is to be applied is made.

In this section, we give some of the main theoretical properties of the rewriting system of LoTREC.

10.4.1 Termination

In Section 9.4.2 of Chapter 9, we define termination of graph rewriting systems, we recall the undecidability of proving this property in general and we give two criteria under which a rewriting system is proved to be terminating. The first criterion, given by D. Plump in [Plu95], consists in having only, what the author calls, *forward closure* rules. Thus a rule should remove, during its application, at least one element of the subgraph that matches its LHS, i.e. the redex which makes the rule applicable. It is clear that non-forward closure rules would be infinitely applicable.

However, this can never happen in LoTREC, in fact, due to the tuned notion of rule applicability (see Section 10.2.3), a rule in LoTREC is, by default, never applicable twice on the same pattern.

The second criterion of termination, known as *layered graph grammars* and presented in [EEdL$^+$05], consists in stratifying the rules in different *ordered* layers such that the rules of a given layer are applied as much as possible, but never applied once it is the next layer's turn. This avoids inter-layers cycles. However, an analysis of the rules of each layer is still necessary to prevent intra-layers cycles.

In LoTREC, it is often the case that all the rules constitute only one layer, since all the rules have to be called as much as possible (*saturation* principle) to complete the models construction for a given formula of some logic. So the layered graph grammars approach would not guarantee in general the termination of the decision procedures defined in LoTREC.

In fact, a rule may generate a *new* pattern that would trigger the application of the same rule once again, ending up by having a non-terminating system. In such case, we may prove termination on the basis of the *general termination theorem* given in [GHS06a] for the rules used in the model construction methods of various modal logics. A rule of this kind would only rewrite an expression into (often smaller) expressions and generate new finite graph structures. For such kind of rules, the termination is guaranteed in two cases:

- when the rules generate on the RHS *strict* subexpressions of the expressions appearing in the LHS,

- when the rules generate on the RHS (*non-strict*) subexpressions of the expressions appearing in the LHS, with, in addition, a check for node-inclusion or node-equality loops, to be called at the right moment in the strategy.

Note that in the last case, the strategy becomes complicated which makes harder the completeness proofs of the underlying methods.

We may also have to deal with other kinds of rules for which termination is not guaranteed. In such cases we should adopt other solutions to guarantee the termination. Generally, a working solution consists in adding NACs to the rules to prevent them from being applicable, and thus avoid their application.

To summarize this section we recall that:

- deciding on the termination of a set of rewriting rules and a strategy in LoTREC is case-dependant,

- some results from the graph rewriting community and other general results on termination in LoTREC can be used to ease the task of proving the termination of user-defined rules and strategies,

- the most general and golden trick that helps in ensuring the termination in practice is: "the use of NACs".

10.4.2 Completeness, locality and invertibility

Every rule defined in LoTREC is compiled before its application and its definition is never accepted if it may lead to confusions at the run-time. The compilation takes into account the DPO conditions and avoid problematic rules such as those defined in Example 12, page 199. Hence, according to Definition 48 and its discussion in Section 9.4.3, we assume that the rewriting system of LoTREC is complete.

Locality of rewriting in LoTREC is based on the locality of each rule application, as already shown in Section 10.2.4.

When the rules are defined according to the SPO approach, such as in LoTREC, their invertibility is not straightforward (see Section 9.3.2). In order to define the inverse of such rules, we should save more information on how the rules are applied at run-time . However, un-applying a rule in LoTREC amounts to un-applying each elementary action, since the RHS of a rule consists of a set of elementary actions. Hence, the information to be saved can be easily defined for each action, making any rule invertible.

For example, the actions `unmark` and `unmarkExpressions` are already defined in LoTREC and used in the model checking method as the inverse actions of `mark` and `markExpressions` actions. These are the only two invertible actions defined so far in LoTREC, since invertible rules are not essential in our application domain in logic.

10.4.3 Parallelism, confluence and convergence

The definitions of sequential and parallel independence, addressed in Section 9.4.5, are the same in LoTREC, since the rules definition is based on the same algebraic approach used for those definitions. The Church-Rosser theorem (Theorem 5) holds too.

On the other hand, the confluence of a set of rules cannot be defined without taking into consideration the strategy calling them. In fact, given an input graph, a *unique normal form* is derived in LoTREC only according to a *specific* strategy.

According to this definition of confluence, and to the termination criteria given above, a set of rules and a strategy are convergent in LoTREC if and only if they define a terminating and confluent rewriting system. Furthermore, a formal proof which states the equivalence of rewriting in LoTREC and rewriting in the usual classical sense, is given for Theorem 6 in the next chapter.

Complexity As shown in Section 9.4.7, the main source of time consumption in the whole rewriting process is the pattern matching process. This is why we postpone the discussion about the complexity of rewriting in LoTREC to the next chapter, where we talk about its matching process in depth.

Conclusion

In this chapter, we gave an overview of the graph rewriting system of LoTREC and its relation to the theory of graph transformation.

We specified the similarities between models and graphs, between premodel construction and graph transformation and between LoTREC rules and graph rewriting rules. This should help the reader to better understand the background of the model construction methods given throughout the chapters 3 to 7.

We also gave the specificities of the rewriting system of LoTREC in contrast to existing and traditional rewriting systems. This allows us to discuss some of the theoretical properties of our tool. This also shows that we may reuse some existing results, that are well-established in the domain of graph rewriting, to characterize our model construction methods in logic.

The language used, in this chapter and the previous one, is the one talked by the community of graph transformation. This allows us to introduce to them an original graph rewriting system, even if this system was developed for specific purposes in a different domain. Written from scratch, our tool has a competent and performant rewriting engine, as we shall see in the next chapter, which makes its techniques worth being studied and adopted by developers of graph transformation tools.

Chapter 11

Semantics of event-based graph rewriting system of LoTREC

Introduction

The large variety of graph transformation tools have mainly the same one-step rule application mechanism and usually differ by the techniques used for the graph pattern matching step, which is considered to be the most crucial in the overall performance of a graph transformation process. This process consists in mapping the elements of the left-hand side graph L of a rewriting rule to the elements of the host graph G. Hence, a naive implementation, which computes every possible mapping and which browses the whole host graph structure at each iteration to look up for matches, leads to $O(|G|^{|L|})$ time complexity in general. Furthermore, when dealing with model construction for modal logics, this search becomes more expensive, since these structures are often of exponential size w.r.t. the input formula, .

The classical approach, used in [Rud00] for AGG [Tae99], is to solve the pattern matching problem as a *constraint satisfaction problem.* Other systems, such as PROGRES [Z9̈6] and FUJABA [FNTZ00], use *local search* techniques consisting of matching a single node by some heuristics and extending the matching step-by-step by neighboring nodes and edges. G. Varró and D. Varró introduce recently in [VV04] the *incremental update* technique which aims to keep track of all possible matchings identified by the rules in database tables, and update these tables incrementally to exploit the fact that rules typically perform only local modifications to graphs.

In the graph rewriting system of our theorem prover LoTREC, we implement an original technique that lies between these three approaches. Before running the rewriting process, we analyse the left-hand side conditions of every rule in

order to specify the possible ways (i.e. *search plans*) a matching process could be established and achieved. During the rewriting process, we keep track of the local changes made by the rules at each step, using a special data structure called *events*. This is achieved in JAVA by using the *event dispatching* paradigm [Ham97]. When it is called by the strategy, a rule uses these events to establish a *local* pattern matching process, with respect to its different search plans. Finally, the established match is completed using, as usual, a CSP-like procedure which instantiates the rest of the pattern variables.

On one hand, this technique reduces the time-cost to $O(|G_i \setminus G_{i-1}|^{|L|})$ at each step i of the rewriting process, where $G_i \setminus G_{i-1}$ is the set of elements (nodes, edges, attributes...) that are added at the previous step $i-1$. Thus, this technique does not change the complexity of the matching process itself. However, in some application domains, such as in model construction for logics, the rewriting rules are only applicable on k subgraphs at each step, where k is a constant factor or a linear factor of the input problem. In such cases, the average performance is increased by reducing the time-cost to $O(|k|^{|L|})$ using our event-based technique.

On the other hand, an important aspect of programming model construction methods concerns the "distance" between their theoretical formulations on paper and their implementations involving programming tricks and various optimizations that are not in the initial formulation. This may raise some doubts on completeness and correctness proofs that usually are given using the theoretical formulation. We claim that LoTREC allows to keep the implementation of these methods closer to their theoretical formulation due to its declarative language. It may seem that the use of events makes this "distance" greater, but we show in fact that rewriting with events preserves completeness, soundness and termination.

In this chapter, we formally define the semantics of the language of our generic prover LoTREC, and we formalize its event-driven pattern detection, as it has been presented in [GSS09, SG08]. Then we prove that running with or without this optimisation is equivalent in terms of possible computations.

In the first section we introduce the term notation used to easy the definitions and the proofs given in the sequel. We show how to define a premodel and a rule in this formalism. Then, we define the notions of rules applicability and application both in a naive system (Section 11.1.4) and in LoTREC (Section 11.1.6) and we prove their equivalence. Next, we extend those notions and equivalence to strategies. Finally we compare our technique to those of existing graph rewriting tools, and we give some experimental results.

11.1 Modelling graph rewriting with terms

In this section, we give a representation of the graph rewriting based on terms. The benefit is to have a simple and uniform notation for all the graph elements. Nodes, edges and attributes are all represented as terms in the graphs and in the rewriting rules.

I should insist on the fact that the term formalism is used to easy the definitions, without loss of generality. However, it is not implemented in LoTREC as it is defined here, since the access to typed objects (nodes, edges, formulas...) is clearly less expensive than browsing a whole set of un-typed terms. It should also be clear for the reader that term rewriting problems are usually transformed to graph rewriting problems (*term graph rewriting* in [Plu99]), and not the inverse, in order to be solved efficiently[1].

First, we recall some basic notions about terms. You can also refer to [BN98].

11.1.1 Recalling terms

Definition 55 (set of terms)**.** *Let L be a set of symbols. We define the set of terms $\mathcal{T}(L)$ over L as the smallest set such that:*

- $L \subseteq \mathcal{T}(L)$*;*
- *for all $n \in \mathbb{N}$, for all $t_1, \ldots, t_n \in \mathcal{T}(L)$, $(t_1, \ldots, t_n) \in \mathcal{T}(L)$.*

$\mathcal{C}$ denotes the infinitely countable set of constant symbols and $\mathcal{V}$ the infinitely countable set of variable symbols. We define $\mathcal{GT} = \mathcal{T}(\mathcal{C})$ the set of *ground terms* and $\mathcal{VT} = \mathcal{T}(\mathcal{C} \cup \mathcal{V})$ the set of *variable terms.*

Given a variable term $u \in \mathcal{VT}$, $Var(u)$ denotes the set of variables occurring in u. Given a set of variable terms U, $Var(U) = \bigcup_{u \in U} Var(u)$.

Definition 56 (substitution)**.** *A (ground) substitution σ is a mapping from $\mathcal{V}$ to $\mathcal{GT}$ which extends as usual to elements of $\mathcal{VT}$. We denote by Σ the set of all possible substitutions.*

Example 27. Let $\mathtt{w}, \mathtt{A} \in \mathcal{V}$, and let $\{containsformula,\ w, \Diamond, P, Q\} \subseteq \mathcal{C}$. If $\sigma(\mathtt{w}) = w$ and $\sigma(\mathtt{A}) = (P \vee Q)$, then we have $\sigma(containsformula(\mathtt{w}\ ((\Diamond\mathtt{A}) \wedge \mathtt{A}))) = (containsformula(w\ ((\Diamond(P \vee Q)) \wedge (P \vee Q)))$.

11.1.2 Encoding a premodel with terms

In chapter 2, we showed how to represent models with graphs w.r.t. Kripke semantics. In the last chapter, Section 10.1, we showed how premodels can be also represented as graphs. For the sake of simplicity, we abstractly represent such graphs by a set of ground terms, but we keep on calling them premodels. We introduce a set of constant symbols $\mathcal{C}$ containing:

- symbols to compose terms encoding the formulas, mainly:
  - the set of connectors, $Conn = \{\Diamond, \Box, \wedge, \vee, \ldots\} \subseteq \mathcal{C}$;
  - and the set of atomic propositions $\mathcal{P} = \{P, Q, \ldots\} \subseteq \mathcal{C}$;
- and symbols to create terms which encode a graph, such as:

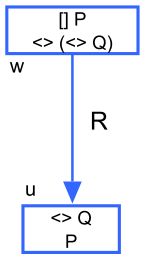
[1] *Term graph rewriting* has many interesting properties vis-à-vis term rewriting, such as subgraph sharing and confluence.

- $\{w, u, v \ldots\} \subseteq \mathcal{C}$ to denote nodes;
- $\{R, R_I, R_J \ldots\} \subseteq \mathcal{C}$ to denote edge labels
- and $\{world, link, containsformula, \ldots\} \subseteq \mathcal{C}$ to define a graph structure.

