Building Logic Toolboxes

ILLC Dissertation Series DS-2003-03

Institute for Logic, Language and Computation

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-525 6051
fax: +31-20-525 5206
e-mail: `illc@science.uva.nl`
homepage: `http://www.illc.uva.nl/`

Building Logic Toolboxes

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.mr. P.F. van der Heijden
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op donderdag 4 december 2003, te 12.00 uur

door

Juan Martín Heguiabehere

geboren te Buenos Aires, República Argentina.

Promotor: Prof.dr. D.J.N. van Eijck

Co-promotor: Dr. M. de Rijke

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Printed and bound by PrintPartners Ipskamp.

ISBN: 90–5776–115–7

Contents

Acknowledgments

Mis amigos son gente cumplidora
que acuden cuando saben que yo espero
si les roza la muerte, disimulan,
que para ellos la amistad es lo primero.

– *"Malas compañías", Joan Manuel Serrat*

Once an engineer, always an engineer. It's very hard to get rid of some habits, and even harder to abandon one's worldview; that might be why when I arrived at the ILLC what I set to do was to build *tools*. It was truly fortunate for me that my supervisors, Jan van Eijck and Maarten de Rijke, not only tolerated but even encouraged my impulse to transform theories into computer programs. A few of those programs were deemed good enough to be mentioned in the rest of this book; as such, they occupy enough space already and I will say no more about them here.

There are lots of people I have to thank, and I hope I remember all of them. First of all my parents: they were my first great teachers, and continue to be.

When one moves to another continent, several things are left behind: most notoriously family and friends. When I arrived, however, some friends were here already. Thanks to Carlos, to Juan Manuel and Susanne, to Alejandro and Iris, to Nacho and Ana, to Bárbara, and to Roberto, for giving a Buenos Aires flavor to some places in Europe. Especially to Carlos, who even went so far as to become a Ph.D. student in the ILLC and avail himself of many great friends in Amsterdam before I even thought of leaving home. Said friends became immediately my friends too, as could be expected, and it was a group that always grew, never diminishing even if some members moved away. For shared friendship, cinema, food, evenings, travel, parties, and good times in general, I want to thank Nikos Massios, Claudia Bedini, Gabriel Infante López, Rosella Gennari, Piero Spinnato, Caterina Caracciolo, Marta García Matos, Jon Calsamiglia, Catarina Dutilh-Novaes, Shai Berger, Stasinos Konstantopoulos, Balder ten Cate, Joost Joosten, Marco Vervoort, Raffaella Bernardi, Patrick Blackburn, Renata Wasserman, Stefan Schlobach, Valentin Jijkoun, Detlef Prescher, Karin Müller, Aline Honingh, Neta Spiro, Gabriele Musillo, Breanndán Ó Nualláin, Bernadette Martínez, Mónica Naef, Andrea Rocco, Miranna Portal, Alessandra Palmigiano, Marco Aiello, Claudia Rosetti, Jelle Gerbrandy, Elena Brosio, Carla Piazza, Massimo Franceschet, and Miguel Valero Espada. It has been good to meet you all.

Far but not lost, I want to also thank those friends who remained in Argentina and kept writing, who reclaimed my presence every now and then, and obligingly showed up whenever I went back for a short holiday. Thanks to Fernando for dispelling the illusion that you can't spend time with your friends just because they are not on the same country; to Silvia, for having her birthday fall during my visits; to Ricardo for teaching me how to make pizza; to Leo for the 'slinky effect' and so many odysseys in Campana; to Alejandro for reappearing after so many years; to Alf for keeping his sanity whilst working for a corporation; to Gaby, for putting up with more engineers than recommended by the Public Health Authority; to Sergio, for sending me pictures of a strangely familiar goofy person; to Juan José, for remaining close even after moving to Silicon Valley; and to Gabriel, Javier, Adrio, and Sebas, for years of dungeons and dragons and music and books.

Coming back to the subject of the thesis, I want to thank again Balder, Breanndán, Carlos and Rosella for the stimulating work we did together and for allowing me to use material from it here; Aline for helping me with the samenvatting; Marco Vervoort, for his very handy timetable for making sure a promotion happens when and as planned; Ria Rettob and Marjan Veldhuisen for all their help.

Finally, I wish to thank my supervisors, Maarten and Jan, for taking the risk of allowing a relative outsider to the field (me) to come into the ILLC, for giving me the right mix of guidance and freedom, for being there whenever I needed help of any sort, and for all I learned with them.

Amsterdam
October, 2003.

Juan Heguiabehere

Chapter 1

Introduction

The purpose of computing is insight,
not numbers.
– Richard Hamming

1.1 What is Logic *good for*?

Formal logic is the study of necessary truths and of systematic methods for clearly expressing and rigorously demonstrating such truths.

When confronted with a modeling task, logic can be used to capture a situation (a property of the world, a machine state, a cognitive state, the state of a database, ...); given the inference mechanisms allowed by the logic, we can then derive implicit or explicit information about the situation being modeled. During its long history, logic has been used to analyze phenomena ranging from planning in robotics or scheduling in railways to natural language processing [CGM$^+$97, BBKdN98, CGV02]. The value of logic as a tool comes from its power to validate complex assertions; if the premises are true and our reasoning is correct, our conclusion is guaranteed to be also true.

In this thesis we are interested in "classical" logics, that is, in logics that work on exact ("crisp") input and that only admit two truth values: *true* and *false*. While this may not always be the best choice, it is very much the norm for the settings in which we are interested in in this thesis: *modal and hybrid satisfiability testing* and *dynamic first order logic theorem proving.*

1.2 Working with Logic

Logic is useful in any context in which the notion of inference is relevant. In particular, logic can be used to certify that computer programs perform their assigned task (if formally stated) [Hoa69], or that reactive systems have the desired behavior, or that a theory is consistent. Sometimes, however, the task of determining whether or not a statement follows from a theory is so huge as to be intractable for humans; this led to the development of programs to automate the inference tasks [Rob65, DP60, Smu68]. More specifically, automated tools exist to support the following reasoning tasks (among others)

- satisfiability checking: the task of determining whether a given formula or set of formulas in a certain logic is *possibly* true.
- validity checking: the task of determining whether a given formula or set of formulas in a certain logic is *necessarily* true.
- model checking: the task of determining whether a given formula in a certain logic is true, given a model.
- model generation: the task of finding out which model, if any, makes a given formula true.

But there is also a further reason to develop general-purpose automated reasoning tools: Having a computer program carry out the reasoning tasks lets us experiment with theories, concentrate on the modeling tasks, handle bigger problems than we could on our own. Ideas become more tangible, and if a tool is well implemented, it is possible that people will use it for things the authors never dreamed of. In a sense, having a reasoning tool *empowers* a logic to come out of the books and get its hands dirty (hopefully for a clean cause). This thesis is then about automated reasoning *tools*: how one can make a tool for automated reasoning, how to tell if it is any good, how to make it better, and how it can be useful.

In this thesis we will focus almost exclusively on satisfiability checking. The purpose of this work is to explore some of the algorithms that enable computers to perform automated satisfiability checking, as well as their implementation and assessment. We will discuss some of the ways in which logic can be put to use through automated reasoning, and the importance of testing in the evolution of automated reasoning tools.

1.3 The Road Ahead

The rest of this thesis is organized in two main parts. Part I, Evaluation in Modal and Hybrid Theorem Proving, deals with current and existing efforts in the field of modal and hybrid logic theorem proving, and the importance of evaluation in the design and comparison of theorem provers as well as in the evaluation of the benchmarks themselves. In Chapter 2 we'll review the evolution of benchmarking in modal logic theorem proving, and introduce a hybrid logic benchmark. In Chapter 3 we talk about the different methods for translating Modal Logic to First Order Logic (FOL), to take advantage of the years of development that went into FOL theorem proving, and how different methods compare. In Chapter 4 we describe another approach to theorem proving in non-classical logics: developing your own specialized theorem prover. We describe the theory and implementation of HyLoRes, a resolution-based theorem prover for hybrid logics; we also describe how testing was an integral part of development.

In Part II, Programming with Dynamic First Order Logic, we explore the use of Dynamic First Order Logic (DFOL) as a programming language. In Chapter 5 we give some background to the 'formulas as programs' paradigm; we introduce the concept of an executable interpretation of DFOL($\cup, \sigma$), and describe two increasingly faithful approximations to the interpretation. In Chapter 6 we explain why DFOL($\cup, \sigma$) is a good candidate for a programming language and describe a Hoare calculus for it. In Chapter 7 we describe a tableau calculus for DFOL($\cup, \sigma$) which gives an even better approximation to the executable interpretation of DFOL($\cup, \sigma$) and can be used as a programming language engine, and in Chapter 8 we describe the implementation of such an engine and show some example runs.

In Part III, Conclusion, we reflect on what was learned from Parts I and II, what they had in common, and where they would meet.

Parts of Chapter 2 were originally published in [HdR01] and [AH03]; Chapter 3 contains material from [AGHdR00]; Chapter 4 is an extension of [AH02a]. Most of Chapter 7 was originally published in [vEHN01].

Before embarking on our trip, we will review the notions and notation required for reading the material in later chapters. In addition, the next few sections provide the reader with an overview that should help situate the logics and issues investigated in this thesis.

