DEVELOPMENT OF A TABLEAUX RESOLUTION PROVER

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2009

By
Rawan Ghali AlBarakati
School of Computer Science

Contents

Abstract **7**

Declaration **8**

Copyright **9**

Dedication **10**

Acknowledgements **11**

1 Introduction **12**

2 Modal logic tableaux **15**

2.1 Traditional modal logic 15
2.2 Dynamic modal logics 18
2.3 Tableau for modal logic 21
2.4 Formula normalization 25

3 Resolution **28**

3.1 First-order resolution 28
3.2 Ordered hyper-resolution 32
3.3 The resolution theorem prover SPASS 33

4 Simulation of tableaux via first-order resolution **39**

4.1 Structural transformation for first-order logic 39
4.2 Structural transformation for dynamic modal logic 41
4.3 Simulating tableau for $K_{(m)}(\wedge, \vee, \smile)$ 46

5 Using SPASS as a Tableau Resolution Prover **50**

5.1 Normalisation of input 50

5.2 Associating modal formulae with first-order translations 51
5.3 A new approach to renaming 53
5.4 From resolution to tableau 62
5.5 Setting tableau step indentation 68
5.6 Tableau proofs and models 72
5.7 Branching comes last 74

6 Adding relational frame conditions 78
6.1 Simulating relational conditions 78
6.2 Redefining tableau rules 80
6.3 Representing rule nodes in the output 86
6.4 Making all rules structural 88

7 Tableau simulation results 90
7.1 Expected input 90
7.2 Avoiding redundancy 91
7.3 Testing renaming of negated formulae 92
7.4 Testing renaming of relational formulae 95
7.5 Testing proof indentation 97
7.6 Applying non-branching rules first 98
7.7 Intelligent backtracking in SPASS 101
7.8 Testing frame conditions on traditional modal logics 103
7.9 Relational frame conditions in dynamic modal logic 106

8 Conclusion 108

A Full input example 111

B Full output example 112

Bibliography 115

List of Figures

2.1 Defined operators 16
2.2 Basic modal logic semantics 17
2.3 Modal logic axioms, corresponding properties and first-order definitions 18
2.4 Dynamic modal logic semantics 19
2.5 Directed graph example of a Kripke frame 20
2.6 Translation to first-order logic for $K_{(m)}(\wedge, \vee, \smile)$ 21
2.7 First-order translation of $[r_1][r_2](p \wedge \neg q)$ 21
2.8 Basic modal logic tableaux Calculus 22
2.9 Structural tableau rules for axioms T, D, B, 4 and 5 23
2.10 Tableaux Calculus for $K_{(m)}(\wedge, \vee, \smile)$ 23
2.11 Tableau branching rules with semantic branching 25
2.12 Comparison between two derivations of equivalent formulae 26
2.13 Rewrite rules 27

3.1 Resolution derivation example 29
3.2 The resolution calculus $R^{\text{red}}_{\text{sp}}$ 31
3.3 Comparison 33
3.4 Input problem example in DFG syntax 35
3.5 SPASS ouput 36

4.1 Definition of the translation mappings π' and τ' 43
4.2 Definitional clausal forms for $K_{(m)}(\wedge, \vee, \smile)$ 44
4.3 Transformation of $\Xi(\varphi)$ into clausal form 46
4.4 Using hyper-resolution H_{sp} on N from Figure 4.3 48

5.1 Mapping first-order translations to original modal formulae 52
5.2 Introducing fresh Skolem predicates for first-order formulae associated with non-atomic modal formulae 54

5.3 Results for applying structural transformation and the produced clausal form 55
5.4 Converse simplification rewrite rules 58
5.5 Association of first-order and dynamic formulae, and introduced symbols 61
5.6 Structural transformation and clausal form of renaming example . 61
5.7 Tableau steps data structure with examples 62
5.8 Encoding of tableau rules 64
5.9 Algorithm for translating input definitional clauses to tableau rules 65
5.10 Using indentation to represent tableau branches 69
5.11 Corresponding tree representation 69
5.12 Example of a branches table 71
5.13 Illogical proof structure 75
5.14 Tree representation of illogical proof structure 76
5.15 Algorithm for choosing a clause not suitable for splitting 77

6.1 Definitions and produced clausal form for axioms T, D, B, 4 and 5 78
6.2 Tableau rules for axiom properties 81
6.3 Tableau application rules for traditional modal logic 82
6.4 New tableaux calculus for $K_{(m)}(\wedge, \vee, \smile)$ 86
6.5 Properties clauses and rule translations 87
6.6 Derivation of $[r][r]\neg p \wedge [r]p$ in KD4 with defined worlds 89

7.1 Avoidance of redundancy 91
7.2 Testing negated formulae renaming - 1 93
7.3 Testing negated formulae renaming - 2 94
7.4 Testing relational formula renaming 95
7.5 Testing proof indentation 97
7.6 Testing the logical order of derivation steps - 1 99
7.7 Testing the logical order of derivation steps - 2 100
7.8 Intelligent backtracking 102
7.9 Symmetry and axiom B 103
7.10 Reflexivity and axiom T 104
7.11 Seriality and axiom D 104
7.12 Transitivity and axiom 4 105
7.13 Euclideanness and axiom 5 105

7.14 Relational frame conditions on dynamic logic 106

Abstract

Proof systems carry great benefit for the field of artificial intelligence because of the reasoning services they provide for AI applications. Semantic tableau and first-order resolution are two proof systems that operate considerably differently from each other. When choosing which proof system to use for an application, the decision bears a trade-off between efficiency and readability. Semantic tableau is considered more comprehensible and user-friendly, but when it comes to performance, it is hard to decide which is better because there are many factors to consider when benchmarking [HS99]. Some researchers studied the two proof systems in search of a link between them and found that despite the differences in operation resolution provers can be adapted to simulate semantic tableau derivations. The result of this simulation enables testing the performance of the two systems on common grounds. In this thesis, I extend the first-order resolution theorem prover SPASS so that it simulates semantic tableau and provide tableau proofs and models translated from resolution proofs and saturated sets of clauses. The logics considered are traditional modal logics and dynamic modal logics defined over relations closed under intersection, union and converse with the support of adding relational frame conditions.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of Computer Science (or the Vice-President).

Dedication

I dedicate this thesis to the memory of my brother Dr. Rakahn Ghali AlBarakati. May God have mercy on his soul.

Acknowledgements

First and above all, I am grateful to God -the one and only- who is worthy of every praise for the opportunity, impetus and inspiration that lead me to completing and presenting this thesis.

I would also like to express my sincere appreciation to my supervisor Dr. Renate A. Schmidt. Her constructive remarks, guidance and enlightenment on my work have been invaluable.

I am sincerely grateful to my parents Mrs. Elham AlBarakati and Dr. Ghali G. AlBarakati and the rest of my family who are the reason I am who I am today, and for the care and love they have always provided. I am especially thankful to my mother for her continuous prayers and blessings.

I am forever in debt to my husband Khaled Ali Fawaz for his utmost understanding, and ultimate support, love and encouragement.

Last but not least, I am grateful to my country the Kingdom of Saudi Arabia and precisely the Ministry of Education for supporting and financially aiding my studies throughout this year.

Chapter 1

Introduction

The aim of this dissertation is to implement a tableau-resolution prover using the first-order resolution prover SPASS.

The simulation of semantic tableau calculi for various modal and description logics via first-order resolution has been studied and discussed in several papers including [HS99, HdNS00, HS02, GHS03, Sch06, SH07, Sch08]. These studies show that the simulation becomes possible by mainting a link to original modal logic subformulae. A modal logic formula is translated into first-order logic and transformed into a clausal form that first-order resolution can operate on. The link between the clausal form and the modal logic formula is obtained by using a controlled form of a standard technique called *structural transformation*, which is also referred to as *renaming*. By this technique, new predicates that in our case represent subformulae of the modal logic problem replace their first-order translation. Because of the retained information on the original problem, the clauses produced by the resolution derivation can easily be translated back to original modal or description logic subformulae.

In the latest papers on this subject by Schmidt [Sch06, Sch08], a refined framework of resolution called hyper-resolution performed over range-restricted clauses is used to produce the exact inferences that simulate tableau. Range-restricted clauses are clauses where all clause variables appear in negative clause literals and all positive clauses are ground. In the cited papers, these clauses are obtained by applying a special kind of first-order translation and structural transformation used specifically for dynamic modal logic formulae that is both sound and

complete.

In this thesis, I focus on the simulation of traditional modal logics and the dynamic modal logic $K_{(m)}(\wedge, \vee, \smile)$, but the implementation can easily be extended to include an even wider range of logics.

Dynamic modal logics allow the necessity and possibility modal logic operators to be defined over relational formulae. This kind of expressivity makes them closely connected to description logics and multi-agent systems. Modal and description logics can actually be viewed as syntactic variants of each other. Description logics has lately become the focal point of research because of the reasoning services it provides for the semantic web.

My implementation of the tableau-resolution prover relies to a big extent on [Sch06, Sch08]. However, instead of the suggested new version of structural transformation, I introduce a new technique that utilises standard first-order translation and structural transformation to acquire the range-restricted clauses. Furthermore, the clauses produced by the technique I introduce are fewer and maintain soundness and completeness. In addition, I explain how generic relational frame conditions can naturally be imposed in first-order resolution and justified in tableau by redefining the modal logic tableau calculus in a way that reflects what happens in the simulation.

Extending a first-order resolution prover to simulate semantic tableau results in providing users with the option of obtaining tableau justification for no extra cost. Semantic tableau is viewed to be more comprehensible and user-friendly than resolution. This simulation also means that the simulated tableau derivations, proofs and models immediately inherit the advanced optimizations provided by resolution procedures.

The rest of this thesis is constructed as follows: Chapter 2 describes traditional and dynamic modal logics, and gives the tableau calculi for these logics. Chapter 3 explains first-order resolution, the hyper-resolution refinement, and gives a brief introduction to the first-order resolution theorem prover SPASS. Chapter 4 describes the two kinds of structural transformation used in the literature and the important role structural transformation plays in the simulation of ground

semantic tableau via first-order resolution. In Chapter 5, I describe my extension of SPASS that simulates ground semantic tableau and prints out tableau translated proofs and models. The chapter addresses the problems that I faced during the implementation and what solutions I used to overcome these problems. It also introduces a new approach I devised for structural transformation so that it utilises the standard techniques already implemented in SPASS. In Chapter 6, I explain how relational frame conditions can be introduced in first-order resolution for traditional and dynamic modal logics and translated directly to tableau rules. In the chapter, I redefine the tableau calculi for traditional and dynamic modal logic in a way that justifies the approach I use. In Chapter 7, I provide and explain a number of derivation results produced by the extended version of SPASS, which serve as an evaluation metric to the implemented work and introduced solutions.

Chapter 2

Modal logic tableaux

This chapter gives an overview of the tableau calculus that the implementation of this project simulates. This includes giving background on the extensions the modal logics that are considered for the tableau simulation. Modal logics, which are viewed as syntactic variants of description logics are playing an increasing role in computer science and especially when it comes to the hot topic of ontology languages and the semantic web.

The rest of this chapter is organized as follows: Section 2.1 explains the syntax and semantics of traditional modal logics. Section 2.2 explains the syntax and semantics of the modal logic $K_{(m)}(\wedge, \vee, \smile)$. It also gives the definition of the standard translation of modal logic formulae into first-order logic, which is essential for the simulation via first-order resolution. Section 2.3 presents the semantic tableau calculus for traditional modal logic and the dynamic modal logic $K_{(m)}(\wedge, \vee, \smile)$ on the underlying Kripke semantics. Finally, Section 2.4 explains the benefits of eliminating defined operators and explains the process of normalization.

2.1 Traditional modal logic

Modal logics are logics that were introduced to model notions of knowledge and belief and allow us to reason about these notions. Modal logics have been applied in several fields other than computer science including philosophy, linguistics and mathematics [Sch09b]. The basic modal logic extends propositional logic with two operators: the box ($\Box$) operator and diamond ($\Diamond$) operator also called the

necessity and *possibility* operators, respectively.

Logic operators can be categorised into two categories: basic operators and defined operators. The choice of basic operators may differ according to preference, but the point is to choose a minimal subset of operators for expressing formulae. In this thesis, we choose as basic operators for modal logics: $\bot, \neg, \wedge$ and $\Box$. Any modal logic formula can be expressed using these four operators. Figure 2.1 defines the defined operators in terms of basic operators.

formula using defined operator	equivalent form using basic operators
$\top$	$\neg\bot$
$\varphi \leftrightarrow \phi$	$(\varphi \wedge \neg\phi) \wedge (\phi \wedge \neg\varphi)$
$\varphi \rightarrow \phi$	$\varphi \wedge \neg\phi$
$\varphi \vee \phi$	$\neg(\neg\varphi \wedge \neg\phi)$
$\Diamond\varphi$	$\neg\Box\neg\varphi$

Figure 2.1: Defined operators

The benefits of reducing a formula to the minimum number of basic operators for tableaux is discussed later on in Section 2.4.

[Sch09b] is the main reference used for defining the syntax and semantics of traditional modal logics in this section.

Syntax

Let p_i be a countably finite set of propositional variables. The following definition is the basic modal logic definition adapted from [Sch09b]. The difference is that the definition here is limited to the chosen set of basic operators.

1. every propositional symbol in the set p_i is a modal formula.
2. $\bot$ (false) is a modal formula.
3. if ϕ and φ are modal formulae then: $\neg\phi$, $(\phi \wedge \varphi)$, and $\Box\phi$ are all modal formulae.

An atomic modal formula is either a propositional variable or $\bot$. These formulae are called *atomic* because they cannot be disassembled into subformulae. All other modal formulae are non-atomic and can be disassembled into smaller subformulae.

We assume the following precedence of the operators from highest to lowest: $\neg \quad \Box \quad \wedge$. $\wedge$ is assumed to be an associative operator.

Semantics

Modal logics are widely defined on *Kripke semantics* [Kri63]. Kripke defines modal logics over frames. A Kripke frame $\mathcal{F}$ is a tuple (W, R_r) consisting of a non-empty set of worlds W and a relation R_r.

A Kripke model is a tuple $\mathcal{M} = (\mathcal{F}, v)$, where $\mathcal{F}$ is a Kripke frame and v is a mapping from propositional variables to subsets of W. The mapping function v denotes which propositional symbols are true in which world. $\mathcal{M}, x \models \phi$ denotes that a formula ϕ is true at world x in model $\mathcal{M}$. A modal formula ϕ is said to be satisfiable if and only if there is a world x in a model $\mathcal{M}$ such that $\mathcal{M}, x \models \phi$. A formula is said to be valid if and only if it is satisfiable in every world of all models.

The truth of modal formulae is defined in Figure 2.2.

Truth of basic modal logic formulae
$\mathcal{M}, x \nvDash \bot$
$\mathcal{M}, x \vDash p$ iff $x \in v(p)$
$\mathcal{M}, x \vDash \neg\phi$ iff $\mathcal{M}, x \nvDash \phi$
$\mathcal{M}, x \vDash \phi \wedge \psi$ iff both $\mathcal{M}, x \vDash \phi$ and $\mathcal{M}, x \vDash \psi$
$\mathcal{M}, x \vDash \Box\phi$ iff $(x, y) \in R_r$ implies $\mathcal{M}, y \vDash \phi$, for any $y \in W$

Figure 2.2: Basic modal logic semantics

$\mathcal{M}, x \nvDash \bot$ means that in every world in $\mathcal{M}$, $\bot$ is false. $\mathcal{M}, x \vDash \Box\phi$ means that for every world y accessible from the world x via the accessibility relation R_r, ϕ must be true.

The basic modal logic represented by K can be extended with certain relational properties. The relational properties impose frame conditions. The result is that certain formulae become valid on the frames satisfying associated relational properties. Formulae that are valid under certain relational conditions are called *axioms*. Figure 2.3 features a set of well-known axioms, the corresponding relational properties and the first-order definitions of properties. For instance, we say that a frame $\mathcal{F}$ validates axioms T, D and 4 iff R_r is reflexive, serial and transitive.

Name	Axiom	Property	First-order definition
T	$\Box p \rightarrow p$	reflexivity	$\forall x R_r(x,x)$
D	$\Box p \rightarrow \Diamond p$	seriality	$\forall x \exists y R_r(x,y)$
B	$p \rightarrow \Box\Diamond p$	symmetry	$\forall x R_r(x,y) \rightarrow R_r(y,x)$
4	$\Box p \rightarrow \Box\Box p$	transitivity	$\forall x,y,z((R_r(x,y) \wedge R_r(y,z)) \rightarrow R_r(x,z))$
5	$\Diamond p \rightarrow \Box\Diamond p$	euclideanness	$\forall x,y,z((R_r(x,y) \wedge R_r(x,z)) \rightarrow R_r(y,z))$

Figure 2.3: Modal logic axioms, corresponding properties and first-order definitions

If Σ is a sequence of symbols chosen from D, T, B, 4, 5, then $K\Sigma$ is a modal logic with frames satisfying the properties associated with the axioms indicated by the symbols in Σ [Sch09b].

2.2 Dynamic modal logics

Syntax

Multi-modal logic $K_{(m)}$ is the basic modal logic but with multi $\Box$ operators. A multi modal logic box operator is denoted in this thesis by $[r_i]$, where r_i is a relational variable. Dynamic modal logics can be viewed as being extensions of the multi-modal logic $K_{(m)}$ in which the modal operators can be parametrised by relational formulae. As an example, $[r_1 \wedge r_2]$ is a dynamic modal operator and not a multi-modal operator. Having relational formulae in dynamic modal logic makes it closely related to description logics [Sch08]. The different relations and relational formulae can be used to formalise dynamic notions such as actions or programs and are useful in linguistic and AI applications [SH06].

$K_{(m)}(\wedge, \vee, \smile)$ is the multi-modal logic defined over frames in which the relations are closed under intersection, union and converse. Let p_j be a countably finite set of propositional variables, and r_i a countably finite set of relational variables. $K_{(m)}(\wedge, \vee, \smile)$ is defined as follows:

1. every propositional symbol in the set p_j is a dynamic modal formula.
2. $\perp$ (false) is a dynamic modal formula.
3. if ϕ and φ are dynamic modal formulae and α is a relational formula, then: $\neg\phi$, $(\phi \wedge \varphi)$, and $[\alpha]\phi$ are dynamic modal formulae.
4. if α and β are relational formulae, then so are: $\alpha \wedge \beta$, $\alpha \vee \beta$ and $\alpha^{\smile}$.

Semantics

The difference between the semantics of dynamic modal logic and basic modal logic is that for dynamic modal logic the Kripke frame $\mathcal{F}$ is a tuple (W, R) consisting of a non-empty set of worlds W and a mapping function R from relational formulae to relations.

The truth of dynamic modal formulae is defined in Figure 2.4.

Truth of dynamic modal formulae
$\mathcal{M}, x \not\models \perp$
$\mathcal{M}, x \models p$ iff $x \in v(p)$
$\mathcal{M}, x \models \neg\phi$ iff $\mathcal{M}, x \not\models \phi$
$\mathcal{M}, x \models \phi \wedge \psi$ iff both $\mathcal{M}, x \models \phi$ and $\mathcal{M}, x \models \psi$
$\mathcal{M}, x \models [\alpha]\phi$ iff $(x, y) \in R_\alpha$ implies $\mathcal{M}, y \models \phi$, for any $y \in W$
Relational conditions
$R_{\alpha\wedge\beta} = R_\alpha \cap R_\beta$
$R_{\alpha\vee\beta} = R_\alpha \cup R_\beta$
$R_{\alpha^{\smile}} = R_\alpha^{\smile}$

Figure 2.4: Dynamic modal logic semantics

$\mathcal{M}, x \models [\alpha]\phi$ means that for every world y accessible from the world x via the accessibility relation formula defined in $\Box$, ϕ must be true. For instance, $\mathcal{M}, x \models [r_1 \vee r_2]\ p$ means that every world y accessible from x via either one of

the two relations Rr_1 and Rr_2, p must be true. $R_\alpha^\smile$ denotes the inverse of the relation.

Kripke frames can be depicted by labelled directed graphs. The nodes of a graph represent worlds, and connecting arrows represent accessibility relations. Figure 2.5 depicts the frame:

$$.\mathcal{F} = (\{1,2,3,4\}, \{R_{r_1}(1,2), R_{r_2}(1,4), R_{r_2}(3,3), R_{r_1}(4,2), R_{r_1}(4,1)\})$$

There are four nodes in the graph representing the four worlds 1, 2, 3 and 4. Labelling the nodes with propositional variables denotes that the variable is true in the respective world.

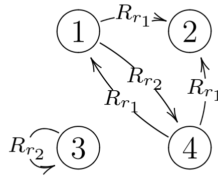


Figure 2.5: Directed graph example of a Kripke frame

Dynamic modal logic translation to first-order logic

The translation of modal logic to first-order logic is discussed here because simulation of tableau calculus is done by using first-order resolution. The standard translation of modal formulae into-first order logic is straight forward. For a given modal formula ψ, the translation function Π translates any modal formula into first-order logic by the following definition:

$$\Pi(\psi) = \exists x. \pi(\psi, x)$$

The definition of π and the relational formulae translation function τ for the logic $K_{(m)}(\wedge, \vee, \smile)$ is given in Figure 2.6.

Let us workout a small example that will be used for illustration later on. Let $\varphi = [r_1][r_2](p \wedge \neg q)$. Figure 2.7 shows the translation steps that result from applying the translation rules given in Figure 2.6.

Propositional translation	Relational translation
$\pi(\bot, x) = \bot$	$\tau(r_i, x, y) = R_i(x, y)$
$\pi(p_j, x) = P_j(x)$	$\tau(\alpha \wedge \beta, x, y) = \tau(\alpha, x, y) \wedge \tau(\beta, x, y)$
$\pi(\neg\psi, x) = \neg\pi(\psi, x)$	$\tau(\alpha \vee \beta, x, y) = \tau(\alpha, x, y) \vee \tau(\beta, x, y)$
$\pi(\psi \wedge \varphi, x) = \pi(\psi, x) \wedge \pi(\varphi, x)$	$\tau(\alpha^{\smile}, x, y) = \tau(\alpha, y, x)$
$\pi([\alpha]\psi, x) = \forall y(\tau(\alpha, x, y) \rightarrow \pi(\varphi, y))$	

Figure 2.6: Translation to first-order logic for $K_{(m)}(\wedge, \vee, \smile)$

$$
\begin{aligned}
\Pi(\varphi) &= \exists x.\, \pi([r_1][r_2](p \wedge \neg q), x) \\
&= \exists x.\, \forall y.\, (\tau(r_1, x, y) \rightarrow \pi([r_2](p \wedge \neg q), y)) \\
&= \exists x.\, \forall y.\, (R_{r_1}(x, y) \rightarrow (\forall z.\, \tau(r_2, y, z) \rightarrow (\pi(p \wedge \neg q, z)))) \\
&= \exists x.\, \forall y.\, (R_{r_1}(x, y) \rightarrow (\forall z.\, R_{r_2}(y, z) \rightarrow (\pi(p, z) \wedge \pi(\neg q, z)))) \\
&= \exists x.\, \forall y.\, (R_{r_1}(x, y) \rightarrow (\forall z.\, R_{r_2}(y, z) \rightarrow (P_p(z) \wedge \neg\pi(q, z)))) \\
&= \exists x.\, \forall y.\, (R_{r_1}(x, y) \rightarrow (\forall z.\, R_{r_2}(y, z) \rightarrow (P_p(z) \wedge \neg P_q(z))))
\end{aligned}
$$

Figure 2.7: First-order translation of $[r_1][r_2](p \wedge \neg q)$

2.3 Tableau for modal logic

A semantic tableau is a formal proof procedure based on refutation; given an initial formula φ, it tries to show that $\neg\varphi$ is unsatisfiable. A tableau derivation is usually presented with a branching tree. The nodes of the tree represent labelled formulae. For the formula φ, the tableau derivation has the set $\{a : \varphi\}$ as its root node. a is a constant representing a world where φ is true.

In general, inference rules, which are called expansion rules are of the form:

$$\frac{X}{X_1 \mid ... \mid X_n}$$

where X and X_i denote the sets of *premises* and *conclusions* respectively. Each rule is applied only once to the same set of instances of premises of a rule.

A tableau branch is expanded at its leaf node creating up to n successor nodes by applying the relevant inference rule. A branch in a tableau tree stops expanding if no more rules are applicable or if a contradiction is found. A contradiction results from having a formula and its negation in the same world in a branch. If such a case exists, then $\bot$ is derived. A branch containing $\bot$ is said to be a

closed branch. If no contradiction is found, and no more rules are applicable, then the branch is referred to as an *open branch*. A closed branch is unsatisfiable. A tableau with no open branches indicates that the formula is unsatisfiable. If a formula is proved to be unsatisfiable, then its negation is valid. On the other hand, a tableau containing an open branch with no more rules applicable is satisfiable.

Different logics are defined with different tableau rules but they all share common grounds. For instance, all tableaux must have a closure rule that derives contradictions. The closure rule for modal logics is defined as:

$$(\bot)\ \frac{s:\varphi\ ,\ s:\neg\varphi}{\bot} \qquad \text{where } s \text{ is a constant representing a world}$$

Other than the contradiction rule, basic modal logic tableau is made-up of 4 expansion rules. These rules are defined in figure 2.8.

$$(\wedge)\ \frac{s:\varphi\wedge\phi}{s:\varphi, s:\phi} \qquad (\neg\wedge)\ \frac{s:\neg\,(\varphi\wedge\phi)}{s:\sim\!\varphi \mid s:\sim\!\phi}$$

$$(\Box)\ \frac{s:\Box\varphi, (s,t)\!:\!R_r}{t:\varphi} \qquad (\neg\Box)\ \frac{s:\neg\Box\varphi}{(s,t):R_r,\ t:\sim\!\varphi} \text{ where } t \text{ is new to the branch}$$

Figure 2.8: Basic modal logic tableaux Calculus

In [SH06, HS02], the simulation of $K\Sigma$ tableau where Σ is a sequence of symbols chosen from the axioms T, D, B, 4, and 5 via first-order resolution has been described based on the single-step tableaux calculi of Massacci [Mas98, Mas00]. Single-step tableau rules are also referred to as propagational rules because the rules propagate formulae to worlds without asserting relational edges between worlds. The reader can refer to the references for more on these rules.

In [CFdCGHg97], two kind of rules are identified. The first kind is propagational rules that are introduced by Massacci and also structural rules. Structural rules do not add formulae but rather asserts relational edges between worlds.

Structural rules for axioms T, D, B, 4, and 5 can be defined as given in Figure 2.9.

$$(T)\ \frac{\cdot}{(s,s):r} \qquad (D)\ \frac{\cdot}{(s,t):r}$$

$$(B)\ \frac{(s,t):r}{(t,s):r} \quad (4)\ \frac{(s,t):r,(t,u):r}{(s,u):r} \quad (5)\ \frac{(s,t):r,(s,u):r}{(t,u):r}$$

where t in rule (D) represents a new constant on the branch.