Definition 57 (premodel)**.** *A premodel T is a finite set of ground terms. The set of all premodels is noted $\mathcal{PM}$.*

Example 28. $\{((world\ w), (world\ u), (containsformula\ w\ (\Box P)), (link\ w\ u\ R_I)\}$ encodes the premodel $\bullet_w^{\{\Box P\}} \xrightarrow{R_I} \bullet_u^{\emptyset}$.

Another example is given below:



$T = \{$
$(world\ w)$,
$(containsformula\ w\ (\Box P))$,
$(containsformula\ w\ (\Diamond\Diamond Q))$,
$(world\ u)$,
$(link\ w\ u\ R)$,
$(containsformula\ u\ \Diamond Q)$
$(containsformula\ u\ P)$,
$\}$

For the sake of brevity, we settle for these two examples and we omit the details of how we can represent any premodel by a set of terms.

11.1.3 Coding rules as term rewriting rules

As shown in the previous chapter, Section 10.2.2, a model construction rule consists of conditions (describing a graph pattern) and actions (describing new graph objects to be added[2]). If the graph pattern is found in the premodel, then actions are performed. Conditions are modeled by two sets of variable terms: positive conditions (PC) and negative conditions (NC). PC is a set of variable terms to be unified with a subset of the premodel. NC is a set of variable terms which have to be not unifiable.

We may model actions by a set of variable terms (A) denoting new ground terms to be added to the premodel when the rule is applied. Instead we use $\mathcal{A} = \{A_1, \ldots, A_n\}$ since our rules can be non deterministic, i.e. the system makes a choice between different sets of terms to be added (see Section 10.2.4, and see Examples 29, 30 below). When the rule is deterministic $Card(\mathcal{A}) = 1$. Formally:

Definition 58 (rule)**.** *A rule r is a triple $(PC, NC, \mathcal{A})$ where:*

- $\emptyset \subsetneq PC \subseteq \mathcal{VT}$;

[2]Since we only use monotonic rules.

- *$NC \subseteq \mathcal{VT}$;*
- *$Var(NC) \subseteq Var(PC)$;*
- *a non empty finite set $\mathcal{A} \subseteq 2^{\mathcal{VT}}$ such that for all $A \in \mathcal{A}$, $Var(A) \subseteq Var(PC)$.*

We write it $r = \frac{PC,NC}{\mathcal{A}}$. Given a rule r, PC_r, NC_r and $\mathcal{A}_r$ denote respectively positive conditions set, negative conditions set and actions set of r.

A rule with an empty set of conditions is always applicable on any premodel (Section 10.2.1, page 222), and it is not checked during the pattern matching process. Thus, such kind of rules is irrelevant here in this work. This is why we assume that $PC \neq \emptyset$ w.l.o.g., and we only consider rules which are applicable on non empty patterns.

As our rules only add elements (Section 10.2.4, page 228), only positive conditions are used to instantiate variables, and negative conditions are only verified on these instances. This is why we have the third condition $Var(NC) \subseteq Var(PC)$.

We also impose $Var(A) \subseteq Var(PC)$, supposing that new added objects are denoted by *Skolem* terms whose arguments are terms of PC (see examples 31 and 32).

In order to make the variable symbols used in the sequel self-explanatory, we use $\mathtt{w}, \mathtt{u}, \mathtt{v}, \ldots$ to denote variables that are unifiable with constant nodes $w, u, v, \ldots$, and we use $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots$ to denote variables that are unifiable with formulas. Note that in the equivalent definitions given in LoTREC's language, we use `variable A`, `variable B`, `variable C`, ... instead.

In the following examples, the actions are given in green, and each set of action is surrounded by {blue braces}.

Example 29.

$$r_\vee = \frac{\{(containsformula\ \mathtt{w}\ (\mathtt{B}\vee\mathtt{C}))\}\ ,\ \emptyset}{\{\{(containsformula\ \mathtt{w}\ \mathtt{B})\},\{(containsformula\ \mathtt{w}\ \mathtt{C})\}\}}$$ is the rule dealing with the "or" operator.

Example 30.

$$r_3 = \frac{\{(world\ \mathtt{w}),(world\ \mathtt{u})\}}{\{\{(link\ \mathtt{w}\ \mathtt{u})\},\ \{(link\ \mathtt{u}\ \mathtt{w})\}\}}$$ is the rule corresponding to the *connectedness* property (for instance in the modal logic S4.3).

The following example gives an incorrect encoding of the $\Diamond$-rule:

Example 31.

$$r_\Diamond = \frac{\{(world\ \mathtt{w}),(containsformula\ \mathtt{w}\ (\Diamond\ \mathtt{B}))\}\ ,\ \emptyset}{\{\{(world\ \mathtt{u}),(link\ \mathtt{w}\ \mathtt{u}),(containsformula\ \mathtt{u}\ \mathtt{B})\}\}}$$ is not a correct rule since $Var(A) \nsubseteq Var(PC)$.

Instead of dealing with a new node $\mathtt{u}$, we name it with the skolem term $(r_\Diamond\ \mathtt{w}\ (\Diamond\mathtt{B}))$ encoding the fact that the node has been created by the rule $r_\Diamond$ applied on the node $\mathtt{w}$ with the formula $\Diamond\mathtt{B}$.

Example 32. A correct encoding of the $\Diamond$-rule shall be:

$$r_\Diamond = \frac{\{(world\ \mathtt{w}),(containsformula\ \mathtt{w}\ (\Diamond\ \mathtt{B}))\}\ ,\ \emptyset}{\{\{(world\ (r_\Diamond\ \mathtt{w}\ (\Diamond\mathtt{B}))),(link\ \mathtt{w}\ (r_\Diamond\ \mathtt{w}\ (\Diamond\mathtt{B}))),(containsformula\ (r_\Diamond\ \mathtt{w}\ (\Diamond\mathtt{B}))\ \mathtt{B})\}\}}$$

11.1.4 Rewriting rule application

Applying a rule r on a premodel T consists in:

1. finding a substitution σ unifying positive conditions of r with a subset of T;
2. verifying that negative conditions are respected;
3. and finally performing the actions when 1. and 2. succeed.

When a given substitution σ succeeds on 1. and 2., we usually say that r is applicable on T and the application of r on T will result in a new $T' = \sigma(A) \cup T$. If r is applied on T' by considering the same substitution σ, then T' will remain unchanged. While practically, we are rather interested in applying r using a different substitution σ', if there are any. This can be achieved by keeping a history of found substitutions. Nevertheless, we guarantee it by testing that the skolem terms added by the application of r and σ on T have not been already added: for all $A \in \mathcal{A}$, $\sigma(A) \not\subseteq T$. That's why we tune the usual definitions of rule *applicability* and rule *application* to ensure that a new different substitution is considered when the same rule is applied sequentially twice.

Definition 59 (rule applicability with the substitution σ)**.** *Let $T \subseteq \mathcal{GT}$, $r = \frac{PC,NC}{\mathcal{A}}$ and $\sigma \in \Sigma$. We say that r is applicable on T with the substitution σ, denoted by $r \downarrow_\sigma T$ iff*

- *$\sigma(PC) \subseteq T$;*
- *$\sigma(NC) \not\subseteq T$;*
- *and for all $A \in \mathcal{A}$, $\sigma(A) \not\subseteq T$.*

Definition 60 (rule applicability)**.** *We say that r is applicable on T, denoted by $r \downarrow T$ iff there exists $\sigma \in \Sigma$ such that $r \downarrow_\sigma T$.*

Definition 61 (rule application)**.** *Let $T, T' \subseteq \mathcal{GT}$ and $r = \frac{PC,NC}{\mathcal{A}}$. We write $T \xrightarrow{r} T'$ iff there exists $\sigma \in \Sigma$, $A \in \mathcal{A}$ such that $r \downarrow_\sigma T$ and $T' = T \cup \sigma(A)$.*

Example 33. Let us consider $r_\Diamond$ from Example 32 and the following premodels (see Figure 11.1):

$$\begin{aligned}
T\ &= \{\ \ (world\ w), (containsformula\ w\ (\Diamond(\Diamond P)))\}\ ;\\
T_2 &= \{\ \ (world\ w), (containsformula\ w\ (\Diamond(\Diamond P))),\\
&\qquad (world\ u), (link\ w\ u\ R), (containsformula\ u\ (\Diamond P))\};\\
T_3 &= \{\ \ (world\ w), (containsformula\ w\ (\Diamond(\Diamond P))),\\
&\qquad (world\ u), (link\ w\ u\ R), (containsformula\ u\ (\Diamond P)),\\
&\qquad (world\ v), (link\ u\ v\ R), (containsformula\ v\ P)\};
\end{aligned}$$

where $u = (r_\Diamond\ w\ (\Diamond(\Diamond P)))$ and $v = (r_\Diamond\ u\ (\Diamond P))$.

We have $T \xrightarrow{r_\Diamond} T_2 \xrightarrow{r_\Diamond} T_3$. But we do not have $T \xrightarrow{r_\Diamond} T_2 \xrightarrow{r_\Diamond} T_2$.

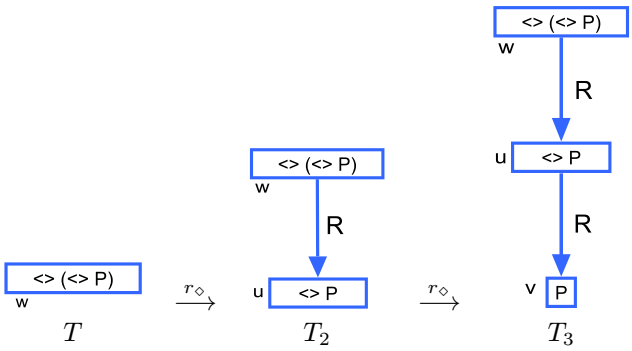


Figure 11.1: Graphical representation of Example 33.

The following proposition recalls that our rules are monotonic and it shall be used later in the proof of theorem 6.

Proposition 4. *Let $T, T' \subseteq \mathcal{GT}$ and a rule r: $T \xrightarrow{r} T'$ implies $T \subseteq T'$.*

11.1.5 Discussion about pattern matching process

Let us consider Example 33. A naive rewriting system will achieve $T \xrightarrow{r_\diamond} T_2$ by using a substitution σ such that $\sigma(\mathtt{w}) = w$ and $\sigma(\mathtt{B}) = (\diamond P)$. Then, at the rewriting step $T_2 \xrightarrow{r_\diamond} T_3$, the system will consider again the same substitution σ, before it realizes that it will not succeed. This is because a naive system would browse the whole premodel T_2 looking up for a possible match.

Due to the high time-cost of the matching process, we would rather want to reduce the number of established substitutions. Moreover, a rule can be infinitely applicable using the same substitution. Except if we store the whole information about all the already achieved substitutions. Which is another source of time and space consumption.

In LoTREC, we do not need to store such information, since it is coded in the graphs themselves. In fact, new successful patterns at a given step may only rise from new objects added at the previous step. Hence, we keep track of these objects, during *only one step* transition, to guide the following matching processes.

In fact, unifications are established only on new objects. For instance, at the step $T \xrightarrow{r_\diamond} T_2$, $(world\ u), (link\ w\ u\ R)$ and $(containsformula\ u\ (\diamond P)$ are reported as new objects. Whereas $(world\ w), (containsformula\ w\ (\diamond(\diamond P))$ are not. So at step $T_2 \xrightarrow{r_\diamond} T_3$, the substitution σ is no more considered.

Keeping track of these new objects can be done "by hands" by the programmer of the model construction method. For instance, for the logic K, rules can be defined in such a way that they are only applied on one active world (the current world). However, in a more complex logic (e.g. K plus universal modality etc.) the notion of active world is hard to define. Thus in LoTREC, we aim

to make the notion of active worlds *implicit* for the programmer, so he or she has to give only high-level declaration of rules.

In the next subsection, we describe the event-driven pattern detection which implements this automatic management of new objects in LoTREC. Next we prove that using this automatic management our rewriting system is still sound w.r.t a naive rewriting system.

11.1.6 LoTREC rewriting system

In LoTREC, new objects, called *events*, are managed with the paradigm of event-based programming [Ham97]. When a new object is added to the premodel, it is dispatched to *all* the rules in form of a new occurring event.

Thus, a state of the machine in LoTREC is the current premodel and a structure defining the launched events, whereas , in a naive rewriting system, the state of the machine is only the current premodel.

In these different settings, the notions of rule applicability and rule application become slightly different. And that is why, we shall give the precise semantics for the LoTREC rewriting system.

In the sequel, R denotes a non empty finite set of rules.