1.4 A Plethora of Logics

We present now the general logical framework in which this work is set. Outside of propositional logic [GvMW00], the best known logic, the one which has the most tools developed for it, is First Order Logic (FOL). The satisfiability problem for first order logic is undecidable, in spite of which a myriad of reasoning tools exist; see [CAS]. These tools have reached impressive levels of optimization, but the fact remains that the underlying problem is undecidable. So, if the problem at hand can be stated in terms of a less expressive logic which has a decision procedure, that's already an improvement (at least in principle). Also, sometimes FOL does not offer the right perspective for the task at hand, so that a logic with the same expressive power, yet different syntax or semantics, will be better suited. Specifically, the following three logics will play a leading role in this thesis:

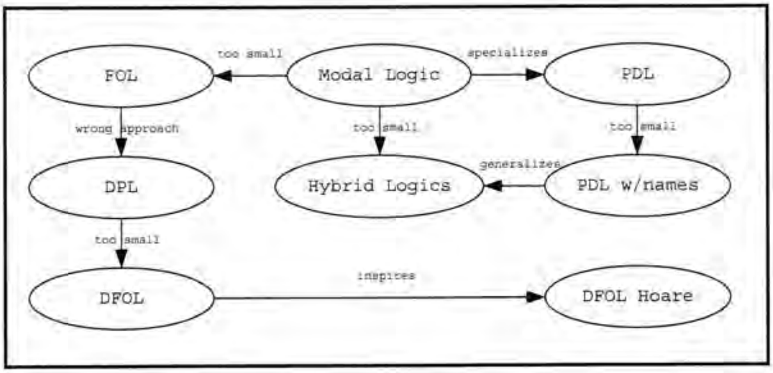


Figure 1.1: Relationships between the logics introduced

- first order logic [Fit96].
- modal and hybrid logic [BdRV01].
- dynamic first order logic [GS91].

But we will encounter even more logics. In Figure 1.1 we provide a diagrammatic overview of the logics we will shortly introduce. The labels on the arrows indicate some aspect of the relation between the logics involved.

We will now provide formal definitions as well as some examples and discussions that should help understand their *raison d'être*.

1.4.1 First Order Logic

First order logic, by far the most widely studied logic, was first formulated in 1879 by Frege. It provides a formal framework for quantified expressions of the form 'all computers use Windows' or 'there is a computer that does not crash'. Even though it cannot quantify over properties, its satisfiability problem is already *undecidable*: for some sentences of FOL, it is not possible to ascertain whether they could be true or not. We will now introduce the logic proper.

1.4.1. Definition. [Syntax] Let REL be a countable set of relational symbols, each with its own arity, let FUN be a countable set of function symbols, each with its own arity, and let CON and VAR be countable sets of constant and variable symbols respectively. We call $\mathcal{S} = \langle \mathsf{REL}, \mathsf{FUN}, \mathsf{CON}, \mathsf{VAR} \rangle$ a *signature*. The well-formed *terms* over this signature are defined as follows:

$$\mathsf{TERMS} := c \mid x \mid f(t_1, \ldots, t_n),$$

where $c \in \mathsf{CON}, x \in \mathsf{VAR}$, $f \in \mathsf{FUN}$ with arity n, and $t_1, \ldots, t_n \in \mathsf{TERMS}$. The well-formed formulas over the signature are

$$\mathsf{FORMS} := \top \mid R(t_1, \ldots, t_n) \mid \neg\phi \mid (\phi_1 \wedge \phi_2) \mid \exists x(\phi),$$

where $R \in \mathsf{REL}$ with arity n, $t_1, \ldots, t_n \in \mathsf{TERMS}$, $x \in \mathsf{VAR}$ and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$. We take $\vee, \rightarrow, \leftrightarrow$ and $\forall$ as defined symbols.

1.4.2. Definition. [Semantics of FOL]: A *model* over a signature $\mathcal{S}$ is a pair $\mathcal{M} = (D, I)$, where D is a nonempty set, called the *domain* of $\mathcal{M}$, and I is an *interpretation*; to every $f \in \mathsf{FUN}$ of arity n, it associates a function $f_I : D^n \rightarrow D$, and to every $R \in \mathsf{REL}$ of arity n, a relation $R_I \subseteq D^n$. To every element c of CON, it associates an element of D. An *assignment* in a model $\mathcal{M} = (D, I)$ is a mapping $g : \mathsf{VAR} \rightarrow D$. Given an assignment g for $\mathcal{M}$, $x \in \mathsf{VAR}$ and $m \in D$, we define g^x_m (an *x-variant* of g) by $g^x_m(x) = m$ and $g^x_m(y) = g(y)$, for $y \neq x$. Now, given a model $\mathcal{M}$ and an assignment g every term in the language can be evaluated to an element of D:

$$\begin{array}{rcl} I(x) & = & g(x) \\ I(f(t_1, \ldots, t_n)) & = & I(f)(I(t_1), \ldots, I(t_n)). \end{array}$$

And the satisfiability relation, then, is as follows:

$$\begin{array}{rcl} \mathcal{M} \models \top[g] & & \text{always} \\ \mathcal{M} \models R(t_1, \ldots, t_n)[g] & \text{iff} & I(R)(I(t_1), \ldots, I(t_n)) \\ \mathcal{M} \models \neg\phi[g] & \text{iff} & \mathcal{M} \not\models \phi[g] \\ \mathcal{M} \models \phi_1 \wedge \phi_2[g] & \text{iff} & \mathcal{M} \models \phi_1[g] \text{ and } \mathcal{M} \models \phi_2[g] \\ \mathcal{M} \models \exists x(\phi)[g] & \text{iff} & \mathcal{M} \models \phi[g^x_m] \text{ for some } m \in D. \end{array}$$

1.4.2 Restrictions

Sometimes the full expressive power of FOL is not necessary; in those cases, we might be able to model our problem using logics that are less expressive but more tractable. It is also possible that a logic is as complex as FOL, but is better suited at describing the situation at hand. In this subsection we review a small number of restrictions of first order and second order logic.

Modal Logic. Modal logic is a powerful and flexible tool for working with relational structures [BdRV01]. It's very well behaved, robustly decidable [Var97], and allows us to reason about relational structures such as those found in mathematics, computer science and linguistics. Modal and modal-like logics such as temporal logic, description logic, and feature logic, have had a long history in artificial intelligence, both as an area of foundational research and as a source for useful representation formalisms and reasoning methods [FHMV95, HS97], and the recent advent of agent-based technologies and the Semantic Web have dramatically increased the need for efficient automated reasoning methods for modal logic [FHMV95, PSHvH02]. But there are things that can't be expressed in modal logic: the gains in decidability have a price in expressiveness.

1.4.3. Definition. [Syntax] Let REL be a countable set of *relational symbols*, and PROP a countable set of *propositional variables*. The well-formed formulas of the modal language $\mathcal{ML}$ in the signature $\langle \mathsf{REL}, \mathsf{PROP} \rangle$ are

$$\mathsf{FORMS} := \top \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid [R]\phi$$

where $p \in \mathsf{PROP}$, $R \in \mathsf{REL}$ and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$.

The operator $\langle R \rangle$ is defined as $\neg[R]\neg$, i.e. they are *dual* operators.

1.4.4. Definition. [Semantics] Given a signature $\langle \mathsf{REL}, \mathsf{PROP} \rangle$, a (modal) *model* $\mathcal{M}$ is a triple $\mathcal{M} = \langle M, \{R_i\}, V \rangle$ such that M is a non-empty set, $\{R_i\}$ is a set of binary relations on M, and $V : \mathsf{PROP} \to Pow(M)$.

Let $\mathcal{M} = \langle M, \{R_i\}, V \rangle$ be a model, $m \in M$. Then the satisfiability relation is defined as follows:

$$\begin{array}{llll}
\mathcal{M}, m \Vdash & \top & \text{always} & \\
\mathcal{M}, m \Vdash & p & \text{iff} & m \in V(p), p \in \mathsf{PROP} \\
\mathcal{M}, m \Vdash & \neg\phi & \text{iff} & \mathcal{M}, m \nVdash \phi \\
\mathcal{M}, m \Vdash & \phi_1 \wedge \phi_2 & \text{iff} & \mathcal{M}, m \Vdash \phi_1 \text{ and } \mathcal{M}, m \Vdash \phi_2 \\
\mathcal{M}, m \Vdash & [R]\phi & \text{iff} & \forall m'.(R(m, m') \implies \mathcal{M}, m' \Vdash \phi)
\end{array}$$

Modal logic allows us then to talk about properties of elements of a given domain, which are themselves connected to each other by one or more relations. What we can't do with modal logic, however, is tell these elements apart; two different elements of the same model can satisfy the same set of modal formulas and therefore be indistinguishable to the logic. Now, we could go back to FOL, but we can also see if we can add expressive power to the modal language and still preserve decidability. This has been carried out in a number of ways, as we shall see below.