Figure 2.9: Structural tableau rules for axioms T, D, B, 4 and 5

$$(\bot)\ \frac{s:\varphi, s:\neg\varphi}{\bot}$$

$$(\wedge)_1\ \frac{s:\varphi\wedge\phi}{s:\varphi} \qquad (\wedge)_2\ \frac{s:\varphi\wedge\phi}{s:\phi} \qquad (\neg\wedge)\ \frac{s:\neg(\varphi\wedge\phi)}{s:\sim\varphi \mid s:\sim\phi}$$

$$(\neg[\alpha])_1\ \frac{s:\neg[\alpha]\varphi}{(s,t):\alpha} \qquad (\neg[\alpha])_2\ \frac{s:\neg[\alpha]\varphi}{t:\sim\varphi} \qquad ([\alpha])\ \frac{(s,t):\alpha, s:[\alpha]\varphi}{t:\varphi}$$

$$(\smile)\ \frac{(s,t):\alpha^{\smile}}{(t,s):\alpha} \qquad (\smile)_I\ \frac{(t,s):\alpha}{(s,t):\alpha^{\smile}}$$

$$(\wedge)^r_1\ \frac{(s,t):\alpha\wedge\beta}{(s,t):\alpha} \qquad (\wedge)^r_2\ \frac{(s,t):\alpha\wedge\beta}{(s,t):\beta} \qquad (\wedge)^r_I\ \frac{(s,t):\alpha,(s,t):\beta}{(s,t):\alpha\wedge\beta}$$

$$(\vee)^r\ \frac{(s,t):\alpha\vee\beta}{(s,t):\alpha \mid (s,t):\beta} \qquad (\vee)^r_{I,1}\ \frac{(s,t):\alpha}{(s,t):\alpha\vee\beta} \qquad (\vee)^r_{I,2}\ \frac{(s,t):\beta}{(s,t):\alpha\vee\beta}$$

$$(contr)\ \frac{s:\neg(\psi\wedge\psi)}{s:\sim\psi} \qquad (contr)^r\ \frac{(s,t):\alpha\vee\alpha}{(s,t):\alpha}$$

(i) t in the rules $(\neg[\alpha])_1$ and $(\neg[\alpha])_2$ represents a new constant on the branch.
(ii) The rules $(\wedge)^r_I$, $(\vee)^r_I$ and $(\smile)_I$ have the side condition that the relational formulae in the conclusions occur as subformulae of a box in the input problem.

Figure 2.10: Tableaux Calculus for $K_{(m)}(\wedge,\vee,\smile)$

The tableaux calculus for $K_{(m)}(\wedge,\vee,\smile)$ as defined in [Sch06, Sch08] is given in Figure 2.10.

s and t in the rules of Figure 2.10 are constants representing worlds. The closure rule $(\bot)$ is applied when a world s contains a formula φ and its negation $\neg\varphi$. The conclusion of the rule is $\bot$, which denotes the unsatisfiability of the

branch.

For rules $(\wedge)_1$ and $(\wedge)_2$, when a world s is labelled with a formula of the form $\varphi \wedge \phi$, this formula can be broken-down so that the world s is labelled with both φ and ϕ.

$(\neg\wedge)$ is a branching rule. The separator | denotes that two branches are produced from applying the rule. Branching means that there are several possibilities to consider. Hence, branching introduces non-determinism. For the application of this rule, one branch would contain $s : {\sim}\varphi$ and the other would contain $s : {\sim}\phi$. $\sim$ denotes the complement, which means that if $\varphi = \neg\psi$, then ${\sim}\varphi = \psi$, otherwise, ${\sim}\varphi = \neg\varphi$.

The rule $(\neg[\alpha])_1$ is applied when we have a world s labelled with $\neg[\alpha]\varphi$. The conclusion asserts an edge between s and a new world on the branch t. Rule $(\neg[\alpha])_2$ labels the new world t with the complement of the formula φ.

Rule $[\alpha]$ is applicable when we have two connected worlds (s, t) and which are labelled with the relational formula α and also have that s is labelled with $[\alpha]\varphi$. In this case, the conclusion labels the successor t of s with φ.

$\smile$ denotes the converse operator. When the converse operator is applied on a relational formula $\alpha^\smile$ that labels (s, t), then the conclusion labels (t, s) with α.

$(\wedge)^r$ is similar to $(\wedge)$ but is applied on relational formulae.

$(\vee)^r$ is another branching rule but on relational formulae where the formula contains relational union. When the rule is applied, two branches are explored with the different possibilities.

Relational rules have opposite introductory rules which are denoted by $(o)_I$, where o here represents $\smile, \wedge^r or \vee^r$.

The contraction rules $(contr)$ and $(contr)^r$ are simplification rules that are applied on propositional and relational formulae respectively whenever redundancy is found, where branching maybe applied.

Usually in tableaux, there is a single rule for each of $(\wedge)$ and $(\neg\Box)$. However, for back-translation purposes, it is convenient to express each of them in two rules instead of one. The reasons for this become apparent once the simulation via first-order resolution is discussed in Chapter 4.

The rules $(\neg\wedge)$ and $(\vee)^r$ as defined in Figure 2.10 follow standard tableau branching rules. The tableau search using these rules is referred to in the literature as *syntactic branching search*. Syntactic branching has been proven to be wasteful and highly inefficient. This is because unsatisfiable disjunctions can reoccur when considering alternative branches and there is nothing to prevent this from happening [Hor97, Hor98]. This problem is overcome by forcing the branches to become disjoint. The branches become disjoint when the negation of the unsatisfiable disjunct is added to alternative branches. This enhanced search is referred to as *semantic branching search*.

When semantic branching is utilized, the branching rules are slightly changed as in Figure 2.11.

$$(\neg\wedge)\ \frac{s : \neg(\varphi \wedge \psi)}{s : {\sim}\varphi \ \mid\ s : \varphi, s : {\sim}\psi} \qquad\qquad (\vee^r)\ \frac{(s,t) : (\alpha \vee \beta)}{(s,t) : \alpha \ \mid\ (s,t) : {\sim}\alpha, (s,t) : \beta}$$

Figure 2.11: Tableau branching rules with semantic branching

2.4 Formula normalization

Lots of tableau algorithms described in the literature assume that input formulae are in negation normal form (NNF). Formulae in negation normal form are much simpler to describe but in practice and for efficiency reasons the input in automated provers is not transformed to NNF [HHSS07].

Take for instance a formula φ where $\varphi = (p \wedge q) \wedge \neg(p \wedge q)$. When this formula is translated into negation normal form, it is transformed to $(p \wedge q) \wedge (\neg p \vee \neg q)$. The original formula is clearly unsatisfiable because the subformula $(p \wedge q)$ clashes with $\neg(p \wedge q)$. Pushing the negation inwards to obtain the negation normal form causes this obvious contradiction to become unclear. Figure 2.12 shows the two

tableaux derivations. The derivation to the left is of the original formula. The derivation to the right is of its negation normal form. In the first derivation, a contradiction is concluded right after applying the ($\wedge$) rule. Whereas in the second derivation, the formula needs to break-down further and derive more inference steps. Drawing contradictions and concluding the unsatisfiability of the problem only becomes possible at the lowest level after branching and backtracking. The example is fairly simple but if the formula is huge, then the overhead becomes quite significant and wasteful. This shows why negation normal form regardless of its simplicity is not the preferred form for satisfiability testing [HHSS07].

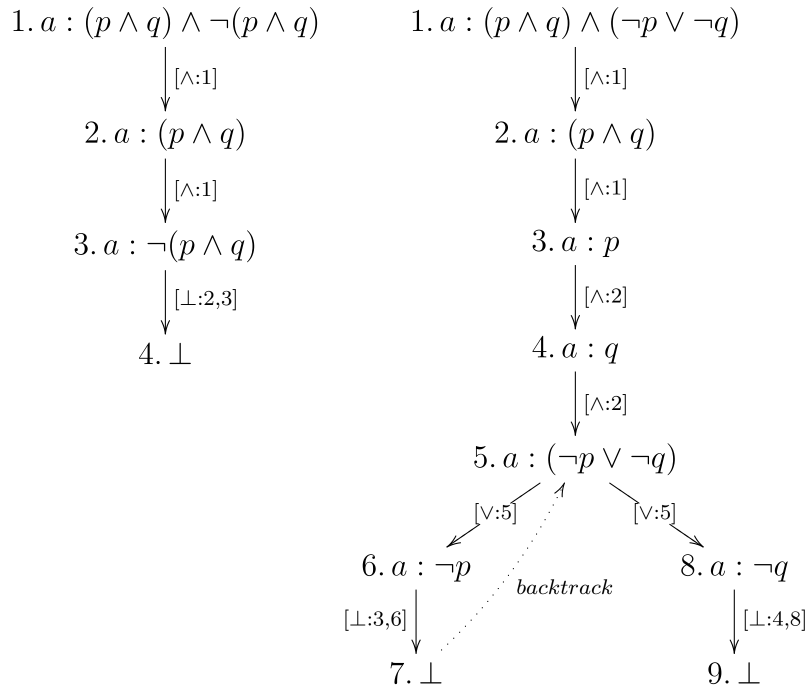


Figure 2.12: Comparison between two derivations of equivalent formulae

Using a minimum number of operators in formulae helps detect cases of early contradictions and saves the overhead of unnecessary inference steps. Take for instance the modal formula $\varphi = \Diamond p \wedge \Box\neg p$. The normal form of φ using the rewrite rules in Table 2.13 is:

$$\neg\Box\neg p \wedge \Box\neg p$$

In this form, it becomes apparent that the formula is not satisfiable because of

the obvious contradiction between $\neg\Box\neg p$ and $\Box\neg p$.

The motive for using basic operators only for expressing dynamic modal syntax becomes clear; it helps in detecting contradictions faster. *Normalization* is used in the context of this thesis as a preprocessing step for transforming modal formulae by removing defined operators as removing double negation for simplification. The process of normalization involves recursively applying a set of rewrite rules until no more rules are applicable. Figure 2.13 lists down the rewrite rules for removing defined operators.

formula	rewrite rule
$\top$	$\neg\bot$
$\neg\neg\varphi$	φ
$\varphi \leftrightarrow \phi$	$(\varphi \rightarrow \phi) \wedge (\phi \rightarrow \varphi)$
$\varphi \rightarrow \phi$	$\neg\varphi \vee \phi$
$\varphi \vee \phi$	$\neg(\neg\varphi \wedge \neg\phi)$
$\Diamond\varphi$	$\neg\Box\neg\varphi$

Figure 2.13: Rewrite rules

Chapter 3

Resolution

This chapter gives an overview of the resolution calculus and modern frameworks of resolution that are implemented in today's theorem provers.

This chapter is organized as follows: Section 3.1 describes resolution as a refutation system for first-order logic by exploiting a technique called *unification*. Section 3.2 describes a refined resolution calculus called hyper-resolution, which along with a process called *structural transformation* (discussed in Chapter 4) simulates tableau systems for dynamic modal logic. Section 3.3 presents the first-order resolution theorem prover SPASS [Wei05, WDF$^+$09], which is used in this project for the implementation of the tableau-resolution simulator.

3.1 First-order resolution

Resolution [Rob65] is a deduction system that just like tableaux is also based on refutation. A refutation system proves the entailment of a formula φ by proving that $\neg\varphi$ is unsatisfiable.

The information in this section is based on Chapters 3 and 7 in [Fit90], Chapters 5 and 9 in [Kel97], and [Sch09a, Vor09].

Resolution requires the input it works on to be in a clausal form usually represented by sets of clauses. For first-order logic, the clausal form is obtained by applying several transformations to a first-order formula; transformation to conjunctive normal form, Skolemization [Sko55] and structural transformation [Sch06].

Skolemization ensures the clausal form is transformed to a quantifier free form. The implicit assumption for any variable in this form is that it falls under a universal quantifier.

Let Cls be a function transforming first-order formulae to clausal form. We have that $\models \varphi$, iff $\text{Cls}(\neg\varphi)$ is unsatisfiable [Sch06]. This says that the entailment of any first-order formula is proven if and only if the clausal form of its negation is unsatisfiable.

The basic resolution calculus is based on two expansion/inference rules; the resolution rule and the factoring rule. For propositional logic, the resolution rule states that for two clauses $A \vee B$ and $\neg B \vee C$, B and $\neg B$ can be resolved together and the result $A \vee C$ is obtained. The factoring rule is a form of contraction of redundant clauses. Factoring is applicable to clauses of the form $A \vee B \vee B$ and the result $A \vee B$ is obtained.

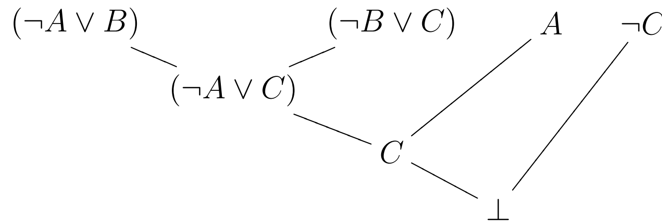


Figure 3.1: Resolution derivation example

Figure 3.1 shows an example of how resolution is used on the following set of clauses: $\{(\neg A \vee B), (\neg B \vee C), A, \neg C\}$. The first line in the graph displays the input clauses. The first two clauses are resolvable and upon resolution derive $(\neg A \vee C)$. Then, the result is resolved with the third input clause A and the result C is obtained. Finally C is resolved with $\neg C$ resulting in the empty clause $\bot$. This means that the input set of clauses is unsatisfiable.

Predicate logic is more complicated than propositional logic because it contains variables. We find that in order to apply resolution rules on predicate logic, we must first apply substitutions. The idea is that in order for literals to become resolvable, substitutions that make occurrences of literals identical must be used.

This kind of substitution is referred to in the literature as *unifying substitutions*, and the process is called *unification* [Kel97]. Applying unification with basic resolution rules makes a sound and refutationally complete inference system for full first-order logic and clause logic [Sch06]. However, it is found that implementing resolution in its basic form makes the prover highly inefficient for most cases [BG01]. The inefficiency comes from deriving inference steps that may have been derived before and are therefore redundant.

Refinements of basic resolution provide better control over the application of rules and the redundancy of inference steps. The ordering parameter $\succ$ and the selection function S are two ways to limit inference steps and enhance the efficiency of theorem provers. With the ordering parameter, inference rules are not applied unless on a maximal literal according to $\succ$. The selection function overrides ordering. This means that negative literals selected by S are preferred over literals that are possibly larger under the ordering $\succ$ [Sch06]. The search space with the restrictions implied by $\succ$ and S becomes much smaller than the search space of unrestricted resolution. A smaller search space enhances the performance of the prover. The proof of soundness and completeness of ordered resolution with selection can be found in [Sch09a].

In resolution expansion rules have the form:

$$\frac{N}{N_1 \mid ... \mid N_n}$$

where N and N_i denote sets of clauses.

Figure 3.2 displays the definition of a resolution calculus with redundancy elimination and splitting as given in [HS02, Sch06, Sch08]. We use the notation $R^{\text{red}}_{\text{sp}}$ to refer to this calculus.

The deduce rule represents the basic rules, which are ordered resolution and factoring.

The delete and simplify rules are important because they eliminate redundancies offering better management of the search space. However, redundancy

Deduce:	$\dfrac{N}{N \cup \{C\}}$	if C is a factor or resolvent of premises in N
Delete:	$\dfrac{N \uplus \{C\}}{N}$	if C is redundant with respect to N
Simplify:	$\dfrac{N}{(N \setminus M) \cup M'}$	if $(N \setminus M) \cup M'$ is satisfiable when N is satisfiable and every clause in M is redundant with respect to $(N \setminus M) \cup M'$
Split:	$\dfrac{N \uplus \{C \vee D\}}{N \cup \{C\} \mid N \cup \{D\}}$	if C and D are variable-disjoint
alternatively	$\dfrac{N \uplus \{C \vee D\}}{N \cup \{C\} \mid N \cup \{\neg C, D\}}$	an alternative splitting rule referred to as *complement splitting*

Resolvents and factors are computed with:

Ordered resolution: $\dfrac{C \vee A \quad \neg B \vee D}{(C \vee D)\sigma}$

(i) where σ is the most general unifier of A and B such that $A\sigma = B\sigma$
(ii) no literal is selected in C, and $A\sigma$ is strictly $\succ$-maximal with respect to $C\sigma$
(iii) $\neg B$ is either selected, or $\neg B\sigma$ is maximal with respect to $D\sigma$ and no literal is selected in D.

Ordered factoring: $\dfrac{C \vee A \vee B}{(C \vee A)\sigma}$

(i) σ is the most general unifier of A and B
(ii) no literal is selected in C and $A\sigma$ is $\succ$-maximal with respect to $C\sigma$

Figure 3.2: The resolution calculus $R^{\mathrm{red}}_{\mathrm{sp}}$

testing can be an expensive operation and thus, the implementation of redundancy elimination is usually restricted to computable forms of redundancy elimination [Sch06].

The symbol $\uplus$ in the split and delete rules denotes variable-disjoint union. Instead of refuting $N \cup \{C \vee D\}$, the prover splits the clause and refutes both $N \cup \{C\}$ and $N \cup \{D\}$. The alternative form of splitting known as *complement splitting* refutes both $N \cup \{C\}$ and $N \cup \{\neg C, D\}$. Splitting is actually adapted

from tableau. Splitting introduces non-determinism. As in tableaux, splitting requires backtracking. In tableaux, complement splitting is referred to as *semantic branching*. It is a form of refined splitting that eliminates redundancies caused by disjunction by making the two created branches disjoint [Li08].

3.2 Ordered hyper-resolution

Hyper-resolution is a refinement of the resolution calculus. In hyper-resolution, positive clauses are resolved with non-positive clauses such that the conclusion is always positive or the empty clause $\emptyset$ [GHS03]. The selection function S in hyper-resolution selects exactly the set of all negative literals in any non-positive clause [Sch06].

Hyper-resolution is defined with the following rule [HS02, Sch06, Sch08]:

$$\frac{C_1 \vee A_1 \;\; ... \;\; C_n \vee A_n \quad \neg B_1 \vee ... \vee \neg B_n \vee D}{(C_1 \vee ... \vee C_n \vee D)\sigma}$$

where:

(i) σ is the most general unifier such that $A_i\sigma = B_i\sigma$ for every i, $1 \leq i \leq n$

(ii) $A_i\sigma$ is strictly $\succ$-maximal with respect to $C_i\sigma$ and the C_i are positive clauses, for every i, $1 \leq i \leq n$

(iii) for every $i, 1 \leq i \leq n, \neg B_i$ is selected and D is a positive clause.

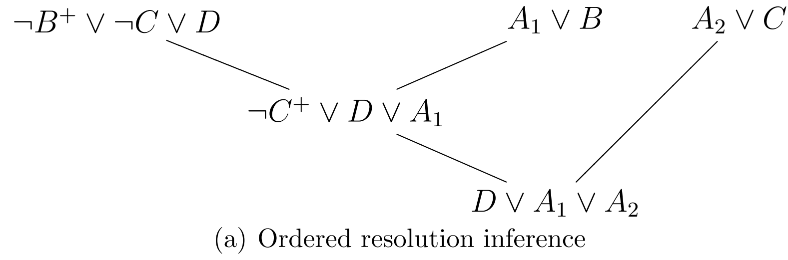
The factoring rule is given by:

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

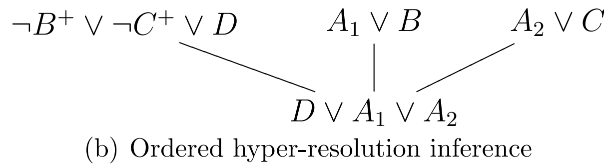
(i) σ is the most general unifier of A and B

(ii) no literal is selected in C and $A\sigma$ is $\succ$-maximal with respect to $C\sigma$

Figure 3.3 sketches a small example showing the difference between ordered resolution and ordered hyper-resolution on the same set of clauses. Figure 3.3(a) displays ordered resolution behaviour on the given set of clauses $\{\neg B \vee \neg C \vee D, A_1 \vee B, A_2 \vee C\}$. Assuming that the literal $\neg B$ in the first clause is selected by the selection function S, it resolves with B from the second clause. The result is



(a) Ordered resolution inference



(b) Ordered hyper-resolution inference

Figure 3.3: Comparison between (a) ordered resolution and (b) ordered hyper-resolution

$\neg C \vee D \vee A_1$. The selection function now selects the only negative literal $\neg C$ to resolve with C from the third clause and the obtained result is $D \vee A_1 \vee A_2$. In comparison with ordered hyper-resolution of Figure 3.3(b), the selection function selects both negative literals of the first clause and resolves on both of them simultaneously deriving the result in a single step. Hyper-resolution results in fewer inference steps with only positive conclusions.

3.3 The resolution theorem prover SPASS

SPASS is a saturation-based automated resolution theorem prover for full first-order logic with equality. SPASS is extended in this project to provide modal logic tableau simulation via first-order resolution. There are other first-order resolution provers such as E [Sch02] and Vampire [RV02] that could have been used just as well. However, not only was SPASS essentially required by the project's supervisor, but it also included the following important points:

- The latest versions of SPASS includes the implementation of MSPASS [Sch99, HS00], which is provided as an integral part. MSPASS translates modal logic formulae, description logic formulae, and formulae of relational calculus into first-order logic for deduction. This functionality of translating modal logic formulae into first-order logic is needed for this project since the goal is to simulate modal logic tableau via first-order resolution.

- SPASS includes a sophisticated renaming module. The renaming module applies structural transformation on first-order formulae. The importance of structural transformation for this project is discussed in Chapter 4.
- SPASS already has an implementation of the hyper-resolution refinement of the resolution calculus.
- When the input problem is satisfiable, SPASS can output a saturated-set of clauses, which represent an open branch. This means that SPASS can be used as a model finder, which is a useful reasoning service.

SPASS is operated from the commandline by running the command

```
SPASS [options] inputfile.dfg
```

`inputfile.dfg` is the input file containing the problem in DFG format.

The input format for SPASS is explained in detail in the input syntax manual [CWT07] and a brief tutorial is present online [Wei06]. For the purpose of this project, only relevant syntax is discussed this thesis. As mentioned earlier, this project is interested in modal logics and therefore the syntax explained here is relevant for the support added by MSPASS.

Figure 3.4 shows an example of an input problem in DFG syntax for illustration.

The syntax of a problem has three different parts; the description part, the logical part, and the settings part. In Figure 3.4, the settings part begins at line 2 and ends at line 7. The parameters included are mandatory to any problem. Status (line 5) indicates one of three possibilities: satisfiable, unsatisfiable, or unknown. More parameters such as version, logic and date are optional to include.

The logical part has three sections:

- `list_of_symbols` (lines:9-12)
  - `predicates[].` is used to define predicate symbols as pairs specifying the name of the predicate and its arity as `(symbol,arity)`

```
1.  begin_problem(Example_1).
2.  list_of_descriptions.
3.    name({*Problem: Example 1*}).
4.    author({*Rawan AlBarakati*}).
5.    status(unknown).
6.    description({* -([r](p \/ <r> p) /\ -p) *}).
7.  end_of_list.
8.
9.  list_of_symbols.
10. predicates[(r,0), (p,0), (R,2)].
11. translpairs[(r,R)].
12. end_of_list.
13.
14. list_of_special_formulae(axioms,EML).
15. formula(forall([x], R(x,x)),reflexivity). % R is reflexive
16. % rel_formula(implies(id,r)).   % r is reflexive
17. end_of_list.
18.
19. list_of_special_formulae(conjectures,EML).
20. prop_formula(implies(box(r,p),p)).
21. end_of_list.
22.
23. list_of_settings(SPASS)
24. {*
25.    set_flag(PGiven,0).
26.    set_flag(DocProof,1).
27.    set_flag(Auto,0).
28.    set_flag(IOHy,1).
29. *}
30. end_of_list.
31. end_problem.
```

Figure 3.4: Input problem example in DFG syntax

  – `translpairs[].` is used to create a link between different symbols. When the propositional predicate is translated into first-order logic, the paired symbol is used instead of introducing a new one. For the example in the figure, r is paired up with R. r denotes a relational variable in modal logic, whereas R denotes the corresponding accessibility relation.

- `list_of_special_formulae(axioms,EML)` (lines:14-17)
  This section carries declarations of axiom formulae according to their type.

- `list_of_special_formulae(conjectures,EML)` (lines:19-21)
  Again, formulae are declared in this section according to their type.

The input problem is formed from the conjunction of axioms and negated conjectures.

A formula in first order logic is declared with `formula()`. In the example, line 15 contains a first order formula declaration falling into the axioms part. This formula imposes the reflexivity property on the accessibility relation R. Remember that R is defined as the translation pair of r found in the modal formula. Formulae can have associated labels. The first-order formula imposing reflexivity in the example is labelled with its property.

```
--------------------------SPASS-START-----------------------------
1. Input Problem:
2. 1[0:Inp] ||  -> R(U,U)*.
3. 2[0:Inp] || P(skc1)* -> .
4. 3[0:Inp] || R(skc1,U)* -> P(U).
5.   This is a first-order Horn problem without equality.
6.   This is a problem that has, if any, a finite domain model.
7.   There are no function symbols.
8.   Axiom clauses: 1 Conjecture clauses: 2
9.   Inferences: IOHy=1
10.  Reductions:
11.  Extras    : No Input Saturation, Dynamic Selection, Full Splitting, Full
Reduction,  Ratio: 5, FuncWeight: 1, VarWeight: 1
12.  Precedence: nequal > div > id > r > s > t > p > q > w > v > a > b > c > d >
 R > S > P > Q > skc0 > skc1
13.  Ordering  : KBO
14. Processed Problem:
15.
16. Worked Off Clauses:
17.
18. Usable Clauses:
19. 2[0:Inp] || P(skc1)* -> .
20. 1[0:Inp] ||  -> R(U,U)*.
21. 3[0:Inp] || R(skc1,U)* -> P(U).
22. SPASS V 3.5
23. SPASS beiseite: Proof found.
24. Problem: example1.dfg
25. SPASS derived 2 clauses, backtracked 0 clauses, performed 0 splits and kept
4 clauses.
26. SPASS allocated 39989 KBytes.
27. SPASS spent 0:00:00.12 on the problem.
28.             0:00:00.05 for the input.
29.             0:00:00.03 for the FLOTTER CNF translation, of which
30.             0:00:00.00 for the translation from EML to FOL.
31.             0:00:00.00 for inferences.
32.             0:00:00.00 for the backtracking.
33.             0:00:00.00 for the reduction.
34.
35. Here is a proof with depth 2, length 5 :
36. 1[0:Inp] ||  -> R(U,U)*.
37. 2[0:Inp] || P(skc1)* -> .
38. 3[0:Inp] || R(skc1,U)* -> P(U).
39. 4[0:OHy:3.0,1.0] ||  -> P(skc1)*.
40. 5[0:OHy:2.0,4.0] ||  -> .
41. Formulae used in the proof : reflexivity conjecture0
--------------------------SPASS-STOP------------------------------
```

Figure 3.5: SPASS ouput

A boolean type formula is declared with either `prop_formula` used for propositional modal logics or `concept_formula` used for declaring concepts of description logics. Line 20 is an example of a propositional formula.

A relational formula is declared with `rel_formula` for relations in modal logics or `role_formula` for relations in description logics. Line 21 is an example of a relational formula which makes `r` reflexive. Note that the line starts with a (`%`), which causes the line to be treated as a comment.