Definition 62 (state of the LoTREC machine)**.** *A state of the LoTREC machine is a couple* (T, E) *where* $T \in \mathcal{PM}$ *and* $E : R \longrightarrow\subseteq \mathcal{GT}$*.*

T is the current premodel and $E(r)$ (also denoted by E_r) is the set of launched events received by the rule r. Notice that an event is simply a ground term.

At the beginning of the tableau method, we start the process with an initial state (T, E_T) where T is the initial premodel[3] and E_T contains all terms appearing in T (as if all objects of T are declared as new events). Formally:

Definition 63 (initial events)**.** *Given* $T \subseteq \mathcal{GT}$ *and a set of rules* R*, we define the map of initial events of a premodel* T *by the map* $E_T : R \longrightarrow\subseteq \mathcal{GT}$ *as for all* $\rho \in R$*,* $E_T(\rho) = T$*.*

Remark 17. We use ρ when we quantify over a set of rules. We use r instead to denote a specific rule, for example $r_\diamond$.

Now, we define the notion of rule applicability and application on a given state of the LoTREC machine.

Definition 64 (rule applicability in LoTREC by e, σ)**.** *Let* (T, E) *be a state of LoTREC machine,* $r = \frac{PC,NC}{\mathcal{A}} \in R$ *and* $\sigma \in \Sigma$*,* $e \in \mathcal{GT}$*. We say that* r *has been awakened by the event* e *and is applicable in LoTREC on* (T, E) *with the substitution* σ*, denoted by* $r \downarrow^{LoTREC}_{e,\sigma} (T, E)$*, iff*

- $e \in E_r$*;*
- $e \in \sigma(PC)$*;*

[3]Usually a single world with the input formula.

- *and* $r \downarrow_\sigma T$.

Practically, the application of the rule r is driven by the event e. In fact, the event e can be viewed as a *pin* that guides our quest of finding a fruitful substitution σ, making the rule r applicable on T, by establishing the matching process *locally* around the new added object e (since $e \in \sigma(PC)$).

Definition 65 (rule applicability in LoTREC)**.** *We say that r is applicable in LoTREC on (T, E), denoted by $r \downarrow^{LoTREC} T$ iff there exists $e \in E_r$, $\sigma \in \Sigma$ such that $r \downarrow_{e,\sigma}^{LoTREC} T$.*

Definition 66 (transition of the LoTREC machine)**.** *Let (T, E) and (T', E') be two states of the LoTREC machine and $r \in R$. We write $(T, E) \overset{r}{\twoheadrightarrow} (T', E')$ iff there exists $e \in E_r$, $\sigma \in \Sigma$, $A \in \mathcal{A}_r$ such that:*

- *$r \downarrow_{e,\sigma}^{LoTREC} T$ and $T' = T \cup \sigma(A)$;*
- *and for all $\rho \in R$, $E'_\rho = E_\rho \cup \sigma(A)$.*

Intuitively, we write $(T, E) \overset{r}{\twoheadrightarrow} (T', E')$ if and only if we can find a substitution σ holding one of the events stored in $E(r)$. In this case, we apply r on T by performing the add actions and we obtain the new premodel T'. Finally we send the new created objects to every rule and we obtain E'.

In order to avoid trying to apply r on the same pattern twice, we delete already exploited events. We describe this simplification process in the following:

Definition 67 (simplification transition of the LoTREC machine)**.** *Let (T, E) and (T, E') be two states (with the same premodel). We write $(T, E) \rightsquigarrow (T, E')$ iff for all $\rho \in R$,*

$$E'_\rho = E_\rho \setminus \{e \in E_\rho \ \ \textit{such that for all } \sigma \in \Sigma,\ e \in \sigma(PC_\rho) \textit{ implies } \rho \not\downarrow_\sigma T\ \}.$$

where $\rho \not\downarrow_\sigma T$ means that we have not $\rho \downarrow_\sigma T$.

Example 34. We will give a remake of example 33. The process is:

$$(T, E_T) \overset{r_\diamond}{\twoheadrightarrow} (T_2, E'_2) \rightsquigarrow (T_2, E_2) \overset{r_\diamond}{\twoheadrightarrow} (T_3, E'_3) \rightsquigarrow (T_3, E_3)$$

where:

- T, T_2, T_3 are as defined in Example 33;
- E_T is such that $E_{T_{r_\diamond}} = T$;
- E'_2 is such that $E'_{2_{r_\diamond}} = E_{T_{r_\diamond}} \cup \{(world\ u), (link\ w\ u\ R), (containsformula\ u\ (\diamond P))\}$;
- E_2 is such that $E_{2_{r_\diamond}} = E'_{2_{r_\diamond}} \setminus \{(world\ w), (containsformula\ u\ (\diamond(\diamond P)))\}$,
  i.e. $E_{2_{r_\diamond}} = \{(world\ u), (link\ w\ u\ R), (containsformula\ u\ (\diamond P))\}$;
- E'_3 is such that $E'_{3_{r_\diamond}} = E_{2_{r_\diamond}} \cup \{(world\ v), (link\ u\ v\ R), (containsformula\ v\ P)\}$;

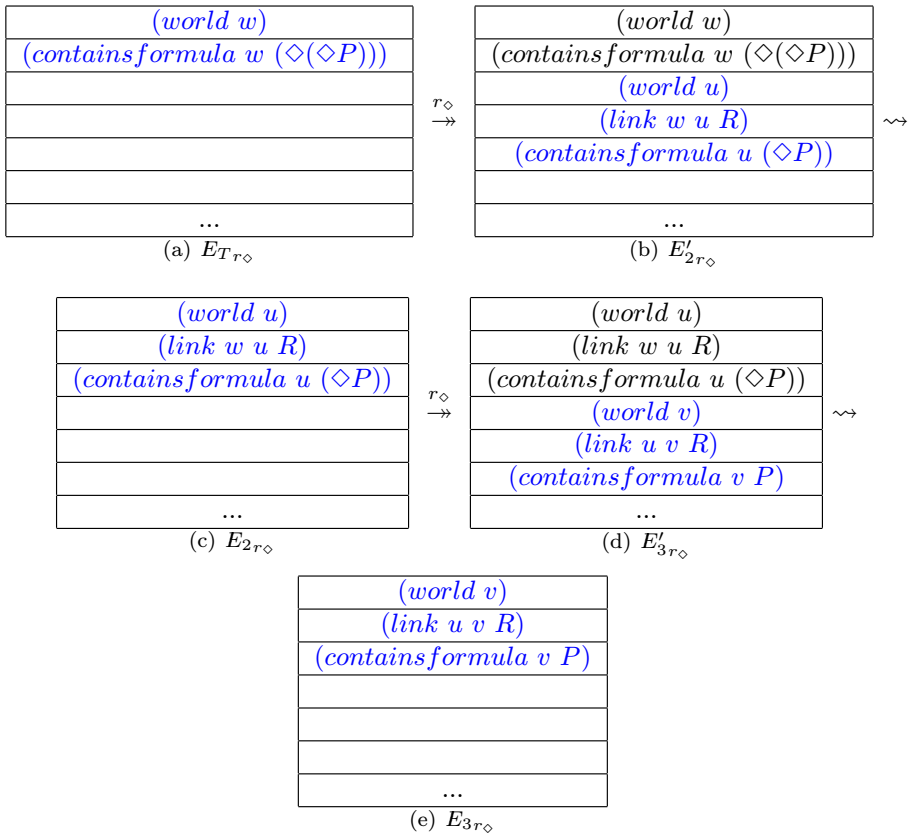


Figure 11.2: The changes of the event queue of the rule $r_\diamond$ during the transformations shown in Example 34. Blue terms indicate newly added events. Black ones are exploited ones. They are deleted during simplification ($\rightsquigarrow$) steps.

- E_3 is such that $E_{3_{r_\diamond}} = E'_{3_{r_\diamond}} \backslash \{(world\ u), (link\ w\ u\ R), (containsformula\ u\ (\diamond P))\}$, i.e. $E_{3_{r_\diamond}} = \{(world\ v), (link\ u\ v\ R), (containsformula\ v\ P)\}$.

These changes are illustrated in Figure 11.2.

Definition 68. *We write* $(T, E) \overset{r}{\twoheadrightarrow\rightsquigarrow} (T', E')$ *iff there exists* E'' *such that* $(T, E) \overset{r}{\twoheadrightarrow} (T', E'') \rightsquigarrow (T', E')$. *We write* $(T, E) \overset{r_1 \ldots r_n}{\twoheadrightarrow\rightsquigarrow} (T', E')$ *iff there exists* $((T_i, E_i))_{i \in \{2 \ldots n-1\}}$ *such that* $(T, E) \overset{r_1}{\twoheadrightarrow\rightsquigarrow} (T_2, E_2) \overset{r_2}{\twoheadrightarrow\rightsquigarrow} \ldots \overset{r_n}{\twoheadrightarrow\rightsquigarrow} (T', E')$.

11.1.7 Equivalence between usual rewriting system and LoTREC in terms of rules

We defined in subsection 11.1.4 the classical notion of rule application $T \xrightarrow{r} T'$. We also gave the semantics of rule application in our rewriting system LoTREC

in subsection 11.1.6. In the following, we prove the equivalence of these two notions.

One direction is trivial: what is achieved in LoTREC can be achieved exactly in a classical naive rewriting system.

Proposition 5. *Let (T, E) and (T', E') be two states of the LoTREC machine and $r \in R$. $(T, E) \overset{r}{\twoheadrightarrow} (T', E')$ implies $T \xrightarrow{r} T'$.*

However proving the other direction is not that obvious since we are loosing some events during the simplification steps $\rightsquigarrow$. In what follows, we prove that those simplifications will not ban the applicable rules from being applied.

Theorem 6. *Let R be a set of rules. Let $r_1 \dots r_n$ a sequence of rules of R. For all $T_1, \dots, T_n \in \mathcal{PM}$, we have equivalence between:*

- *the classical rewriting*

$$T_1 \xrightarrow{r_1} T_2 \xrightarrow{r_2} \dots \xrightarrow{r_{n-1}} T_n;$$

- *and, that under LoTREC, there exists $(E_i)_{i \in \{2,\dots,n\}}$ such that:*

$$(T_1, E_{T_1}) \overset{r_1}{\twoheadrightarrow\rightsquigarrow} (T_2, E_2) \overset{r_2}{\twoheadrightarrow\rightsquigarrow} \dots \overset{r_{n-1}}{\twoheadrightarrow\rightsquigarrow} (T_n, E_n).$$

Proof. $\boxed{\Uparrow}$ Straightforward with proposition 5.

$\boxed{\Downarrow}$ We prove this direction by induction on n. For $n = 1$, it is trivial. Suppose it for one $n \geq 1$. Let us prove it for $n + 1$.

Suppose $T_1 \xrightarrow{r_1} T_2 \xrightarrow{r_2} \dots \xrightarrow{r_{n-1}} T_n \xrightarrow{r_n} T_{n+1}$. By induction there exists $(E_i)_{i \in \{2,\dots,n\}}$ and $(E'_i)_{i \in \{2,\dots,n\}}$ such that

$$(T_1, E_{T_1}) \overset{r_1}{\twoheadrightarrow} (T_2, E'_2) \rightsquigarrow (T_2, E_2) \overset{r_2}{\twoheadrightarrow} (T_3, E'_3) \rightsquigarrow \dots \overset{r_{n-1}}{\twoheadrightarrow} (T_n, E'_n) \rightsquigarrow (T_n, E_n).$$

We have to prove there exists E_{n+1} and E'_{n+1} such that:

$$(T_n, E_n) \overset{r_n}{\twoheadrightarrow} (T_{n+1}, E'_{n+1}) \rightsquigarrow (T_{n+1}, E_{n+1}). \qquad (*)$$

As $T_n \xrightarrow{r_n} T_{n+1}$, there exists $\sigma \in \Sigma$ such that $r_n \downarrow_\sigma T_n$.

From now on, we will note $r = r_n$, $PC = PC_{r_n}$,$NC = NC_{r_n}$. Here we just need to prove that we can find $e \in \mathcal{GT}$ such that $r \downarrow_{e,\sigma}^{\text{LoTREC}} T_n$. First we will define $e \in \sigma(PC)$ and secondly we will prove that $e \in E_n(r)$.

As $r \downarrow_\sigma T_n$, we have $\sigma(PC) \subseteq T_n$. Consider the set $I = \{i \geq 1 \mid \sigma(PC) \subseteq T_i\}$. It is a non empty (because $n \in I$) set of positive integers. So we can consider the minimum $i_0 = min(I)$.

First let us define $e \in \sigma(PC)$:

- if $i_0 = 1$, as $\sigma(PC) \neq \emptyset$ by definition 58, we simply take $e \in \sigma(PC)$;
- if $i_0 > 1$, as $T_{i_0-1} \subsetneq \sigma(PC)$ by definition of i_0, let $e \in \sigma(PC) \setminus T_{i_0-1}$.