Hybrid Logics. In hybrid logics, the relational structures of modal logics are kept, but we add the capability to refer to individual elements of M, thus going beyond the expressive power of modal logic. Some of the additions increase the complexity of the satisfiability problem, while others go so far as making it undecidable, but basically hybrid logics can be tailored so that their expressivity and complexity are matched to the problem at hand. Hybrid logics can be said to span the expressivity and complexity gap between modal logic and FOL; see [HyL] for a thorough introduction and extensive bibliography. Still, hybrid logics are not the only way in which one can extend modal logic; we'll look at some more ways later.

1.4.5. DEFINITION. [Syntax] Let REL be a countable set of *relational symbols*, PROP a countable set of *propositional variables*, NOM a countable set of *nominals*, and SVAR an infinite, countable set of *state variables*. We assume that these sets are pairwise disjoint. We call $\mathsf{SSYM} = \mathsf{NOM} \cup \mathsf{SVAR}$ the set of *state symbols*, and $\mathsf{ATOM} = \mathsf{PROP} \cup \mathsf{NOM} \cup \mathsf{SVAR}$ the set of *atoms*. The well-formed formulas of the hybrid language $\mathcal{H}(@, \downarrow)$ in the signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM}, \mathsf{SVAR} \rangle$ are

$$\mathsf{FORMS} := \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid [R]\phi \mid @_s\phi \mid \downarrow x.\phi,$$

where $a \in \mathsf{ATOM}$, $x \in \mathsf{SVAR}$, $s \in \mathsf{SSYM}$, $R \in \mathsf{REL}$ and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$.

Note that all types of atomic symbol (i.e., proposition symbols, nominals and state variables) are *formulas*. Further, note that the above syntax is simply that of ordinary (multi-modal) propositional logic extended with clauses for $@_s\phi$ and $\downarrow x.\phi$. Finally, the difference between nominals and state variables is simply that nominals cannot be bound by $\downarrow$, whereas state variables can.

The notions of *free* and *bound* variable are defined as in first order logic, with $\downarrow$ as the only binding operator. A *sentence* is a formula containing no free state variables.

The basic hybrid language is $\mathcal{H}$, basic modal logic extended with nominals. Further extensions are usually named by listing the added operators; we are interested in the logics $\mathcal{H}(@)$ and $\mathcal{H}(@, \downarrow)$, which also adds state variables.

1.4.6. DEFINITION. [Semantics] A (hybrid) *model* $\mathcal{M}$ is a triple $\mathcal{M} = \langle D, \{R_i\}, V \rangle$ such that D is a non-empty set, $\{R_i\}$ is a set of binary relations on D, and $V : \mathsf{PROP} \cup \mathsf{NOM} \to Pow(D)$ is such that for all nominals $i \in \mathsf{NOM}$, $V(i)$ is a singleton subset of D.

An *assignment* g for $\mathcal{M}$ is a mapping $g : \mathsf{SVAR} \to D$. Given an assignment g, g^x_m is defined as for FOL. Assignments are not needed when dealing with $\mathcal{H}(@)$.

Let $\mathcal{M} = \langle D, \{R_i\}, V \rangle$ be a model, $m \in D$, and g an assignment. For any atom a, let $[V, g](a) = \{g(a)\}$ if a is a state variable, and $V(a)$ otherwise. Then the satisfiability relation is defined as follows:

$$\begin{array}{lcll}
\mathcal{M},g,m \Vdash & \top & \text{always} & \\
\mathcal{M},g,m \Vdash & a & \text{iff} & m \in [V,g](a),\ a \in \mathsf{ATOM} \\
\mathcal{M},g,m \Vdash & \neg\phi & \text{iff} & \mathcal{M},g,m \nVdash \phi \\
\mathcal{M},g,m \Vdash & \phi_1 \wedge \phi_2 & \text{iff} & \mathcal{M},g,m \Vdash \phi_1 \text{ and } \mathcal{M},g,m \Vdash \phi_2 \\
\mathcal{M},g,m \Vdash & [R]\phi & \text{iff} & \forall m'.(R(m,m') \implies \mathcal{M},g,m' \Vdash \phi) \\
\mathcal{M},g,m \Vdash & @_s\phi & \text{iff} & \mathcal{M},g,m' \Vdash \phi, \text{ where } [V,g](s) = \{m'\} \\
\mathcal{M},g,m \Vdash & \downarrow x.\phi & \text{iff} & \mathcal{M},g^x_m,m \Vdash \phi.
\end{array}$$

Named elements can now be distinguished, and we can express properties which were not expressible before: the formula $(\downarrow x.[R]\neg x)$ is true in every element of a model if and only if the accessibility relation R for that model is irreflexive, something not expressible in modal logic.

Propositional Dynamic Logic. While *Propositional Dynamic Logic*(PDL) is a modal logic, by all accounts, it is not a restriction of first order logic (as modal and hybrid logic), but rather a restriction of second order logic. Propositional Dynamic Logic deals with actions as modalities; usually, the represented actions are atomic programs, and the elements of the domain therefore reflect the relevant state of the computer running them. With this interpretation in mind, many natural operators on programs (i.e., relations) suggest themselves, such as $\cup$ (non-deterministic choice), ; (sequential composition), and the Kleene star $*$ (iteration). See [HKT84] for a thorough introduction. Here's a brief overview of the standard repertoire of PDL operators, with their intended meanings:

$[\alpha]A$	After every execution of α, A holds
$\alpha_1;\alpha_2$	Do α_1 and then do α_2
$\alpha_1 \cup \alpha_2$	Do either α_1 or α_2 non-deterministically
α^*	repeat α some finite number (possibly zero) of times
$A?$	Test A; continue if A is true, otherwise fail.

1.4.7. DEFINITION. [Syntax] Let AP be a set of atomic programs, and PROP a set of atomic formulas. Then the formulas A and the programs α are defined as:

$$\begin{array}{ll}
\mathsf{FORMS} & := \bot \mid p \mid A_1 \to A_2 \mid [\alpha]A, \\
\mathsf{PROGS} & := \pi \mid \alpha_1;\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \mid A?
\end{array}$$

where $p \in \mathsf{PROP}, \pi \in \mathsf{AP}, A, A_1, A_2 \in \mathsf{FORMS}$, and $\alpha, \alpha_1, \alpha_2 \in \mathsf{PROGS}$.

1.4.8. DEFINITION. [Semantics] A model for this language would be a structure of the form $\mathcal{M} = (S,\ \{R_\alpha : \alpha \in \mathsf{PROGS}\}, V)$ with R_α a binary relation on S for each program α and $V : \mathsf{PROP} \to S$ a valuation. We want to consider models that reflect the intended meanings of the program combinations; a model is considered *standard* if the R_α satisfy the following conditions:

$$\begin{array}{lll} R_{\alpha_1 \cup \alpha_2} & = & R_{\alpha_1} \cup R_{\alpha_2}, \\ R_{\alpha_1 ; \alpha_2} & = & R_{\alpha_1} R_{\alpha_2} = \{(s,t) \mid \exists v (s R_{\alpha_1} v \wedge v R_{\alpha_2} t)\}, \\ R_{\alpha^*} & = & (R_\alpha)^* = \bigcup_{k<\omega} (R_\alpha)^k, \\ R_{A?} & = & \{(s,s)\} \mid s \models A\}, \end{array}$$

The semantics of a PDL formula, then, are as follows:

$$\begin{array}{lcl} \mathcal{M}, s \Vdash & \bot & \text{never} \\ \mathcal{M}, s \Vdash & p & \text{iff } s \in V(p) \\ \mathcal{M}, s \Vdash & A_1 \to A_2 & \text{iff } \mathcal{M}, s \Vdash A_1 \text{ implies } \mathcal{M}, s \Vdash A_2 \\ \mathcal{M}, s \Vdash & [\alpha]A & \text{iff } s R_\alpha t \text{ implies } \mathcal{M}, t \models A \end{array}$$

Combinatory PDL. Next we consider an extension of PDL: Combinatory PDL [PT85, PT91], which adds nominals and the universe program. This brings about a huge increase in expressive power, accompanied by undecidability. The main insight behind Combinatory PDL was the search for a dynamic logic that would allow for an axiomatic definition of the intersection between two modalities; this is particularly relevant for parallel, or concurrent, computing [Pel85].

1.4.9. Definition. [Syntax] Let AP and PROP be the sets of atomic programs and atomic formulas, as in PDL, and NOM be the set of *names.* The letter $\nu \notin \mathsf{NOM} \cup \mathsf{PROP} \cup \mathsf{AP}$ will be called the *universe program.* Then the formulas FORMS and programs PROGS of the language are defined as:

$$\begin{array}{ll} \mathsf{FORMS} & := \bot \mid p \mid n \mid A_1 \to A_2 \mid [\alpha]A, \\ \mathsf{PROGS} & := \pi \mid \nu \mid \alpha_1 ; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \mid A? \end{array}$$

where $p \in \mathsf{PROP}, \pi \in \mathsf{AP}, n \in \mathsf{NOM}, A, A_1, A_2 \in \mathsf{FORMS}$, and $\alpha, \alpha_1, \alpha_2 \in \mathsf{PROGS}$.