For the implementation of the tableau-resolution prover of this project, only first-order formulae are expected in the axioms part to impose relational conditions on modal logic frames, and in the conjectures part only propositional formulae are expected with limited relational operators.

The settings part carries flag settings for the concerned system. Each flag can be declared only once by the line `set_flag(flag,value)`. This part is completely optional, and the flags can alternatively be declared in the commandline as `-flag=value`.

For any problem, the settings part is optional but the other two parts are mandatory.

Figure 3.5 is the output from running the example of Figure 3.4. Lines 1-4 is the input problem as a list of clauses after translation to first-order logic of the modal formulae and transformation to clausal form. The star * denotes that the literal is maximal according to the specified or default ordering. Lines 5-8 are analysis information of the problem by SPASS. Lines 9-13 are based on flag settings. In our problem, we specified the inference type as ordered hyper resolution by setting on the option `IOHy=1`.

Next comes the answer to the problem. The most important part in the output is in line 23 `SPASS beiseite: Proof found.` This line specifies the problem result. Since the formula in the conjectures part is negated for refutation, proof found denotes that the problem is actually valid. The proof is output in lines 35-40. Line 41 specifies the formula lables contributing to the proof. If the result says `Completion found`, then it means that SPASS has found an open branch

making up a model. When switching on the DocProof flag setting, if completion is found, SPASS outputs a model.

Chapter 4

Simulation of tableaux via first-order resolution

This chapter is an overview on how tableau simulation is achieved using hyper-resolution on range-restricted clauses obtained from structural transformation. The rest of this chapter is structured as follows: Section 4.1 explains structural transformation for first-order logic and how the introduced predicates can be linked to modal formulae. Papers following this approach assume the input is in negation normal form. We have seen that negation normal form is not suitable due to efficiency reasons. However, the approach in this section is explained because it will be referred to in consequent chapters. Section 4.2 explains a special kind of structural transformation specifically introduced for dynamic modal formulae. The way definitions are introduced in this version result in a more efficient and effective derivation. Section 4.3 illustrates with a practical example how the simulation is done using the structural transformation described in Section 4.2.

4.1 Structural transformation for first-order logic

Structural transformation (also known as renaming) [1] involves the replacement of subformulae with fresh predicates. For a formula $\psi \vee \phi$, one can introduce a new symbol $Q_\vee$ that replaces $\psi \vee \phi$ such that $Q_\vee$ is satisfiable if and only if $Q_\vee \wedge (Q_\vee \leftrightarrow (\psi \vee \phi))$ is satisfiable.

[1] Structural transformation and renaming will be used interchangeably throughout this dissertation

An optimised version of structural transformation further takes the polarity of the replaced subformula into account. The polarity of a subformula in first-order logic can be *positive*, *negative*, or have *zero polarity*. Following [Sch09b], if a formula ψ contains neither $\leftrightarrow$ nor $\rightarrow$, a subformula ϕ:

- has a *positive polarity* in ψ if it occurs under an even number of negation symbols.
- has a *negative polarity* in ψ if it occurs under an odd number of negation symbols.
- has *zero polarity* in ψ if it occurs under both positive and negative scopes.

First order structural transformation is described in [HdNS00, HS02] as follows: Let λ denote the position of a first-order subformula in φ. $\mathrm{Pos}(\varphi)$ is the set of all the positions of subformulae of φ. $\varphi|\lambda$ denotes the subformula of φ at position λ and $\varphi[\psi \mapsto \lambda]$ is the result of replacing the subformula in φ at position λ with ψ.

Let $\Lambda \subseteq \mathrm{Pos}(\varphi)$. Each element λ of Λ is associated with a fresh predicate symbol Q_λ and a literal $Q_\lambda(x_1, ..., x_n)$, where $x_1, ..., x_n$ are free variables of $\varphi|\lambda$. Let

$$\mathrm{Def}^+_\lambda(\varphi) = \forall x_1...x_n(Q_\lambda(x_1, ..., x_n) \rightarrow \varphi|\lambda)$$
$$\mathrm{Def}^-_\lambda(\varphi) = \forall x_1...x_n(\varphi|\lambda \rightarrow Q_\lambda(x_1, ..., x_n))$$

The definition of Q_λ is the formula

$$\mathrm{Def}_\lambda(\varphi) = \begin{cases} \mathrm{Def}^+_\lambda(\varphi) & \text{if } \varphi|\lambda \text{ has positive polarity} \\ \mathrm{Def}^-_\lambda(\varphi) & \text{if } \varphi|\lambda \text{ has negative polarity} \\ \mathrm{Def}^+_\lambda(\varphi) \wedge \mathrm{Def}^-_\lambda(\varphi) & \text{if } \varphi|\lambda \text{ has zero polarity} \end{cases}$$

The corresponding clauses are called *definitional clauses*. Although the result of the optimized structural transformation is not logically equivalent to the original formula, it preserves satisfiability, which is sufficient for resolution theorem proving [NW01].

In [HS02], the assumption is that formulae are in negation normal form. Negations in a formula in negation normal form only occur in front of predicates. The consequence is that the polarity of all introduced symbols is positive. Thus, we

can say that $\text{Def}_\lambda(\varphi) = \text{Def}^+_\lambda(\varphi)$. The main advantage of introducing new predicates is to link the predicates to original modal formulae so that the connection is not lost. One can manage this connection by introducing fresh predicates only for subformulae corresponding to non-literal subformulae of the original modal formula [HdNS00]. A mapping function is used for this purpose. For a given modal formula φ and its translation into first-order logic $\varphi' = \Pi(\varphi)$, the mapping Def_Λ is applied with following definition from [HdNS00]:

$$\Lambda = \{\lambda \mid \text{there is a non-literal subformula } \varphi|\lambda' \text{ of } \varphi \text{ and } \varphi'|\lambda = \Pi(\varphi|\lambda')\}.$$

For example, let us take the first-order translation of the modal formula $[r_1][r_2](p \wedge \neg q)$ that was previously obtained in Figure 2.7:

$$\exists x.\, \forall y.\, (R_{r_1}(x, y) \rightarrow (\forall\ z\ R_{r_2}(y, z) \rightarrow (P_p(z) \wedge \neg P_q(z))))$$

The structural transformation of formulae corresponding to original modal formulae results in the conjunction of the following clauses:

- $\exists x.\, Q_{[r1][r2](p\wedge\neg q)}(x)$
- $\forall x.\, Q_{[r1][r2](p\wedge\neg q)}(x) \rightarrow (\forall y.\, R_{r_1}(x, y) \rightarrow Q_{[r2](p\wedge\neg q)}(y))$
- $\forall x.\, Q_{[r2](p\wedge\neg q)}(x) \rightarrow (\forall y.\, R_{r_2}(x, y) \rightarrow Q_{p\wedge\neg q}(y))$
- $\forall x.\, Q_{p\wedge\neg q}(x) \rightarrow (P_p(x) \wedge Q_{\neg q}(x))$
- $\forall x.\, Q_{\neg q}(x) \rightarrow \neg P_q(x)$

Notice that not all subformulae are replaced with a predicate. For instance, we do not introduce a new predicate to replace $R_{r_1}(x, y) \rightarrow Q_{[r2](p\wedge\neg q)}(y))$ because this subformula does not have a modal logic match.

4.2 Structural transformation for dynamic modal logic

[Sch06, SH07, Sch08] are used as the main references for this section. In the mentioned references, Schmidt and Hustadt used a structural transformation version specifically introduced for dynamic modal logics. The previous papers [HdNS00, HS99, HS98a, HS02] used the structural transformation of Section 4.1. The latter references assumed problem formulae are in negation normal form. Dropping this

assumption makes the classical structural transformation introduce negative definitions for subformulae under negative polarity. The definitions that we would obtain are not suitable for the hyper-resolution framework that we would like to use. Let us see why this is the case by considering a simple example:

Let $\varphi = \neg(p \wedge q)$. The first-order translation to the formula is $\exists x.\, \neg(P_p(x) \wedge P_q(x))$.

By applying structural transformation we get:
$\exists x.\, Q_\neg(x)$
$\wedge\, \forall x.\, Q_\neg(x) \rightarrow \neg Q_\wedge(x)$
$\wedge\, \forall x.\, (P_p(x) \wedge P_q(x)) \rightarrow Q_\wedge(x)$

The clausal form gives the following set:

$$\{\ Q_\neg^*(a),\ \neg Q_\neg^+(x) \vee \neg Q_\wedge^+(x),\ \neg P_p(x)^+ \vee \neg P_q(x)^+ \vee Q_\wedge(x)\ \}$$

Hyper-resolution for this set of clauses does not produce any inference steps because it needs both $Q_\neg^+(x)$ and $Q_\wedge^+(x)$ of the second clause to occur positively in order to make an inference step. In which case, the empty set would be derived and the derivation would stop. In tableau however, there is a $(\neg\wedge)$ rule that needs to be applied and branching to take place. It is not sufficient to end the derivation this early and conclude that the formula is satisfiable because this may not be the case.

The version of structural transformation introduced in [Sch06, SH07, Sch08] solves this problem by introducing three definitions for each formula; one for the formula itself, one for the formula's complement, and a third that states the two formulae are complements of each other. Also, definitions are created for relational formulae. The assumption here is that all occurrences of double negation have been eliminated. If $\psi = \neg\phi$, then $\sim\psi = \phi$. Otherwise, $\sim\psi = \neg\phi$.

Following the mentioned references, let Def′ be a transformation function for dynamic modal formulae and relational formulae defined as follows:

Def$'(\psi)$ is the *definition* of Q_ψ, where Q_ψ is a new predicate symbol uniquely associated with the modal formula ψ.

$$\begin{aligned}\mathrm{Def}'(\psi) \quad =^{def} \quad & \forall x(Q_\psi(x) \rightarrow \pi'(\psi, x)) \\ & \wedge \forall x(Q_{\sim\psi}(x) \rightarrow \pi'(\sim\psi, (x)) \\ & \wedge \forall x(Q_\psi(x) \rightarrow \neg Q_{\sim\psi}(x)),\end{aligned}$$

Def$'(\alpha)$ is the *definition* of R_α, where R_α is a new relational symbol uniquely associated with the relational formula α.

$$\begin{aligned}\mathrm{Def}'(\alpha) \quad =^{def} \quad & \forall x, y.\,(R_\alpha(x, y) \rightarrow \tau'(\alpha, x, y)) \\ & \wedge \forall x, y.\,(\tau'(\alpha, x, y) \rightarrow R_\alpha(x, y)).\end{aligned}$$

The functions $\pi'$and τ' are translation functions of modal and relational formulae to first order logic. They are defined in [Sch08] as given by Figure 4.1 where z is any variable distinct from x.

$\pi'(\bot, x)$	$=$	$\bot$
$\pi'(\neg\bot, x)$	$=$	$\neg Q_\bot(x)$
$\pi'(p, x)$	$=$	$\top$
$\pi'(\neg p, x)$	$=$	$\neg Q_p(x)$
$\pi'(\psi \wedge \phi, x)$	$=$	$Q_\psi(x) \wedge Q_\phi(x)$
$\pi'(\neg(\psi \wedge \phi), x)$	$=$	$Q_{\sim\psi}(x) \vee Q_{\sim\phi}(x)$
$\pi'([\alpha]\psi, x)$	$=$	$\forall z(R_\alpha(x, z) \rightarrow Q_\psi(z))$
$\pi'(\neg[\alpha]\psi, x)$	$=$	$\exists z(R_\alpha(x, z) \wedge Q_{\sim\psi}(z))$
$\tau'(r, x, y)$	$=$	$R_r(x, y)$
$\tau'(\alpha \wedge \beta, x, y)$	$=$	$R_\alpha(x, y) \wedge R_\beta(x, y)$
$\tau'(\alpha \vee \beta, x, y)$	$=$	$R_\alpha(x, y) \vee R_\beta(x, y)$
$\tau'(\alpha^\smile, x, y)$	$=$	$R_\alpha(y, x)$

Figure 4.1: Definition of the translation mappings π' and τ'

Let φ' be the result of applying structural transformation to a modal formula φ as per the definition of Def′. Let N be the set of clauses obtained from applying conjunctive normal form transformation, inner Skolemization, and clausification on φ'. Every clause in N is either a unit clause $Q_\varphi(a)$, where a is a Skolem constant, or represents one of the definitional clauses of Figure 4.2. The set of clauses N preserves the satisfiability of the original formula φ [Sch08].

Subformula θ	Definitional clauses associated with θ
$\bot$	$\neg Q_\bot(x)^+$
$\psi \wedge \phi$	$\neg Q_{\psi\wedge\phi}(x)^+ \vee Q_\psi(x)$
	$\neg Q_{\psi\wedge\phi}(x)^+ \vee Q_\phi(x)$
$\neg(\psi \wedge \phi)$	$\neg Q_{\neg(\psi\wedge\phi)}(x)^+ \vee Q_{\sim\psi}(x) \vee Q_{\sim\phi}(x)$
$[\alpha]\psi$	$\neg Q_{[\alpha]\psi}(x)^+ \vee \neg R_\alpha(x,y)^+ \vee Q_\psi(y)$
$\neg[\alpha]\psi$	$\neg Q_{\neg[\alpha]\psi}(x)^+ \vee R_\alpha(x, f_{\neg[\alpha]\psi}(x))$
	$\neg Q_{\neg[\alpha]\psi}(x)^+ \vee Q_{\sim\psi}(f_{\neg[\alpha]\psi}(x))$
$\alpha \wedge \beta$	$\neg R_{\alpha\wedge\beta}(x,y)^+ \vee R_\alpha(x,y)$
	$\neg R_{\alpha\wedge\beta}(x,y)^+ \vee R_\beta(x,y)$
	$R_{\alpha\wedge\beta}(x,y) \vee \neg R_\alpha(x,y)^+ \vee \neg R_\beta(x,y)^+$
$\alpha \vee \beta$	$\neg R_{\alpha\vee\beta}(x,y)^+ \vee R_\alpha(x,y) \vee R_\beta(x,y)$
	$R_{\alpha\vee\beta}(x,y) \vee \neg R_\alpha(x,y)^+$
	$R_{\alpha\vee\beta}(x,y) \vee \neg R_\beta(x,y)^+$
$\alpha^\smile$	$\neg R_{\alpha^\smile}(x,y)^+ \vee R_\alpha(y,x)$
	$R_{\alpha^\smile}(x,y) \vee \neg R_\alpha(y,x)^+$

Figure 4.2: Definitional clausal forms for $K_{(m)}(\wedge, \vee, \smile)$

Figure 4.2 shows all possible definitional clausal forms for $K_{(m)}(\wedge, \vee, \smile)$ as given in [Sch08]. Negative literals are marked by + which denotes that they are selected by the selection function S.

Let Ξ be a structural transformation function, where $\Xi(\varphi) = \exists x Q_\varphi(x) \wedge \mathrm{Def}'(\varphi)$. By applying Ξ to $\varphi = \neg(\neg \wedge \Box(r_1, p)) \wedge \Box(r_1 \vee r_2, \neg p) \wedge \neg q$, we get the following formula:

$$
\begin{array}{rcl}
\Xi(\varphi) & = & \exists x.Q_{(\wedge_1)}(x) \\
& & \wedge\, \forall x.Q_{(\wedge_1)}(x) \rightarrow (Q_{(\neg\wedge_2)}(x) \wedge Q_{(\Box_1)}(x) \wedge Q_{(\neg q)}(x)) \\
& & \wedge\, \forall x.Q_{(\neg\wedge_1)}(x) \rightarrow (Q_{(\wedge_2)}(x) \vee Q_{(\neg\Box_1)}(x) \vee Q_q(x)) \\
& & \wedge\, \forall x.\, Q_{(\wedge_1)}(x) \rightarrow \neg Q_{(\neg\wedge_1)}(x) \\
& & \wedge\, \forall x.\, Q_{(\neg\wedge_2)}(x) \rightarrow (Q_q(x) \vee Q_{(\neg\Box_2)}(x)) \\
& & \wedge\, \forall x.\, Q_{(\wedge_2)}(x) \rightarrow (Q_{(\neg q)}(x) \wedge Q_{(\Box_2)}(x)) \\
& & \wedge\, \forall x.\, Q_{(\neg\wedge_2)}(x) \rightarrow \neg Q_{(\wedge_2)}(x) \\
& & \wedge\, \forall x.\, Q_q(x) \rightarrow \top \\
& & \wedge\, \forall x.\, Q_{(\neg q)}(x) \rightarrow \neg Qq(x) \\
& & \wedge\, \forall x.\, Q_q(x) \rightarrow \neg Q_{(\neg q)}(x) \\
& & \wedge\, \forall x.\, Q_{(\neg\Box_2)}(x) \rightarrow (\exists z.(R_{r_1}(x,z) \wedge Q_{(\neg p)}(z))) \\
& & \wedge\, \forall x.\, Q_{(\Box_2)}(x) \rightarrow (\forall z.(R_{r_1}(x,z) \rightarrow Q_p(z))) \\
& & \wedge\, \forall x.\, Q_{(\neg\Box_2)}(x) \rightarrow \neg Q_{(\Box_2)}(x) \\
& & \wedge\, \forall x.\, Q_p(x) \rightarrow \top \\
& & \wedge\, \forall x.\, Q_{\neg p}(x) \rightarrow \neg Q_p(x) \\
& & \wedge\, \forall x.\, Q_p(x) \rightarrow \neg Q_{(\neg p)}(x) \\
& & \wedge\, \forall x.\, Q_{(\Box_1)}(x) \rightarrow (\forall z.(R_{\vee^r}(x,z) \rightarrow Q_p(z))) \\
& & \wedge\, \forall x.\, Q_{(\neg\Box_1)}(x) \rightarrow (\exists z.(R_{\vee^r}(x,z) \wedge Q_{(\neg p)}(z))) \\
& & \wedge\, \forall x.\, Q_{(\Box_1)}(x) \rightarrow \neg Q_{(\neg\Box_1)}(x) \\
& & \wedge\, \forall x,y.\, R_{\vee^r}(x,y) \rightarrow (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \\
& & \wedge\, \forall x,y.\, (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \rightarrow R_{\vee^r}(x,y)
\end{array}
$$

Figure 4.3 represents the clausal set of the formula, which will be used for illustration throughout the rest of this chapter.

1.$Q_{(\wedge_1)}(a)$
2.$\neg Q_{(\wedge_1)}(x)^+ \vee Q_{(\neg\wedge_2)}(x)$
3.$\neg Q_{(\wedge_1)}(x)^+ \vee Q_{(\Box_1)}(x)$
4.$\neg Q_{(\wedge_1)}(x)^+ \vee Q_{(\neg q)}(x)$
5.$\neg Q_{(\neg\wedge_1)}(x)^+ \vee Q_{\wedge_2}(x) \vee Q_{(\neg\Box_1)}(x) \vee Q_q(x)$
6.$\neg Q_{(\wedge_1)}(x)^+ \vee \neg Q_{(\neg\wedge_1)}(x)^+$
7.$\neg Q_{(\neg\wedge_2)}(x)^+ \vee Q_q(x) \vee Q_{(\neg\Box_2)}(x)$
8.$\neg Q_{(\wedge_2)}(x)^+ \vee Q_{(\neg q)}(x)$
9.$\neg Q_{(\wedge_2)}(x)^+ \vee Q_{(\Box_2)}(x)$
10.$\neg Q_{(\neg\wedge_2)}(x)^+ \vee \neg Q_{(\wedge_2)}(x)^+$
11.$\neg Q_{(\neg q)}(x)^+ \vee \neg Qq(x)^+$
12.$\neg Q_q(x)^+ \vee \neg Q_{(\neg q)}(x)^+$
13.$\neg Q_{(\neg\Box_2)}(x)^+ \vee R_{r_1}(x, f_{(\neg\Box_2)}(x))$
14.$\neg Q_{(\neg\Box_2)}(x)^+ \vee Q_{(\neg p)}(f_{(\neg\Box_2)}(x))$
15.$\neg Q_{(\Box_2)}(x)^+ \vee \neg R_{r_1}(x, y)^+ \vee Q_p(y)$
16.$\neg Q_{(\neg\Box_2)}(x)^+ \vee \neg Q_{(\Box_2)}(x)^+$
17.$\neg Q_{(\neg p)}(x)^+ \vee \neg Qp(x)^+$
18.$\neg Q_p(x)^+ \vee \neg Q_{(\neg p)}(x)^+$
19.$\neg Q_{(\Box_1)}(x)^+ \vee \neg R_{\vee^r}(x, y)^+ \vee Q_p(y)$
20.$\neg Q_{(\neg\Box_1)}(x)^+ \vee R_{\vee^r}(x, f_{(\neg\Box_1)}(x))$
21.$\neg Q_{(\neg\Box_1)}(x)^+ \vee Q_{(\neg p)}(f_{(\neg\Box_1)}(x))$
22.$\neg Q_{(\Box_1)}(x)^+ \vee \neg Q_{(\neg\Box_1)}(x)^+$
23.$\neg R_{\vee^r}(x, y)^+ \vee R_{r_1}(x, y) \vee R_{r_2}(x, y)$
24.$\neg R_{r_1}(x, y)^+ \vee R_{\vee^r}(x, y)$
25.$\neg R_{r_2}(x, y)^+ \vee R_{\vee^r}(x, y)$

Figure 4.3: Transformation of $\Xi(\varphi)$ into clausal form

4.3 Simulating tableau for $K_{(m)}(\wedge, \vee, \smile)$

This section describes how hyper-resolution and the range-restricted clauses obtained in Section 4.2 can simulate ground semantic tableau for dynamic modal logic following [Sch06, Sch08].

Let N be the set of clauses obtained in Figure 4.3. These clauses and all clauses obtained from the structural transformation of Section 4.2 are *range-restricted*. Variables in range-restricted clauses must occur in the negative part of the clause [GHS03]. All positive range-restricted clauses are ground. This means that hyper-resolution and factoring inference steps on range-restricted clauses always produce positive ground clauses [Sch08]. As a consequence, factoring and splitting in hyper-resolution are always applied to positive ground non-unit clause [Sch08].

In H_{sp}, which denotes hyper-resolution with splitting, the order of the application of rules on each clause is: factoring, then splitting, and last comes the hyper-resolution rule. As a result of this ordering, all non-unit ground clauses are first factored or split into a single unit clause before it is used as a positive premise in any hyper-resolution inference step [Sch08].

Ground unit clauses resulting from applying inference rules on dynamic modal formulae as defined in Section 2.2 are associated with either a relational formula α or dynamic modal formula φ. $Q_\varphi(a)$ where a is a Skolem constant translates to the associated labelled formula $\{a : \varphi\}$. Similarly, $\{R_a(a, b)\}$ translates to $(a, b) : \alpha$.

Every definitional clause in Figure 4.2 has a corresponding expansion or closure tableau rule as per the tableau calculus defined in Figure 2.10. For any definitional clause, the negative literals correspond to the set of premises and the positive literals correspond to the conclusions of a tableau rule. When there is more than one positive literal in the clause, they are separated by | to represent tableau disjunction. The variables of the clause are substituted with constants. Take as an example the definitional clause of $(\neg\psi \wedge \phi)$:

$$\neg Q_{\neg(\psi\wedge\phi)}(x)^+ \vee Q_{\sim\psi}(x) \vee Q_{\sim\phi}(x), \quad \text{becomes the } (\neg\wedge) rule : \quad \frac{s : \neg(\psi \wedge \phi)}{s : \sim\psi \;\mid\; s : \sim\phi}$$

The only tableau rule that is not obtained in this way is the contraction rule and it corresponds to the factoring rule in hyper-resolution.

The tableau calculus for $K_{(m)}(\wedge, \vee, \smile)$ has the relational introduction rules $(\wedge)_I^r$, $(\vee)_I^r$ and $(\smile)_I$. These rules have a side condition that they must not by applied unless the conclusion formula occurs in the input problem. Notice that Def′ (a) has two parts:

- $\forall x, y.(R_\alpha(x, y) \to \tau'(\alpha, x, y))$, which creates definitional clauses corresponding to tableau relational rules.
- $\forall x, y.(\tau'(\alpha, x, y) \to R_\alpha(x, y))$, which creates definitional clauses corresponding to relational introduction rules.

Since the definitions are only introduced for relational formulae occurring in the input problem, the mentioned rules' side condition is satisfied in hyper-resolution derivations.

When hyper-resolution is performed on the range-restricted clauses of Figure 4.2, then every tableau rule application is simulated by one or two hyper-resolution inference steps [Sch08].

26. [OHy : 1, 2] $Q_{(\neg\wedge_2)}(a)$	
27. [OHy : 1, 3] $Q_{(\Box_1)}(a)$	
28. [OHy : 1, 4] $Q_{(\neg q)}(a)$	
29. [OHy : 7, 26] $Q_q(a) \vee Q_{(\neg\Box_2)}(a)$	
30. [Spt : 29.0] $Q_q(a)$	32. [Spt : 29.1] $Q_{(\neg\Box_2)}(a)$
31. [OHy : 12, 28, 30] $\emptyset$	33. [OHy : 13, 32] $R_{r_1}(a, f_{(\neg\Box_2)}(a))$
	34. [OHy : 14, 32] $Q_{(\neg p)}(f_{(\neg\Box_2)}(a))$
	35. [OHy : 24, 33] $R_{\vee^r}(a, f(a))$
	36. [OHy : 19, 27, 35] $Q_p(f(a))$
	37. [OHy : 18, 34, 36] $\emptyset$

Figure 4.4: Using hyper-resolution H_{sp} on N from Figure 4.3

Figure 4.4 shows the hyper-resolution derivation with splitting denoted by H_{sp} on the set of clauses N that was produced previously in Figure 4.3.

The modal tableau formula associated with the symbol $Q_{(\wedge_1)}$ is the original formula φ. The ground clause as a whole corresponds to the pair $\{a : \varphi\}$, which represents the root node in a tableau derivation.

The H_{sp} derivation starts with an inference with the only positive clause $Q_{(\wedge_1)}(a)$. The first derivation resolves $1.Q_{(\wedge_1)}(a)$ with $2.\neg Q_{(\wedge_1)}(x)^+ \vee Q_{(\neg\wedge_2)}(x)$ by applying the substitution $\sigma = \{x/a\}$.

Clause number 2 corresponds to the $(\wedge)$ rule in tableau. This correspondence becomes clear once one thinks of negative literals as the set of premises and the positive ones as the set of conclusions. The following two derivations (number 27 and 28) are similar to the first derivation and are the application of the same rule.

Since the resolution inference steps for a conjunction depends on the number of conjuncts, it becomes clear why the tableau $(\wedge)$ rule, and similarly the $(\neg\Box)$, is broken down into two rules.

In the case of the negated conjunction $(\neg\wedge)$, the hyper-resolution inference is followed by splitting. Each split literal represents the beginning of a new

branch as splitting simulates tableau branching. Clause number 29 is an inference step resulting from negated conjunction. Recall the order of inference rules in H_{sp}, which implies that clause 29 must first be split before applying the hyper-resolution rule. Clauses 30 and 32 originate from splitting clause 29. This results in two branches in the tree-like derivation simulating tableau.