Secondly our aim is to prove $e \in E_n(r)$. We prove it by proving that for all $k \in \{i_0, \ldots, n\}$, $e \in E_k(r)$ by induction on k.

- Initial case: we have $e \in E_{i_0}(r)$. Indeed, if $i_0 = 1$, $E_1 = E_{T_1}$ so $e \in \sigma(PC) \subseteq T_1 = E_1(r)$. If $i_0 > 1$, we have: there exists $\sigma_{i_0-1} \in \Sigma$ such that $r_{i_0-1} \downarrow_{\sigma_{i_0-1}} T_{i_0-1}$. There exists $A \in \mathcal{A}_{r_{i_0-1}}$ such that $T_{i_0} = T_{i_0-1} \cup \sigma_{i_0-1}(A)$. As $e \in \sigma(PC) \setminus T_{i_0-1}$, we have $e \in \sigma_{i_0-1}(A)$. By definition of $\overset{r_{i_0-1}}{\twoheadrightarrow}$, we have $E'_{i_0}(r) = E_{i_0-1}(r) \cup \sigma_{i_0-1}(A)$. So, $e \in E'_{i_0}(r)$. Then we have $e \in \sigma(PC)$ and $r \downarrow_\sigma T_{i_0}$. Indeed:

  - $\sigma(PC) \subseteq T_{i_0}$;
  - $\sigma(NC) \not\subseteq T_{i_0}$; (because $\sigma(NC) \not\subseteq T_n$ and $T_{i_0} \subseteq T_n$)
  - for all $A \in \mathcal{A}$ such that $\sigma(A) \not\subseteq T_{i_0}$. (because for all $A \in \mathcal{A}$ such that $\sigma(A) \not\subseteq T_n$ and $T_{i_0} \subseteq T_n$)

  So $e \in E_{i_0}(r)$.

- Now, let us prove the induction case: for all $k \in \{i_0, n-1\}$, $e \in E_k(r)$ implies $e \in E_{k+1}(r)$. If $e \in E_k(r)$, as $E_k(r) \subseteq E'_{k+1}(r)$ we have $e \in E'_{k+1}(r)$. Then for the same reason as in the initial case we have $e \in \sigma(PC)$ and $r \downarrow_\sigma T_{k+1}$. Hence $e \in E_{k+1}(r)$.

Finally, we have $e \in E_n(r)$ and $e \in \sigma(PC)$ and $r \downarrow_\sigma T_n$: hence $r \downarrow^{\text{LoTREC}} (T_n, E_n)$! So (*) holds and this concludes the proof of theorem 6.

□

The following definition allows us to state propositions 6 and 7 as direct reformulation of theorem 6 that we use in the sequel:

Definition 69 (E is rich for T)**.** *Let (T, E) be a state of the LoTREC machine. We say that E is rich for T iff there exists a sequence u of rules in R, T_0 such that $(T_0, E_{T_0}) \overset{u}{\twoheadrightarrow\rightsquigarrow} (T, E)$. (if the sequence $u = \epsilon$ it is equivalent to the condition $E = E_T$)*

"E is rich for T" means that E contains enough events to preserve rules applicability on T.

Proposition 6. *Let (T, E) a state of the LoTREC machine. If E is rich for T then $r \downarrow T$ iff $r \downarrow^{LoTREC} (T, E)$.*

Proposition 7. *Let (T, E) a state of LoTREC machine and $T' \in \mathcal{PM}$. If E is rich for T, we have $T \xrightarrow{r} T'$ iff there exists E' such that $(T, E) \overset{r}{\twoheadrightarrow} (T', E')$ (and of course E' is rich for T')*

11.2 Strategies

Defining strategies over rules is the traditional way to declare complex scenarios of rule applications: sequences and repetitive applications of rules. This is done in LoTREC by a simple and high-level declarative language, which was introduced in Section 10.3, page 229.

In this section, we define a formal syntax definition of this language. Then we give two semantics of strategies:

- We extend the notions of applicability and applications in a naive rewriting system (definitions 60 and 61);
- We extend the notions of applicability and applications in LoTREC (definition 65 and 66).

Finally, we extend the result of theorem 6, i.e. we show that the two semantics are equivalent.

11.2.1 Syntax

The syntax is basically inspired from the theory of regular expressions.

Definition 70 (strategy)**.** *Let R be a set of rules. The set of all strategies S over R is defined as the smallest set such that:*

- $R \subseteq S$;
- *if* $s_1, s_2 \in S$, *then* $s_1 \mathbin{:} s_2 \in S$;
- *if* $s_1, s_2 \in S$, *then* $s_1 \mid^{or}_{else} s_2 \in S$;
- *if* $s \in S$, $s^{\text{☼}} \in S$.

Example 35. If $r_\diamond$ and $r_\wedge$ are two rules, $(r_\diamond \mathbin{:} r_\wedge)^{\text{☼}} \mid^{or}_{else} r_\wedge$ is a strategy.

In order to avoid semantical ambiguity, we have introduced the new operators **:** and ☼ instead of using the classical ; and * operators of regular expressions.

11.2.2 Standard semantics

A strategy consisting of one single rule r has the same definition of applicability and application. The sequence operator **:** is used to create an order between the application of two strategies: $s_1 \mathbin{:} s_2$ means that we try to apply s_1 first and then we try to apply s_2. The strategy $s_1 \mid^{or}_{else} s_2$ means that we try to apply s_1 and if it is not possible then we try to apply s_2[4]. The strategy $s^{\text{☼}}$ means that we apply s as long as it is possible. These semantics reflects the semantics given in Section 10.3, and they are detailed in the following definitions.

[4]Note that this strategy is not fair.

Definition 71 (strategy applicability)**.** *We define $s \downarrow_{str} T$ by induction:*

- *if $r \in R$, $r \downarrow_{str} T$ iff $r \downarrow T$;*
- *if $s_1, s_2 \in S$, $s_1 \ \mathbf{:} \ s_2 \downarrow_{str} T$ iff $s_1 \mid^{or}_{else} s_2 \downarrow_{str} T$ iff $s_1 \downarrow_{str} T$ or $s_2 \downarrow_{str} T$;*
- *if $s \in S$, $s^{\text{☼}} \downarrow_{str} T$ iff $s \downarrow_{str} T$.*

Notice that $\mathbf{:}$ and $\mid^{or}_{else}$ have the same notion of applicability but not the semantics.

Definition 72 (rewriting transition with strategy)**.** *Given $T, T' \in \mathcal{PM}$, $s \in S$ we define $T \xrightarrow{\overline{s}} T'$ (the premodel T can be rewritten in T' by applying the the strategy s) by induction:*

1. *If $r \in R$, $T \xrightarrow{\overline{r}} T'$ iff $T \xrightarrow{r} T'$;*
2. *If $s_1, s_2 \in S$, $T \xrightarrow{\overline{s_1 \ \mathbf{:} \ s_2}} T'$ iff:*
   (a) *there exists T'' such that $T \xrightarrow{\overline{s_1}} T''$ and $T'' \xrightarrow{\overline{s_2}} T'$;*
   (b) *or else $T \xrightarrow{\overline{s_1}} T'$ and $s_2 \not\downarrow_{str} T'$;*
   (c) *or else $s_1 \not\downarrow_{str} T$ and $T \xrightarrow{\overline{s_2}} T'$.*
3. *If $s_1, s_2 \in S$, $T \xrightarrow{\overline{s_1 \mid^{or}_{else} s_2}} T'$ iff:*
   (a) *we have $T \xrightarrow{\overline{s_1}} T'$;*
   (b) *or $s_1 \not\downarrow_{str} T$ and $T \xrightarrow{\overline{s_2}} T'$.*
4. *If $s \in S$, $T \xrightarrow{\overline{s^{\text{☼}}}} T'$ iff there exists $n \in \mathbb{N}$, $T_1, T_2, \ldots T_n \in \mathcal{PM}$, such that $T \xrightarrow{\overline{s}} T_1 \xrightarrow{\overline{s}} T_2 \xrightarrow{\overline{s}} \ldots \xrightarrow{\overline{s}} T_n \xrightarrow{\overline{s}} T'$ and $s \not\downarrow_{str} T'$.*

A straightforward induction shows that the following holds:

Proposition 8. *Let $T \in \mathcal{PM}$. If there exists $T' \in \mathcal{PM}$ such that $T \xrightarrow{\overline{s}} T'$ then it implies $s \downarrow_{str} T$.*

Be careful: the other direction is false. We can have at the same time $s \downarrow_{str} T$ and no $T' \in \mathcal{PM}$ such that $T \xrightarrow{\overline{s}} T'$. This is the case when a strategy does not terminate. Thus with our settings, proving that a strategy terminates amounts to proving that there exists such a T'.

Example 36. Consider the rule $r_{ser} = \frac{\{(world\ \mathtt{w})\}\ ,\ \emptyset}{\{\{(world\ (r_{ser}\ \mathtt{w})),(link\ \mathtt{w}\ (r_{ser}\ \mathtt{w}))\}\}}$ corresponding to the seriality property of a Kripke frame, and let $T = \{(world\ w)\}$. We have $r_{ser}{}^{\text{☼}} \downarrow_{str} T$ because $r_{ser} \downarrow_{str} T$. But there is no T' such that $T \xrightarrow{\overline{r_{ser}{}^{\text{☼}}}} T'$ because the execution will always continue to add a new successor to the last created world.

Our strategy language is similar to TWB language inspired by Angel [MGW96]. The only difference is that *skip* and *fail* tactics of Angel are embedded in LoTREC rules. In order to clarify the semantics of our strategy language we would compare it with programs of the logic PDL with sequence, iteration and test [HKT00]. In fact, we can express a strategy into the PDL program language by the following translation tr:

- $tr(r) = r$ (rules are atomic programs);
- $tr(s_1 \,:\, s_2) = (tr(s_1); (?[tr(s_2)]\bot \mid tr(s_2)) \mid (?([tr(s_1)]\bot); tr(s_2))$;
- $tr(s_1 \mid^{or}_{else} s_2) = tr(s_1) \mid (?([tr(s_1)]\bot); tr(s_2))$;
- $tr(s^{\maltese}) = tr(s)^*; ?[tr(s)]\bot$.

The reader can verify that $(s_1 \mid^{or}_{else} s_2)^{\maltese}$ and $(s_1^{\maltese} \,:\, s_2)^{\maltese}$ have the same semantics.

11.2.3 Semantics of rewriting with strategies in LoTREC

In this subsection, we give the semantics in LoTREC of the above defined strategies.

Definition 73 (strategy applicability in LoTREC)**.** *We define* $s \downarrow^{LoTREC}_{str} (T, E)$ *by induction:*

- *if* $r \in R$, $r \downarrow^{LoTREC}_{str} (T, E)$ *iff* $r \downarrow^{LoTREC} (T, E)$;
- *if* $s_1, s_2 \in S$, $s_1 \,:\, s_2 \downarrow^{LoTREC}_{str} (T, E)$ *iff* $s_1 \mid^{or}_{else} s_2 \downarrow^{LoTREC}_{str} (T, E)$ *iff* $s_1 \downarrow^{LoTREC}_{str} (T, E)$ *or* $s_2 \downarrow^{LoTREC}_{str} (T, E)$;
- *if* $s \in R$, $s^{\maltese} \downarrow^{LoTREC}_{str} T$ *iff* $s \downarrow^{LoTREC}_{str} (T, E)$.

Definition 74 (LoTREC rewriting transition with strategy)**.** *Given* (T, E), $(T', E') \in \mathcal{PM}$, $s \in S$, *we define* $(T, E) \overset{\overline{s}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$ *(the state* (T, E) *can be rewritten in* (T', E') *by applying the strategy* s*) by induction:*

1. *If* $r \in R$, $(T, E) \overset{\overline{r}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$ *iff* $(T, E) \overset{r}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$;
2. *If* $s_1, s_2 \in S$, $(T, E) \overset{\overline{s_1 \,:\, s_2}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$ *iff:*
   (a) *there exists* (T'', E'') *such that* $(T, E) \overset{\overline{s_1}}{\twoheadrightarrow\!\rightsquigarrow} (T'', E'')$ *and* $(T'', E'') \overset{\overline{s_2}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$;
   (b) *or else* $(T, E) \overset{\overline{s_1}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$ *(and* $s_2 \not\downarrow^{LoTREC}_{str} (T', E')$*);*
   (c) *or* $s_1 \not\downarrow^{LoTREC}_{str} (T, E)$ *and* $(T, E) \overset{\overline{s_2}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$.
3. *If* $s_1, s_2 \in S$, $(T, E) \overset{\overline{s_1 \mid^{or}_{else} s_2}}{\twoheadrightarrow\!\rightsquigarrow} (T', E')$ *iff:*

(a) *we have* $(T,E) \xrightarrow{\overline{s_1}} (T',E')$;

(b) *or* $s_1 \not\downarrow_{str}^{LoTREC} (T,E)$ *and* $(T,E) \xrightarrow{\overline{s_2}} (T',E')$.