1.4.10. Definition. [Semantics] A *model* for CPDL is a quadruple $\mathcal{M} = (M, R, \chi, V)$, where M is a non-empty set (the set of *states*), and the other three are functions:

$$\begin{array}{rl} R: & \mathsf{PROGS} \to Pow(M^2), \\ \chi: & \mathsf{NOM} \to M, \\ V: & \mathsf{FORMS} \to Pow(M), \end{array}$$

R satisfies the following requirements:

$$\begin{array}{lll} R_\nu & = & M^2, \\ R_{\alpha \cup \beta} & = & R_\alpha \cup R_\beta, \\ R_{\alpha ; \beta} & = & R_\alpha R \beta = \{(s,t) \mid \exists v (s R_\alpha v \wedge v R_\beta t)\}, \\ R_{\alpha^*} & = & (R_\alpha)^* = \bigcup_{k<\omega} (R_\alpha)^k, \\ R_{A?} & = & \{(s,s)\} \mid s \models A\} \end{array}$$

where $sR_\alpha t$ is $(s,t) \in R_\alpha$.
Given this model, the semantics of CPDL are as follows:

$$\begin{array}{lcl}
\mathcal{M}, s \Vdash & \bot & \text{never} \\
\mathcal{M}, s \Vdash & n & \text{iff } s = \chi(n), \text{ for } n \in \mathsf{NOM} \\
\mathcal{M}, s \Vdash & p & \text{iff } s \in V(p) \\
\mathcal{M}, s \Vdash & A_1 \to A_2 & \text{iff } \mathcal{M}, s \Vdash A_1 \text{ implies } \mathcal{M}, s \Vdash A_2 \\
\mathcal{M}, s \Vdash & [\alpha]A & \text{iff } sR_\alpha t \text{ implies } \mathcal{M}, t \models A
\end{array}$$

1.4.3 Extensions

After having reviewed a number of restrictions of first order and second order logic, we will now examine some extensions of first order logic that play an important role in this thesis.

Dynamic First Order Logic

Dynamic Predicate Logic (DPL) was introduced by Groenendijk and Stokhof [GS91] as a first step towards a compositional, non-representational theory of discourse semantics. Like we did with hybrid logics, we will now present the family of dynamic first order logics, obtained by using DPL as a base logic and extending it with additional operators, some of which we will use to arrive at an useful executable program interpretation.

The difference between DPL and first order logic proper resides mostly in their semantics, in that the meaning of a DPL sentence is not captured by its truth conditions but by the way it changes the information state of the interpreter; a sentence takes us from one state of information to another, and its meaning is given by *how* it does so [GS91]. This feature of DPL makes it very straightforward to supply it with an executable program interpretation, with the advantage that any programming language based on such an interpretation will have a clear and natural semantics.

For example, the FOL formula $\phi_1 \wedge \phi_2$ is true in a model $\mathcal{M}$ under an assignment g iff both ϕ_1 and ϕ_2 are true under that assignment, while in DPL the formula $\phi_1; \phi_2$ (sequential composition, the DPL analogue to $\wedge$) carries us from an assignment s to an assignment u iff there is an assignment t such that ϕ_1 carries us from s to t and ϕ_2 carries us from t to u. We will now formally introduce DPL.

1.4.11. Definition. [Syntax of DPL] Let PRED be a countable set of predicate symbols, each with its own arity, and let CON and VAR be countable sets of constant and variable symbols respectively. The DPL language is then given by the following production rules:

$$\begin{array}{lcl}
\mathsf{TERMS} & := & v \mid c \\
\mathsf{FORMS} & := & \exists v \mid P\bar{t} \mid t_1 \doteq t_2 \mid \neg(\phi) \mid \phi_1; \phi_2
\end{array}$$

where $v \in \mathsf{VAR}, c \in \mathsf{CON}, P \in \mathsf{PRED}, t, t_1, t_2 \in \mathsf{TERMS}$, and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$.

DPL has been extended with a variety of operators, some([Vis98, GS90]) coming from the original linguistic perspective and some [vEHN01] from the 'formulas as programs' perspective; a survey of extensions of DPL can be found in [tCvEH01]. In this work we will consider the logics resulting of the extension of DPL with operators for nondeterministic choice ($\phi \cup \phi$), explicit substitutions (σ), local variable declaration ($\exists v(\phi)$) and iteration (ϕ^*), as well as the inclusion of function symbols in the signature. We will give the name Dynamic First Order Logic (DFOL) to the extension of DPL with function symbols.

1.4.12. DEFINITION. [DFOL and Extensions] Given a signature for FOL, the syntax for DFOL and extensions is the appropriate fragment of the following:

$$\begin{array}{lcll}
\mathsf{TERMS} & := & v \mid c \mid f(t_1, \ldots, t_n) & \text{(Terms)} \\
\mathsf{FORMS} & := & \exists v \mid P\bar{t} \mid t_1 \doteq t_2 \mid \neg(\phi) \mid \phi_1; \phi_2 \mid (\phi_1 \cup \phi_2) \mid & \text{(Formulas)} \\
 & & \mid (\phi_1 \cap \phi_2) \mid \exists v(\phi) \mid \sigma \mid \breve{\sigma} \mid \breve{\phi} \mid \phi^* &
\end{array}$$

where $v \in \mathsf{VAR}, c \in \mathsf{CON}, P \in \mathsf{PRED}, f \in \mathsf{FUN}, t_1, t_2, \ldots, t_n \in \mathsf{TERMS}$, and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$. We will write $\bar{t}$ for $(t_1, \ldots, t_n)$. The names for the extensions are given as DFOL(X), where X is a subset of $\{\cup, \cap, \exists, \sigma, \breve{\sigma}, \breve{}\}$. A *substitution* σ is a function $\mathsf{VAR} \to \mathsf{TERMS}$ that makes only a finite number of changes, i.e., σ has the property that $dom(\sigma) = \{v \in \mathsf{VAR} \mid \sigma(v) \neq v\}$ is finite. We will use $rng(\sigma)$ for $\{\sigma(v) \mid v \in dom(\sigma)\}$. During the rest of this work, we will use the letters ρ, θ, σ to denote substitutions. An explicit form (or: a representation) for substitution σ is a sequence

$$[\sigma(v_1)/v_1, \ldots, \sigma(v_n)/v_n],$$

where $\{v_1, \ldots, v_n\} = dom(\sigma)$, (i.e., $\sigma(v_i) \neq v_i$, for only the *changes* are listed), and $i \neq j$ implies $v_i \neq v_j$ (i.e., all variables in the domain are mentioned only once). We will use $[]$ for the *empty* substitution, i.e. the substitution that has empty domain and therefore changes nothing. We will call these representations *bindings*. A definition we will need is the one of syntactic composition of bindings:

1.4.13. DEFINITION. [Syntactic composition] The *syntactic composition* of two bindings θ and ρ (notation $\theta \circ \rho$) is defined in the following way:
Let $\theta = [t_1/v_1, \ldots, t_n/v_n]$ and $\rho = [r_1/w_1, \ldots, r_m/w_m]$ be binding representations. Then $\theta \circ \rho$ is the result of removing from the sequence

$$[\theta(r_1)/w_1, \ldots, \theta(r_m)/w_m, t_1/v_1, \ldots, t_n/v_n]$$

the binding pairs $\theta(r_i)/w_i$ for which $\theta(r_i) = w_i$, and the binding pairs t_j/v_j for which $v_j \in \{w_1, \ldots, w_m\}$.

We will omit parentheses where it doesn't create syntactic ambiguity, and allow the usual abbreviations: we write $\bot$ for $\neg([])$, $\neg P\bar{t}$ for $\neg(P\bar{t})$, $t_1 \neq t_2$ for $\neg(t_1 \doteq t_2)$, $\phi_1 \cup \phi_2$ for $(\phi_1 \cup \phi_2)$. Similarly, $(\phi \to \psi)$ stands for $\neg(\phi; \neg(\psi))$, $\forall v(\phi)$ for $\neg(\exists v; \neg(\phi))$, ϕ^n for $\underbrace{(\phi; \cdots; \phi)}_{n}$ and $\bigcup^{v}_{M..N} \phi$ for $(([M/v]; \phi) \cup \cdots \cup ([N/v]; \phi))$, assuming $M, N \in \mathbb{N}$ and $M \leq N$. A formula ϕ is a *literal* if ϕ is of the form $P\bar{t}$ or $\neg P\bar{t}$, or of the form $t_1 \doteq t_2$ or $t_1 \neq t_2$. The complement $\overline{\phi}$ of a formula ϕ is given by: $\overline{\phi} := \psi$ if ϕ has the form $\neg(\psi)$ and $\overline{\phi} := \neg(\phi)$ otherwise. We abbreviate $\neg\neg(\phi)$ as $(\!(\phi)\!)$, and we will call formulas of the form $(\!(\phi)\!)$ *block* formulas.

We can think of formula ϕ as built up from units U by concatenation. For formula induction arguments, it is sometimes convenient to read a unit U as the formula $U; []$ (recall that $[]$ is the empty binding), thus using $[]$ for the 'true' formula. This formula has the same semantics as U; see Definition 1.4.16. In other words, we will silently add the $[]$ at the end of a formula list when we need its presence in recursive definitions or induction arguments on formula structure.