The left branch is explored first. The two clauses $28.Q_{(\neg q)}(a)$ and $30.Q_q(a)$ are resolved with $11.\neg Q_{(\neg q)}(x)^+ \vee \neg Qq(x)^+$ in one hyper-resolution step. The conclusion is the empty set $\emptyset$. This corresponds to deriving a contradiction in a tableau branch between $a : q$ and $a : \neg q$ by applying the closure rule and concluding $\perp$. Just like tableau, this branch is now closed and the second branch is explored.

The split literal from clause 29 makes the clause $32.Q_{(\neg\Box_2)}(a)$. This clause is resolved with two input clauses in two inference steps. It resolves once with $13.\neg Q_{(\neg\Box_2)}(x)^+ \vee R_{r_1}(x, f_{(\neg\Box)}(x))$ and produces $R_{r_1}(a, f_{(\neg\Box)}(a))$. Then it resolves with $14.\neg Q_{(\neg\Box_2)}(x)^+ \vee Q_{(\neg p)}(f_{(\neg\Box)}(x))$ and produces $Q_{(\neg p)}(f_{(\neg\Box)}(a))$. These two steps simulate the two $(\neg\Box)$ rules in tableau.

Since the relational formula $r_1 \vee r_2$ is present in the original modal formula, then the $(\vee_I^r)$ rule is applicable on r_1. We find that clause 35 is the result of the introduction rule. Notice that $(\smile^r)$ rule on the other hand is not applied.

With the new conclusion derived from the relational introduction rule, another inference step becomes possible. The inference step resolves $19.\neg Q_{(\Box_1)}(x)^+ \vee \neg R_{\vee^r}(x, y)^+ \vee Q_p(y)$ with clauses 27 and 35 to produce the result in clause 36.

Finally, clauses 18, 34 and 46 resolve together to conclude the empty set $\emptyset$. This step represents the application of the closure rule in tableau. By this inference steps the derivation ends.

With this example we have practically seen how tableau is simulated step-by-step via first-order hyper-resolution and dynamic modal logic structure transformation.

Chapter 5

Using SPASS as a Tableau Resolution Prover

In this chapter I describe how I extended the resolution theorem prover SPASS so that it behaves like a tableau theorem prover.

There rest of this chapter is organized as follows: Section 5.1 describes the preprocessing steps needed on input dynamic modal logic formulae but the translation to first-order logic takes place. A formula is transformed into a normal form by iteratively applying a set of rewrite rules. Section 5.2 describes how modal formulae and their first-order translations are linked together in a form of a data structure through binding their pointers. Section 5.3 gives a solution that makes use of the structural transformation presented in Section 4.1 rather than the new version described in Section 4.2. Section 5.4 describes how inference steps output by SPASS are translated into tableau derivation steps. In Section 5.5, explanation is given on the presentation of a tableau tree-like derivation by indenting steps to represent branches. Finally, Section 5.7 explains a logical problem caused by inference steps order and the solution to the problem.

5.1 Normalisation of input

In SPASS, all input formulae are first translated to first-order logic. Input formulae are passed to the EML (extended modal logic) module for this process. We have seen in Section 2.4 that it is better to have modal formulae normalized by removing defined operators. Removing defined operators enables early

detection of contradictions in a tableau. So before translation to first-order logic takes course, a formula is passed to a normalisation method that I have added. A formula in this method goes through a number of transformation steps:

1. **Removal of $\leftrightarrow$ and $\rightarrow$**
   The removal of equivalence $\leftrightarrow$ and implication $\rightarrow$ is achieved by a call to a method already available in SPASS. Every subformula of the form $\varphi \leftrightarrow \phi$, the subformula is replaced with $(\varphi \rightarrow \phi) \wedge (\phi \rightarrow \varphi)$. Then, for every subformula of the form $\varphi \rightarrow \phi$, it is substituted with $\neg\varphi \vee \phi$. When the formula is returned back by the method, it is equivalence and implication free.

2. **Removal of $\vee$ and $\langle\alpha\rangle$**
   The implementation of the method that removes $\vee$ and $\langle\alpha\rangle$ is similar to SPASS's implementation of the removal of $\leftrightarrow$ and $\rightarrow$. A vector is used to maintain propositional subformulae. Every subformula of the form $\psi \vee \phi$ is replaced with $\neg\psi \wedge \neg\phi$. Every formula of the form $\langle\alpha\rangle\psi$ is replaced with $\neg[\alpha]\neg\psi$. By the end of this step, all occurrences of $\langle\alpha\rangle$ are eliminated and all occurrences of $\vee$ are over relational formulae.

3. **Removal of double negation**
   This is a simple step that removes occurrences of $\neg\neg$, which are mostly produced by the elimination of the defined symbols in the previous two steps.

4. **Removal of obvious redundancy**
   By obvious redundancies I mean the occurrences of $\varphi \wedge \varphi$ and $\alpha \wedge (\vee)\ \alpha$, where α is a relational formula. A formula of the form $\varphi \wedge \varphi$ is reduced to φ, and similarly $\alpha \wedge (\vee)\ \alpha$ is reduced to α.

5.2 Associating modal formulae with first-order translations

The simulation of modal tableau derivation is done via first-order resolution. This means that we will need to translate first-order derived clauses back to the modal form. Hence, it is important to store pointers to the subformulae of original modal formulae. In SPASS, the modal formulae are destructively changed

when translated to first-order formulae. To overcome this problem, I created two hashmaps: one for storing pointers to propositional formulae and the other for storing pointers to relational formulae. We will find that we will often have to check if the subformula is a relational formula or a propositional formula. Saving each kind in a distinct hashmap is thus beneficial as it facilitates knowing the origin of the subformula.

In a hashmap, each entry is identified by a unique key. An entry in a hashmap stores the pointer to a subformula of the original modal logic formula. The key to this entry is the pointer to the corresponding first-order translation as shown in Figure 5.1.

In the figure, k denotes the key of the hashmap entry and v denotes the value. The dashed triangle to the left represents a first-order formula. The inner triangle is a first-order subformula. The black circles are the keys to the hashmap values but are also pointers to first-order subformulae. Hashmap values are denoted by the white circles, which are pointers to subformulae of the original modal logic formula. Modal logic formulae are represented by the triangles to the right.

This solution enables retrieving the the back-translation of any first-order formula by simply passing its pointer to the hashmap retrieve function. If there is a key in the hashmap, which is the same as the passed pointer, the function sends back the pointer to the corresponding modal formula.

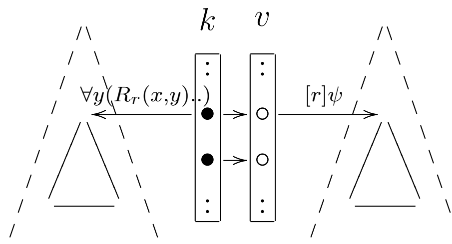


Figure 5.1: Mapping first-order translations to original modal formulae

The association is done during the translation of modal formulae to first-order formulae by first creating a mirror copy of the original modal formula passed to

the translation method. I updated the translation method so that it also accepts as input the modal logic formula copy. The modal logic formula and its copy are then traversed simultaneously. Whenever a first-order formula is constructed to replace a modal formula in a destructive way, both first-order and modal logic formula pointers are stored in one of the two hashmaps according to the type of the modal logic subformula. At the end of the translation process, the original modal logic formula is destructively changed to its first-order equivalent, but all pointers to subformulae are associated with the modal logic formulae in the copy.

5.3 A new approach to renaming

Implementing the structural transformation suggested in [Sch06, SH07, Sch08] and described in Section 4.2 requires fundamental changes to how SPASS handles renaming. In this chapter I introduce a new solution that makes use of the structural transformation module implemented in SPASS.

There are two main differences between the structural transformation presented in [HdNS00, HS02] and in the mentioned references. First, in [Sch06, SH07, Sch08] positive definitions are introduced for negated formulae. The second difference is that definitions are also introduced for relational formulae. In this section, I explain how I accommodated these two points in the implementation of this project and give a detailed example.

Renaming negated formulae

SPASS already has an implementation of a sophisticated renaming module for first-order logic. I tried to find a different solution that makes use of the code with small acceptable changes or additions. The goal was to find a formula that produces the same definitional clauses for $\neg(\psi \wedge \phi)$ and $\neg[\alpha]\varphi$ as the structural transformation in Section 4.2 produces.

I was able to produce the same results by doing the following:

- I conjugated every non-atomic negated formula with an equivalent form where De Morgan's law is applied once.

- I updated the hashmap which associates first-order formulae with original modal logic subformulae. The update associates the new conjugated formula with the modal logic subformula instead of the default association.
- I made all introduced predicates have positive definitions even though some replaced subformulae may have negative polarity.
- The mapping function λ that has been previously defined in Section 4.1, is used to introduce a new Skolem predicate for each subformula associated with a non-atomic modal logic subformula.

Take for instance the first-order formula $\exists x. \neg(p(x) \land q(x))$, which is the first-order translation of the modal logic formula $\neg(p \land q)$. We transform this formula into the equivalent form $\exists x. \neg(p(x) \land q(x)) \land (\neg p(x) \lor \neg q(x))$. Then, we remove the association between the modal logic formula $\neg(p \land q)$ and the original first-order translation $\neg(p(x) \land q(x))$, and make the modal logic formula associated with the new dual form $\neg(p(x) \land q(x)) \land (\neg p(x) \lor \neg q(x))$ instead.

Figure 5.2 shows each first-order subformula eligible for replacement, the modal logic subformula it is associated with, and the Skolem predicate introduced to replace it.

First-order formula	modal formula	Skolem predicate
$\neg(p(x) \land q(x)) \land (\neg p(x) \lor \neg q(x))$	$\neg(p \land q)$	$Q_{\neg(p \land q)}$
$p(x) \land q(x)$	$p \land q$	$Q_{p \land q}$
$\neg p(x)$	$\neg p$	$Q_{\neg p}$
$\neg q(x)$	$\neg q$	$Q_{\neg q}$

Figure 5.2: Introducing fresh Skolem predicates for first-order formulae associated with non-atomic modal formulae

Figure 5.3 shows the positive definitions obtained from applying structural transformation to the first-order formula, and the clauses after conversion to conjunctive normal form, and applying Skolemization and clausification.

The first clause is the unit clause representing the formula and the rest are definitional clauses. By comparison to what we get from applying the structural

Applying structural transformation	Clausal form
$\exists x.Q_{\neg(p\wedge q)}(x)$	1. $Q_{\neg(p\wedge q)}(a)$
$\wedge\forall x.Q_{\neg(p\wedge q)}(x) \rightarrow (Q_{\neg p}(x) \vee Q_{\neg q}(x))$	2.$\neg Q_{\neg(p\wedge q)}(x)^{+} \vee Q_{\neg p}(x) \vee Q_{\neg q}(x)$
$\wedge\forall x.Q_{\neg(p\wedge q)}(x) \rightarrow \neg Q_{p\wedge q}(x)$	3.$\neg Q_{\neg(p\wedge q)}(x)^{+} \vee \neg Q_{p\wedge q}(x)^{+}$
$\wedge\forall x.Q_{p\wedge q}(x) \rightarrow (Q_{\neg p}(x) \wedge Q_{q}(x))$	4.$\neg Q_{p\wedge q}(x)^{+} \vee p(x)$
	5.$\neg Q_{p\wedge q}(x)^{+} \vee q(x)$
$\wedge\forall x.Q_{\neg p}(x) \rightarrow \neg Q_{p}(x)$	6.$\neg Q_{\neg p}(x)^{+} \vee \neg p(x)^{+}$
$\wedge\forall x.Q_{\neg q}(x) \rightarrow \neg Q_{q}(x)$	7.$\neg Q_{\neg q}(x)^{+} \vee \neg q(x)^{+}$

Figure 5.3: Results for applying structural transformation and the produced clausal form

transformation of Section 4.2, we find that the result when compared to Figure 4.2 is identical.

- Clause number 2 and the consequences 6 and 7 map to the result of applying the first part of the definition $\forall x(Q_{\psi}(x) \rightarrow \pi'(\psi, x))$, where $\psi = \neg(p \wedge q)$.
- Clause number 4 and 5 map to the result of the second part $\forall x(Q_{\sim\psi}(x) \rightarrow \pi'(\sim\psi, (x))$.
- Clause number 3 maps to the result of the last part $\forall x(Q_{\psi}(x) \rightarrow \neg Q_{\sim\psi}(x))$.

Consequently, any negated non-literal formula can be transformed to produce the exact set of required definitions. This is implemented as a preprocessing step performed on all first-order translations of modal logic formulae just before renaming is applied.

Let δ be the function that performs this step. δ is applied to $\Pi(\varphi)$. Recall that Π is the standard first-order translation. In this step, any first-order subformula of the form:

- $\neg(\pi(\psi, x) \wedge \pi(\phi, x))$ is transformed to:
  $\neg(\pi(\psi, x) \wedge \pi(\phi, x)) \;\wedge\; \delta(\sim\pi(\psi, x) \vee\; \sim\pi(\phi, x))$
- $\neg(\forall\, y\; \tau(\alpha, x, y) \rightarrow \pi(\psi, x))$ is transformed to:
  $\neg(\forall\, y\; \tau(\alpha, x, y) \rightarrow \pi(\psi, x)) \wedge \delta(\exists\, y\; \tau(\alpha, x, y) \wedge\; \sim\pi(\psi, x))$

The version of structural transformation described in Section 4.2 introduces three definitional clauses for any subformula. However, introducing the extra definitions for anything other than negated non-atomic formulae is not necessary for the simulation and is superfluous. Therefore, the new technique introduced here produces fewer clauses while maintaining the problem's satisfiability result. It also achieves the purpose of reusing the current implementation of renaming in SPASS. A detailed example that compares the results between the two techniques is given after explaining relational renaming.

The `CNFRenaming` flag in SPASS controls which kind of subformulae are replaced by new predicates. I extended this flag with a new option that controls the replacement of subformulae for our purpose. In this new option, and just as we have seen in the example of Figure 5.2, all first-order subformulae associated with original non-atomic subformulae are eligible for renaming. All other subformulae are not.

Renaming relational formulae

Let us look again at how structural transformation definitions are introduced for relational formulae.

$$\begin{aligned}\text{Def'}(\alpha) \quad =^{def} \quad & \forall x, y.\, (R_\alpha(x,y) \rightarrow \tau'(\alpha, x, y)) \\ & \wedge \forall x, y.\, (\tau'(\alpha, x, y) \rightarrow R_\alpha(x,y)).\end{aligned}$$

Notice that the definition can be viewed as the definition of formulae with zero polarity because the first part is equal to a positive definition and the second part is equal to a negative definition. Thus assigning relational formulae a zero polarity value ensures the two required definitions are obtained. This is one of the areas where having a separate hashmap for mapping relational formulae proves fruitful. A first-order formula resulting from a relational subformula in the original modal logic input is easily identified by checking if the relational formulae hashmap returns an association. In this project, the definitions of relational formulae are simply produced by assigning relational formulae a zero polarity value.

This solution however, did not produce accurate results when the converse operator was present. The reason for this is that the first-order translation of $\smile$ does not involve an equivalent operator similar to how the translation of a modal $\wedge$

involves the $\wedge$ operator of the first-order language, but rather switches the order of the variables of the relation it maps to. Take for instance the subformulae $R_r(x, y)$. From just looking at this subformula, we cannot know if it requires renaming because results from the translation of $r^\smile$ or not. Furthermore, there may be another subformula $R_r(u, v)$. In first-order logic, these two subformulae are syntactically equivalent, but they may not be equivalent when looking at the modal logic associations as one may result from translating $r^\smile$ and the over from r.

Relational operators can be applied to relational variables as well as relational formulae. Let us look at the two cases for the converse operator by example and see the problems that arise for each case in more detail:

- Let $\varphi = [r^\smile]p$. The first-order translation for this formula is:
  $\forall x, y.\, R_r(y, x) \rightarrow P_p(x)$.
  Creating the right structural transformation for $R_r(y, x)$ involved three adaptations:

  1. The term $R_r(y, x)$ is an atom. New predicates do not need to be introduced for atoms because it serves no value. In our case however, the atom term represents a complex relational formula and we need to reflect that in the clausal form. This has been overcome by checking if relational atoms are associated with converse relational formulae.
  2. To minimize redundancy, implementations of structural transformation introduce only one predicate for syntactically equivalent formulae. In SPASS, syntactically equivalent formulae are referred to as *further matches*. Further matches for a renamed formula are all replaced with the same introduced predicate even if the further matches were not eligible for renaming. This also caused a problem because the two occurrences of R_r in a formula translated from $[r^\smile]p \wedge \neg[r]q$ get replaced with the introduced predicate for $r^\smile$. This problem is solved by creating a method that returns first-order matches not according to their first-order syntax but according to the syntax of associated modal formulae. This way, the translation of r is not confused as a further match to the translation of $r^\smile$

3. The last adaptation is required when the definitional formulae are created. The converse property has to be reflected in the definition in order for it to be meaningful. Without the added code, SPASS would create a definition of the form $Rel_r(y,x) \rightarrow R_r(y,x) \wedge R_r(y,x) \rightarrow Rel_r(y,x)$, which is meaningless because it achieves nothing as it replaces a literal with another logically equivalent literal. This is corrected by swapping the variables of the introduced predicate wherever it occurs to mirror the converse effect. The inserted definition becomes $Rel_r(x,y) \rightarrow R_r(y,x) \wedge R_r(y,x) \rightarrow Rel_r(x,y)$.

- Now let $\varphi = [(r_1 \vee r_2)^{\smile}]p$
  The problem here is that we need two new predicates for the same first-order formula $R_{r_1}(y,x) \vee R_{r_2}(y,x)$; one for the converse operator and one for the disjunction operator.
  The definitions must be introduced such that:
  $\forall x,y.\, R_{(\smile)}(y,x) \rightarrow R_{(\vee^r)}(x,y)$
  $\wedge\, \forall x,y.\, R_{(\vee^r)}(x,y) \rightarrow R_{(\smile)}(y,x)$
  $\wedge\, \forall x,y.\, R_{(\vee^r)}(x,y) \rightarrow (R_{r_1}(x,y) \vee R_{r_2}(x,y))$
  $\wedge\, \forall x,y.\, (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \rightarrow R_{(\vee^r)}(x,y)$

The problem is more complicated if the relational formula is $((\alpha)^{\smile})^{\smile}$ or even $(((\alpha)^{\smile})^{\smile})^{\smile}$. Such relational formulae are better simplified. The simplification is done by performing a preprocessing step that distributes the converse operator recursively and applies the idempotency law such that the converse is only applied on relational variables. A set of rewrite rules are applied for this step is given in Figure 5.4.

Original formula	Simplified formula
$(\alpha^{\smile})^{\smile}$	α
$(\alpha \wedge \beta)^{\smile}$	$\alpha^{\smile} \wedge \beta^{\smile}$
$(\alpha \vee \beta)^{\smile}$	$\alpha^{\smile} \vee \beta^{\smile}$

Figure 5.4: Converse simplification rewrite rules

When the new predicates are created to replace a first-order subformula, the modal logic formula or relational formula the subformula is associated with is retrieved. The returned modal or relational formula needs to be associated with

the newly introduced symbol. For this purpose I have extended the symbol data structure in SPASS with a new field that is dedicated to storing associated modal/relational formulae. It is also possible to store the first-order formula at this stage in along with the modal logic formula and extend the current implementation slightly to produce not only modal logic tableau but also first-order tableau.

Example on renaming

Now let us apply the new method of renaming on the example used in Section 4.2 for comparison

$$\varphi = \neg(\neg q \wedge [r_1]p) \wedge [r_1 \vee r_2]\neg p \wedge \neg q$$

The standard first-order translation $\Pi(\varphi) of \varphi$ can be obtained as follows:

$$\begin{aligned}
\Pi(\varphi) &= \exists x.\, \pi(\neg(\neg q \wedge [r_1]p) \wedge [r_1 \vee r_2]\neg p \wedge \neg q, x) \\
&= \exists x.\, (\pi(\neg(\neg q \wedge [r_1]p), x) \wedge \pi([r_1 \vee r_2]\neg p, x) \wedge \pi(\neg q, x)) \\
&= \exists x.\, (\neg\pi(\neg q \wedge [r_1]p, x) \wedge (\forall y.\, \tau(r_1 \vee r_2, x, y) \rightarrow \pi(\neg p, y)) \wedge \neg\pi(q, x)) \\
&= \exists x.\, (\neg(\pi(\neg q, x) \wedge \pi([r_1]p, x)) \\
&\quad \wedge (\forall y.\, (\tau(r_1, x, y) \vee \tau(r_2, x, y)) \rightarrow \neg\pi(p, y)) \wedge \neg P_q(x)) \\
&= \exists x.\, (\neg(\neg\pi(q, x) \wedge (\forall y.\, \tau(r_1 x, y) \rightarrow \pi(p, y)) \\
&\quad \wedge (\forall y.\, (R_{r_1}(x, y) \vee R_{r_2}(x, y)) \rightarrow \neg P_p(y)) \wedge \neg P_q(x)) \\
&= \exists x.\, (\neg(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x, y) \rightarrow P_p(y)) \\
&\quad \wedge (\forall y.\, (R_{r_1}(x, y) \vee R_{r_2}(x, y)) \rightarrow \neg P_p(y)) \wedge \neg P_q(x))
\end{aligned}$$

Applying the previously defined *prepare for renaming* function denoted by δ gives the following:

$$\begin{aligned}
\delta(\Pi(\varphi)) &= \delta(\exists\, x\, (\neg(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x, y) \rightarrow P_p(y)) \\
&\quad \wedge (\forall y.\, (R_{r_1}(x, y) \vee R_{r_2}(x, y)) \rightarrow \neg P_p(y) \wedge \neg P_q(x)))
\end{aligned}$$

When δ is applied to a formula, the method checks if the formula is a translation of $\neg\wedge$ or $\neg\Box$. If not, then δ is applied to subformulae.

$$\begin{aligned} = \quad & \exists\, x\ (\delta(\neg(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))) \\ & \wedge\, \delta(\forall y.\, (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \rightarrow \neg P_p(y) \wedge \neg P_q(x)) \end{aligned}$$

The function δ at this point finds the subformula $\neg(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))$, which needs transformation. This formula is transformed so that it is conjugated with the equivalent form $P_q(x) \vee \neg(\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))$. δ is also applied to the newly added part.

$$\begin{aligned} = \quad & \exists x.\, ((\neg(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))) \\ & \wedge\, (\delta(P_q(x)) \vee \delta(\neg\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))) \\ & \wedge\, (\forall\ y\ (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \rightarrow \delta(\neg P_p(y) \wedge \neg P_q(x))) \end{aligned}$$

From the new addition $\neg(\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))$ needs transformation. δ conjugates this formula with $\exists y.\, R_{r_1}(x,y) \wedge \neg P_p(y)$, and again δ is applied on the new addition.

$$\begin{aligned} = \quad & \exists x.\, ((\neg(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y))) \\ & \wedge\, (P_q(x) \vee ((\neg\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y)) \wedge (\exists y.\, R_{r_1}(x,y) \wedge \delta(\neg P_p(y))))) \\ & \wedge\, (\forall y.\, (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \rightarrow \delta(\neg P_p(y)) \wedge \delta(\neg P_q(x))) \end{aligned}$$

At this point, we find that δ cannot apply more transformations as all occurrences of $\neg$ are applied to atomic formulae. The final first-order formula transformed by δ is the following:

$$\begin{aligned} = \quad & \exists x.\, ((\neg(\neg P_q(x) \wedge (\forall\ y\ R_{r_1}(x,y) \rightarrow P_p(y))) \\ & \wedge\, (P_q(x) \vee ((\neg\forall y.\, R_{r_1}(x,y) \rightarrow P_p(y)) \wedge (\exists y.\, R_{r_1}(x,y) \wedge \neg P_p(y)))) \\ & \wedge\, (\forall y.\, (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \rightarrow \neg P_p(y) \wedge \neg P_q(x)) \end{aligned}$$

Recall that δ also changes the associations of first-order translations with modal logic formulae for transformed subformulae. Figure 5.5 breaks down the obtained first-order formula showing which first-order subformulae correspond to original modal logic formulae. The last column lists the predicates that will replace each subformula.

First-order formula	Associated modal formula	Symbol
$\exists x.\,($	$\neg(\neg q \wedge [r_1]p) \wedge [r_1 \vee r_2]\neg p \wedge \neg q$	Q_φ
$(\neg$	$\neg(\neg q \wedge [r_1]p)$	$Q_{(\neg\wedge)}$
$(\neg P_q(x) \wedge (\forall y.\, R_{r_1}(x,y) \to P_p(y)))$	$\neg q \wedge [r_1]p$	$Q_{(\wedge)}$
$\wedge\, (P_q(x)$	q	
$\vee\, ((\neg\forall y.\, R_{r_1}(x,y) \to P_p(y))$	$\neg[r_1]p$	$Q_{(\neg\Box_2)}$
$\wedge\, (\exists y.\, R_{r_1}(x,y)$		
$\wedge\, \neg P_p(y))))$	$\neg p$	$Q_{(\neg p)}$
$\wedge\, (\forall y.$	$[r_1 \vee r_2]p$	$Q_{(\Box_1)}$
$\wedge\, (R_{r_1}(x,y) \vee R_{r_2}(x,y))$	$(r_1 \vee r_2)$	$R_{\vee^r}$
$\to \neg P_p(y))$	$\neg p$	$Q_{(\neg p)}$
$\wedge\, \neg P_q(x))$	$\neg q$	$Q_{(\neg q)}$

Figure 5.5: Association of first-order and dynamic formulae, and introduced symbols

Figure 5.6 displays the application of structural transformation on the formula displayed in Figure 5.5 and its clausal form.