4. *If* $s \in S$, $(T,E) \overset{\overline{s^{\text{☼}}}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$ *iff there exists* $n \in \mathbb{N}$, $(T_1,E_1),(T_2,E_2),\ldots(T_n,E_n)$, *such that* $(T,E) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T_1,E_1) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T_2,E_2) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} \ldots \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T_n,E_n) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$ *and* $s \not\downarrow_{str}^{LoTREC} (T',E')$.

11.2.4 Equivalence between usual rewriting system and LoTREC in terms of strategies

Proposition 9. *If* $T \in \mathcal{PM}$ *and* E *rich for* T *then* $s \downarrow_{str} T$ *iff* $s \downarrow_{str}^{LoTREC} (T,E)$.

Proof. We extend proposition 6 by induction on s. □

The following theorem is exactly what is stated in theorem 6 and proposition 7 extended to the strategies case.

Theorem 7 (Equivalence Theorem)**.** *Let* (T,E) *and* (T',E') *be two states of the LoTREC machine such that* E *is rich for* T. $T \xrightarrow{\overline{s}} T'$ *iff there exists* E' *such that* $(T,E) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$.

Proof. We show by induction on s the two following properties:

1. If E is rich for T, $T \xrightarrow{\overline{s}} T'$ iff there exists E' such that E' is rich for T' and $(T,E) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$;

2. there exists E' such that $(T,E) \overset{\overline{s}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$ implies $T \xrightarrow{\overline{s}} T'$.

This proof is tedious and we give just an extract of it. We give the most difficult points.

$\boxed{\text{1. in case of a rule } r\text{:}}$ If $T \xrightarrow{\overline{r}} T'$, by definition 72 we have $T \xrightarrow{r} T'$. Proposition 7 gives that there exists E' rich for T' such that $(T,E) \overset{r}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$. By definition 74, $(T,E) \overset{\overline{r}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$.

$\boxed{\text{1. in case of a sequence:}}$

Suppose $T \overset{\overline{s_1 \,\vdots\, s_2}}{\longrightarrow} T'$. We have either (2a), (2b) or (2c) of Definition 72. Let us consider (2a): there exists T'' such that $T \xrightarrow{\overline{s_1}} T''$ and $T'' \xrightarrow{\overline{s_2}} T'$. By induction applied on $T \xrightarrow{\overline{s_1}} T''$, we can say that: there exists (T'',E'') such that $(T,E) \overset{\overline{s_1}}{\twoheadrightarrow\!\!\rightsquigarrow} (T'',E'')$ and E'' is rich for T''. We can use induction on $T'' \xrightarrow{\overline{s_2}} T'$ and that there exists E' such that $(T'',E'') \overset{\overline{s_2}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$ and E' is rich for T'. Thus we obtain: $(T,E) \overset{\overline{s_1 \,\vdots\, s_2}}{\twoheadrightarrow\!\!\rightsquigarrow} (T',E')$. Cases (2b) and (2c) are left to the reader. □

This result allows to reason on strategies without being involved in optimisation details. A strategy for LoTREC is complete (resp. sound, resp. terminating) if and only if it is complete (resp. sound, resp. terminating) w.r.t. naive rewriting.

11.3 Related works

In PROGRES [Sch97], a rule r is compiled first. To instantiate the variables appearing in the left-hand side graph L_r of the rule r, a (most optimal) plan is chosen among the $|L_r|!$ different possible ways. A usual CSP-like pattern matching process follows this step. In LoTREC, we consider a fixed number of $|L_r|$ different search plans during the rule compilation. However, we achieve the pattern matching cleverly, due to our event mechanism. In fact, most of these plans are just ignored, since a plan is escaped as soon as the test of one of its conditions fails; and it is so often the case that it is the first condition of the plan which fails in matching any of the events received by the rule r.

In addition, although PROGRES supports an incremental technique called *attributes update*, this technique detects only the *invalid* variable assignments, i.e. unfruitful substitutions (what we call *learning from failure*). Thus it does not exploit the whole information embedded in the changes made on the graphs. In LoTREC we use these information to detect only new *valid* assignments (we can call it *seeking success*).

The *incremental update* [VV04] is the most similar to our technique, in the sense that both avoid restarting already-done pattern matching processes. However, our technique is less space and time-consuming. In fact, in our system, a rule keeps only the events occurred since its last application, and releases them all after being applied once again. Whereas the incremental update technique needs to keep successful matches in tables during the whole run-time.

Furthermore, these tables need a considerable amount of preprocessing at initialization time and they are maintained by continuous updates, while the events stored temporary in our rules need neither initialization nor update, and thus have no additional time-cost.

11.4 Experimental results

In this section, we would like to evaluate LoTREC's rewriting optimization in comparison with existing rewriting tools. However, it is not easy to setup the necessary experiments for many technical reasons: some tools are not maintained anymore, some others are hard to be installed, and even the easy to use ones are not customizable to be called automatically in a large set of benchmarks.

The only tool that is easy to install (as for me) and to run on the fly is AGG [Tae99] (written in Java and cross-platforms). However, AGG does not seem to be a good choice for the evaluation. When run with only one rule, the

confluence rule[5], it takes more than 6 secondes to achieve ten steps in depth. Whereas this is achieved in less than 0.5 seconds in LoTREC.

Without getting so far in these benchmarks, I realized that such evaluations are not fair anyway, mainly because in LoTREC we perform, in addition to the graph pattern matching, a complex process of formulas matching which cannot be easily integrated in the core of AGG or of other rewriting tools.

So I decided to settle for a simple evaluation: we run LoTREC in the hardest formulas of some logics, with and without the optimization, then we compare the number of tentatives of pattern matching processes established in each case. This would necessarily reflect the average time-cost reduction due to our event-based approach.

11.4.1 The LWB benchmark suite

LWB authors presented in [BHS00] a benchmark for the logics K, KT and S4. For each logic they give about twenty classes of formulas characterized by different types of difficulty (non-deterministic choices, branching factor etc.). There are 21 numbered formulas in each class. The highest number corresponds to the most complex formula in a class. The authors propose to compare theorem provers by running them on the same machine over these sets of formulas. By fixing 100 seconds as a ceil for the run time, provers handling formulas with higher numbers are the better in terms of performance.

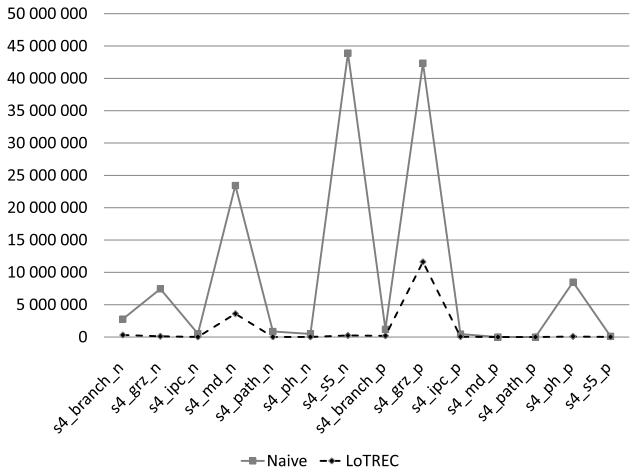
11.4.2 Evaluation of the event-based technique

To evaluate our event-based optimization, we run LoTREC with and without the optimisation on the benchmark suite of LWB presented above and by considering the hardest formulas that LoTREC can treat within 100 seconds. Then we count for each test the number of tentatives of pattern matching established in LoTREC.

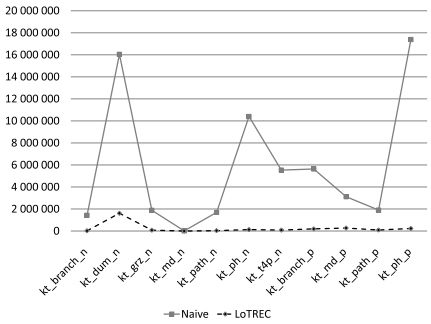
When LoTREC runs using the event-driven mechanism, this number is equal to the sum of the number of relevant events treated by the rules during all the iterations. Whereas when running without events, this number is equal to the sum of the objects (mainly nodes and formulas) found in the premodels at each iteration.

The results of these experiments are illustrated in Figure 11.3. In the first chart, treating the hardest formula of the class "S4_grz_n" (the 10th formula) leads to about 116 000 tentatives of pattern matching in optimised LoTREC, and to about 7 445 000 tentatives in the naive version of LoTREC, that is to say 64 times more of unfruitful and time consuming matching processes. We notice that generally the difference is more important when the formula that LoTREC can handle is harder.

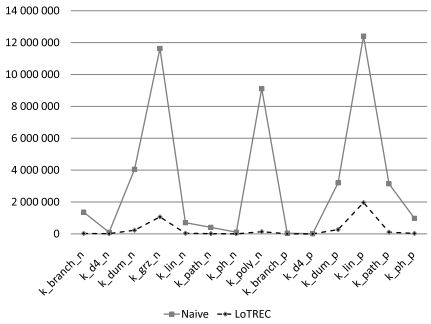
[5]for every two nodes linked to a common one, the rule creates a common successor - see Section 4.5.



(a) Results for S4 formulas



(b) Results for KT formulas



(c) Results for K formulas

Figure 11.3: LoTREC running with and without optimisation on the hardest formulas of the LWB benchmark suite.

Conclusion

LoTREC is a graph rewriting system adapted to theorem proving by tableau-like model construction methods for modal logics. The theoretical semantics of rewriting is purely functional and not optimised which made it suitable for proving properties like termination, soundness and completeness of the logical methods. However, a direct implementation of these semantics is not efficient, especially that the pattern matching process is known to be NP-complete.

In LoTREC, we propose a local and event-driven mechanism for limiting the effect of the graph pattern matching problem. At a given rewriting step, we establish the match process only on the local subgraphs changed by the rules during the previous step. It is clear that this technique has no advantage in specific applications where the redexes matched at each step are as numerous

as size of the whole host graph G. However, in the applications where the rules are applied alternatively in k redexes at each step, where k is likely to be far less than $|G|$, our technique reduces the $|G|$ factor of $O(|G|^{|L|})$ time complexity of this problem to k.

Nevertheless, such optimizations may raise some questions about the soundness of the rewriting process w.r.t. the theoretical semantics. Another main aim in this chapter was to fill the gap between this semantics and its implementation in LoTREC.

We have presented the rewriting system of LoTREC as a term rewriting system. We gave a semantics for model construction rules and their application that we may have in a naive rewriting system. We also gave a semantics for rules application in LoTREC rewriting system which has a global and automatic optimization based on events. Then we proved that this optimisation is sound and complete. Finally we gave a semantics to strategies both in naive system and in LoTREC and we proved their equivalence which in turn ensures that general properties of naive strategies propagates to optimised ones.

To conclude the chapter, we presented some of the related works and an evaluation of the even-based optimization.

Conclusion

In the first part of this thesis, we gave a "gentle" introduction to modal logics from an application point of view (chapters 1 and 2), and via a tableaux-like *model construction* procedure. We showed how to write such decision procedures for a wide variety of logics that we investigated throughout the chapters 3 to 7.

We also showed how to implement these methods in our generic model builder LoTREC, how to run and debug them with critical examples. Using this pedagogical style, we are planning to complete this (pedagogically speaking) preliminary work in the near future in form of a courseware book in logic, in joint work with O. Gasquet, A. Herzig and F. Schwarzentruber.

A special model construction method was used to investigate the frame properties and the complexity of a particular family of logics called *layered logics* (chapter 8). To keep the size of the models constructed in our method within a size exponential in that of the input formula we used a special filtration technique that we call *dynamic filtration*. The resulting method proved the class of these logics to be NEXPTIME-complete. I also gave the way to implement such filtration techniques in LoTREC.

In the second part of this thesis, I started with an overview of graph rewriting (chapter 9). It introduces various graph definitions, and the standard way to define graph rewriting rules with a solid theoretical basis. I briefly discussed some interesting theoretical properties that one would like to have in a graph rewriting system.

In chapter 10, I showed how LoTREC is to be defined as a graph rewriting system. This makes precise which kind of graphs it deals with, how the rules and strategies are defined, and what the main constraints and restrictions are that we had to impose on these definitions in LoTREC. This leads to a discussion of some of the properties of LoTREC viewed as a graph rewriting system.

In the last chapter, I presented the event-driven pattern matching mechanism of LoTREC (chapter 11). Using a term notation, the semantics of rewriting using this mechanism are formalized, then proved to be sound w.r.t. usual rewriting semantics. At the end of this chapter, I compare LoTREC's technique with existing related ones, and I give some experimental results that show the effect of our technique in enhancing the average performance of the rewriting process.