Binding in DFOL($\sigma, \cup$)

The extension of DFOL that we will be using as the core of most of Part II is DFOL($\sigma, \cup$); DFOL augmented with nondeterministic choice and simultaneous bindings. Here follow some definitions and results that we will need later on.

Bindings θ are lifted to (sequences of) terms and (sets of) formulas in the familiar way:

1.4.14. Definition. [Binding in DFOL($\sigma, \cup$)]

$$\begin{array}{rcl}
\theta(ft_1 \cdots t_n) & := & f\theta(t_1) \cdots \theta(t_n) \\
\theta(\rho) & := & \theta \circ \rho \\
\theta(\rho; \phi) & := & (\theta \circ \rho)\phi \\
\theta(\exists v; \phi) & := & \exists v; \theta'\phi \text{ where } \theta' = \theta \backslash \{t/v \mid t \in \mathsf{TERMS}\} \\
\theta(P\bar{t}; \phi) & := & P\theta\bar{t}; \theta\phi \\
\theta(t_1 \doteq t_2; \phi) & := & \theta t_1 \doteq \theta t_2; \theta\phi \\
\theta((\phi_1 \cup \phi_2); \phi_3) & := & \theta(\phi_1; \phi_3) \cup \theta(\phi_2; \phi_3) \\
\theta(\neg(\phi_1); \phi_2) & := & \neg(\theta\phi_1); \theta\phi_2
\end{array}$$

Note that it follows from this definition that

$$\theta((\!(\phi_1)\!); \phi_2) = (\!(\theta\phi_1)\!); \theta\phi_2.$$

Thus, binding distributes over block: this accounts for how $(\!(\cdots)\!)$ insulates dynamic binding effects.

The composition $\theta \cdot \rho$ of two bindings θ and ρ has its usual meaning of 'θ after ρ', which we get by means of $\theta \cdot \rho(v) := \theta(\rho(v))$. It can be proved in the usual way, by induction on term structure, that the definition has the desired effect, in the sense that for all $t \in T$, for all binding representations θ, ρ: $(\theta \circ \rho)(t) = \theta(\rho(t)) = (\theta \cdot \rho)(t)$.

Here is an example of how to apply a binding to a formula:

$$\begin{aligned} & [a/x]Px; (Qx \cup \exists x; \neg Px); Sx \\ = \; & Pa; [a/x](Qx \cup \exists x; \neg Px); Sx \\ = \; & Pa; ([a/x]Qx; Sx \cup [a/x]\exists x; \neg Px; Sx) \\ = \; & Pa; (Qa; Sa; [a/x] \cup \exists x; \neg Px; Sx) \end{aligned}$$

The binding definition for DFOL fleshes out what has been called the 'folklore idea in dynamic logic' (Van Benthem [vB96]) that syntactic binding $[t/v]$ works semantically as the program instruction $v := t$ (Goldblatt [Gol92]), with semantics given by ${}_s[v := t]^{\mathcal{M}}_u$ iff $u = s[[t]^M_s/v]$. To see the connection, note that $v := t$ can be viewed as DFOL shorthand for $\exists v; v = t$, on the assumption that $v \notin \mathit{var}(t)$. To generalize this to the case where $v \in \mathit{var}(t)$ and to simultaneous binding, auxiliary variables must be used. The fact that we have simultaneous binding represented in the language saves us some bother about these.

In standard first order logic, sometimes it is not safe to apply a binding to a formula, because it leads to accidental capture of free variables. The same applies here. Applying binding $[x/y]$ to $\exists x; Rxy$ is not safe, as it would lead to accidental capture of the free variable y. The following definition defines safety of binding.

1.4.15. DEFINITION. [Binding θ is safe for ϕ]

$$\begin{array}{rcl} \theta \text{ is safe for } \rho & & \text{always} \\ \theta \text{ is safe for } \rho; \phi & :\Longleftrightarrow & \theta \circ \rho \text{ is safe for } \phi \\ \theta \text{ is safe for } P\bar{t}; \phi & :\Longleftrightarrow & \theta \text{ is safe for } \phi \\ \theta \text{ is safe for } t_1 \doteq t_2; \phi & :\Longleftrightarrow & \theta \text{ is safe for } \phi \\ \theta \text{ is safe for } \exists v; \phi & :\Longleftrightarrow & v \notin \mathit{var}(\mathit{rng}\, \theta') \text{ and } \theta' \text{ is safe for } \phi \\ & & \quad \text{where } \theta' = \theta \backslash \{t/v \mid t \in \mathsf{TERMS}\} \\ \theta \text{ is safe for } \neg(\phi_1); \phi_2 & :\Longleftrightarrow & \theta \text{ is safe for } \phi_1 \text{ and } \theta \text{ is safe for } \phi_2 \\ \theta \text{ is safe for } (\phi_1 \cup \phi_2); \phi_3 & :\Longleftrightarrow & \theta \text{ is safe for } \phi_1; \phi_3 \text{ and } \theta \text{ is safe for } \phi_2; \phi_3 \end{array}$$

Note that there are ϕ with [] not safe for ϕ. E.g., [] is not safe for $[y/x]\exists y; Rxy$, because $[y/x]$ is not safe for $\exists y; Rxy$.

Given a first order signature and a model $\mathcal{M} = (D, I)$, the semantics of DFOL is given as a binary relation on the set D^{VAR}, the set of all variable maps (valuations) in the domain of the model. We impose the usual non-empty domain constraint of FOL: any DFOL model $\mathcal{M} = (D, I)$ has $D \neq \emptyset$. If $s, u \in D^{\mathsf{VAR}}$, we use $s \sim_v u$ to indicate that s, u differ at most in their value for v, and $s \sim_X u$ to indicate that s, u differ at most in their values for the members of X. If $s \in D^{\mathsf{VAR}}$ and $v, v' \in \mathsf{VAR}$, we use $s[v'/v]$ for the valuation u given by $u(v) = s(v')$, and $u(w) = s(w)$ for all $w \in \mathsf{VAR}$ with $w \neq v$.

$\mathcal{M} \models_s P\bar{t}$ indicates that s satisfies the predicate $P\bar{t}$ in $\mathcal{M}$ according to the standard truth definition for classical first order logic. $\llbracket t \rrbracket_s^{\mathcal{M}}$ gives the denotation of t in $\mathcal{M}$ under s. If σ is a substitution and s a valuation (a member of D^{VAR}), we will use s_σ for the valuation u given by $u(v) = \llbracket \sigma(v) \rrbracket_s^{\mathcal{M}}$. Then, the semantics of DPL and its extensions is defined inductively:

1.4.16. DEFINITION. [Semantics of extensions of DPL]

$$\begin{array}{rll}
{}_s\llbracket \exists v \rrbracket_u^{\mathcal{M}} & \text{iff} & s \sim_v u \\
{}_s\llbracket P\bar{t} \rrbracket_u^{\mathcal{M}} & \text{iff} & s = u \text{ and } \mathcal{M} \models_s P\bar{t} \\
{}_s\llbracket t_1 \doteq t_2 \rrbracket_u^{\mathcal{M}} & \text{iff} & s = u \text{ and } \llbracket t_1 \rrbracket_s^{\mathcal{M}} = \llbracket t_2 \rrbracket_s^{\mathcal{M}} \\
{}_s\llbracket \neg(\phi) \rrbracket_u^{\mathcal{M}} & \text{iff} & s = u \text{ and } \neg\exists t \text{ with } {}_s\llbracket \phi \rrbracket_t^{\mathcal{M}} \\
{}_s\llbracket \phi_1 ; \phi_2 \rrbracket_u^{\mathcal{M}} & \text{iff} & \exists t \; s.t. \; {}_s\llbracket \phi_1 \rrbracket_t^{\mathcal{M}} \text{ and } {}_t\llbracket \phi_2 \rrbracket_u^{\mathcal{M}} \\
{}_s\llbracket \phi_1 \cup \phi_2 \rrbracket_u^{\mathcal{M}} & \text{iff} & {}_s\llbracket \phi_1 \rrbracket_u^{\mathcal{M}} \text{ or } {}_s\llbracket \phi_2 \rrbracket_u^{\mathcal{M}} \\
{}_s\llbracket \phi_1 \cap \phi_2 \rrbracket_u^{\mathcal{M}} & \text{iff} & {}_s\llbracket \phi_1 \rrbracket_u^{\mathcal{M}} \text{ and } {}_s\llbracket \phi_2 \rrbracket_u^{\mathcal{M}} \\
{}_s\llbracket \exists v(\phi) \rrbracket_u^{\mathcal{M}} & \text{iff} & \exists s', u' \; s.t. \; s \sim_v s', u \sim_v u', \; {}_{s'}\llbracket \phi \rrbracket_{u'}, \quad \text{and } \llbracket v \rrbracket_s^{\mathcal{M}} = \llbracket v \rrbracket_u^{\mathcal{M}} \\
{}_s\llbracket \sigma \rrbracket_u^{\mathcal{M}} & \text{iff} & u = s_\sigma \\
{}_s\llbracket \breve{\sigma} \rrbracket_u^{\mathcal{M}} & \text{iff} & s = u_\sigma \\
{}_s\llbracket \check{\phi} \rrbracket_u^{\mathcal{M}} & \text{iff} & {}_u\llbracket \phi \rrbracket_s^{\mathcal{M}} \\
{}_s\llbracket \phi^* \rrbracket_u^{\mathcal{M}} & \text{iff} & s = u \;\text{ or }\; \exists t \;\text{ s.t. }\; {}_s\llbracket \phi \rrbracket_t^{\mathcal{M}} \text{ and } {}_t\llbracket \phi^* \rrbracket_u^{\mathcal{M}}
\end{array}$$

We will denote by $\llbracket \phi \rrbracket_s^{\mathcal{M}}$ the set of all assignments u such that ${}_s\llbracket \phi \rrbracket_u^{\mathcal{M}}$.