Applying structural transformation	Clausal form
$\exists x.\, Q_{(\varphi)}(x)$	1. $Q_{(\varphi)}(a)$
$\forall x.\, Q_{(\varphi)}(x) \to (Q_{(\neg\wedge)}(x)$	2. $\neg Q_{(\varphi)}(x)^+ \vee Q_{(\neg\wedge)}(x)$
$\wedge\, Q_{(\Box_1)}(x)$	3. $\neg Q_{(\varphi)}(x)^+ \vee Q_{(\Box_1)}(x)$
$\wedge\, Q_{(\neg q)}(x))$	4. $\neg Q_{(\varphi)}(x)^+ \vee Q_{(\neg q)}(x)$
$\forall x.\, Q_{(\neg\wedge)}(x) \to \neg Q_{(\wedge)}(x)$	5. $\neg Q_{(\neg\wedge)}(x)^+ \vee \neg Q_{(\wedge)}(x)^+$
$\forall x.\, Q_{(\neg\wedge)}(x) \to (Q_{(q)}(x) \vee Q_{(\neg\Box_2)}(x))$	6. $\neg Q_{(\neg\wedge)}(x)^+ \vee Q_{(q)}(x) \vee Q_{(\neg\Box_2)}(x)$
$\forall x.\, Q_{(\wedge)}(x) \to (Q_{(\neg q)}(x)$	7.$\neg Q_{(\wedge)}(x)^+ \vee Q_{(\neg q)}(x)$
$\wedge\, Q_{(\Box_2)}(x))$	8.$\neg Q_{(\wedge)}(x)^+ \vee Q_{(\Box_2)}(x)$
$\forall x.\, Q_{(\neg q)}(x) \to \neg Q_{(q)}(x)$	9. $\neg Q_{(\neg q)}(x)^+ \vee \neg Q_{(q)}(x)^+$
$\forall x.\, Q_{(\Box_2)}(x) \to \forall y(R_{r_1}(x,y) \to Q_{(p)}(y))$	10. $\neg Q_{(\Box_2)}(x)^+ \vee \neg R_{r_1}(x,y)^+ \vee Q_{(p)}(y)$
$\forall x.\, Q_{(\neg\Box_2)}(x) \to \neg Q_{(\Box_2)}(x)$	11. $\neg Q_{(\neg\Box_2)}(x)^+ \vee \neg Q_{(\Box_2)}(x)^+$
$\forall x.\, Q_{(\neg\Box_2)}(x) \to (\exists y R_{r_1}(x,y)$	12. $\neg Q_{(\neg\Box_2)}(x)^+ \vee R_{r_1}(x,y)$
$\wedge\, Q_{(\neg p)}(y))$	13. $\neg Q_{(\neg\Box_2)}(x)^+ \vee Q_{(\neg p)})(y)$
$\forall x.\, Q_{(\neg p)}(x) \to \neg Q_{(p)}(x)$	14. $\neg Q_{(\neg p)}(x)^+ \vee \neg Q_{(p)}(x)^+$
$\forall x.\, Q_{(\Box_1)}(x) \to (\forall R_{\vee^r}(x,y) \to Q_{(\neg p)}(x))$	15. $\neg Q_{(\Box_1)}(x)^+ \vee \neg R_{\vee^r}(x,y)^+ \vee Q_{(\neg p)}(x)$
$\forall x,y.\, R_{\vee^r}(x,y) \to (R_{r_1}(x,y) \vee R_{r_2}(x,y))$	16. $\neg R_{\vee^r}(x,y)^+ \vee R_{r_1}(x,y) \vee R_{r_2}(x,y)$
$\forall x,y.\, (R_{r_1}(x,y) \vee R_{r_2}(x,y)) \to R_{\vee^r}(x,y)$	17. $\neg R_{r_1}(x,y)^+ \vee R_{\vee^r}(x,y)$
	18. $\neg R_{r_2}(x,y)^+ \vee R_{\vee^r}(x,y)$

Figure 5.6: Structural transformation and clausal form of renaming example

When we compare the number of clauses produced by the new renaming method in this example and the clauses produced in Figure 4.3, we find that the new method produced only 18 clauses whereas the previous method produced 25. The extra 7 clauses produced in Figure 4.3 are superfluous to the derivation because they result from introducing definitional clauses of formulae that do not exist in the input formula. For example, applying renaming on $(p \wedge q)$ using the structural transformation of Section 4.2 would also include definitional clauses for $\neg(p \wedge q)$, $\neg p$ and $\neg q$ all of which do not occur in the original formula.

5.4 From resolution to tableau

A tableau derivation step consits of two parts:

- the modal logic (sub)formula obtained in that step
- and the justification. The justification indicates the rule that was applied and the formula(e) that it was applied on. For instance $(\wedge, 3)$ says that the derivation step is produced by applying the $(\wedge)$ rule on step 3.

Modal formulae and subformulae are associated with the introduced predicates during renaming. This makes the first part of translating a derivation step just a matter of retrieving the modal formula associated with the literal symbol. To know what rule has been applied however, requires some analysis.

In order to store the tableau derivation steps, I created a new data structure in SPASS. Pointers to all created structures are collected and maintained in a list.

Fields	example1	example2
indentation level	1	2
step number	1	2
rule origin	given	$\neg\wedge$
label	Conjecture0	null
parents	null	1
worlds	(skc0)	(skc0)
modal term	$\neg(q \wedge \neg([r \vee s]\neg q))$	$[r \vee s]\neg q$

Figure 5.7: Tableau steps data structure with examples

Figure 5.7 illustrates the fields of the data structure created for saving tableau derivation information along with two small examples. Fields of the data structure are explained here briefly.

Setting the *indentation level* is done at a stage after the translation is complete and is explained later on in Section 5.5. The indentation level is used when printing out a tableau proof so that it looks like a branching tree. *Step number* is an incremental ID number to identify derivation steps when they are referenced in the reasoning part. *Rule origin* is an enumeration of values; one for each rule plus *unknown* if the translator failed to conclude a rule. *Label* is associated with a user input formula. Printing a label for given formulae is most beneficial when a large number of formulae are input and a proof is output because it identifies which formulae from the input set contributed to the proof. *Parents* store pointers to steps that the rule was applied on. *Worlds* is a list of one or two Skolem constants labelling modal or relational formulae respectively. Finally, *modal term* saves the modal logic subformula.

The production of tableau derivations is done in two steps: translation of input clauses, and translation of derived clauses.

I use a *resolution-tableau association table* to store pointers to each resolution step along with the pointer to the equivalent tableau step.

Translating input clauses

The input set of clauses is passed to an input translation function. According to the applied structural transformation, each unit clause in the input set is positive, ground and corresponds to a dynamic modal logic formula input by the user. Thus, for each unit-clause, a tableau derivation step is created.

Each tableau step is associated with a unique incremental ID number starting from one. For derivations corresponding to an input clause, the rule origin is set to *given* indicating it is input by the user. The clause literal argument, which is a Skolem constant since the clause is ground, represents the world that labels the formula. SPASS allows labelling input formulae, and creates a generic label

if the user did not input a label. This label is also saved in the tableau derivation data structure to identify which input formula it corresponds to.

All other input clauses represent tableau rules. The translation method does not produce tableau derivations for these. Instead it analyses the clause and finds the rule it corresponds to. The number of each of these clauses and the rule name the clause represents are stored in a *rules table*, which is referenced during the production of derived clauses. The names for rules in the implementation are given in Figure 5.8. The only tableau rule that does not take a form of a definitional clause is *contraction* but it is easily identified if the inference rule that is applied is factoring.

Tableau rule	encoding in SPASS
$\wedge$	`And`
$\neg\wedge$	`NAnd`
$\Box$	`Box`
$\neg\Box$	`NBox`
$\wedge^r$	`AndR`
$\wedge^r_I$	`AndRI`
$\vee^r$	`OrR`
$\vee^r_I$	`OrRI`
$\smile$	`Conv`
$\smile_I$	`ConvI`
Cont	`Cont`
$\bot$	`Clash`
unknown	`unknown`

Figure 5.8: Encoding of tableau rules

The algorithm for analyzing the definitional clause and setting a rule is given in Figure 5.9. The algorithm refers to negative literals as premises and to positive literals as conclusions. In SPASS these are called antecedents and succedents respectively. The algorithm basically inspects the premises and conclusions and their arguments. When a predicate has two arguments, then the predicate represents a relational formula and a propositional formula otherwise.

The algorithm starts in line 1 by checking if the clause has got no conclusion. In which case, the conclusion is actually the empty set. A clause of this form

Algorithm: Find rule translation
1. **if** conclusion literals = 0 **then** set *rule* to `Clash`
2. **else if** conclusion literals > 1 **then**
3.   **if** premise literal arguments > 1 **then** set *rule* to `ORR`
4.   **else** set *rule* to `NAnd`
5. **else if** premise literals > 1 **then**
6.   **if** conclusion literal arguments > 1 **then** set *rule* to `AndRI`
7.   **else** set *rule* to `Box`
8. **else if** premise literals = 1 AND conclusion literals = 1 **then**
9.   **if** premise literal arguments = 1 **then**
10.   **if** conclusion literal arguments > 1
11.   OR conclusion literal has new argument
12.   **then** set rule to `NBox`
13.   **else** set rule to `And`
14.   **else if** premise literal arguments = 2
15.   AND conclusion literal arguments = 2 **then**
16.   **if** conclusion literal arguments =
17.   reversed premise literal arguments
18.   **then** set *rule* to `Conv`
19.   **else** set *rule* to `unknown`

Figure 5.9: Algorithm for translating input definitional clauses to tableau rules

represents a closure rule which is named in the implementation as `Clash`. Otherwise, the clause has got a conclusion, and so the algorithm checks if the conclusion is made of more than one literal. If so, then the clause represents a branching rule, which is either $(\vee^r)$ or $(\neg\wedge)$. We differentiate the two rules by checking the number of arguments of the premise. If it contains more than one argument, then the predicate is a binary predicate, which means that it represents a relational formula. In this case, the clause represents a $(\vee^r)$ denoted in the implementation by `ORR`. Otherwise, it represents $(\neg\wedge)$ denoted by `NAnd`.

Reaching line 5 means that there is only one conclusion literal in the clause. So, the algorithm checks if the premise literals of the clause are more than one. If so, then it is either a $([\alpha])$ rule or a $(\wedge^r_I)$ rule. The conclusion of a $([\alpha])$ rule is a unary predicate, whereas the conclusion of a $(\wedge^r_I)$ is a binary predicate. Thus, checking the arguments of the conclusion is sufficient to know what rule the clause represents.

Line 8 checks if the premise literals are equal to 1 and the conclusion literals are also equal to 1, which is actually the only case left to possibly produced definitional clauses. Line 9 does a further check on the premise literal if it contains a single argument. If so, then it checks in line 10 if the conclusion is binary or if the conclusion has a new argument. These two cases represent the two $(\neg[\alpha])$ rules. If the conclusion is not binary and has not got a new argument, then line 13 sets the rule to $(\wedge)$ denoted by `And`.

Lines 14 and 15 check if the clause is binary by checking that the premise and conclusion are binary. In this case the rule represents one of the relational rules $(\smile)$, $(\smile_I)$, $(\wedge^r)$ or $(\vee^r_I)$. Knowing exactly which rule the clause represents is not done at this stage just yet. However, lines 16 and 17 check if the arguments of the premise and conclusion are switched. This indicates that the clause represents one of the two converse rules. For the two rules $(\wedge^r)$ and $(\vee^r_I)$, the algorithm cannot detect which has been applied just yet because the conclusion and premise parts are similar. This is why on line 19 the algorithm sets the rule to `unknown`.

Translating derived clauses

During the derivation of resolution inference steps, the derived clauses are passed to a derivation translation function that I have added to translate the clauses to tableau steps. Derived clauses are all ground and take one of three forms:

- **Positive non-unit clause**
  Positive non-unit clauses are clauses that will eventually be split into unit clauses. Since they have no corresponding formulae in the tableau derivation, I associate them in the resolution-tableau association table with a tableau step previously created. This is done by checking which of the parent clauses returns a corresponding tableau step.

  Non-unit clauses are also important in setting the indentation level of branches according to the number of literals it contains. Explanation on this part is given in Section 5.5.

- **Positive unit clause**
  Most derived clauses fall under this category. Each of these result from an

application of a rule. A derived clause in SPASS has a set of parent clauses. The parent clauses are the set of clauses that resolved together to produce the inference step. If the inference origin is *factoring*, then the tableau rule origin is set to `Cont` denoting contraction. Factoring is applied to only one parent clause. The parent clause's number is used to retrieve the tableau derivation step it represents from the association table. The tableau parent step is stored in *parents*.

If the origin of the inference step is *splitting*, then the origin of the tableau step is one of two: the $(\neg\wedge)$ rule or the $(\vee^r)$ rule. The two rules are differentiated as explained before by checking the clause is unary or binary. A unary clause means the derivation results from $(\neg\wedge)$, and if it is a binary clause then it results from $(\vee^r)$. The parent of a split clause is a positive non-unit clause. As explained before, we do not produce a tableau step for non-unit clauses because they are considered redundant to a previous tableau step. However, I associate non-unit clauses that do not have a tableau translation with the previous logically equivalent tableau derivation step. For instance, both $Q_{\neg(p\wedge q)}(a)$ and $Q_{\neg p}(a) \vee Q_{\neg q}(a)$ are associated with the tableau translation $\neg(p \wedge q)$. So when $Q_{\neg p}(a) \vee Q_{\neg q}(a)$ is referenced as the parent clause, the tableau parent clause would be $\neg(p \wedge q)$.

If a clause results from applying *hyper-resolution*, then one of the parent clauses has to be a definitional clause from the input set corresponding to rules. Recall that we stored all definitional clauses and the rules they correspond to in a *rules table*. Thus, we get the rule origin by retrieving the rule the definitional parent clause represents from the *rules table*. Other parent clauses are used to retrieve tableau parents by checking the resolution-tableau association table. If the retrieved tableau rule is *unknown*, then it is either resulting from $(\vee_I^r)$ or $(\wedge^r)$. The two are differentiated by checking with the relational formula of the parent step. If the conclusion formula is part of the parent formula, then it results from $(\wedge^r)$. Otherwise, it results from $(\vee_I^r)$. Also, if the retrieved tableau rule is *Conv*, then it is either $(\smile)$ or $(\smile_I)$. These are also differentiated in the same way by checking with the relational formula of the parent step. If the conclusion formula is part of the parent formula, then it results from $(\smile)$. Otherwise, it results from $(\smile_I)$.

- **Negative unit clause**
  SPASS uses complement splitting because it is more efficient when deriving inferences. The use of complement splitting affects tableau branching rules slightly. As a reminder, the definitions of branching rules with complement splitting take the following form:

$$(\neg\wedge)\ \frac{s : \neg(\varphi \wedge \psi)}{s : \sim\varphi \ \mid \ s : \varphi, s : \sim\psi} \qquad\qquad (\vee^r)\ \frac{(s,t) : (\alpha \vee \beta)}{(s,t) : \alpha \ \mid \ (s,t) : \sim\alpha, (s,t) : \beta}$$

  Due to complement splitting, negative unit clauses may be derived. When translating these clauses, the complement of the modal/relational formula that is associated with literal symbol is acquired. This means that if the associated formula is already negated, then the complement of the formula is the formula that occurs under the negation.

5.5 Setting tableau step indentation

In order for a tableau proof to be comprehensible, it is important to easily differentiate the different branches in a derivation tree. The idea is mimic to a treelike structure through indenting tableau derivation steps according to a set of rules. In general, when branching occurs, the indentation level is increased, which denotes the beginning of a sub-branch. On the other hand, when the closure rule is applied closing-up a branch, then the indentation level is decreased, which means that we have moved to another sub-branch.

This indentation style is similar to the ones produced by pdl-tableau [Sch07] and MLTP [Li08] systems. Pdl-tableau is a prototypical implementation of the tableau calculus for PDL implemented by Renate Schmidt. MLTP is a efficient and generic modal logic prover implemented by Zhen Li.

Figure 5.10 is a small example showing this indentation style in representation a tableau derivation tree. This is followed by Figure 5.11, which displays the corresponding abstract tree representation of the derivation.

Derivation step	Applied rule
1 a : $\neg\big(\neg(d \wedge \neg d)$ $\wedge\neg(\neg(p \wedge q \wedge s) \wedge s \wedge q \wedge p)\big)$	(given)
2 a : $(d \wedge \neg d)$	($\neg\wedge$:1)
3 a : d	($\wedge$:2)
4 a : $\neg d$	($\wedge$:2)
5 a : $\perp$	($\perp$:3,4)
6 a : $(\neg(p \wedge q \wedge s) \wedge s \wedge q \wedge p)$	($\neg\wedge$:1)
7 a : s	($\wedge$:6)
8 a : q	($\wedge$:6)
9 a : p	($\wedge$:6)
10 a : $\neg(p \wedge q \wedge s)$	($\wedge$:6)
11 a :$\neg s$	($\neg\wedge$:10)
12 a : $\perp$	($\perp$:7,11)
13 a :$\neg q$	($\neg\wedge$:10)
14 a: $\perp$	($\perp$:8,13)
15 a :$\neg p$	($\neg\wedge$:10)
16 a: $\perp$	($\perp$:9,15)

Figure 5.10: Using indentation to represent tableau branches

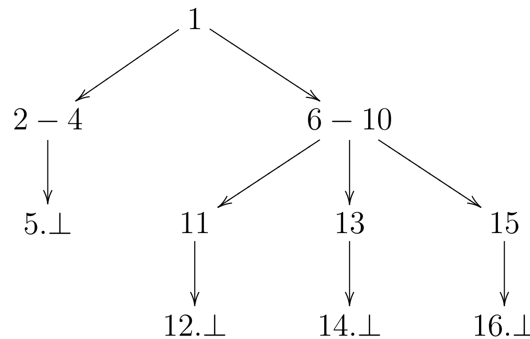


Figure 5.11: Corresponding tree representation

In the example, 1 is the formula at the root of the tree since no formulae occur above it. The formulae 2 to 5 occur on the left branch below 1, while 6 to 10 occur on the right branch below 1.

The left-most branch of the tree derivation is given by 1 to 5. This branch is closed when 5. $a : \perp$ is derived. At this point, backtracking takes place and the derivation continues by exploring the right branch, which is started by 6 a :

$(\neg(p \wedge q \wedge s) \wedge s \wedge q \wedge p)$. Steps 7 to 10 are the result of applying the $\wedge$ rule.

Since a formula may consist of several disjuncted subformulae as in 10 a : $\neg(p \wedge q \wedge s)$, every branch is indented with enough room for the indentation of the rest of branches. Step 10 results into 3 distinct branches, the first sub-branch started by step 11 is indented 2 units inwards. The second sub-branch started by step 13 is indented a single unit. The last sub-branch started by step 15 does not need indentation.

For a given formula in the derivation, the branch on which it occurs is given by the sequence of formulae above it with the same indentation level or a smaller indentation level. If a previous step has a smaller indentation level, then again only steps with the same indentation level or a smaller indentation level are part of the branch.

For instance, in the example in Figure 5.10, if we wanted to identify the branch that caused the clash in step 14, we look at the previous steps. Step 13 has the same indentation level as 14, so we know it belongs to the branch. Steps 11 and 12 do not belong to the branch because they have a bigger indentation level. Step 10 has a smaller indentation level so it belongs to the branch and it also becomes the new measure. Steps 6 to 9 are all on the same indentation level as 10, so they are part of the branch. Steps 2 to 5 have a bigger indentation so they are not. Finally, step 1, which is the root is of course part of the branch. Thus, the branch consists of the sequence $\{1, 6, 7, 8, 9, 10, 11, 13, 14\}$. Although 2, 3, 4 and 5 have the same indentation level as 14, they are not part of the branch as they belong to the left-most branch as illustrated in Figure 5.11.

Calculating the indentation level is done in two steps. The first step is to create a *branches table* where each tableau step known to cause branching has an entry associated with the number of branches that it would produce as well as a branching counter initialized with the same number. This is done during the back translation of resolution derivation steps.

According to the way renaming is defined, resolving a clause of the form $Q_{\sim(p\wedge q)}(a)$ with the definitional clause for $Q_{\sim(p\wedge q)}$ produces this resolvent:

$$Q_{\sim p}(a) \vee Q_{\sim q}(a)$$

Splitting is applied on the latter clause. However, the resolvent is actually semantically equivalent to the previous clause when translated into tableau and thus is not transformed into a tableau step. However, the number of succedent literals from this extra step is what we use in setting up the entries in the *branches table* and associating them with the respective tableau formula.

For the example used in Figure 5.10 for instance, the *branches table* would have two entries as follows:

Clause	branches	branch_counter
$1'$	2	2
$10'$	3	3

Figure 5.12: Example of a branches table

$1'$ and $10'$ represent resolution clauses that are not transformed into tableau derivation steps but are semantically equivalent to steps 1 and 10 respectively.

The second step is setting up an indentation level for every formula contributing to the proof. If completion is found when the input is satisfiable, then setting the indentation level becomes irrelevant and this step is ignored. This is so because indentation is meant to represent a tree-like structure. When the input is satisfiable, the printed tableau consists only of a single open branch rather than a derivation tree.

By looping through the proof derivation steps, the rule origin of the equivalent tableau translation is examined as follows:

- If the applied rule is the $(\neg\wedge)$ rule or the $(\vee_r)$ rule then branching takes place. A *branch_counter* associated with the parent clause is retrieved from the *branches table*. Recall that the *branch_counter* is initialized with

the number of branches the step would produce. The general indentation counter (originally initialized to 1) is increased by n, where $n =$ *branch_counter*-1. The indentation value n is pushed into a stack. The step indentation level is set equal to the new value of the general indentation counter. The value of the *branch_counter* is decreased by 1 and updated in the *branches table*.

For example, step 2 in Figure 5.10 is a result of applying the $(\neg\wedge)$ rule. We retrieve the *branch_counter* value associated with $1'$ from Figure 5.12. The retrieved value is 2. Thus, n is equal to 1. The counter is then decreased by 1 extra indentation level. So when the second branch at step 6 is reached, the retrieved counter value will be 1 and n will be equal to 0 indicating the branch does not need indentation.

When the *branch_counter* reaches its minimum value, it gets re-initialized to the original value of *branches*. This is because in some derivations, the same formula might need to reproduce its sub-branches on more than one branch. Not reinitializing the counter would cause all the sub-branches of the second branching attempt to have the same indentation level.

- If the applied rule is a `Clash` $(\perp)$, then the formula indentation level is assigned the value of the general indentation counter. The counter is then decreased by the indentation value popped-out from the stack for backtracking.
- Otherwise, the formula indentation level is set to the indentation counter value.

5.6 Tableau proofs and models

At the end of the search in SPASS, if the problem is satisfiable, then a saturated set of clauses is output representing a model. Otherwise, a proof is produced. In both cases, the output does not always mirror the resolution derivation steps. For example, when a derivation explores a branch and it turns out to be closed, it continues with other branches until it finds an open branch. If an open branch is found, then all derivation clauses that do not belong to the open branch are

eliminated from the saturated set of clauses. Similarly, we find the output of a proof includes only derivation steps that contribute to the proof. It follows that not all translated tableau steps are needed for a proof or a model. For this reason, I have created a method that selects the necessary tableau steps for the proof or model from the translated derivation set.

The implementation prints out a tableau single open branch representing a model if the input is satisfiable. A single open branch is displayed in a linear format. Otherwise, if all tree branches are closed and a proof is found, then the proof tree is output in an indented style as explained in Section 5.5.

We have two kinds of formulae: given formulae and derived formulae. The notation for a given formula is displayed as:

$$n[\texttt{Giv} : \texttt{label}] \mid\mid w \ : \ \textit{modal_formula}$$

n denotes the tableau step number which is unique for each derived formula. `Giv` denotes that the formula's origin is given by the user. The given formula is identified by a unique `label`, which has either been specified by the user in the input set or it was generically produced by SPASS. The two parallel lines || are just a separator between the formula information and the labelled formula itself. w is a Skolem term representing a world in a model.

Derived formulae have a slightly different notation:

$$n[\texttt{origin} : \{parent\}^{+}] \mid\mid w \ : \ \textit{modal_formula}$$
$$n[\texttt{origin} : \{parent\}^{+}] \mid\mid (w, v) \ : \ \textit{relational_formula}$$

The difference between a given formula and a derived one, is the [`origin` : $parent^{+}$] part. `origin` represents what rule has been applied to derive the step. Rule names can be found in Figure 5.8. Some derived formulae are relational formulae. In which case, relational formulae are labelled with a set of two Skolem terms (w, v).

$\{parent\}^{+}$ represents the formula(e) that the rule was applied on. The plus sign denotes that for any derived formula, there must be at least one formula that the rule was applied on. Derived formulae usually have utmost 2 parents

depending on the applied rule. However, for the ($\wedge_I^r$) rule, the number of parents depends on the number of arguments the relational $\wedge$ has in the original modal formula. If we make the $\wedge$ operator a binary operator, then all rules would have at most 2 parents.

5.7 Branching comes last

During the implementation of the resolution-tableau prover, a problem was noticed when looking at some of the tableau proof tree output. In a tableau tree, each node contributing to the branch must be present on the branch. In other words, any node that is not appearing on a specific branch is not considered part of it. For some test cases, some nodes seemed missing from where they should appear.

We have seen in Section 5.10 how indentation is set and how branch nodes are identified. As a reminder, the indentation of a tableau formula is increased when branching and decreased after a branch is closed by a closure rule. Inference steps in between the beginning of a branch and its closure inherit the indentation level of the branch it is on. This means that nodes resulting from inference steps that could have been derived before the main branch splits will inherit one of the smaller branch's indentation level. This will make this step seem to be belonging to one branch without the others. Thus the step will seem to be *missing* from other branches.

Let us look at the proof example in Figure 5.13 to understand the problem more clearly. Nodes 9 and 10 result from applying the ($\neg\Box$) rule on node 8. At this stage, there are two applicable rules: the ($\Box$) rule is applicable on nodes 7 and 9, and the ($\neg\wedge$) rule is applicable on node 10. The derivation in the figure, applied the branching rule ($\neg\wedge$) first and concluded node 11, which started the left branch. Node 12 is the result of applying the ($\Box$) rule, which could have been applied before branching. Since node 12 is derived after branching took place, the node inherited the indentation level of left branch. After a few derivation steps, the left branch is closed and the right branch is explored at node 36. Notice now that node 37 is the result of applying the ($\neg\wedge$) rule on node 12. Since node 12 is referenced by another node on the right branch, it seems missing from this branch.

Derivation	Justification
$\vdots$	
7. $a : [r]\neg(p \wedge a)$	
8. $a : \neg[r](\neg p \wedge \neg q)$	
9. $(a, f_1(a)) : r$	$(\neg\Box : 8)$
10. $f_1(a) : \neg(\neg p \wedge \neg q)$	$(\neg\Box : 8)$
11. $f_1(a) : p$	$(\neg\wedge : 10)$
12. $f_1(a) : \neg(p \wedge a)$	$(\Box : 7, 9)$
13. $f_1(a) : \neg p$	$(\neg\wedge : 12)$
$\bot$	$(\bot : 11, 13)$
15. $f_1(a) : \neg a$	$(\neg\wedge : 12)$
$\vdots$	
$\bot$	
36. $f_1(a) : q$	$(\neg\wedge : 10)$
37. $f_1(a) : \neg p$	$(\neg\wedge : 12)$
$\vdots$	
$\bot$	
42. $f_1(a) : \neg a$	$(\neg\wedge : 12)$
$\vdots$	
$\bot$	

Figure 5.13: Illogical proof structure

Notice that the attempt of changing the indentation level of node 12 to the indentation level of the main branch it actually belongs to would not solve the problem. This is because node 11 would then look like an open ended branch and not part of the branch containing nodes 12, 13, and 15.

Figure 5.14 shows the tree representation of the example in Figure 5.13.

If we go back to how tableau operates, we find that if a branching rule is applied before an applicable non-branching rule, then the non-branching rule has to be applied for every created branch. However, the heuristic for applying rules imply applying branching rules as late as possible because it results in a better, more efficient tableaux [Kel97]. Delaying splitting in resolution until no other inference step is applicable is exactly what we need for the solution of the missing nodes problem.