Beyond that, some other goals I had stated at the beginning of this thesis were reached: first, LoTREC's platform was maintained and enhanced through-

out the duration of my thesis; second, the tool was made more user-friendly and more accessible to the intended public; and third, several new logics were implemented, while keeping the language as small as possible.

LoTREC and the other theorem provers

In this section we give some comparisons and contrasts for LoTREC with existing tools according to their characteristics and purposes.

Educative and playful tools

OOPS[6] [GvVV09] shares many characteristics with LoTREC: it is open source, cross-platform, and aimed at education in modal logics. It has its own integrated general purpose scripting language (Lua), it allows tableaux to be visualized and it can generate counter models for formulas that are not provable.

However, the main difference with LoTREC is that OOPS is not generic: it is a tableau prover only for $\mathsf{S5}_n$.

Molle[7] (Modal Logic Loony Evaluator) or Mollicino is another nice tool. It is similar to LoTREC in being an educative, open-source and cross-platform (written in Java) prover. Like LoTREC, Molle implements a tableaux algorithm to prove the validity of modal formulas, and generates example and counterexample models.

However, Molle only implements the basic modal logic K and allows to choose basic constraints on the accessibility relations, such as reflexivity. Molle has a special "cool" display of models and counter models with the Earth, Mars and other planets to represent possible worlds.

Special purpose tools

We should recall that LoTREC is a generic framework which does not aim at implementing a specific method for only one logic.

It is clear that for a specific logic, any direct implementation would be simpler and more efficient than LoTREC. For example, we find many provers for the simple hybrid logic HL(@) (many of them are listed and compared in [Var09]). Some focus on performance. Some others have a simple implementation, thus they stick better to the theory , and they have more elegant proofs for termination and completeness.

However, these very specific provers may not be able to deal with K+Confluence or LTL or other logics as LoTREC does. They may not have an easy high-level language accessible to non-programmers. They may not be cross-platform web-executable softwares with graphical user interface and pretty print visualizations. Hence, they may not be the right candidates for prototyping and learning in logic and philosophy as, we hope LoTREC is.

[6] http://wiki.github.com/gertvv/oops
[7] http://molle.sourceforge.net/

Performant and painful tools

Due to its pedagogical aims, LoTREC was not designed to compete with other provers in terms of performance.

On one hand, we want to keep it as generic as possible, so it can handle new logics with various semantics. This entails that no special heuristics are implemented in LoTREC at all.

On the other hand, we want it to be a model builder: it should keep track of all premodels and should allow the user to analyse these premodels at the end of the satisfiability check. This imposes duplicating the premodel data structures at each non-deterministic choice, instead of dealing with it by backtracking on only one premodel. It follows that LoTREC loses time when managing this supplementary memory.

Nevertheless, we wanted to compare LoTREC against other well established theorem provers. We should note that it was not easy to setup the necessary experiments, since most of the existing tools are not maintained anymore or hard to be installed. On the other hand, installable provers were not customisable in order to be called automatically on a set of significant formulas. This is why we settle for the following basic results.

The authors of TWB published in [Aba07] the results of the comparison of TWB and LWB on the basis of the LWB benchmark (section 11.4.1). By using the same benchmark in the same conditions (used amount of memory...) we give in the column (a) of Figure 11.4 the results of the comparison of LoTREC to LWB and TWB theorem provers.

The charts should be read as follows: in the first chart of column (a) above the class name "S4_branch_n", we notice that LWB can treat the 15th formula within 100 seconds, TWB can handle the 14th whereas LoTREC can maximally deal with the 3rd. In the same chart, but considering the class "S4_S5_p", LoTREC and LWB can treat the hardest formula, the 21th, whereas TWB can treat at max the 16th.

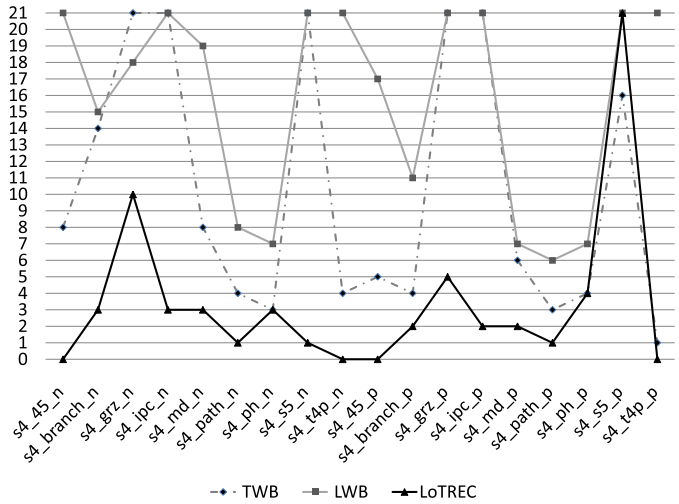
These results show that LoTREC is generally less efficient than other theorem provers even if it does better in some classes of formulas. It seems that LoTREC has some advantages on formulas where there are less non-deterministic choices, even if they have a high branching factor.

Similar results are shown for formulas of K and KT in Figure 11.4.

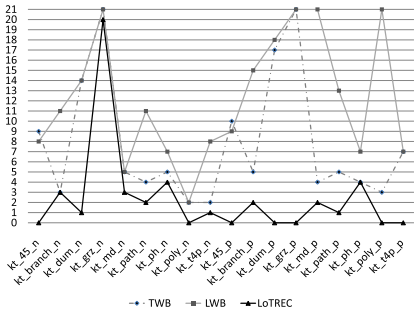
Some implementations get complicated

The implementation of some logics requires some special techniques, as we have seen in the first chapters of this thesis. Some of these techniques are complicated.

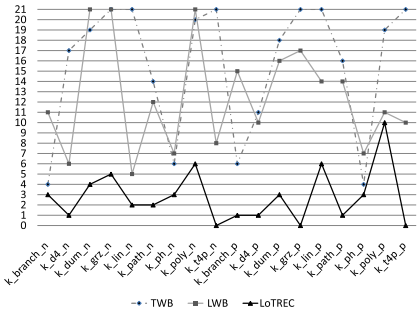
The first example is the need for a common root node linked to every node, such as in S5 (section 4.6) or HL(@) (section 5.3). In such cases, the problem is that we are dealing with *universal* modalities or formulas whose truth value should be *globally* shared between all the (or many) nodes in the premodel. This would have simply been replaced by a single condition of the form



(a) S4 benchmark



(b) KT benchmark



(c) K benchmark

Figure 11.4: Experimental results of benchmark with other provers.

`thereIsANode n` (such that, some other conditions hold in it).

The second example is the list of complex manipulations given in page 118 for HL(@). These manipulations would have been easier if we had in LoTREC a simple action to replace them all, such as `merge node1 node2`.

However, we always prefer not to extend LoTREC's language when a new logic with new semantic conditions is treated, as long as it can be treated by the same set of predefined conditions and actions. This simple language/complex method dilemma is best quoted by Larry Wall from *Programming Perl* [SC97]:

> *Minimalism: The belief that "small is beautiful". Paradoxically, if you say something in a small language, it turns out big, and if you say it in a big language, it turns out small. Go figure.*

Our choice is to keep on our philosophy of small language, even if what we

aim to say will turn out big.

LoTREC in action

LoTREC website is often visited and run via the web pages of several courses in logics, such as:

- *automated reasoning*, Prof. C. Pecheur[8], at Université catholique de Louvain, Belgium;
- *FGI 3 - Logik*[9], Dr. C. Eschenbach, University of Hamburg, Germany;
- *Logique, informatique et sciences cognitives*[10], Prof. R. Villemaire, University of Quebec At Montreal, Canada.

Last year, LoTREC was the subject of a tutorial course given at *Tableaux 2009*[11]. This spring, LoTREC is accompanying a tutorial course that will be held at the *3rd World Congress and School on Universal Logic*[12].

This summer, we shall be giving an introductory course on modal logics at *ESSLLI 2010*[13]. LoTREC will be used by the students to achieve the various exercises.

LoTREC is used for prototyping purposes by the members of our research team LILaC. It was also used by others [BGMS08] to implement a tableaux method for a logic of time sub-intervals.

The tool is executable directly from a web browser by visiting LoTREC's home page:

`http://www.irit.fr/Lotrec`

Ongoing work

The work on a generic platform such as LoTREC must always be a"work in progress". New logics with different semantics emerge while research in logic and philosophy goes on. Hence, new decision procedures are to be designed for these logics, and thus new implementation techniques have to be settled for these procedures.

Currently I am revisiting the logics with the converse modalities, especially transitive logics, in both monomodal and multimodal cases, in order to better understand the behavior of transitive logics with converse and with iteration, such as PDL with converse.

[8] `http://www.info.ucl.ac.be/~pecheur`

[9] `http://www.informatik.uni-hamburg.de/WSV/teaching/vorlesungen/Logik_WiSe09.¬shtml`

[10] `http://www.info2.uqam.ca/~villemaire_r/9305.html`

[11] `http://heim.ifi.uio.no/martingi/Tableaux09/pmwiki.php/Tableaux2009/Tutorials`

[12] `http://www.uni-log.org/t3-kripkeworld.html`

[13] `http://www.irit.fr/ACTIVITES/LILaC/Pers/Herzig/CTableaux/Esslli10.html`

I will be also working on the implementation of other logics that seem more exotic than usual modal logics but implementable in our framework, , such as multi-valued logics, since their semantics can be represented in terms of Kripke models.

On the practical side, we would like to have a backtracking mechanism in order to avoid model duplication if demanded by the user, as if LoTREC is asked to run in a satisfiability-checker mode. I believe that this is possible but not easy. In fact, since there is no automatic backtrack mechanism in Java, we have to implement our own backtrack mechanism. This may be easily done if we did not have to deal with the event-driven mechanism, which is very sensitive to any code modification. This is why this task would be tough.

The philosophy of LoTREC seems to be successful: we can implement a wide variety of logics with a simple language and as few tweaks as possible. However, it seems that simple extensions may make some methods simpler and easier as discussed above.

A promising solution is to offer the user a query-like language which allows nesting of queries. This would allow to nest different quantifiers, which makes the definition of some rules with complex conditions easier, such as in the case of CTL. Triggers and transactions can be used in a query language to roll-back on incoherent updates in databases. This seems appealing since it simulates backtracking on failure choices. On the other hand, we are not yet sure if such a language will be easy-to-learn by non-computer scientists, as is the case for the currently used language. Moreover, we are not sure how to proceed in order to keep on the nice properties of our pattern matching process in such a database environment.

Conclusion

Dans la première partie de cette thèse, j'ai donné une introduction ludique et pedagogique à la logique modale. Cette introduction est donné dans une approche applicative (Chapitre 1 et Chapitre 2), et via *la méthode de construction de modèles* qui ressemble à la méthode de tableaux.

Tout au long des chapitres 3, 4, 5 et 7, j'ai présenté des procédures de décision pour une grande variété de logiques modales. Également, j'ai montré comment implémenter ces procédures, comment les exécuter et comment les déboguer dans notre plateforme générique LoTREC.

Ce travail, ayant un style pédagogique, sera complété dans le future proche sous la forme d'un livre, à accomplir en collaboration avec O. Gasquet, A. Herzig et F. Schwarzentruber, et qui servira comme introduction ludique et sympathique à la logique modale.

Une méthode particulière de construction de modèles a été utilisée pour étudier les propriétés des "frames" et la complexité d'une famille particulière de logiques, qu'on appelle *Layered Modal Logics* ou LML (chapitre 8). Pour conserver une taille exponentielle des modèles construits, nous avons utilisé une technique de filtration spéciale que nous appelons *filtration dynamique*. La méthode résultante prouve que la classe de ces logiques est NEXPTIME-complet. En plus, je montre dans ce chapitre comment implémenter pratiquement de telles techniques de filtration dans notre outil LoTREC.

Dans la seconde partie de cette thèse, j'ai commencé par une introduction générale à la réécriture de graphes (chapitre 9). Elle introduit les différents modèles mathématiques utilisés pour définir les graphes et les règles de réécriture de graphes avec une base théorique solide. A la fin de ce chapitre, j'ai discuté brièvement certaines des propriétés théoriques intéressantes et que l'on aimerait bien avoir dans un système de réécriture de graphes.

Au chapitre 10, j'ai présenté le système de réécriture de graphes de LoTREC. Cela consiste à préciser le type de graphes traités par LoTREC, la manière dont les règles et les stratégies sont définies, et les principales contraintes et restrictions que nous imposons sur ces définitions. Ce chapitre aboutit à sa fin à une discussion sur certaines des propriétés théoriques de LoTREC, considéré comme un système de réécriture de graphes.