The connection between syntactic binding and semantic assignment is formally spelled out in the following:

1.4.17. LEMMA (BINDING LEMMA FOR DFOL($\sigma, \cup$)). *For all models $\mathcal{M}$, all $\mathcal{M}$-valuations s, u, all formulas ϕ, all bindings θ that are safe for ϕ:*

$$ {}_s\llbracket \theta\phi \rrbracket_u^{\mathcal{M}} \textit{ iff } {}_s\llbracket \theta ; \phi \rrbracket_u^{\mathcal{M}}. $$

Proof. Induction on the structure of ϕ. ⊣

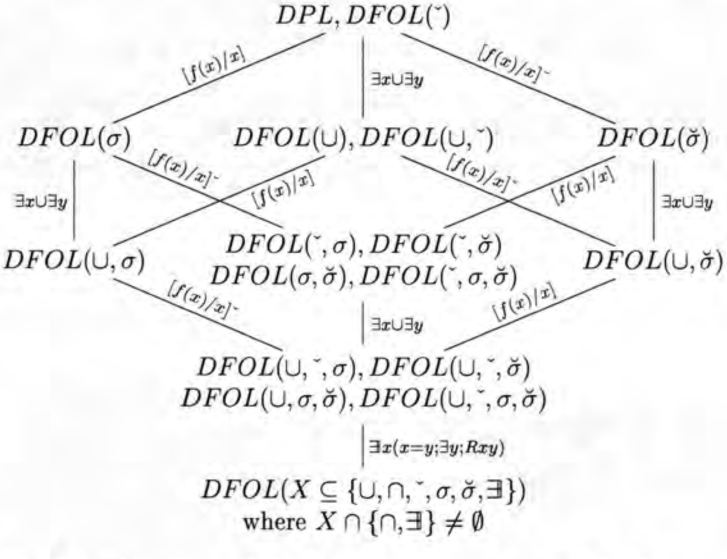
Immediately from this we get the following:

1.4.18. PROPOSITION. *DFOL($\sigma, \cup$) has greater expressive power than DFOL($\sigma, \cup$) with quantification replaced by definite assignment $v := d$.*

Proof. If ϕ is a DFOL($\sigma, \cup$) formula without quantifiers, every binding θ is safe for ϕ. By the binding lemma for DFOL($\sigma, \cup$), ϕ is equivalent to a DFOL($\sigma, \cup$) formula without quantifiers but with trailing bindings. It is not difficult to see that both satisfiability and validity of quantifier free DFOL($\sigma, \cup$) formulas with binding trails is decidable, while DPL is known to be as expressive as FOL [GS91], which is undecidable. $\dashv$

In fact, the tableau system presented in Chapter 7 constitutes a decision algorithm for satisfiability or validity of quantifier free DFOL($\sigma, \cup$) formulas, while the trailing bindings summarize the finite changes made to input valuations.

The Lattice of DPL Extensions. The following figure represents the lattice of all possible extensions of DPL with operators from $\{\cup, \cap, \breve{}, \sigma, \breve{\sigma}, \exists\}$ (union, intersection, converse, simultaneous substitution, converse substitution, hiding) [tCvEH01]. It indicates which operators can be defined in terms of which; the labels on the arrows indicate counterexamples to equal expressivity, i.e., formulae from the lower language that don't have a counterpart in the upper language.


$DPL, DFOL(\breve{})$
$[f(x)/x]$
$\exists x \cup \exists y$
$[f(x)/x]\breve{}$
$DFOL(\sigma)$
$DFOL(\cup), DFOL(\cup, \breve{})$
$DFOL(\breve{\sigma})$
$[f(x)/x]\breve{}$
$[f(x)/x]$
$[f(x)/x]\breve{}$
$[f(x)/x]$
$\exists x \cup \exists y$
$\exists x \cup \exists y$
$DFOL(\cup, \sigma)$
$DFOL(\breve{}, \sigma), DFOL(\breve{}, \breve{\sigma})$
$DFOL(\sigma, \breve{\sigma}), DFOL(\breve{}, \sigma, \breve{\sigma})$
$DFOL(\cup, \breve{\sigma})$
$[f(x)/x]\breve{}$
$\exists x \cup \exists y$
$[f(x)/x]$
$DFOL(\cup, \breve{}, \sigma), DFOL(\cup, \breve{}, \breve{\sigma})$
$DFOL(\cup, \sigma, \breve{\sigma}), DFOL(\cup, \breve{}, \sigma, \breve{\sigma})$
$\exists x(x{=}y; \exists y; Rxy)$
$DFOL(X \subseteq \{\cup, \cap, \breve{}, \sigma, \breve{\sigma}, \exists\})$
where $X \cap \{\cap, \exists\} \neq \emptyset$

Note that all 64 combinations of the six operators are present in the diagram. The diagram makes immediately clear which extensions of DPL are closed under converse: precisely those which are in the same node of the lattice as the corresponding version of DPL *with* converse operator. Adding Kleene star gives an isomorphic lattice for DFOL($*$) and its extensions: none of the distinctions collapse because the same counter-examples to equal expressivity still work.

1.5 The Correctness Problem

How do we know if a program will always perform the task it was written to carry out? We can do some trial runs for which we know the intended output, but that is not a guarantee of correctness; the process is only informative if the program fails, or if we can run it on every possible input, which is usually not feasible. But we can turn to logic for an answer: in general, the purpose of a program is to achieve a desired state transformation, and a specification is a "declarative" description of such a transformation. That is, it specifies the desired net effect of a transformation without concerning itself about how this effect is achieved in a particular implementation. The classical method of Hoare [Hoa69] presents a specification as a pair (A, B) of expressions in a FOL over an underlying data structure, meaning that the task of the required program is to bring the data structure from any state satisfying A to a state satisfying B. Then, a way of checking whether a program fulfills a specification is to have a language that lets us talk about specifications and programs and a calculus that lets us reason in that language. If we can prove that the calculus preserves correctness and covers all the possible correct combinations, then we can check any program against its specifications, or use the calculus to help build the program.

1.5.1 Alma-0 : Executable First Order Logic

The correctness analysis of a program in the manner just described is made much simpler if the programming language has a faithful translation into logical formulas: this is one of the insights behind the Alma-0 programming language (see [AB98, ABPS98]). Alma-0 extends a subset of Modula-2 (an *imperative* programming language) with a number of *declarative* constructs inspired by the logic programming paradigm. A translation is given from the extensions into FOL, and the semantics of the extensions is then stated in terms of an executable interpretation of FOL [Apt00, Ver03]. We will give more details on this perspective in Chapter 5, where we give the executable program interpretation of DFOL.

1.5.2 DFOL and correctness

In the usual correctness reasoning, we distinguish between *partial* and *total* correctness, the difference being that total correctness ensures termination. In DFOL, negation is expressed as a test of failure to terminate successfully; therefore, even for partial correctness we must examine at the same time both correctness and termination. We distinguish two main kinds of correctness rules for DFOL: universal and existential. The existential rules guarantee termination and the existence of at least one output state which satisfies the postcondition, while universal rules are equivalent to partial correctness: i.e. they guarantee that all resulting states will satisfy the postcondition but do not guarantee successful termination. We express existential correctness by $(A)\phi(B)$, and universal correctness as $\{A\}\phi\{B\}$. Total correctness is proved when we derive both universal correctness and existential correctness for the same program, although $\top$ will suffice as the postcondition for the existential case. Of course, existential correctness might result in a different precondition, but then the conjunction of the universal and existential preconditions will guarantee total correctness. Formally, the two kinds of correctness boil down to the following:

$$\begin{array}{lcl} \mathcal{M} \models (A)\phi(B) & \iff & \forall g\,(\mathcal{M} \models_g A \implies \exists h\,({}_g[\phi]_h^{\mathcal{M}} \wedge \mathcal{M} \models_h B)) \\ \mathcal{M} \models \{A\}\phi\{B\} & \iff & \forall g\,(\mathcal{M} \models_g A \implies \forall h\,({}_g[\phi]_h^{\mathcal{M}} \implies \mathcal{M} \models_h B)) \end{array}$$

1.5.3 *Dynamo*

Dynamo is an imperative programming language whose semantics are defined in terms of DFOL($\cup, \sigma$), in a similar manner as Alma-0 is defined in terms of an executable interpretation of FOL. The Hoare calculus for DFOL($\cup, \sigma$) mentioned above is then directly applicable to *Dynamo*. *Dynamo* programs have a purely declarative dynamic semantics. There are no side effects, and no control features. See Van Eijck[vE98a, vE99b] for a more thorough introduction.