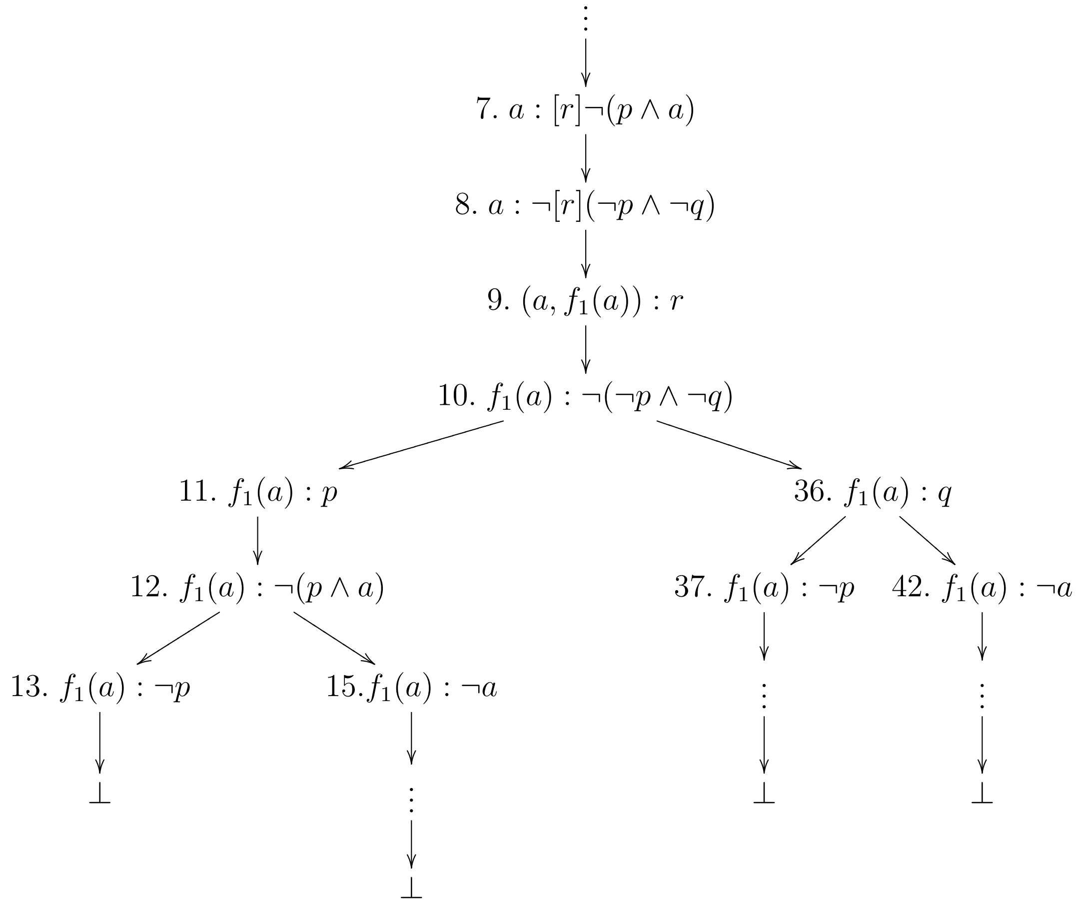


Figure 5.14: Tree representation of illogical proof structure

The technical implementation of the solution requires brief understanding on how SPASS derives clauses. A detailed description on how SPASS derives clauses can be found in [Wei07]. Briefly, the overall search loop with splitting in SPASS deals with two main sets and a stack: a set of usable clauses, a set of worked-off clauses, and a split stack. The basic inference procedure loops until either the usable clauses are exhausted or a contradiction is detected while the split stack is empty. If backtracking is not needed, then a new clause called *given* is selected from the usable set according to some heuristics. If the clause is splittable, then it gets split and the split stack is updated. The inference rules are then applied and the clause is moved to the worked-off clauses. At the end, if the usable clause set is empty then, the problem is satisfiable. Otherwise, the problem is unsatisfiable.

The goal is to adjust the order of applied rules on the usable set as a whole. This means adding a heuristic to how the clauses are chosen from the usable set. The idea is to choose a clause that is not splittable as long as one exists in the usable clauses set. As mentioned before, the implementation of structural transformation produces only range-restricted clauses. Along with ordered hyper-resolution, splitting is always applied on positive ground clauses.Consequently, I created a method that returns a clause with less than *two* positive literals as long as one exists. Notice that clauses of the form $\neg Q_1(x) \vee Q_2(x) \vee Q_3(x)$ are not splittable at this point, but these clauses represent a branching rule and thus the method does not return occurrences of these clauses.

Algorithm: Return unsplittable clause

Input: Usable clauses set
Return: A clause that cannot be split or null
1. set *Result* to null
2. set $s = 2$
3. **for all** c in *usable_clauses*
4. **if** succedent literals of c are less than s
5. *Result* $= c$
6. $s =$ number of succedent literals of c
7. **end if**
8. **end for**
9. **return** *Result*

Figure 5.15: Algorithm for choosing a clause not suitable for splitting

The algorithm in Figure 5.15 shows how the method for choosing a clause that is not splittable is implemented. Succedent literals denote positive literals of a clause in SPASS. The algorithm returns either a clause with zero or 1 positive literals or a null value indicating that all clauses are suitable for splitting.

Chapter 6

Adding relational frame conditions

This chapter is organized as follows: Section 6.1 describes how relational frame conditions can be imposed in first-order resolution and the derivation steps are affected. In Section 6.2, I redfine the tableau calculus so that it reflects what happens in the first-order resolution derivations. In Section 6.3 I explain how clauses obtained from adding relational conditions are translated and presented in the output. In Section 6.4 I propose another solution to supporting relational conditions, which is based on defining worlds.

6.1 Simulating relational conditions

In SPASS, relational frame conditions can be imposed by adding first-order logic formulae to the axioms part of the input file. Figure 6.1 shows the first-order formulae and the obtained clausal form for axioms T, D, B, 4 and 5.

	First-order definition	Clausal Form
T	$\forall x.R_{r_i}(x,x)$	$R_{r_i}(x,x)$
D	$\forall x \exists y.R_{r_i}(x,y)$	$R_{r_i}(x,f_i(x))$
B	$\forall x.R_{r_i}(x,y) \rightarrow R_{r_i}(y,x)$	$\neg R_{r_i}(x,y)^+ \vee R_{r_i}(y,x)$
4	$\forall x,y,z(R_{r_i}(x,y) \wedge R_{r_i}(y,z)) \rightarrow R_{r_i}(x,z)$	$\neg R_{r_i}(x,y)^+ \vee \neg R_{r_i}(y,z)^+ \vee R_{r_i}(x,z)$
5	$\forall x,y,z(R_{r_i}(x,y) \wedge R_{r_i}(x,z)) \rightarrow R_{r_i}(y,z)$	$\neg R_{r_i}(x,y)^+ \vee \neg R_{r_i}(x,z)^+ \vee R_{r_i}(y,z)$

Figure 6.1: Definitions and produced clausal form for axioms T, D, B, 4 and 5

The clausal form of the axioms' definitions can be transformed to rules that are very similar to the structural rules previously defined in Figure 2.9. The difference is that rules produced here are defined with variables and not constants. By viewing a clause's negative literals as premises and positive literals as conclusions, we obtain the relevant tableau rule. Literals of the form $R_r(x, y)$ are transformed to $(x, y) : r$.

For instance, the clause $\neg R_{r_i}(x, y)^+ \vee R_{r_i}(y, x)$ represents the rule $(B)\ \dfrac{(x, y) : r}{(y, x) : r}$.

By looking back at the clauses of Figure 6.1, we find that we can categorise them into two categories. One category contains positive clauses, which are the first two. These clauses translate to rules with no premise. The other category contains the rest of clauses, which translate to rules that have premises.

In tableau, all these axiom rules explicitly bind a world and its successor because the application of the rule implies asserting its conclusion. However, using first-order hyper-resolution means that the conclusions of the category with no premises can never be derived in their ground form. Instead, we find that these clauses have one of two effects: they either resolve with other relational rules to create a new rule, which is also not ground, or act as a catalyst to applying box rules, which has a propagational effect on formulae.

Now let us see an example on how new rules are produced from combining more than one relational condition by looking at how the corresponding clauses can be resolved using hyper-resolution:
The clause $R_{r_i}(x, f_i(x))$ can resolve with both $\neg R_{r_i}(x, y)^+$ and $\neg R_{r_i}(y, z)^+$ from $\neg R_{r_i}(x, y)^+ \vee \neg R_{r_i}(y, z)^+ \vee R_{r_i}(x, z)$ by applying the substitution
$\sigma = \{f_i(x)/y, f_i(f_i(x))/z\}$. The result is $R_{r_i}(x, f_i(f_i(x)))$. This result translates to a new rule of the first category with no premise. We will denote them by PR standing for *propagational rule*. The derived rule translation can be written out as follows: $(PR)_j\ \dfrac{\cdot}{(x, f_i(f_i(x))) : r_i}$.

We find that also derived rules can produce new rules. For instance, the rule we just derived can by iterated in the same way to produce $(PR)_{j+1}\ \dfrac{\cdot}{(x, f_i(f_i(f_i(x)))) : r_i}$

The other kind of inference that propagational rules contribute to as we have mentioned is resolving with a clause representing a box rule. If we take the following set of clauses as an example, where x and y represent variables and a represents a constant:

$R_{r_i}(x, x)$
$\neg Q_{\Box p}(x)^+ \vee \neg R_{r_i}(x, y)^+ \vee Q_p(y)$
$Q_{\Box p}(a)$

We find that the three clauses resolve together and produce $Q_p(a)$.

It is important to mention that the behaviour of propagational rules causes a problem because although the right derivations are produced, conventional tableau definitions fail to justify some of the produced steps. This is why in the next section, I redefine tableau rules.

Including other first-order formulae imposing relational conditions with multiple relations for dynamic modal logic is also possible. For instance, we can add the first order-formula

$$\forall x.\exists y.R_{r1}(x, x) \rightarrow (R_{r1}(x, y) \wedge R_{r2}(x, y))$$

This first-order formula imposes relational conditions such that for any world x in our model, if this world is connected to itself via a R_{r1} relation, then the world has a successor that is connected via two relations: R_{r1} and R_{r2}. This first-order formula will result into two clauses:

$$\neg R_{r1}(x, x) \vee R_{r1}(x, f_1(x))$$
$$\neg R_{r1}(x, x) \vee R_{r2}(x, f_1(x))$$

These two clauses represent two *structural rules* denoted by SR:

$$(SR)_1 \ \frac{(x, x) : r_1}{(x, f_1(x)) : r_1} \qquad\qquad (SR)_2 \ \frac{(x, x) : r_1}{(x, f_1(x)) : r_2}$$

6.2 Redefining tableau rules

We have seen in Section 6.1 how clauses resulting from applying relational conditions resolve with other clauses. This behaviour needs to be explained and

represented in terms of tableau rules. This section is an attempt to redefine tableau rules in a way that reflects what happens in the simulation via first-order resolution.

The axioms T, D, B, 4 and 5 are associated with rules that I define as given in Figure 6.2. These rules are defined similarly to how tableau structural rules are defined but the difference is that the definitions involve variables instead of constants as we have seen before. Doing this creates two kinds of rules: rules that are defined over **variables** and rules that are defined over **constants**. Throughout the rest of this thesis, I will refer to rules with constants as *application rules*, and rules with variables as *property rules*.

$$(PR)_T \frac{.}{(x,x):r} \qquad (PR)_D \frac{.}{(x,f_i(x)):r}$$

$$(SR)_B \frac{(x,y):r}{(y,x):r} \quad (SR)_4 \frac{(x,y):r,(y,z):r}{(x,z):r} \quad (SR)_5 \frac{(x,y):r,(x,z):r}{(y,z):r}$$

where x, y and z are variables

Figure 6.2: Tableau rules for axiom properties

Each of the property rules has a relevant application rule that is defined with constants. We have seen that *propagational rules* combine with the box application rule to create a new box rule. Thus, the associated application rules for the T and D propagational rules are:

$$(\Box)_T \frac{T,s:[r]\varphi}{s:\varphi} \qquad (\Box)_D \frac{D,s:[r]\varphi}{t:\varphi}$$

Figure 6.3 shows the definitions of application rules for traditional modal logic. Notice that property rules form part of the premise in the axioms application rules.

Property rules are added to the input problem. This addition activates relative application rules and make them applicable according to the problem.

For instance, take the problem $[r]p$ in K. We find that no rule from Figure 6.3 is applicable.

$$1.\, a\ :\ [r]p$$

$$(\wedge)_1\ \frac{s:(\varphi\wedge\phi)}{s:\varphi} \qquad (\wedge)_2\ \frac{s:(\varphi\wedge\phi)}{s:\phi} \qquad (\neg\wedge)\ \frac{s:\neg(\varphi\wedge\phi)}{s:\sim\varphi \mid s:\varphi, s:\sim\phi}$$

$$(\neg\Box)_1\ \frac{s:\neg[r]\varphi}{t:\sim\varphi} \qquad (\neg\Box)_2\ \frac{s:\neg[r]\varphi}{(s,t):r} \qquad (\Box)\ \frac{(s,t):r, s:[r]\varphi}{t:\varphi}$$

$$(\Box)_T\ \frac{T, s:[r]\varphi}{s:\varphi} \qquad (\Box)_D\ \frac{D, s:[r]\varphi}{t:\varphi} \qquad (\text{Rule})_B\ \frac{B,(s,t):r}{(t,s):r}$$

$$(\text{Rule})_4\ \frac{4,(s,t):r,(t,u):r}{(s,u):r} \qquad (\text{Rule})_5\ \frac{5,(s,t):r,(s,u):r}{(t,u):r}$$

Figure 6.3: Tableau application rules for traditional modal logic

Now if we test $[r]p$ in KT, then we add the T property rule to the input set and then we can apply the application rule $(\Box)_T$

$1.\, T$ (given rule)

↓

$2.\, a : [r]p$ (given)

↓ ($\Box$:1,2)

$3.\, a : p$

Similarly, testing $[r]p$ in KTD would make the $(\Box)_D$ applicable as well.

$1.\, T$ (given rule)

↓

$2.\, D$ (given rule)

↓

$3.\, a : [r]p$ (given)

↓ ($\Box$:1,3)

$4.\, a : p$

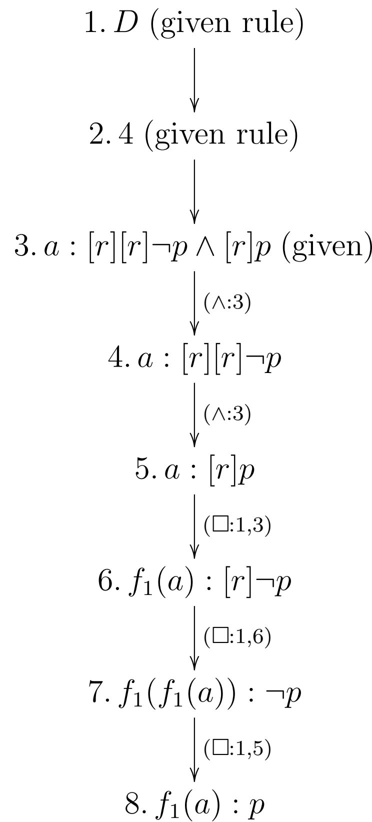
↓ ($\Box$:2,3)

$5.\, f_1(a) : p$

We previously defined the nodes of the branching tree representation of a tableau derivation as being labelled formulae. Now, we have two kinds of nodes,

rule nodes and labelled formulae nodes.

In practice, this given definition of tableau rules is not quite sufficient. The reason is that combining different relational properties may have consequences that are not covered by the definition. These combinations create derived rules as we have seen in the previous section. For example, if we test the satisfiability of $[r][r]\neg p \wedge [r]p$ in $KD4$ with the defined rules in Figure 6.3 we get the following derivation:



The first two nodes of the derivation tree are the given rules implied by D and 4. Node 3 is the input problem. The two following nodes 4 and 5 are the result of applying the (∧) expansion rule. Then the (□) rule is applied on node 3 with the given rule associated with D. This step creates node 6 with the formula $[r]\neg p$ labelled with $f_1(a)$ which is a successor for a. The (□) rule is applied again on node 6 with the given rule associated with D. This creates node 7 with the formula $\neg p$ labelled with $f_1(f_1(a))$, which is a successor for $f_1(a)$. The last derivation results from applying the (□) rule on node 5 with the given rule D,

and this creates the formula p labelled with $f_1(a)$.

Notice that the transitivity rule should be applied and produce $(a, f_1(f_1(a))) : r$ but since $(a, f_1(a)) : r$ and $(f_1(a), f_1(f_1(a))) : r$ are not asserted explicitly, this inference step does not take place. In fact, with the correct derivation steps, the node $f_1(f_1(a)) : p$ should be produced and a contradiction should be derived with node 7 closing up the tree.

To overcome this problem for this particular example, we add a rule for the combination of seriality with transitivity as follows:

$$(D4)\ \frac{.}{(x, f_i(f_i(x)))}$$

which creates the derived application rule

$$(\Box)_{D4}\ \frac{D4, s : [r]\varphi}{f_i(f_i(s)) : \varphi}$$

Other axiom combinations will have other consequences. I do not attempt here to cover all possibilities, which may not be practical in the first place. However, we have seen that first-order hyper-resolution automatically derives these rules for any given combination by applying hyper-resolution and unification. In fact, we find that using first-order resolution has the added value that allows us to impose generic relational frame conditions on dynamic modal logics without worrying about defining related tableau rules.

Take as an example a rule of the form $(SR)_i\ \frac{(x, y) : r}{(x, x) : r}$. This rule says that for any two arbitrary worlds x and y, if there is a connection from x to y via R_r, then x is also connected to itself via the same relation. Since this rule has got a premise, then the application rule takes the form

$$(\text{Rule})_{SR_i}\ \frac{SR_i, (s, t) : r}{(s, s) : r}$$

In general, we can say that

$$(\text{Rule})_{SR_i}\ \frac{SR_i, ground(prem(SR_i))}{ground(conc(SR_i))}$$

gives the application rule for any structural rule, where $ground(prem(SR_i))$ gives the ground premise of the rule and $ground(conc(SR_i))$ gives the ground conclusion of the rule.

Propagational rules in dynamic modal logic are also affected by the relational introductory rules $(\wedge_I^r)$, $(\vee_I^r)$, and $(\smile_I)$.

For example, if the converse introductory rule for r_j is present and we had the rule

$$(PR(r_j)) \quad \frac{\cdot}{(x, f_i(x)) : r_j}$$

then we also get the derived rule

$$(PR(r_j^\smile)) \quad \frac{\cdot}{(f_i(x), x) : r_j^\smile}$$

In general, we can say that for any given or derived propagational rule $(PR(\alpha))_i$, we have a relevant application rule that takes the form:

$$([\alpha])_i \quad \frac{PR_j(\alpha), s : [\alpha]\varphi}{t : \varphi}$$

where s and t are the result of applying unification.

The general notation makes arbitrary relational conditions in our simulator for dynamic modal logic justified in tableau.

Figure 6.4 gives the new definition of tableau calculus for dynamic modal logic with relation conditions.

$$(\bot)\ \frac{s:\varphi, s:\neg\varphi}{\bot}$$

$$(\wedge)_1\ \frac{s:\varphi\wedge\phi}{s:\varphi} \qquad (\wedge)_2\ \frac{s:\varphi\wedge\phi}{s:\phi} \qquad (\neg\wedge)\ \frac{s:\neg(\varphi\wedge\phi)}{s:\sim\varphi \mid s:\varphi, s:\sim\phi}$$

$$(\neg[\alpha])_1\ \frac{s:\neg[\alpha]\varphi}{(s,t):\alpha} \qquad (\neg[\alpha])_2\ \frac{s:\neg[\alpha]\varphi}{t:\sim\varphi} \qquad ([\alpha])\ \frac{(s,t):\alpha, s:[\alpha]\varphi}{t:\varphi}$$

$$(\smile)\ \frac{(s,t):\alpha^{\smile}}{(t,s):\alpha} \qquad (\smile)_I\ \frac{(t,s):\alpha}{(s,t):\alpha^{\smile}}$$

$$(\wedge)^r_1\ \frac{(s,t):\alpha\wedge\beta}{(s,t):\alpha} \qquad (\wedge)^r_2\ \frac{(s,t):\alpha\wedge\beta}{(s,t):\beta} \qquad (\wedge)^r_I\ \frac{(s,t):\alpha, (s,t):\beta}{(s,t):\alpha\wedge\beta}$$

$$(\vee)^r_{I,1}\ \frac{(s,t):\alpha}{(s,t):\alpha\vee\beta} \qquad (\vee)^r_{I,2}\ \frac{(s,t):\beta}{(s,t):\alpha\vee\beta}$$

$$(\vee)^r\ \frac{(s,t):\alpha\vee(s,t):\beta}{(s,t):\alpha \mid (s,t):\sim\alpha, (s,t):\beta}$$

$$(contr)\ \frac{s:\neg(\psi\wedge\psi)}{s:\sim\psi} \qquad (contr)^r\ \frac{(s,t):\alpha\vee\alpha}{(s,t):\alpha}$$

$$([\alpha])_i\ \frac{PR_j(\alpha), s:[\alpha]\varphi}{t:\varphi} \qquad (\text{Rule})_{SR_i}\ \frac{SR_i, ground(prem(SR_i))}{ground(conc(SR_i))}$$

(i) t in the rules $(\neg\Box)_1$ and $(\neg\Box)_2$ represents a new constant on the branch.
(ii) The rules $(\wedge)^r_I$, $(\vee)^r_I$ and $(\smile)_I$ have the side condition that the relational formulae in the conclusions occur as subformulae of a box in the input problem.

Figure 6.4: New tableaux calculus for $K_{(m)}(\wedge,\vee,\smile)$

6.3 Representing rule nodes in the output

The translation of input clauses is a little different when a relational condition is added. Each clause in the input set first goes through a method that tests if the clause represents an added relational frame condition. The method does a check on all literals of the clause. All literals of such clauses need to be binary and original. Binary literals indicate they represent accessibility relations. By original literals here I mean that they are not introduced from applying structural transformation. For a clause that passes the test, a tableau derivation step is

created. The step's origin is set to `GRule` indicating that this is a given rule by the user, which could either be a structural rule or a propagational rule. These added rules are inserted in the *rules table* like all other rules indexed by the clause number.

When printing tableau steps, the print method checks if the printed step's origin is `GRule`. If so, then the print method prints the rule as follows:

$$n[\texttt{GRule} : \mathit{Label}]\ \{\mathit{premises}\}/\{\mathit{conclusions}\}$$

where n is the clause number, which will serve here as the rule number. The origin of the step is `GRule` followed by the label that was either generated by SPASS for the formula that generated the clause or added explicitly by the user. The premises and conclusions are separated by a slash (/). These are translated from the literals of the original clause. Each literal in the clause of the form $R_{r_i}(x, y)$ is printed as $(x, y) : r_i$, where r_i is the relational variable associated with the relation R_{r_i}.

Axiom	property clausal form	rule translation
D	$R_r(x, f_i(x))$	`. / (x,fi(x)):r`
T	$R_r(x, x)$	`. / (x,x):r`
B	$\neg R_r(x, y) \vee R_r(y, x)$	`(x,y):r / (y,x):r`
4	$\neg R_r(x, y) \vee \neg R_r(y, z) \vee R_r(x, z)$	`(x,y):r, (y,z):r / (x,z):r`
5	$\neg R_r(x, y) \vee \neg R_r(x, z) \vee R_r(y, z)$	`(x,y):r, (x,z):r / (y,z):r`

Figure 6.5: Properties clauses and rule translations

Figure 6.5 shows the printed translation of property rules for the basic axioms T, D, B, 4 and 5. Any other rule is printed out in a similar way.

For inference steps, when a derivation step is a result of applying a structural rule, then the origin of the rule is set to `Rule` followed by the rule number that was inserted in the rules table and the numbers of nodes that the rule was applied on. Since propagational rules are only used with other rules and never applied on its own, the derivation step will reference the rule number in the parents, but the origin of the derived step will depend on what was applied.

6.4 Making all rules structural

In order to maintain as much as possible the conventional methods of tableau, I came up with a solution that makes all property rules structural. This solution overcomes all the problems previously discussed because eliminating propagational rules means every derived step will be ground and no new rules will be derived.

Due to lack of time, I have not implemented this in the current extension of SPASS and will not discuss the solution in much detail. In general, the idea is to implement a few changes so that we go back to producing only range-restricted clauses as the case was prior to introducing relational frame conditions.

We know that the whole problem is caused by the two clauses

$$R_{r_i}(x, x) \qquad \text{and} \qquad R_{r_i}(x, f_i(x))$$

The first clause says that for any world x, this world is connected to itself via the accessibility relation R_{r_i}. The second clause says that for any world x, this world has a successor $f_i(x)$ via the accessibility relation R_{r_i}. So, in fact being a **world** is the precondition for applying the rules.

Adding $world(x)$ as a precondition gives the following clauses:

$$\neg world(x)^+ \vee R_{r_i}(x, x) \qquad \text{and} \qquad \neg world(x)^+ \vee R_{r_i}(x, f_i(x))$$

With this addition, whenever a new world is introduced, we have to define it as a world. This means that for seriality we also need to produce:

$$\neg world(x)^+ \vee world(f_i(x))$$

Of course this means that adding the seriality property makes the system undecidable except on unsatisfiable problems.

The initial world a also needs to be defined as a world, and the same goes for the new world introduced by the $(\neg\Box)$ rule. One way of producing these extra

clauses is by slightly changing the structural transformation so that it produces

$$\exists x.\, world(x) \wedge Q_{\varphi}(x) \wedge Def(\varphi)$$

This addition would define the initial world.

For $(\neg\Box)$, we give it the following definition:

$$\forall x.\, Q_{\neg\Box\varphi}(x) \rightarrow (\exists y.\, world(y) \wedge R_{r_i}(x,y) \wedge Q_{\neg\varphi}(y))$$

This would make the $\neg\Box$ produce three definitional clauses instead of two, one of which would define the new world.

Let us look at an example with these new clauses. Take for instance the formula we derived in tableau in Section 6.2: $[r][r]\neg p \wedge [r]p$ in KD4.

Input clauses	Derived clauses
1. [input] $Q_{\wedge}(a)$	12. [OHy:1,3] $Q_{\Box\Box\neg}(a)$
2. [input] $world(a)$	13. [OHy:1,4] $Q_{\Box}(a)$
3. [input] $\neg Q_{\wedge} \vee Q_{\Box\Box\neg}(x)$	14. [OHy:2,9] $R_r(a, f_1(a))$
4. [input] $\neg Q_{\wedge} \vee Q_{\Box}(x)$	15. [OHy:2,10] $world(f_1(a))$
5. [input] $\neg Q_{\Box\Box\neg}(x) \vee \neg R_r(x,y) \vee Q_{\Box\neg}(y)$	16. [OHy:8,13,14] $P(f_1(a))$
6. [input] $\neg Q_{\Box\neg}(x) \vee \neg R_r(x,y) \vee Q_{\neg}(y)$	17. [OHy:5,12,14] $Q_{\Box\neg}(f_1(a))$
7. [input] $\neg Q_{\neg}(x) \vee \neg P(x)$	18. [OHy:15,9] $R_r(f_1(a), f_1(f_1(a)))$
8. [input] $\neg Q_{\Box}(x) \vee \neg R_r(x,y) \vee P(y)$	19. [OHy:15,10] $world(f_1(f_1(a)))$
9. [input] $\neg world(x) \vee R_r(x, f_1(x))$	20. [OHy:6,17,18] $Q_{\neg}(f_1(f_1(a)))$
10.[input] $\neg world(x) \vee world(f_1(x))$	21. [OHy:11,14,18] $R_r(a, f_1(f_1(a)))$
11.[input] $\neg R_r(x,y) \vee \neg R_r(y,z) \vee R_r(x,z)$	22. [OHy:7,13,21] $P(f_1(f_1(a)))$
	23. [OHy:7,20,22]$\bot$

Figure 6.6: Derivation of $[r][r]\neg p \wedge [r]p$ in KD4 with defined worlds

Figure 6.6 shows the input and derived clauses for the formula with the new solution. Notice the additions on the input set of clauses containing world definitions. The initial world a is defined as a world in clause 2. Clauses 9 and 10 which represent seriality property associated with axiom D have a premise $world(x)$.