Au dernier chapitre de cette partie, j'ai présenté le mécanisme "event-driven pattern matching" de LoTREC (chapitre 11). En utilisant une notation de termes, la sémantique de la réécriture de graphes qui utilise ce mécanisme événe-

mentiel est formalisée. Ensuite, elle est montré adéquate et correcte par rapport à la sémantique habituelle de réécriture. À la fin de ce chapitre, je compare cette technique événementielle avec d'autres techniques utilisées par la communauté de réécriture de graphes, et je donne certains résultats expérimentaux qui montrent l'effet de notre technique dans l'amélioration de la performance moyenne du processus de réécriture.

Au-delà de ça, les autres objectifs que j'avais évoqués au début de ce manuscrit ont été atteints avec succès. Ils ont été abordés au fur et à mesure et à différentes reprises dans ce manuscrit. Pour en faire un petit bilan, je rappelle que: d'abord, la plate-forme LoTREC a été maintenue et renforcée pendant toute la durée de ma thèse; deuxièmement, l'outil a été rendue plus conviviale et plus accessible au public ciblé; et troisièmement, plusieurs nouvelles logiques ont été implémentée avec succès, tout en gardant le langage de définition des règles la plus simple possible.

Appendix A

Signatures, algebras and terms

A signature consists of a syntactical description of an *algebra* as a set of types called *sorts* and a set *operations* to define computations on these sorts. An algebra can be then defined over a signature to give it a semantics. Thus many algebras can be defined over the same signature. You may consider a signature and a corresponding algebra exactly as an interface and one of its implementations in Object Oriented Programming. For a deeper introduction to algebraic signatures, see [EM85].

Definition 75 (algebraic signature)**.** *An algebraic signature $SIG = (S, OP)$, or signature for short, consists of a set S of sorts and a family OP of operation symbols. OP is the union of two disjoint sets*

- $(K_s)_{s \in S}$ *the set of constant symbols of the sort $s \in S$;*
- $(OP_{w,s})_{(w,s) \in S^+ \times S}$ *the set of operations that take in arguments of sorts $s_1, \ldots, s_n = w \in S^+$ and has a return value belonging to the sort $s \in S$.*

Remark 18. For an operation symbol $op \in OP_{w,s}$, we write $op : w \to s$ or $op : s_1, \ldots, s_n \to s$, where $w = s_1, \ldots, s_n$.

Definition 76 (SIG-algebra)**.** *For a given signature $SIG = (S, OP)$, a SIG-algebra $A = ((A_s)_{s \in S}, (op_A)_{op \in OP})$ is defined by*

- *for each sort $s \in S$, a set A_s, called the carrier set;*
- *for a constant symbol $k \in K_s$ of a sort $s \in S$, a constant $k_A \in A_s$;*
- *for each operation symbol $op : s_1, \ldots, s_n \to s \in OP$, a mapping function $op_A : A_{s_1}, \ldots, A_{s_n} \to A_s$.*

Now we really need some examples to better understand what a signature and an algebra are.

Example 37. The following signature describes the natural numbers interface. It has only one sort *nat*, a constant symbol "zero", and three operation symbols for a successor, an addition and a multiplication operations.

$$\begin{array}{llllll} NAT & = & & & \\ \text{sorts} & : & nat & & \\ \text{operators} & : & zero & : & \to nat \\ & & succ & : & nat \to nat \\ & & add & : & nat, nat \to nat \\ & & mult & : & nat, nat \to nat \end{array}$$

The standard implementation of the NAT signature is the following algebra:

$$\begin{array}{lll} A_{nat} & = & \mathbb{N} \\ zero_A & = & 0 \in A_{nat} \\ succ_A & : & A_{nat} \to A_{nat} \\ & & x \to x+1 \\ add_A & : & A_{nat} \times A_{nat} \to A_{nat} \\ & & (x,y) \to x+y \\ mult_A & : & A_{nat} \times A_{nat} \to A_{nat} \\ & & (x,y) \to x.y \end{array}$$

Some other examples can be found in [EEPT06], Appendix B.

In order to be able to define a category on signatures, we define signature morphism as following:

Definition 77 (signature morphism)**.** *Given signatures $SIG = (S, OP)$ and $SIG' = (S', OP')$, a signature morphism $m : SIG \to SIG'$ is a pair of mappings $m = (m_S : S \to S', m_{OP} : OP \to OP')$ such that $m_{OP}(op) : m_S(s_1), \ldots, m_S(s_n) \to m_S(s) \in OP'$, for all $op : s_1, \ldots, s_n \to s \in OP$.*

Definition 78 (category Sig)**.** *Algebraic signatures together with signatures morphisms define the category* Sig *of signatures.*

Since many algebras may implement the same signature corresponding to different semantics, we define homomorphisms on algebras to analyze relations and similarities between algebras.

Definition 79 (algebra homomorphism)**.** *Given a signature $SIG = (S, OP)$ and two SIG-algebras A and B, an algebra homomorphism $m : A \to B$ is a family $m = (m_s)_{s \in S}$ of mappings $m_s : A_s \to B_s$ such that*

- *for each constant symbol $k \in K_s$, we have $m_s(k_A) = k_B$;*
- *for each operation $op : s_1, \ldots, s_n \to s \in OP$, it holds that $m_s(op_A(x_1, \ldots, x_n)) = op_B(m_{s_1}(x_1), \ldots, m_{s_n}(x_n))$ for all $x_i \in A_{s_i}$.*

Note that:

Definition 80 (category $\mathsf{Alg}(SIG)$)**.** *Given a signature SIG, SIG-algebras and algebra homomorphisms define the category* $\mathsf{Alg}(SIG)$ *of SIG-algebras.*

In the following, we define the *final algebra* of a signature SIG, which can be considered as the terminal object of the category $\mathsf{Alg}(SIG)$.

Definition 81 (final algebra)**.** *The final SIG-algebra of a signature $SIG = (S, OP)$, denoted by Z, is defined by*

$$Z_s = s, \text{ for all } s \in S.$$

A.1 Terms and term algebras

Terms with and without variables can be constructed over a signature SIG and evaluated in each SIG-algebra.

Definition 82 (variables and terms)**.** *Let $SIG = (S, OP)$ be a signature. Let $X = (X_s)_{s \in S}$ be a family of sets, where X_s is the set of variables of sort s, for each sort $s \in S$. We assume that these X_s are pairwise disjoint and are disjoint with OP. The set of terms with variables of sort s, denoted as T_s, is inductively defined as follows:*

- $X_s \cup K_s \subset T_s$,
- $op(t_1, \ldots, t_n) \in T_s$ *for every operation symbol* $op : s_1, \ldots, s_n \to s \in OP$ *and all terms* $t_i \in T_{s_i}$ *for* $i = 1, \ldots, n$,
- *no other terms in* T_s.

The set of ground terms of sort s, denoted as $\mathcal{G}T_s$, is defined similarly to T_s but with an empty set of variables, i.e. as if $X_s = \emptyset$.

We denote by $T = (T_s)_{s \in S}$ the family of terms with variables of SIG, and by $\mathcal{G}T = (\mathcal{G}T_s)_{s \in S}$ the family of terms without variables (i.e. ground terms) of SIG.

Definition 83 (term algebra)**.** *The term algebra over a signature $SIG = (S, OP)$ and a family of variables X is defined as $TA = ((T_s)_{s \in S}, (op_{TA})_{op \in OP})$, where the carriers sets consist of terms with variables, and the operations are defined by*

- $k_{TA} = k \in TA$ *for all constants* $k \in OP$,
- $op_{TA} : T_{s_1} \times \ldots \times T_{s_n} \to T_s$ *such that* $op_{TA}(t_1, \ldots, t_n) = op(t_1, \ldots, t_n)$ *for all* $op : s_1, \ldots, s_n \to s \in OP$.

Appendix B

Technical details

B.1 Graph construction in LoTREC

LoTREC provides three different ways to build a graph:

- drawing a graph by mouse clicks: this is intuitively done using the graphical user interface;
- loading a graph from an file (if it has been first created with LoTREC);
- or using a rule.

Here we show how to use a rule to create a simple greeting graph (Figure B.1):

```
Rule Build_Greeting_Graph
  createNewNode n
  createNewNode n'
  add  n    Welcome
  link n n' To
  add  n'   KripkesWorld
End
```

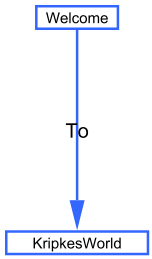


Figure B.1: simple greeting graph

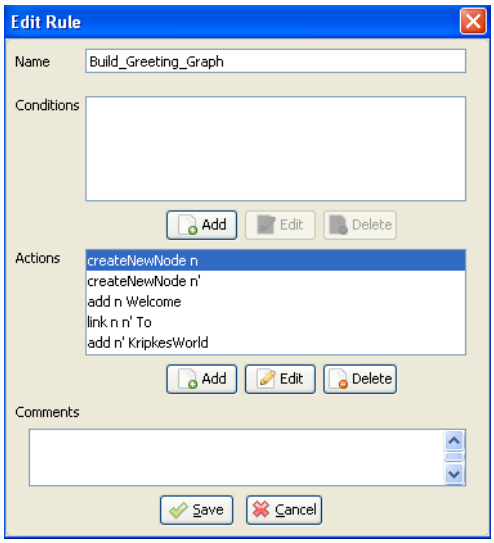


Figure B.2: rule definition window

In the graphical user interface of LoTREC, we only need to fill the program part of the rule "`Build_Greeting_Graph`", as shown in figure B.2.

Remark 19. We do not need to fill the *conditions* part appearing in the rule definition window of figure B.2. The utility of conditions is introduced later in chapter 3.

To apply this rule definition effectively in LoTREC, we need to call it in a sort of program that we call *strategy*. The simplest way is to put the name of the rule in the body of a strategy code (as shown in figure B.3). We may call this strategy `Greeting_Graph_Builder`.

```
Strategy Greeting_Graph_Builder
  Build_Greeting_Graph
End
```

Remark 20. Starting form chapters 3, we introduce more elaborated strategies.

Remark 21. By default, LoTREC draws the graphs using a *Hierarchic* layout. When needed, the user may choose the *Circular* layout.

B.2 Layout and display

Graphs in LoTREC are displayed in a pretty-print graphical interface. They are plotted by default with an hierarchical layout. However, the user may choose other layout algorithms, such as the circular layout, if this would make the graphs more readable. The layout algorithms of LoTREC are based on yFiles

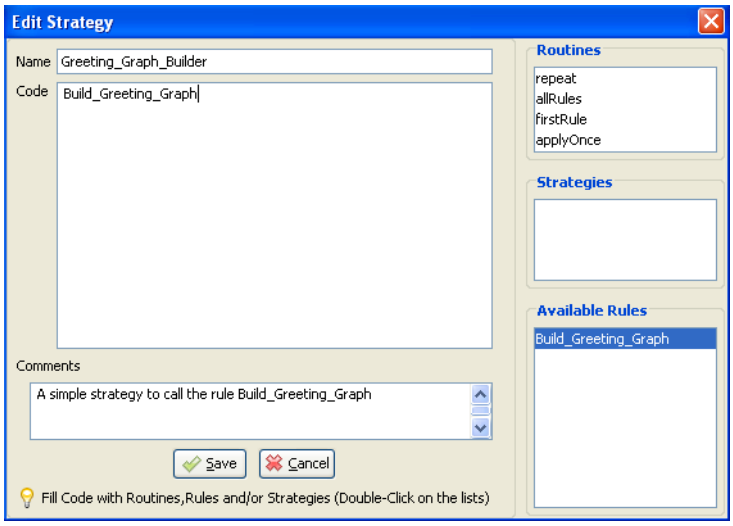


Figure B.3: strategy definition window

[WEK01]: one of the most efficient graph layout Java library, but an expensive commercial one. However, yFiles algorithms are shipped as free plugins in Cytoscape[1], and this is how we integrated them freely in LoTREC.

We should clarify that the position of a node or an edge, calculated according to these layouts, does not infer any syntactic or semantic information. Thus, two different displays of the same graph are equivalent. Exactly as they are two different files of the same program: written on one line or in a logical structure with colored keywords, the program remains the same for the compiler.

For the sake of readability, we adopt in LoTREC the following options in graphs display:

1. an attribute, such as a formula or a mark, has an internal *name*, *type* and *value*. However, only the value is displayed,

2. an edge is represented as an arc. It has exactly one attribute: a formula that is displayed on the middle of the arc,

3. an edge is often referenced by its source and target nodes and its label formula. Thus, edges names are never displayed,

4. nodes names are hidden by default, although the user may get the name of a given node by a simple right-click on it,

5. a node is displayed as a rectangle, and all its attributes are displayed inside this rectangle:

[1] http://www.cytoscape.org/

(a) marks of the node, if there are any, are displayed on the first line at the top of the rectangle within two brackets,

(b) then its formulas are displayed one by line. A formula is followed by its marks between two brackets, if it has any.