Figure 1.2 introduces the *Dynamo* syntax by means of a translation to the language of DFOL. The translation fixes the intended meaning of every *Dynamo* construct.

Figure 1.2: Translation from *Dynamo* to DFOL.

$(\{S_1; \dots ; S_n\})^\circ$	:=	$S_1^\circ; \dots ; S_n^\circ$
$(\{S_1 \mid \dots \mid S_n\})^\circ$	:=	$S_1^\circ \cup \dots \cup S_n^\circ$
$(\texttt{true})^\circ$	:=	$[]$
$(\texttt{false})^\circ$	:=	$\neg[]$
$(t_1 = t_2)^\circ$	:=	$t_1 \doteq t_2$
$(P\bar{t})^\circ$	:=	$P\bar{t}$
$(\texttt{some}\ v)^\circ$	:=	$\exists v$
$(\texttt{some}\ v_1, \dots , v_n)^\circ$	:=	$\exists v_1; \dots ; \exists v_n$
$(v := t)^\circ$	:=	$[t/v]$
$(v + +)^\circ$	:=	$[(v+1)/v]$
$(\texttt{find}\ v\ \texttt{in}\ [N..M]\ \texttt{with}\ S)^\circ$	:=	$\bigcup_{M..N}^{v} S^\circ$
$(\texttt{do}\ N\ \texttt{times}\ S)^\circ$	:=	$(S^\circ)^N$
$(\texttt{if}\ S_1\ S_2\ \texttt{else}\ S_3)^\circ$	:=	$(\neg\neg S_1^\circ; S_2^\circ) \cup (\neg S_1^\circ; S_3^\circ)$
$(\texttt{let}\ v_1 = t_1 \cdots v_n = t_n\ \texttt{in}\ S)^\circ$	:=	$[t_1/v_1, \cdots , t_n/v_n]; S^\circ$
$(\texttt{not}\ S)^\circ$	:=	$\neg S^\circ$

1.6 The Role of Evaluation

Usually, theoretical studies are not enough to provide sufficient insight on the effectiveness and behavior of complex systems such as satisfiability solvers. For one thing, worst-case complexity analysis is never influenced by optimisations, which as we will see have a very strong influence on the behavior of satisfiability solvers. As a complement, then, empirical evaluations have to be used. In the area of propositional satisfiability checking there is large and rapidly expanding body of experimental knowledge; see e.g., [GvMW00]. In contrast, empirical aspects of modal satisfiability checking have only recently drawn the attention of researchers. We now have a number of test sets, some of which have been evaluated extensively [BFH+92, HS96, GS96, HS97, HPSS00]. In addition, we also have a clear set of guidelines for performing empirical testing in the setting of modal logic; these were proposed by Horrocks, Patel-Schneider, and Sebastiani [HPSS00], building on work by Heuerding and Schwendimann [HS96]. We contend that empirical testing is an integral part not only of the design and evaluation of theorem provers, but also of the tests themselves, and can (and should) strongly influence the development of both.

We will now start Part I, with an overview of empirical evaluation in modal and hybrid logics.

Part I

Evaluation in Modal and Hybrid Theorem Proving

Here we will survey current and past methods of evaluating modal and hybrid theorem provers, as well as some ways of approaching the satisfiability problem in modal and hybrid logics. We will introduce first the *indirect* method: translating formulas from our logic into FOL, and then performing resolution on the translated formulas, and then the *direct* method: developing our own theorem prover. Also, we make explicit the role of empirical evaluation in the development of theorem provers.

Chapter 2

How Long is a Ruler?

The police protects us from the bandits.
Who protects us from the police?

Comparing theorem provers

Our aim in this chapter is to discuss empirical evaluation methods for modal theorem provers, and see if an evaluation method for *hybrid* logic provers can be developed, given the strong link between modal and hybrid logic. Now, empirical comparison of theorem provers is conceptually simple: given a representative sample of the problems they are meant to solve, a criterion for comparison is established such as mean run time, and the performances are compared. However, some complications arise when trying to define what 'representative' problems are, and perhaps 'real life' problems are too few or still too difficult. In that case, artificial problems must be supplied, and there are several criteria that the test sets must comply with. Since our goal is to develop an evaluation method for hybrid logics, it's only natural that we study the existing efforts for modal logic. Heuerding and Schwendimann [BHS00] stated a set of criteria for evaluating modal theorem proving benchmarks, which was later expanded by Horrocks, Patel-Schneider and Sebastiani [HPSS00]. We will start, in Section 2.1 by giving an overview of these criteria. We will then review the existing modal test sets, particularly with respect to these criteria, in Sections 2.3, 2.4 and 2.5. Finally, we will discuss a new test methodology for hybrid logic, also in Section 2.5.

2.1 Fitness criteria for modal test sets

To be able to assess the quality of test methodologies for modal theorem proving, we will review a number of 'common sense' criteria that have been proposed in the literature.

Reproducibility. Reproducibility of experiments is fundamental in science; anybody should be able to run the same experiment to confirm the result. Applied to theorem proving, this means that the formulas used, or the algorithm to generate them, must be made available. In the case of random generation, this would include the 'seeding' of the random generator. Also, if the generating algorithm is provided, variants of the test can be developed, for example to extend the target logic [AH03].

Representativeness. Ideally, a test set should cover as much as possible of the input space, and span the whole range of sources of difficulty. Of course, there is no complete catalogue of sources of difficulty, so a test set should at least cover a large area of inputs. If the problems are limited to a narrow area of the input space, we run the risk of not assessing the real capabilities of the provers if they are to be run on arbitrary formulas.

Valid vs. not valid balance. Uncertainty with respect to the satisfiability of the formulas in the test should be maximum: the provers should not *a priori* have any information as to whether the formula is satisfiable or not, and furthermore there should be about as many satisfiable as unsatisfiable formulas in the set; satisfiable and unsatisfiable formulas might present different sources of difficulty, and we want a fast answer from our prover in either case.

Difficulty. The set should provide a challenge to the provers being tested; if the problems are too easy, the resource consumption will reflect mostly startup costs, which do not scale with problem difficulty. Also, some problems should be too hard for the current provers: as the proving techniques evolve, this helps the test remain current.

Termination. The test should terminate in a reasonable amount of time, with a meaningful result. If all inputs are too hard, there will be no information gained even if the benchmark can be run in a short time.

These criteria give rise to the following, more specialized considerations:

Parameterisation. One way to achieve a good coverage of the input space is to make the generating algorithm accept parameters that allow the problems to span large areas of the input space. There should be enough parameters to allow for a good coverage, but not so many that covering a specific part of the input space would take an inordinate amount of experiments.

Control. It is very useful for the generating algorithm to have parameters that control monotonically features of the problems like valid/not valid balance, modal/propositional balance, difficulty, etc. Monotonicity is very important: it allows us to leave out uninteresting areas of the input space, and to control the problem features independently of other parameters.

Modal vs. propositional balance. A modal prover should be adept at both propositional and 'purely modal' reasoning tasks; therefore, a test set should provide enough challenge for both aspects of modal reasoning.

Data organization. It should be possible to summarize the results of the benchmark, and to plot them to see the qualitative behavior of the evaluated provers.

Focus on narrow problems. Special ad-hoc sets may serve to measure the behavior of the systems with respect to specific difficulty sources; even though they do not provide a complete assessment of the capabilities of theorem provers, they are a good complement of a test set that spans large areas of the input space.

Redundancy. Ideally, many of the formulas in a complex problem should play a part in determining its satisfiability status; that is, it should not be decided by a small subset of the formulas. While a solver that recognizes redundancy in a set is desirable, redundant problems should not be a significant part of the test suite, as they can be rendered trivial by the handling of the redundancy.

Triviality. When a small part of a formula dictates the satisfiability of the whole, independently of the rest, the formula is said to be trivial. Trivial problems should not be a significant component of the test set, even if recognizing trivial problems is of course a desirable capability of theorem provers.

Artificiality. If there is an application in mind for the systems, problems generated should be of a similar nature to those coming from application inputs. Otherwise, the results of the test may not reflect the suitability of the systems for the task at hand. Note that 'real life' problems might not fulfill *any* of the other criteria, and indeed a specific system might be the best for the problem type at hand, and not for the general case.

Size. The problems should not be too big with respect to their difficulty; we are not as interested in processing of big files as in algorithm efficiency.

2.2 Real-world Problems

When the logic at hand is used for real-world applications, there is a source of problems whose representativity cannot be contested. They are, after all, the problems the provers should excel at, if we want them to be useful as well as interesting objects of study. Common downsides of this kind of input often include not having enough real-world problems to provide sufficient testing, and that as provers advance old inputs usually become trivial to solve.

In the following sections, we will come back to these criteria as we discuss the merits of the different test methodologies.

2.3 Hand-tailored Problems

The Balsiger, Heuerding and Schwendimann test set

The Balsiger, Heuerding and Schwendimann test set [BHS00] was used in the TANCS '98 comparison, and represents one of the first attempts at having a comprehensive test set for the comparison of modal theorem provers. It consists of nine classes of provable formulas and nine classes of unprovable formulas, parameterized by a number in $\mathbb{N}$. The performance score of a prover in each class is given by the highest numbered problem in that class that the prover can solve in less than 100 seconds. There are nine different types of problem, each with both a satisfiable and an unsatisfiable class associated to it. The purpose of parameterization was to have a test set which could present harder problems as provers became more advanced; the complexity of each formula in a class is expected to be exponential on its parameter. There is a base problem for each class, which is then made more complex using several techniques, and there was an effort to make the problems resistant to simple optimization.