Chapter 7

Tableau simulation results

In this chapter, I show results of the implementation by running test cases. The examples are selected to show how the results follow expected behaviour from previous discussions.

7.1 Expected input

In order for the program to run and produce correct results, it is important for the input to follow the expectations of the implementation.

I have only discussed and implemented a resolution-tableau simulator for the dynamic modal logic $K_m(\wedge, \vee, \smile)$ and traditional modal logics, both of which support arbitrary relational frame conditions.

Relational frame conditions are added as first-order formulae in the axioms part of the input file. The axioms part should not bear any other kind of formulae.

In the conjectures part only propositional formulae are expected with relational operators currently limited to $\wedge$, $\vee$ and $\smile$. Including other relational operators may not always cause a warning or an error but the tableau simulation may not produce full derivation steps or correct justifications to the produced steps.

Appendix A shows an example of an input problem. Appendix B shows the full result from running the problem of the input in Appendix A. The problem is run with two new flag settings: PrfTr=1 and CNFRenaming=4. The first flag is

a new flag that I have added. The flag's name is short for *proof translation* and by turning it on, SPASS provides tableau translation. The other flag controls the renaming procedure. I have extended this flag to include a fourth option that applies modal logic renaming to first-order translated formulae. I have also included a method that turns on flag settings that are important to the simulation and switches off the ones that are not compatible when translation is requested by the user.

There is also an optional third flag setting: PIntSy=1. This flag is short for *print introduced symbols*. By turning this flag on, every introduced predicate by the renaming method is printed out along with the modal logic formula it is associated with. This symbol association table is shown at the beginning of the output result displayed in Appendix B. This table is helpful if the user would like to make sure that the associations are done correctly.

Following are a number of selected examples of output produced by the implementation.

7.2 Avoiding redundancy

Figure 7.1 shows a very small example, where the tableau derivation steps are fewer than what we would normally expect. The purpose of including this example is to raise the awareness about this unusual behaviour and to explain to the reader why it is justified.

```
---------------------------TABLEAU-TRANSLATION-----------------------------
Model found:
 1[Giv:C1] || (skc0):and(not(p),not(and(p,q)))
 2[And:1] || (skc0):not(and(p,q))
 3[And:1] || (skc0):not(p)
--------------------------------SPASS-STOP---------------------------------
```

Figure 7.1: Avoidance of redundancy

In the example, steps 2 and 3 result from applying the $(\wedge)$ rule on the given formula. Although the $(\neg\wedge)$ rule can be applied on node 2 and create a new branch, it is not and we find that the derivation stops declaring that the formula is satisfiable.

The explanation for this behaviour is given in [Sch08]. The explanation says that this is an enhancement to tableau calculus obtained from the use of resolution. In resolution, unnecessary duplication and superfluous inferences are avoided by subsumption checking. This means that if a previously derived clause subsumes a new one, then the inference step is avoided because it results in a redundant clause that does not affect the derivation. The cited paper defines redundancy in this context for tableau. The definition says that an application of a rule is redundant if it results in redundant conclusions.

For example, for any s, $s : \top$ is redundant. In the example of Figure 7.1, we have both $s : \neg p$ and $s : \neg(p \wedge q)$. Applying the $(\neg\wedge)$ rule would result in a redundant conclusion, which is $s : \neg p$. This is why the $(\neg\wedge)$ rule is not applied. No other inferences are possible and the derivation stops.

With this example, we can have a sense of how the simulated semantic tableau automatically and naturally benefits from first-order resolution and the effect this can have on performance results.

7.3 Testing renaming of negated formulae

We have previously seen the purpose of eliminating defined operators rather than transforming the formula to negation normal form. In general, assuming renaming is done correctly, if we have a negated labelled formula, then the derivation derives a contradiction if the labelled formula occurs positively on the same branch. If not, then either the $(\neg\wedge)$ rule or the $(\neg\Box)$ rule is applied producing the conclusion we expect by their definitions.

The example in Figure 7.2 is produced to test and evaluate the new method I introduced and implemented for renaming negated formulae.

The formula that we are testing for satisfiability is the following:

$$\neg\big([r_1]\neg([r_1](p \wedge q) \wedge \neg([r_1](p \wedge q))) \wedge \neg(\neg p \wedge p)\big)$$

Notice that the formula contains two occurrences of $[r_1](p \wedge q)$. When the tableau derivation steps are expanded, we will find that one of the occurrences

```
---------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
1[Giv:C1] || (skc0):not(
                        and(box(r1,not(
                                       and(box(r1,and(p,q)),
                                       not(box(r1,and(p,q)))))),
                        not(and(not(p),p))))
      2[NAnd:1] || (skc0):and(not(p),p)
      4[And:2] || (skc0):not(p)
      3[And:2] || (skc0):p
      5[Clash:4,3] || .
 6[NAnd:1,2] || (skc0):not(and(not(p),p))
 7[NAnd:1] || (skc0):not(box(r1,not(
                                    and(box(r1,and(p,q)),
                                        not(box(r1,and(p,q)))))))
 8[NBox:7] || (skf1 (skc0)):and(box(r1,and(p,q)),
                                not(box(r1,and(p,q))))
 9[NBox:7] || ((skc0),(skf1 (skc0))):r1
 11[And:8] || (skf1 (skc0)):box(r1,and(p,q))
 10[And:8] || (skf1 (skc0)):not(box(r1,and(p,q)))
 12[Clash:10,11] || .
--------------------------------SPASS-STOP------------------------------------
```

Figure 7.2: Testing negated formulae renaming - 1

will be positive and the other will be negative. We expect the derivation to detect that the two subformulae are actually two occurrences of the same subformula in order to derive a contradiction.

The formula that we are testing for satisfiability in the example is negated. If the subformula under the negation exists somewhere else positively, then the closure rule is applied. Since there is no such formula, then the tree is expanded by applying the $(\neg\wedge)$ rule.

A correct expansion indicates renaming has been done correctly. For the $(\neg\wedge)$ rule, we expect branching, where each branch is a complement of a subformula that is an argument of the negated $(\wedge)$. As we expect, the first branch starts at node 2. We find the formula in node 2 to be the complement of the right argument. This branch is soon closed with a clash between nodes 3 and 4.

The right branch is then expanded at node 6. Node 6 is the complement of node 2, which started the left branch. We now expect to find the complement of the left argument, which is what we find in node 7. Node 7 is a negated box formula. Since the box formula does not occur positively anywhere else in the branch, the $(\neg\Box)$ rule must be applied. Again, if renaming is done correctly, we expect two new nodes to result from the application of this rule. One that

carries the complement of the subformula under the box operator labelled with a new successor, and one that asserts the relation by the box operator between the current world and the new successor. We find that these are nodes 8 and 9.

Nodes 10 and 11 are the result of the application of the $(\wedge)$ rule. These are two non-literal formulae that are complements of each other. We find that the closure rule is applied and the derivation stops. This shows that renaming was applied correctly and also that a single predicate was introduced for the two occurrences of $[r_1](p \wedge q)$.

```
---------------------------TABLEAU-TRANSLATION----------------------------
Tableau proof:
1[Giv:C1] || (skc0):not(
                        and(box(r1,not(
                                       and(box(r1,and(p,q)),
                                       not(box(r1,and(q,p)))))),
                        not(and(not(p),p))))
      2[NAnd:1] || (skc0):and(not(p),p)
      4[And:2] || (skc0):not(p)
      3[And:2] || (skc0):p
      5[Clash:4,3] || .
 6[NAnd:1,2] || (skc0):not(and(not(p),p))
 7[NAnd:1] || (skc0):not(box(r1,not(
                                    and(box(r1,and(p,q)),
                                        not(box(r1,and(q,p)))))))
 8[NBox:7] || (skf2 (skc0)):and(box(r1,and(p,q)),
                                not(box(r1,and(q,p))))
 11[And:8] || (skf2 (skc0)):box(r1,and(p,q))
 10[And:8] || (skf2 (skc0)):not(box(r1,and(q,p)))
 13[NBox:10] || ((skf2 (skc0)),(skf0 (skf2 (skc0)))):r1
 12[NBox:10] || (skf0 (skf2 (skc0))):not(and(q,p))
 14[Box:11,13] || (skf0 (skf2 (skc0))):and(p,q)
 16[And:14] || (skf0 (skf2 (skc0))):p
 15[And:14] || (skf0 (skf2 (skc0))):q
      17[NAnd:12] || (skf0 (skf2 (skc0))):not(p)
      18[Clash:17,16] || .
 19[NAnd:12,17] || (skf0 (skf2 (skc0))):p
 20[NAnd:12] || (skf0 (skf2 (skc0))):not(q)
 22[Clash:20,15] || .
---------------------------------SPASS-STOP-----------------------------------
```

Figure 7.3: Testing negated formulae renaming - 2

Figure 7.3 shows another derivation for the same formula. The input formula is changed slightly to:

$$\neg\big([r_1]\neg([r_1](p \wedge q) \wedge \neg([r_1](q \wedge p))) \wedge \neg(\neg p \wedge p)\big)$$

Notice that the second occurrence of $[r_1](p \wedge q)$ now has p and q switched. By comparing with the previous derivation, we find that this derivation is significantly longer. This is because the prover failed to detect that $[r_1](p \wedge q)$ and $[r_1](q \wedge p)$ are actually equivalent. Thus, the derivation did not apply the closure rule when it should have and instead continued with branching and backtracking.

In order for the prover to be more efficient in detecting matching formulae, a simple preprocessing step can be implemented. This preprocessing step reorders the arguments of the $\wedge$ and *vee* operators according to an ordering parameter. If reordering is done before renaming is applied, then all equivalent formulae will be detectable as they will always have the same order.

7.4 Testing renaming of relational formulae

The implementation of relational formula renaming included several adjustments to produce correct renaming for relational formulae especially when the formula includes the converse operator. The purpose of the derivation illustrated in Figure 7.4 is to test if relational formula renaming after the adjustments produce the expected results.

Input formula:
$([r_1](p \wedge q)) \rightarrow ([r_2]\langle(r_1 \vee r_2)^\smile\rangle p)$

Output:

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Model found:
 1[Giv:C1] || (skc0):and(box(r1,and(p,q)),
                         not(box(r2,not(box(or(conv(r1),conv(r2)),not(p))))))
 2[And:1] || (skc0):box(r1,and(p,q))
 3[And:1] || (skc0):not(box(r2,not(box(or(conv(r1),conv(r2)),not(p)))))
 4[NBox:3] || (skf1 (skc0)):box(or(conv(r1),conv(r2)),not(p))
 5[NBox:3] || ((skc0),(skf1 (skc0))):r2
 6[ConvI:5] || ((skf1 (skc0)),(skc0)):conv(r2)
 8[OrRI:6] || ((skf1 (skc0)),(skc0)):or(conv(r1),conv(r2))
 9[Box:4,8] || (skc0):not(p)
---------------------------------SPASS-STOP------------------------------------
```

Figure 7.4: Testing relational formula renaming

In the example of Figure 7.4, we see the effect of relational formula renaming. First, notice that the converse operator that is applied on $(r_1 \vee r_2)$ in the input

problem got distributed over the relational variables in a preprocessing step. The result of this distribution is $(r_1^{\smile} \vee r_2^{\smile})$. This step is important because the converse does not have a matching operator in first-order logic, which caused a number of problems as previously demonstrated in Section 5.3.

The derivation starts by applying the $(\wedge)$ expansion rule on the first node. This results in nodes 2 and 3. Nodes 4 and 5 are the result of applying the $(\neg\Box)$ rule to node 3. We find that the new world $skf1(skc0)$ is connected to $skc0$ as its successor via the accessibility relation R_{r_2}. Before solving problems with the converse operator, step 5 would have produced the labelled formula $((skf1(skc0)), (skc0)) : conv(r_2)$. This is due to the previously discussed problem of identifying different occurrences of $conv(r_2)$ in the first-order translation and that both $conv(r_2)$ and r_2 are syntactically equivalent in first-order logic. This means that both formulae get substituted by the same introduced relational predicate, which should not happen. In the solution that I implemented, when two first-order formulae are found syntactically equivalent, a further check is done on the associated modal/relational formulae. This way, only subformulae that are syntactically equivalent in first-order logic and in modal logic get substituted with the same introduced predicate.

Now since we have a $r_2^{\smile}$ in the problem formula, we find that the introduction rule of converse is applied on $r2$ to produce $conv(r2)$. The same case with $\vee^r$, we have $(r_1^{\smile} \vee r_2^{\smile})$ so the $\vee$ introduction rule is applied on $conv(r_2)$ to produce $or(conv(r_1), conv(r_2))$. This relational formula matches the relational formula of the box operator in of node 4 and with the same world. Thus, the $(\Box)$ rule is applied on both nodes to produce a new formula. No more rules can be applied and no contradictions have been found so the derivation stops.

We find in this example that the derivation of a formula which includes relational formulae behaves as we expect. This is an indication that relational formula renaming introduces correct definitional clauses. We have also seen that previous problems that arose from having the converse operator have been solved satisfactorily.

7.5 Testing proof indentation

Figure 7.5 shows the output of the example used previously in Figure 5.10. The purpose of this test case is to check if the indentation of derivation steps is applied correctly. As a reminder, indentations of derivation steps serve the purpose of representing tree branches.

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):not(and(not(and(not(and(p,q,a)),p,q,a)),
                             not(and(d,not(d)))))
      2[NAnd:1] || (skc0):and(d,not(d))
      4[And:2] || (skc0):d
      3[And:2] || (skc0):not(d)
      5[Clash:3,4] || .
 6[NAnd:1,2] || (skc0):not(and(d,not(d)))
 7[NAnd:1] || (skc0):and(not(and(p,q,a)),p,q,a)
 11[And:7] || (skc0):p
 10[And:7] || (skc0):q
 9[And:7] || (skc0):a
 8[And:7] || (skc0):not(and(p,q,a))
           12[NAnd:8] || (skc0):not(a)
           13[Clash:12,9] || .
 14[NAnd:8,12] || (skc0):a
      15[NAnd:8] || (skc0):not(q)
      16[Clash:15,10] || .
 17[NAnd:8,15] || (skc0):q
 18[NAnd:8] || (skc0):not(p)
 20[Clash:18,11] || .
---------------------------------SPASS-STOP-----------------------------------
```

Figure 7.5: Testing proof indentation

The derivation starts by applying the $\neg(\wedge)$ rule which results in branching. Since there are only two arguments to the $\wedge$, the left branch at 2 is indented one unit. After the closure rule is applied at the end of the left branch at step 5, the derivation starts exploring the right branch. The right branch does not need indentation and this is why it appears on the same level as the root node.

Step 6 is the complement of the first split unit. Formulae that result from complementing a previous split unit have two parents in their justification and not just one. The first parent refers to the step that results in branching. The second parent refers to the formula that is being complemented.

More rules are applied to the right branch until another branching formula is reached in step 8. The $\wedge$ of this formula has got three arguments. This is why the left most branch of the three branches starts with two levels of indentation.

Step 14 is the complement of the first branch. Since the complemented formula affects all subsequent branches, it inherits the indentation level of the formula that result in branching.

The second branch starting at step 15 is indented with a single unit. Then after this branch is closed as well, the final branch is expanded with no indentation.

We find that all derivation steps were indented as expected. The only difference between the derivation shown here and the derivation in Figure 5.10 is that this derivation demonstrated semantic branching that results from SPASS's use of complement splitting.

7.6 Applying non-branching rules first

We have previously seen in Section 5.7 how the order in which derivation steps are produced can have an effect on the logical structure of a proof in indentation style.

Figures 7.6 and 7.7 in this section are used to illustrate two derivations for the same input problem. The first figure illustrates the default derivation, where the structure of the proof is not logical. The second figure shows the derivation after incorporating the solution. The purpose of this section is to compare the two derivations and see if the introduced solution fixed the problem.

In the derivation of Figure 7.6, nodes 3-8 result from applying the ($\wedge$) rule on the first node. Nodes 9 and 10 result from applying the ($\neg\Box$) on node 8, which creates a new successor to $skc0$. Node 13 results from applying the ($\Box$) rule on nodes 3 and 10.

After node 13, the ($\Box$) rule is still applicable on several other nodes, but so is the branching rule ($\neg\wedge$), which is applicable on node 13. The derivation chooses applying ($\neg\wedge$).

The left branch starts at node 15. Then nodes 16 and 17 result from applying the ($\Box$) rule on nodes 4 and 5 respectively. The derivation continues until this branch is closed by node 35.

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(box(r,not(and(not(a),not(b)))),
                          box(r,not(and(p,a))),
                          box(r,not(and(p,b))),
                          box(r,not(and(q,a))),
                          box(r,not(and(q,not(a)))),
                          box(r,not(and(q,p))),
                          not(box(r,and(not(p),not(q)))))
 3[And:1] || (skc0):box(r,not(and(q,not(a))))
 4[And:1] || (skc0):box(r,not(and(q,a)))
 5[And:1] || (skc0):box(r,not(and(p,b)))
 6[And:1] || (skc0):box(r,not(and(not(a),not(b))))
 7[And:1] || (skc0):box(r,not(and(p,a)))
 8[And:1] || (skc0):not(box(r,and(not(p),not(q))))
 9[NBox:8] || (skf0 (skc0)):not(and(not(p),not(q)))
 10[NBox:8] || ((skc0),(skf0 (skc0))):r
 13[Box:3,10] || (skf0 (skc0)):not(and(q,not(a)))
      15[NAnd:13] || (skf0 (skc0)):a
      16[Box:4,10] || (skf0 (skc0)):not(and(q,a))
      17[Box:5,10] || (skf0 (skc0)):not(and(p,b))
           18[NAnd:16] || (skf0 (skc0)):not(a)
           19[Clash:18,15] || .
      20[NAnd:16,18] || (skf0 (skc0)):a
      21[NAnd:16] || (skf0 (skc0)):not(q)
      23[Box:6,10] || (skf0 (skc0)):not(and(not(a),not(b)))
           24[NAnd:9] || (skf0 (skc0)):q
           25[Clash:21,24] || .
      26[NAnd:9,24] || (skf0 (skc0)):not(q)
      27[NAnd:9] || (skf0 (skc0)):p
      29[Box:7,10] || (skf0 (skc0)):not(and(p,a))
           30[NAnd:29] || (skf0 (skc0)):not(a)
           32[Clash:30,15] || .
      33[NAnd:29,30] || (skf0 (skc0)):a
      34[NAnd:29] || (skf0 (skc0)):not(p)
      35[Clash:34,27] || .
 36[NAnd:13,15] || (skf0 (skc0)):not(a)
 37[NAnd:13] || (skf0 (skc0)):not(q)
      40[NAnd:9] || (skf0 (skc0)):q
      41[Clash:37,40] || .
 42[NAnd:9,40] || (skf0 (skc0)):not(q)
 43[NAnd:9] || (skf0 (skc0)):p
      44[NAnd:23] || (skf0 (skc0)):b
           46[NAnd:17] || (skf0 (skc0)):not(b)
           47[Clash:46,44] || .
      48[NAnd:17,46] || (skf0 (skc0)):b
      49[NAnd:17] || (skf0 (skc0)):not(p)
      50[Clash:49,43] || .
 51[NAnd:23,44] || (skf0 (skc0)):not(b)
 52[NAnd:23] || (skf0 (skc0)):a
 54[Clash:36,52] || .
--------------------------------SPASS-STOP------------------------------------
```

Figure 7.6: Testing the logical order of derivation steps - 1

Node 36 begins the right branch. After several derivation steps, we find that node 46 in one of the sub-branches of the right branch is the result of applying the ($\neg\wedge$) rule on node 17, which according to the indentation style belongs to the left branch only and should not be referenced by other branches. In reality, node 17 belongs to the main branch and not just the left most branch.

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(box(r,not(and(not(a),not(b)))),
                         box(r,not(and(p,a))),
                         box(r,not(and(p,b))),
                         box(r,not(and(q,a))),
                         box(r,not(and(q,not(a)))),
                         box(r,not(and(q,p))),
                         not(box(r,and(not(p),not(q)))))
 3[And:1] || (skc0):box(r,not(and(q,not(a))))
 4[And:1] || (skc0):box(r,not(and(q,a)))
 5[And:1] || (skc0):box(r,not(and(p,b)))
 6[And:1] || (skc0):box(r,not(and(p,a)))
 7[And:1] || (skc0):box(r,not(and(not(a),not(b))))
 8[And:1] || (skc0):not(box(r,and(not(p),not(q))))
 9[NBox:8] || (skf0 (skc0)):not(and(not(p),not(q)))
 10[NBox:8] || ((skc0),(skf0 (skc0))):r
 12[Box:3,10] || (skf0 (skc0)):not(and(q,not(a)))
 13[Box:4,10] || (skf0 (skc0)):not(and(q,a))
 14[Box:5,10] || (skf0 (skc0)):not(and(p,b))
 15[Box:6,10] || (skf0 (skc0)):not(and(p,a))
 16[Box:7,10] || (skf0 (skc0)):not(and(not(a),not(b)))
      17[NAnd:14] || (skf0 (skc0)):not(b)
           18[NAnd:16] || (skf0 (skc0)):b
           19[Clash:17,18] || .
      20[NAnd:16,18] || (skf0 (skc0)):not(b)
      21[NAnd:16] || (skf0 (skc0)):a
           24[NAnd:13] || (skf0 (skc0)):not(a)
           25[Clash:24,21] || .
      26[NAnd:13,24] || (skf0 (skc0)):a
      27[NAnd:13] || (skf0 (skc0)):not(q)
           28[NAnd:9] || (skf0 (skc0)):q
           29[Clash:27,28] || .
      30[NAnd:9,28] || (skf0 (skc0)):not(q)
      31[NAnd:9] || (skf0 (skc0)):p
           32[NAnd:15] || (skf0 (skc0)):not(a)
           34[Clash:32,21] || .
      35[NAnd:15,32] || (skf0 (skc0)):a
      36[NAnd:15] || (skf0 (skc0)):not(p)
      37[Clash:36,31] || .
 38[NAnd:14,17] || (skf0 (skc0)):b
 39[NAnd:14] || (skf0 (skc0)):not(p)
      41[NAnd:13] || (skf0 (skc0)):not(a)
           42[NAnd:12] || (skf0 (skc0)):a
           43[Clash:41,42] || .
      44[NAnd:12,42] || (skf0 (skc0)):not(a)
      45[NAnd:12] || (skf0 (skc0)):not(q)
           47[NAnd:9] || (skf0 (skc0)):q
           48[Clash:45,47] || .
      49[NAnd:9,47] || (skf0 (skc0)):not(q)
      50[NAnd:9] || (skf0 (skc0)):p
      51[Clash:39,50] || .
 52[NAnd:13,41] || (skf0 (skc0)):a
 53[NAnd:13] || (skf0 (skc0)):not(q)
      55[NAnd:9] || (skf0 (skc0)):q
      56[Clash:53,55] || .
 57[NAnd:9,55] || (skf0 (skc0)):not(q)
 58[NAnd:9] || (skf0 (skc0)):p
 60[Clash:39,58] || .
---------------------------------SPASS-STOP------------------------------------
```

Figure 7.7: Testing the logical order of derivation steps - 2

The solution I introduced is to include a heuristic in the main search loop of SPASS. The goal of this heuristic is to force all non-branching rules to be applied before the application of any branching rule. For the dynamic modal logic calculus under study, the two branching rules are $(\neg\wedge)$ and $(\vee)^r$.

The example illustrated in Figure 7.7 shows the derivation with the heuristic of delaying application of branching rules being applied. We notice that indeed all non-branching rules are applied before branching ones.

All node references in the justification of each step are to nodes which are on the same branch. The exceptions are nodes introduced by complement splitting. Take as an example node 20. The justification of node 20 says [NAnd: 16, 18]. When a node is introduced by complement splitting, it refers to both the parent node as well as the previous branch that it complements. In this node it refers to node 18 which is the beginning of the first branch.

7.7 Intelligent backtracking in SPASS

In tableau, we have several kinds of backtracking. The naive form is chronological backtracking, which explores every possible branch regardless of whether the formulae contribute to the contradiction or not. More intelligent forms of backtracking disregards unnecessary branch explorations and jumps back to the last branching point of the search tree which contributed to the contradiction on the current branch [HS98b].

In the most recent version of SPASS, a new form of intelligent backtracking is implemented and which is based on labelled splitting [FW08].

The purpose of this section is to show an example where this new form of backtracking occurs. Figure 7.8 illustrates an abstract form of the derivation showing the backtracking case. The tree structure in the figure shows the derivation where several branches are explored and found closed. The important part of the figure is the node numbered (1) where the two dotted arrows originate. This node results from applying the closure rule on p and $\neg p$. Since the branch is closed, backtracking is performed. In this particular case, whether using chronological backtracking or more intelligent forms of backtracking, we would expect

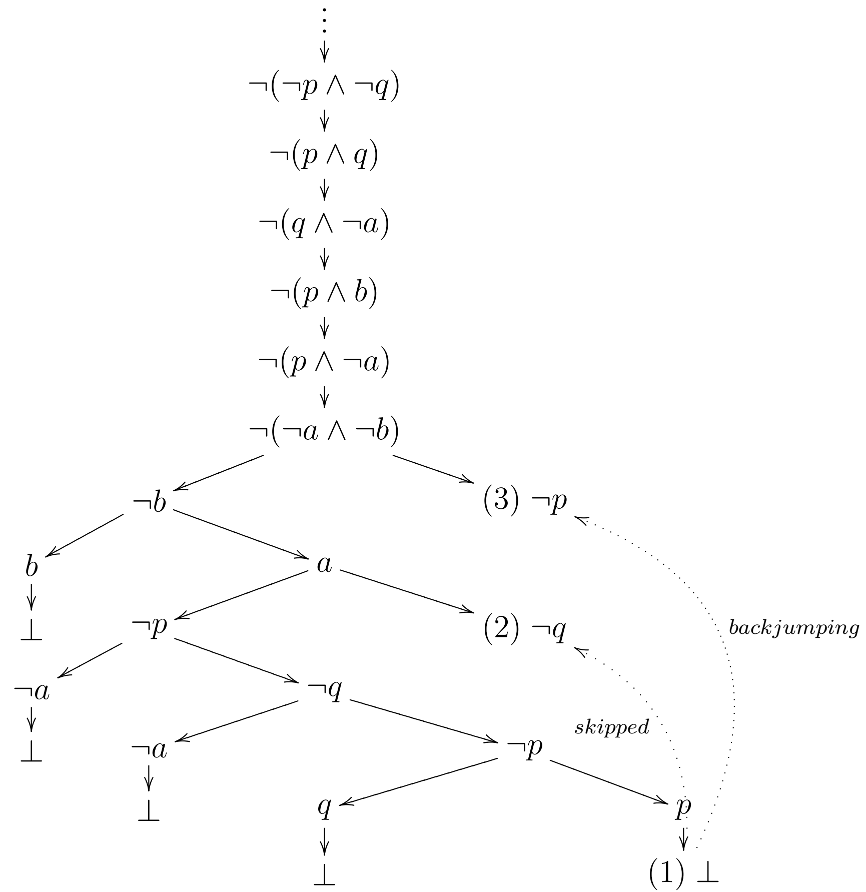


Figure 7.8: Intelligent backtracking

to backtrack to the node numbered (2). The figure explains that backtracking to node (2) is skipped and instead the derivation jumps to node (3) and continue the derivation from there.