Bibliography

[Aba07] Pietro Abate. *The Tableau Workbench: a framework for building automated tableau-based theorem provers.* PhD thesis, Australian National University, 2007.

[ABM01] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: characterization, interpolation and complexity. *The Journal of Symbolic Logic*, 66(3):977–1010, 2001.

[AG03] Pietro Abate and Rajeev Goré. The tableaux work bench. In *TABLEAUX 2003, LNCS*, volume 2796, pages 230–236. Springer, 2003.

[AIK06] Oana Andrei, Liliana Ibanescu, and Hélène Kirchner. Non-intrusive formal methods and strategic rewriting for a chemical application. In *Essays Dedicated to Joseph A. Goguen*, pages 194–215, 2006.

[Bal00] Matteo Baldoni. Normal multimodal logics with interaction axioms. pages 33–57, 2000.

[BdRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic.* Cambridge Tracts in Theoretical Computer Science. University Press, 2001.

[BFG96] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the practical use of graph rewriting. In *Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science*, pages 38–55, London, UK, 1996. Springer-Verlag.

[BFK00] Michael R. Berthold, Ingrid Fischer, and Manuel Koch. Attributed graph transformation with partial attribution. In H. Ehrig and editors G. Taentzer, editors, *Proceedings of GRATRA'2000 (joint appligraph and getgrats workshop on graph transformation systems), ETAPS 2000*, Berlin, March 2000.

[BGMS08] David Bressolin, Valentin Goranko, Angelo Montanari, and Pietro Sala. Tableau-based decision procedures for the logics

of subinterval structures over dense orderings. *Journal of Logic and Computation Advance Access (to appear)*, 2008.

[BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 402–429, London, UK, 2002. Springer-Verlag.

[BHS00] Peter Balsiger, Alain Heuerding, and Stefan Schwendimann. A benchmark method for the propositional modal logics k, kt, s4. *Journal of Automated Reasoning*, 24(3):297–317, 2000.

[BJ01] M. Bauderon and H. Jacquet. Pullback as a generic graph rewriting mechanism. *Applied Categorical Structures*, 9:65–82(18), January 2001.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[CFnDCGH98] Marcos Alexandre Castilho, Luis Fariñas Del Cerro, Olivier Gasquet, and Andreas Herzig. Modal Tableaux with Propagation Rules and Structural Rules (position paper) . janvier 1998.

[Che80] Brian F. Chellas. *Modal logic, an introduction.* Cambridge University Press, 1980.

[Cho02] Noam Chomsky. *Syntactic Structures.* Walter de Gruyter, 2nd edition, December 2002.

[CM97] Serenella Cerrito and Marta Cialdea Mayer. Hintikka multiplicities in matrix decision methods for some propositional modal logics. In *TABLEAUX '97: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 138–152, London, UK, 1997. Springer-Verlag.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [Roz97], pages 313–400.

[dCFG+01] Luis Fariñas del Cerro, David Fauthoux, Olivier Gasquet, Andreas Herzig, Dominique Longin, and Fabio Massacci. Lotrec: The generic tableau prover for modal and description logics. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, volume 2083, pages 453–458, London, UK, 2001. Springer-Verlag.

[dGM00] Giuseppe de Giacomo and Fabio Massacci. Combining deduction and model checking into tableaux and algorithms for converse-pdl. *Inf. Comput.*, 162(1/2):117–137, 2000.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation, Volume 2: applications, languages, and tools*, River Edge, NJ, USA, 1999. World Scientific Publishing Co., Inc.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, pages 1–69, London, UK, 1979. Springer-Verlag.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics (EATCS Monographs in Theoretical Computer Science, No. 6)*. Springer-Verlag, Berlin Heidelberg, 1985.

[EP06] H. Ehrig and U. Prange. Modeling with graph transformations. In G. E. Lalsker and J. Pfalzgraf, editors, *Advances in Multiagent Systems, Robotics and Cybernetics: Theory and Practice. Proceedings of Intern. Conf. on Systems Research, Informatics and Cybernetics 2005*, volume 1, 2006.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2004.

[ER91] Joost Engelfriet and Grzegorz Rozenberg. Graph grammars based on node rewriting: An introduction to nlc graph grammars. In *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 12–23, London, UK, 1991. Springer-Verlag.

[FdCGHS05] Luis Fariñas del Cerro, Olivier Gasquet, Andreas Herzig, and Mohamad Sahade. Modal tableaux: completeness vs. termination. In Serguei Artemov, Howard Barringer, Artur d'Avila Garcez, Luis C. Lamb, and John Woods, editors, *We will show them! Essays in honour of Dov Gabbay*, volume 1, pages 587–614. College Publications, 2005.

[Fit83] Melvin Fitting. *Proof methods for modal and intuitionistic logics.* D. Reidel, Dordrecht, 1983.

[FL77] Michael J. Fischer and Richard E. Ladner. Propositional modal logic of programs. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 286–294, New York, NY, USA, 1977. ACM.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 296–309, London, UK, 2000. Springer-Verlag.

[Gab70] D. M. Gabbay. Selective filtration in modal logic. *Theoria*, pages 30:323–330, 1970.

[Gas06] Olivier Gasquet. On the influence of confluence in modal logics . *Fundamenta Informaticae*, 70(3):227–250, 2006.

[GHS06a] Olivier Gasquet, Andreas Herzig, and Mohamad Sahade. Terminating modal tableaux with simple completeness proof. In Guido Governatori, Ian M. Hodkinson, and Yde Venema, editors, *Advances in Modal Logic*, pages 167–186. College Publications, 2006.

[GHS06b] Olivier Gasquet, Andreas Herzig, and Mohamad Sahade. Terminating modal tableaux with simple completeness proof. In Guido Governatori, Ian Hodkinson, and Yde Venema, editors, *Advances in Modal Logic*, volume 6, pages 167–186. King's College Publications, 2006.

[Gor99] Rajeev Goré. Tableau methods for modal and temporal logics. In Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*,

pages 297–396, Hingham, MA, USA, 1999. Kluwer Academic Publishers.

[GS07] Olivier Gasquet and Bilal Said. Tableaux with Dynamic Filtration for Layered Modal Logics. In Nicola Olivetti, editor, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX), Aix-en-Provence, 03/07/2007-06/07/2007*, number 4548 in LNAI, pages 107–118, http://www.springerlink.com/, 2007. Springer-Verlag.

[GSS09] Olivier Gasquet, Bilal Said, and François Schwarzentruber. A semantics for an event based generic tableau prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX), Oslo, Norway, 06/07/2009-10/07/2009*, http://www.uio.no/, 2009. University of Oslo.

[GvVV09] Elske van der Vaart Gert van Valkenhoef and Rineke Verbrugge. OOPS: An $\mathsf{S5}_n$ prover for educational settings. In *6th Workshop on Methods for Modalities (M4M-6)*, Copenhagen (Denmark), November 2009. Elsevier ENTCS. To appear.

[Ham97] Graham Hamilton. JavaBeans: Api specification. Technical report, Sun Microsystems, 1997.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MA:MIT PRESS, 2000.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 161–176, London, UK, 2002. Springer-Verlag.

[HSZ96] Alain Heuerding, Michael Seyfried, and Heinrich Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In *TABLEAUX '96: Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 210–225, London, UK, 1996. Springer-Verlag.

[IT09] Tetsuo Ida and Hidekazu Takahashi. Origami fold as algebraic graph rewriting. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1132–1137, New York, NY, USA, 2009. ACM.

[Lad77] Richard E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM J. Comput.*, 6(3):467–480, 1977.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & yacc, 2nd edition.* O'Reilly, 1992.

[LS77] E. J. Lemmon and D. S. Scott. *An introduction to modal logic.* Oxford, Blackwell, 1977.

[Mas00] Fabio Massacci. Single step tableaux for modal logics. *J. Autom. Reasoning*, 24(3):319–364, 2000.

[MGW96] A. P. Martin, Paul H. B. Gardiner, and Jim Woodcock. A tactic calculus-abridged version. *Formal Aspects of Computing Journal*, 8(4):479–489, 1996.

[Plu95] Detlef Plump. On termination of graph rewriting. In *WG '95: Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 88–100, London, UK, 1995. Springer-Verlag.

[Plu99] D. Plump. Term graph rewriting. In Ehrig et al. [EEKR99], pages 3–61.

[Pra80] Vaughan R. Pratt. A near-optimal method for reasoning about action. *J. Comput. Syst. Sci.*, 20(2):231–254, 1980.

[RFS08] Maxime Rebout, Louis Féraud, and Sergei Soloviev. A unified categorical approach for attributed graph rewriting. In Edward A. Hirsch, Alexander A. Razborov, Alexei L. Semenov, and Anatol Slissenko, editors, *CSR*, volume 5010 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 2008.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, River Edge, NJ, USA, 1997. World Scientific.

[Rud00] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 238–251, London, UK, 2000. Springer-Verlag.

[Sah04] Mohamad Sahade. The conditions and actions in lotrecŠs language. Technical report, Institut de recherche en informatique de Toulouse, IRIT, 2004.

[SC97] Randal L. Schwartz and Tom Christiansen. *Learning PERL.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. Foreword By-Wall, Larry.

[Sch91] Andy Schürr. Progress: A vhl-language based on graph grammars. In *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 641–659, London, UK, 1991. Springer-Verlag.

[Sch97] A. Schfürr. Programmed graph replacement sytems. [Roz97], pages 479–546.

[Seg68] Krister Segerberg. Decidability of s4.1. *Theoria*, 34(1):7–20, 1968.

[SG08] Bilal Said and Olivier Gasquet. Efficient Graph Rewriting System Using Local Event-driven Pattern Matching. 2008.

[Sha04] Ilya Shapirovsky. On pspace-decidability in transitive modal logic. In Schmidt et al. [SPHRW05], pages 269–287.

[She04] Valentin B. Shehtman. Filtration via bisimulation. In Schmidt et al. [SPHRW05], pages 289–308.

[SPHRW05] Renate A. Schmidt, Ian Pratt-Hartmann, Mark Reynolds, and Heinrich Wansing, editors. *Advances in Modal Logic 5, papers from the fifth conference on "Advances in Modal logic," held in Manchester (UK) in September 2004*. King's College Publications, 2005.

[Tae99] Gabriele Taentzer. Adding visual rules to object-oriented modeling techniques. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 275, Washington, DC, USA, 1999. IEEE Computer Society.

[Var09] M. Cialdea Mayer S. Cerrito E. Benassi F. Giammarinaro C. Varani. Two tableau provers for basic hybrid logic. Technical Report RT-DIA-145-2009, Dipartimento di Informatica e Automazione, Università di Roma Tre, 2009.

[VV04] Gergely Varró and Dániel Varró. Graph transformation with incremental updates. *Electronic Notes in Theoretical Computer Science*, 109:71 – 83, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[WEK01] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yfiles: Visualization and automatic layout of graphs. In *Graph Drawing*, pages 453–454, 2001.

[Z̈96] Albert Zündorf. Graph pattern matching in progres. In *Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science*, pages 454–468, London, UK, 1996. Springer-Verlag.

[ZS92] Albert Zündorf and Andy Schürr. Nondeterministic control structures for graph rewriting systems. In *WG '91: Proceedings of the 17th International Workshop*, pages 48–62, London, UK, 1992. Springer-Verlag.

Index

LoTREC, 13
- connectors, 46, 215
- history, 18, 189
- pattern matching, event based, 241
- rules, 39, 62, 219
- strategies, 65, 229

Completeness, 73
- of a graph rewriting system, 207
- vs. termination, 121

Logics, list of
- HL(@), the basic Hybrid Logic, 113
- K, the basic modal logic, 56
- KB, 78
- K_2 + Inclusion, K+K+Inclusion, 71
- KT, 77
- K4, 99
- LML, Layered Modal Logics, 167
- LTL, Linear Temporal Logic, 139
- K_n, the basic multi-modal logic, 71
- PDL, Propositional Dynamic Logic, 151
- S5, 91
- S5 $_n$, multi-S5, 107
- S4, 104
- K+Confluence, 86
- KB4, 104
- KD4, 104
- KD, 83
- $K.alt_1$, 80

Model Checking, 51, 127, 130

Model construction, 52, 56, 62, *see* Logics

Satisfiability, *see* Model Construction
- definition, 51

Soundness, 73

Tableau methods, *see* Logics, Model Construction, Satisfiability

Techniques
- fulfillment of eventualities, 146, 158
- filtration, 117
  - dynamic filtration, 182
- history, keeping data in a, 103
- loop blocking, 102
- node-equality test, 149, 156
- node-inclusion test, 103, 144
- global data, sharing, 93, 115

Termination, 73, 101, 105, 120, 144, 149, 156
- general theorems, 96, 124
- of a graph rewriting system, 207

Validity, *see* Satisfiability
- definition, 51