This kind of intelligent backtracking has not been previously explained for semantic tableau. To give a justification in tableau for why backtracking performs in this way requires deep understanding of the implementation of labelled splitting in SPASS and the new backtracking rules which included *branch condense* and *right collapse*. Unfortunately and due to lack of time, I do not cover this as it fell out of the scope of this thesis.

7.8 Testing frame conditions on traditional modal logics

Input problem:
Axiom: $\forall x, y.R_{r_1}(x, y) \rightarrow R_{r_1}(y, x)$
Conjecture: $p \rightarrow [r_1]\langle r_1\rangle p$

Tableau output:

```
---------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(p,not(box(r1,not(box(r1,not(p))))))
 2[GRule:Symmetry] || (U,V):r1 / (V,U):r1
 3[And:1] || (skc0):p
 4[And:1] || (skc0):not(box(r1,not(box(r1,not(p)))))
 5[NBox:4] || (skf1 (skc0)):box(r1,not(p))
 6[NBox:4] || ((skc0),(skf1 (skc0))):r1
 7[Rule:2.0,6.0] || ((skf1 (skc0)),(skc0)):r1
 9[Box:5,7] || (skc0):not(p)
 10[Clash:9,3] || .
---------------------------------SPASS-STOP-----------------------------------
```

Figure 7.9: Symmetry and axiom B

Figure 7.9 is a validity test for axiom B in KB. The associated relational property for axiom B is symmetry. The first-order formula representing the symmetric property over the relation R_{r_1} is added in the axiom part of the input problem. Node 2 is the result of translating the clausal form of the symmetric property. The justification part informs that node 2 is a given rule and is given the label (Symmetry), which was specified by the user in the input file.

The tree expansion starts by an application of the ($\wedge$) rule on the formula of the first node, which results in nodes 3 and 4. The ($\neg\Box$) rule is applied on node 4. This results in a new successor $skf1(skc0)$ that labels the formula $[r_1]\neg p$. Node 6 asserts the relation between the world $skc0$ and its successor $skf1(skc0)$.

Node 7 results from applying rule 2 to node 6. Notice the nodes references in the justification part. Each node reference is a set of two digits separated by a dot. The first digit 2 refers to the given rule in node 2. The digit 0 after the dot refers to the part of the rule that is being resolved.

The result of applying the symmetric property can now be used with the box rule on node 5, which gives node 9. Node 9 clashes with node 3, which causes

the application of the closure rule and $\perp$ is derived. Since the conjecture is unsatisfiable, then the axiom is proved to be valid under the local frame set, which matches our expectations.

In a similar fashion, Figures 7.10, 7.11, 7.12 and 7.13 show the validity proofs of axioms T, D, 4 and 5 in KT, KD, K4 and K5 respectively.

Input problem:
Axiom: $\forall x.R_{r_1}(x, x)$
Conjecture: $p \rightarrow [r_1]p$

Tableau output:

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(box(r1,p),not(p))
 2[GRule:reflexivity] ||  / (U,U):r1
 4[And:1] || (skc0):not(p)
 5[And:1] || (skc0):box(r1,p)
 7[Box:5,2] || (skc0):p
 8[Clash:4,7] || .
---------------------------------SPASS-STOP-------------------------------------
```

Figure 7.10: Reflexivity and axiom T

Input problem:
Axiom: $\forall x, \exists y.R_{r_1}(x, y)$
Conjecture: $[r_1]p \rightarrow \langle r_1 \rangle p$

Tableau output:

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(box(r1,p),box(r1,not(p)))
 2[GRule:seriality] ||  / (U,(skf0 U)):r1
 3[And:1] || (skc0):box(r1,not(p))
 4[And:1] || (skc0):box(r1,p)
 5[Box:3,2] || (skf0 (skc0)):not(p)
 6[Box:4,2] || (skf0 (skc0)):p
 7[Clash:5,6] || .
---------------------------------SPASS-STOP-------------------------------------
```

Figure 7.11: Seriality and axiom D

Input problem:
Axiom: $\forall x, y, z.(R_{r_1}(x, y) \wedge R_{r_1}(y, z)) \rightarrow R_{r_1}(x, z)$
Conjecture: $[r_1]p \rightarrow [r_1][r_1]p$

Tableau output:

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(box(r1,p),not(box(r1,box(r1,p))))
 2[GRule:transitivity] || (U,V):r1 , (V,W):r1 / (U,W):r1
 3[And:1] || (skc0):not(box(r1,box(r1,p)))
 4[And:1] || (skc0):box(r1,p)
 5[NBox:3] || (skf1 (skc0)):not(box(r1,p))
 6[NBox:5] || (skf0 (skf1 (skc0))):not(p)
 7[NBox:5] || ((skf1 (skc0)),(skf0 (skf1 (skc0)))):r1
 8[NBox:3] || ((skc0),(skf1 (skc0))):r1
 10[Rule:2.0,8.0,2.1,7.0] || ((skc0),(skf0 (skf1 (skc0)))):r1
 11[Box:4,10] || (skf0 (skf1 (skc0))):p
 12[Clash:6,11] || .
---------------------------------SPASS-STOP----------------------------------
```

Figure 7.12: Transitivity and axiom 4

Input problem:
Axiom: $\forall x, y, z.(R_{r_1}(x, y) \wedge R_{r_1}(x, z)) \rightarrow R_{r_1}(y, z)$
Conjecture: $\langle r_1 \rangle \rightarrow [r_1]\langle r_1 \rangle p$

Tableau output:

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Tableau proof:
 1[Giv:C1] || (skc0):and(not(box(r1,not(p))),not(box(r1,not(box(r1,not(p))))))
 2[GRule:euclideanness] || (U,V):r1 , (U,W):r1 / (V,W):r1
 3[And:1] || (skc0):not(box(r1,not(box(r1,not(p)))))
 4[And:1] || (skc0):not(box(r1,not(p)))
 5[NBox:4] || (skf1 (skc0)):p
 6[NBox:3] || (skf0 (skc0)):box(r1,not(p))
 7[NBox:3] || ((skc0),(skf0 (skc0))):r1
 8[NBox:4] || ((skc0),(skf1 (skc0))):r1
 11[Rule:2.0,7.0,2.1,8.0] || ((skf0 (skc0)),(skf1 (skc0))):r1
 16[Box:6,11] || (skf1 (skc0)):not(p)
 18[Clash:16,5] || .
---------------------------------SPASS-STOP----------------------------------
```

Figure 7.13: Euclideanness and axiom 5

7.9 Relational frame conditions in dynamic modal logic

Input problem:
Axioms: $\forall x.\exists y.R_{r_1}(x,y)$
$\forall x,y,z.(R_{r1}(x,y) \land R_{r_1}(y,z)) \rightarrow R_{r_2}(x,z)$
Conjecture: $\neg((\neg(\neg q \land [r_1]p)) \land [r_1 \lor r_2]\neg p)$

Tableau output:

```
----------------------------TABLEAU-TRANSLATION-----------------------------
Model found:
 1[Giv:C1] || (skc0):and(not(and(not(q),box(r1,p))),box(or(r1,r2),not(p)))
 2[GRule:seriality] ||  / (U,(skf1 U)):r1
 3[GRule:axiom1] || (U,V):r1 , (V,W):r1 / (U,W):r2
 4[And:1] || (skc0):box(or(r1,r2),not(p))
 5[And:1] || (skc0):not(and(not(q),box(r1,p)))
 6[OrRI:2] || / (U,(skf1 U)):or(r1,r2)
 7[Box:4,6] || (skf1 (skc0)):not(p)
 8[Rule:3.0,2.0,3.1,2.0] || / (U,(skf1 (skf1 U))):r2
 9[OrRI:8] || / (U,(skf1 (skf1 U))):or(r1,r2)
 10[Box:4,9] || (skf1 (skf1 (skc0))):not(p)
 11[NAnd:5] || (skc0):not(box(r1,p))
 12[NBox:11] || (skf0 (skc0)):not(p)
 13[NBox:11] || ((skc0),(skf0 (skc0))):r1
 14[OrRI:13] || ((skc0),(skf0 (skc0))):or(r1,r2)
 15[Rule:3.0,13.0,3.1,2.0] || ((skc0),(skf1 (skf0 (skc0)))):r2
 17[OrRI:15] || ((skc0),(skf1 (skf0 (skc0)))):or(r1,r2)
 18[Box:4,17] || (skf1 (skf0 (skc0))):not(p)
 --------------------------------SPASS-STOP-----------------------------------
```

Figure 7.14: Relational frame conditions on dynamic logic

The example in Figure 7.14 serves the purpose of demonstrating how relational frame conditions work with dynamic modal logics.

The input problem contains three formulae. The first two formulae are first-order formulae introduced as relational frame conditions in the axioms part of the file. The third is a dynamic modal logic formula.

Nodes 4 and 5 are the result of applying the $(\land)$ rule to node 1. The formula that results in 4 is $[r_1 \lor r_2]\neg p$. We know from the first axiom that R_{r_1} is serial, which means that box rule can be applied to the formula in 4. However, since the relational formula $r_1 \lor r_2$ of the box operator is replaced with a new predicate by renaming, we need to derive it first.

Node 6 applies the introduction rule of the relational $\vee$ on the given rule in 2. The result contains a variable because it serves as a new rule. The new rule says that either R_{r_1} or R_{r_2} is serial, which is safe to derive. With this new conclusion, the box rule can now be applied on 4, which gives the result found in 7.

The new rule produced in node 6 can also be derived in tableau if we define a property rule for $(\vee)^r_I$ using variables and applied the concept of *unification*:

$$(D)\ \frac{\cdot}{\texttt{(U,(skf1 U))} : r_1} \qquad\qquad (\vee)^r_I\ \frac{\texttt{(U,V)} : r_1}{\texttt{(U,V)} : r_1 \vee r_2}$$

By applying the unification $\sigma = \{\texttt{(skf1 U)/V}\}$, then we can use the conclusion of (D) as the premise in $(\vee)^r_I$. The conclusion would be $\texttt{(U,(skf1 U))} : r_1 \vee r_2$, which is the same conclusion we derived in node 6.

Node 8 results from resolving the two given rules. The second given rule says that for any three consecutive worlds connected with the relation R_{r_1}, then the first world is connected to the third world via R_{r_2}. Since it is given that all worlds have a successor via the R_{r1} accessibility relation, then it makes sense to conclude the result in 8. The rule can also be produced in tableau by using the same idea described for obtaining node 6 by using unification.

The rest of derivation steps continue with similar explanations until no more rules can be produced or applied.

Chapter 8

Conclusion

The objective of this thesis was to extend the first-order resolution prover SPASS so that it appears to users as a modal logic tableau prover.

The implementation had two parts to get to the desired results. The first part was to get SPASS to produce the exact set of derivation steps that correspond to tableau derivation steps. The second part was to perform back-translation of the derived steps that are based on first-order logic to modal logic formulae and give tableau based justifications of each step.

Getting SPASS to perform as we needed required extending previously implemented modules, which included the module responsible for translating modal logic formulae to first-order logic, the renaming module, and the top module performing the main search loop. Any addition or change to the code was introduced carefully as to not disturb existing functionalities of SPASS in any way.

The translation of steps to modal tableau was implemented in a new separate module. This separation offers better control and maintainability in the new code.

The results shown in the previous chapter demonstrate that the objectives of this thesis have been met. The implementation produces tableau proofs when the negation of the input problem is unsatisfiable and models when it is satisfiable by finding an open complete branch. The proofs are printed in linear form with the tree structure being reflected by appropriate indentation. The tests show that

indentations represent branches correctly. The tests also show that the provided justifications are sensible, reflecting the application of tableau rules.

The main contributions of this thesis are:

- Implemented a functional extension of SPASS v3.5 that performs as a tableau simulator via first-order resolution for dynamic modal logic with relational operators closed under intersection, disjunction and converse with the support of including relational frame conditions. The implementation included recommended simplification and normalization techniques that help in detecting contradictions faster, which results in better performance.
- Introduced a new mechanism for renaming negated formulae using standard structural transformation. The mechanism relies on a preprocessing step prior to formula renaming. When standard structural transformation is applied on the transformed formula, all necessary definitional clauses are produced. This new method not only utilized the standard renaming techniques, but also produced fewer definitional clauses than the modal logic structural transformation suggested by the literature.
- Redefined the tableau calculus for traditional and dynamic modal logics to reflect what happens in the first-order resolution simulation when relational frame conditions are imposed. My implementation of handling imposed conditions is not only new but is also much more flexible than what is described in the literature because it allows introducing generic rules.
- Since the way I redefined tableau is so unconventional in that the derivation included not just formulae but also rules defined over variables, I proposed a second solution where worlds are defined. This second solution is also new but is much closer to conventional tableau methods.

The following points can be taken into consideration as further work and enhancement opportunities for this thesis:

- Investigate labelled splitting in SPASS, which results in a new intelligent form of backtracking.
- Further investigation of the tableau derivations involving formulae with variables in the labels and which represent derived rules.

- Further investigation and implementation of the idea of defining worlds, which makes all rules resulting from introducing frame conditions structural and thus prevents deriving formulae with variables in their labels.
- Further extend the current implementation to support a wider range of logics.
- Provide the tableau derivation in first-order logic as a second option.
- Compare the performance of the simulator to the performance of conventional tableau provers.
- Investigate methods and techniques that enhance the performance of the current implementation.

Appendix A

Full input example

```
begin_problem(FullInputExample).

list_of_descriptions.
name({*Input Example*}).
author({*Rawan AlBarakati*}).
status(unknown).
description({*Input Example*}).
end_of_list.

list_of_symbols.
predicates[(r1,0),(r2,0), (p,0), (q,0), (Rr1,2),(Rr2,2)].
translpairs[(r1,Rr1), (r2,Rr2), (p,P), (q,Q)].
end_of_list.

list_of_special_formulae(axioms,EML).
formula(forall([x],exists([y],Rr1(x,y))),seriality).
formula(forall([x,y,z],implies(and(Rr1(x,y),Rr1(y,z)),Rr2(x,z)))).
end_of_list.

list_of_special_formulae(conjectures,EML).
prop_formula(not(and(not(and(not(q),box(r1,p))) , box(or(r1,r2),not(p)))),C1).
end_of_list.

end_problem.
```

Appendix B

Full output example

```
Introduced symbols:
SkP0 : (or (r1) (r2))
SkP1 : (box (or (r1) (r2)) (not (p)))
SkP2 : (not (p))
SkP3 : (not (box (r1) (p)))
SkP4 : (box (r1) (p))
SkP5 : (not (q))
SkP6 : (and (not (q)) (box (r1) (p)))
SkP7 : (not (and (not (q)) (box (r1) (p))))
SkP8 : (and (not (and (not (q)) (box (r1) (p)))) (box (or (r1) (r2)) (not (p))))
--------------------------SPASS-START-----------------------------
Input Problem:
1[0:Inp] ||  -> SkP8(skc0)*.
2[0:Inp] ||  -> Rr1(U,skf1(U))*.
3[0:Inp] || SkP8(U) -> SkP7(U)*.
4[0:Inp] || SkP8(U) -> SkP1(U)*.
5[0:Inp] || SkP6(U) -> SkP5(U)*.
6[0:Inp] || SkP6(U) -> SkP4(U)*.
7[0:Inp] || SkP6(U)* SkP7(U) -> .
8[0:Inp] || Q(U)* SkP5(U) -> .
9[0:Inp] || SkP4(U) SkP3(U)* -> .
10[0:Inp] || SkP3(U) -> SkP2(skf0(U))*.
11[0:Inp] || P(U)* SkP2(U) -> .
12[0:Inp] || SkP7(U) -> SkP3(U) Q(U)*.
13[0:Inp] || SkP3(U) -> Rr1(U,skf0(U))*.
14[0:Inp] || Rr1(U,V)* -> SkP0(U,V).
15[0:Inp] || Rr2(U,V)* -> SkP0(U,V).
16[0:Inp] || Rr1(U,V)* SkP4(U) -> P(V).
17[0:Inp] || SkP0(U,V)* SkP1(U) -> SkP2(V).
18[0:Inp] || SkP0(U,V) -> Rr2(U,V) Rr1(U,V)*.
19[0:Inp] || Rr1(U,V)* Rr1(U1,U)* -> Rr2(U1,V)*.
 This is a first-order Non-Horn problem without equality.
 This is a problem that contains sort information.
 The following monadic predicates have finite extensions: SkP8.
 Axiom clauses: 2 Conjecture clauses: 17
 Inferences: IOHy=1 IOFc=1
 Reductions: RFSub=1 RBSub=1
```

```
 Extras    : No Input Saturation, Always Selection, Full Splitting, Lazy
Reduction,  Ratio: 5, FuncWeight: 1, VarWeight: 1
 Precedence: nequal > div > id > aa > W > r > r1 > r2 > s > t > p > q > w > v >
a > b > c > d > Rr1 > Rr2 > P > Q > SkP0 > SkP1 > SkP2 > SkP3 > SkP4 > SkP5 >
SkP6 > SkP7 > SkP8 > skc0 > skf0 > skf1
 Ordering  : KBO
Processed Problem:

Worked Off Clauses:

Usable Clauses:
1[0:Inp] ||  -> SkP8(skc0)*.
2[0:Inp] ||  -> Rr1(U,skf1(U))*.
5[0:Inp] || SkP6(U) -> SkP5(U)*.
4[0:Inp] || SkP8(U) -> SkP1(U)*.
6[0:Inp] || SkP6(U) -> SkP4(U)*.
3[0:Inp] || SkP8(U) -> SkP7(U)*.
10[0:Inp] || SkP3(U) -> SkP2(skf0(U))*.
8[0:Inp] || SkP5(U) Q(U)* -> .
11[0:Inp] || SkP2(U) P(U)* -> .
7[0:Inp] || SkP7(U) SkP6(U)* -> .
9[0:Inp] || SkP3(U)* SkP4(U) -> .
13[0:Inp] || SkP3(U) -> Rr1(U,skf0(U))*.
12[0:Inp] || SkP7(U) -> Q(U)* SkP3(U).
15[0:Inp] || Rr2(U,V)* -> SkP0(U,V).
14[0:Inp] || Rr1(U,V)* -> SkP0(U,V).
16[0:Inp] || SkP4(U) Rr1(U,V)* -> P(V).
17[0:Inp] || SkP1(U) SkP0(U,V)* -> SkP2(V).
18[0:Inp] || SkP0(U,V) -> Rr1(U,V)* Rr2(U,V).
19[0:Inp] || Rr1(U,V)* Rr1(V,U1)* -> Rr2(U,U1)*.

SPASS V 3.5+
SPASS beiseite: Completion found.
Problem: inputexample.dfg
SPASS derived 20 clauses, backtracked 0 clauses, performed 1 splits and kept 34
clauses.
SPASS allocated 28096 KBytes.
SPASS spent     0:07:39.04 on the problem.
                0:01:46.42 for the input.
                0:01:36.72 for the FLOTTER CNF translation, of which
                0:00:00.12 for the translation from EML to FOL.
                0:00:00.56 for inferences.
                0:00:00.00 for the backtracking.
                0:00:01.37 for the reduction.

---------------------------TABLEAU-TRANSLATION----------------------------
Model found:
 1[Giv:C1] || (skc0):and(not(and(not(q),box(r1,p))),box(or(r1,r2),not(p)))
 2[GRule:seriality] ||  / (U,(skf1 U)):r1
 4[And:1] || (skc0):box(or(r1,r2),not(p))
 5[And:1] || (skc0):not(and(not(q),box(r1,p)))
 6[NAnd:5] || (skc0):not(box(r1,p))
 7[NBox:6] || (skf0 (skc0)):not(p)
 8[NBox:6] || ((skc0),(skf0 (skc0))):r1
```

```
10[OrRI:8] || ((skc0),(skf0 (skc0))):or(r1,r2)
9[OrRI:2] || / (U,(skf1 U)):or(r1,r2)
11[Box:4,9] || (skf1 (skc0)):not(p)
3[GRule:axiom1] || (U,V):r1 , (V,U1):r1 / (U,U1):r2
14[Rule:3.0,8.0,3.1,2.0] || ((skc0),(skf1 (skf0 (skc0)))):r2
15[OrRI:14] || ((skc0),(skf1 (skf0 (skc0)))):or(r1,r2)
13[Rule:3.0,2.0,3.1,2.0] || / (U,(skf1 (skf1 U))):r2
16[Box:4,15] || (skf1 (skf0 (skc0))):not(p)
17[OrRI:13] || / (U,(skf1 (skf1 U))):or(r1,r2)
18[Box:4,17] || (skf1 (skf1 (skc0))):not(p)
---------------------------------SPASS-STOP---------------------------------
```

Bibliography

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001.

[CFdCGHg97] M. A. Castilho, L. Fariñas del Cerro, O. Gasquet, and A. Herzi g. Modal tableaux with propagation rules and structural rules. *Fundamenta Informaticae*, 3–4(32):281–297, 1997.

[CWT07] Renate A. Schmidt Christoph Weidenbach and Dali Topic. *SPASS input syntax version 3.0. Contained in the documentation of SPASS Version 3.0.* Max-Planck-Institut fur Informatik, Stuhlsatzenhausweg 85 66123 Saarbrucken, 2007.

[Fit90] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[FW08] Arnaud Fietzke and Christoph Weidenbach. Labelled splitting. In *IJCAR*, pages 459–474, 2008.

[GHS03] Lilia Georgieva, Ullrich Hustadt, and Renate A. Schmidt. Hyperresolution for guarded formulae. *J. Symb. Comput.*, 36(1-2):163–192, 2003.

[HdNS00] Ullrich Hustadt, Hans de Nivelle, and Renate A. Schmidt. Resolution-based methods for modal logics. *Logic Journal of the IGPL*, 8(3):265–292, 2000.

[HHSS07] I. Horrocks, U. Hustadt, U. Sattler, and R. A. Schmidt. Computational modal logic. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 181–245. Elsevier, Amsterdam, 2007. Commissioned overview paper.

[Hor97] Ian R. Horrocks. *Optimising tableaux decision procedures for description logic.* PhD thesis, The University of Manchester, 1997.

[Hor98] Ian Horrocks. The fact system. In *TABLEAUX*, pages 307–312, 1998.

[HS98a] Ullrich Hustadt and Renate A. Schmidt. Issues of decidability for description logics in the framework of resolution. In *FTP (LNCS Selection)*, pages 191–205, 1998.

[HS98b] Ullrich Hustadt and Renate A. Schmidt. Simplification and backjumping in modal tableau. In *TABLEAUX*, pages 187–201, 1998.

[HS99] Ullrich Hustadt and Renate A. Schmidt. On the relation of resolution and tableaux proof systems for description logics. In *IJCAI*, pages 110–117, 1999.

[HS00] Ullrich Hustadt and Renate A. Schmidt. Mspass: Modal reasoning by translation and first-order resolution. In *TABLEAUX*, pages 67–71, 2000.

[HS02] Ullrich Hustadt and Renate A. Schmidt. Using resolution for testing modal satisfiability and building models. *J. Autom. Reasoning*, 28(2):205–232, 2002.

[Kel97] John J. Kelly. *The Essence of Logic.* Prentice Hall, first edition edition, 1997.

[Kri63] Saul Kripke. Semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift fr Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Li08] Zhen Li. *Efficient and generic reasoning for modal logics.* PhD thesis, The University of Manchester, UK, 2008.

[Mas98] Fabio Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In *TABLEAUX*, pages 217–231, 1998.

[Mas00] Fabio Massacci. Single step tableaux for modal logics. *J. Autom. Reasoning*, 24(3):319–364, 2000.

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 6, pages 335 – 367. Elsevier, Amsterdam, Netherlands, 2001.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of vampire. AI Communications. 15(2-3):91-110, 2002.

[Sch99] Renate A. Schmidt. MSPASS. http://www.cs.man.ac.uk/ schmidt/mspass/, 1999.

[Sch02] S. Schulz. E: A brainiac theorem prover. AI Communications. 15(2-3):111-126, 2002.

[Sch06] Renate A. Schmidt. Developing modal tableaux and resolution methods via first-order resolution. In *Advances in Modal Logic*, pages 1–26, 2006.

[Sch07] Renate A. Schmidt. pdl-tableau. Available at: http://www.cs.man.ac.uk/ schmidt/pdl-tableau/, 2003, February 2007.

[Sch08] R. A. Schmidt. A new methodology for developing deduction methods. *Annals of Mathematics and Artificial Intelligence*, 2008.

[Sch09a] Renate Schmidt. Comp60121 lecture notes: Automated reasoning, part ii advanced topics. University of Manchester, 2008-2009. Available at: http://www.cs.man.ac.uk/ schmidt/COMP60121/2008-2009/PartIIWeek3.pdf.

[Sch09b] Renate Schmidt. Comp6016 lecture notes: Knowledge representation and reasoning modal logic and description logic. University of Manchester, 2008-2009. Available at: http://www.cs.man.ac.uk/ schmidt/COMP6016/2008-2009/mldlWeek1.pdf.

[SH06] R. A. Schmidt and U. Hustadt. First-order resolution methods for modal logics. In A. Podelski, A. Voronkov, and R. Wilhelm, editors, *Volume in memoriam of Harald Ganzinger*, Lecture Notes in Computer Science. Springer, 2006. Invited overview paper, to appear.

[SH07] Renate A. Schmidt and Ullrich Hustadt. The axiomatic translation principle for modal logic. *ACM Trans. Comput. Log.*, 8(4), 2007.

[Sko55] T. Skolem. Peano's axioms and models of arithmetic. *Mathematical Interpretations of formal systems*, pages 1–14, 1955. North-Holland.

[Vor09] Andrei Voronkov. Comp60121 lecture notes: Automated reasoning, part i. University of Manchester, 2008-2009. Available at: http://www.voronkov.com/ar.cgi.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *CADE*, pages 140–145, 2009.

[Wei05] Christoph Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. available at http://spass.mpi-sb.mpg.de/index.html, 2005.

[Wei06] Christoph Weidenbach. SPASS online quick tutorial, January 2006. Available at: http://www.spass-prover.org/tutorial.html.

[Wei07] Christoph Weidenbach. The SPASS handbook, 2007. max planck institut informatik.