However, dramatic advances in the field yielded provers which could solve any formula in most of the categories; the increase in complexity from instance to instance was not exponential any more [HPSS00]. Nevertheless, the test remains very useful for development of modal theorem provers, as it gives a quick way to evaluate improvements to the program, and performance in the different classes might confirm whether optimizations work as planned or not.

Extending the set for hybrid logics Extending the test set to create hybrid formulas is in principle possible, but as we have seen, there would be a poor coverage of the problem space and the tricks to 'hide' the formulas are sooner or later rendered harmless by optimization.

2.4 Random Problems: Modal QBF

Other than the previously described test set, all empirical test sets for modal logic are parameterised random formula generators. The first random generation technique used in testing modal decision procedures, the random $3\mathrm{CNF}_{\Box_m}$ test methodology, was proposed in [GS96]; its subsequent development is described in, for instance, [HPSS00], and the latest version is presented in [PSS03]. After having gone through a series of revisions, this methodology is considered to be well understood. In between revisions of $3\mathrm{CNF}_{\Box_m}$, the random modal quantified Boolean formula test set was proposed by Massacci [Mas99], and used in the 1999 and 2000 editions of the TANCS system performance comparisons [TAN]. We'll examine the Modal QBF set now, and the 3CNF test set family in Section 2.5.

2.4.1 The Random Modal QBF Test Set

The random modal QBF test set is based on the idea of randomly generating quantified boolean formulas (QBFs) and then translating these into modal logic. Let us explain these two steps in more detail.

Generating QBFs

Recall that QBFs have the following shape [GJ79]: $Q_1 v_1 \ldots Q_n v_n \; \mathit{CNF}(v_1, \ldots, v_n)$. That is, QBFs are prenex formulas built up from proposition letters, using the booleans, and $\forall v\, \beta$ and $\exists v\, \beta$ (where v is any proposition letter).

What is involved in evaluating a QBF? We start by peeling off the outermost quantifier; if it's $\exists v$, we choose one of the truth values 1 or 0 and substitute it for the newly freed occurrence of v; if it's $\forall v$, substitute both 1 and 0 for the newly freed occurrences of v. In short, while evaluating QBFs we are generating a tree, where existential quantifiers increase the depth, and universal quantifiers force branching.

In the *random modal QBF test set*, 4 parameters play a role: c, d, v, k:

- The parameter c is the number of clauses of the randomly generated QBF.
- The parameter d is the alternation depth of the randomly generated QBF; it is *not* the modal depth of the modal translation. (More on this below.)
- The parameter v is the number of variables used per alternation.
- And k is the number of different variables used per clause.

The *QBF-validity problem* is the problem of deciding whether a QBF without free variables is valid; it is known to be PSPACE-complete [GJ79]. For every fixed valued of d we can capture the problems in Σ_d^P in the polynomial hierarchy; PSPACE can only be reached by an unbounded value of d.

Here's a concrete example. Using $d = 3$ and $v = 4$ we can generate

$$\underbrace{\forall \overbrace{v_{34}v_{33}v_{32}v_{31}}^{4} \exists \overbrace{v_{24}v_{23}v_{22}v_{21}}^{4} \forall \overbrace{v_{14}v_{13}v_{12}v_{11}}^{4} \exists \overbrace{v_{04}v_{03}v_{02}v_{01}}^{4}}_{3} \; CNF(v_{01},\ldots,v_{34}).$$

Each clause in $CNF(v_{01},\ldots,v_{34})$ has k different variables (default 4) and each is negated with probability 0.5. The first and the third variable (if it exists) are existentially quantified. The second and fourth variable are universally quantified. This aims at eliminating trivially unsatisfiable formulas. Other literals are either universal or existentially quantified variables with probability 0.5. The depth of each literal is randomly chosen from 1 to d.

By increasing the parameter d from odd to even, a layer of existential quantifiers is added at the beginning of the formula, and, conversely, when d increases from even to odd, a layer of universal quantifiers is added. The impact of increasing d on the shape of the QBF trees may be visualized as in Figure 2.1, for the case where $v = 2$.

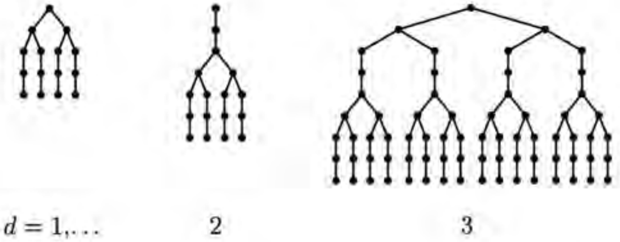


Figure 2.1: The shape of QBF trees for $v = 2$

Translating QBFs into Modal Logic

The QBF that is produced by the random generator is translated into the basic modal logic as introduced in 1.4.3, using a variant of an encoding that is originally due to Ladner [Lad77]. The core idea underlying the translation is to capture, by means of a modal formula, the 'peel off quantifiers and substitute' evaluation process for a given input QBF. The translation forces branching in the structure of the possible model whenever a universal quantifier is found in the original formula, keeps the branches separate, and makes sure there are enough modal levels in the model. It forces the structure of the possible model to be a tree, and the resulting formula is satisfiable iff the original formula is.

Here's a detailed example. The formula $\phi = \forall v_4 v_3 \exists v_2 v_1 \, (v_1 \vee \neg v_2 \vee \neg v_3 \vee \neg v_4)$ (generated with parameters $v = 2$, $d = 1$, $c = 1$, default encoding) translates into the conjunction of the following formulas.

- The matrix ϕ must be true everywhere in the model: $\bigwedge_{m=1}^{4} \Box^m(v_1 \vee \neg v_2 \vee \neg v_3 \vee \neg v_4)$, where $\Box^m$ is a sequence of m occurrences of the $\Box$ operators.
- Keep values of proposition letters forever, adding one per level, in order of quantifier appearance:
  - $\bigwedge_{m=1}^{3}(\Box^m(v_4 \vee \Box(\neg v_4)) \wedge \Box^m(\neg v_4 \vee \Box(v_4))) \wedge$
  - $\bigwedge_{m=1}^{2}(\Box^m(\Box(v_3 \vee \Box(\neg v_3))) \wedge \Box^m(\Box(\neg v_3 \vee \Box(v_3)))) \wedge$
  - $\Box(\Box(\Box(v_2 \vee \Box(\neg v_2)))) \wedge \Box(\Box(\Box(\neg v_2 \vee \Box(v_2))))$
- Force branching on universally quantified variables: $\Diamond v_4 \wedge \Diamond \neg v_4 \wedge \Box(\Diamond v_3) \wedge \Box(\Diamond \neg v_3)$.
- Force tree depth (note that the first two levels are covered by the previous two formulas): $\Box(\Box(\Diamond(\top))) \wedge \Box(\Box(\Box(\Diamond(\top))))$.

The parameters c, k, v and d that are used in the generation process are related to the final modal formula in the following way. The (maximum) number of clauses is $c \cdot k + (v \cdot (d+1))^2 + \lfloor v \cdot (d+1)/2 \rfloor$. The (maximum) number of proposition letters is $v \cdot (d+1)$. And the (maximum) modal depth is $v \cdot (d+1)$. These maximums obtain when c is high enough compared to $v \cdot (d+1)$ to cover all the possible proposition letters. The file size for the translated formula is linear in c, and polynomial in v and d, but usually we are not interested in very big values of the last two, so this is not much of a problem.

Fitness of the test set

Some of the fitness criteria (reproducibility, representativeness, parameterisation for example) can be evaluated by an analysis of its description; others like difficulty, termination or size require empirical testing. We benchmarked a few theorem provers, aiming not to evaluate the state of the field (we left some prominent systems out, for example) but to evaluate the test set itself.

Settings

To evaluate the QBF test set, we used 3 satisfiability solvers for modal logic. First, we used the general first order prover SPASS [SPA], version 1.0.3, extended with the layered translation of modal formulas into first order formulas as presented in [AGHdR00]. Second, we used MSPASS version V 1.0.0t.1.2.a [MSP]. And, third, we used *SAT version 1.3 [Tac99].

Our experiments were run on a Pentium III 800 MHz with 128 MB of memory, running RedHat Linux 7.0.

For our measurements we had to translate the modal QBF files to the formats of the various provers we were using, and in one case we were also were converting the formulas from modal to first order logic. We checked that the resulting file sizes were linear in c, even though the linear coefficient varied from one solver to another.

Our main measurements concerned both CPU time elapsed (with a 10800 second timeout) and a time independent measure: the number of clauses generated for SPASS plus layering and for MSPASS, and the number of unit propagations for *SAT.[1]

Findings

We first ran the standardized tests provided by the TANCS competition: 64 instances randomly generated with $c = 20$, $v = 2$, $d = 2$, and default settings for the remaining parameters. See Figure 2.2.

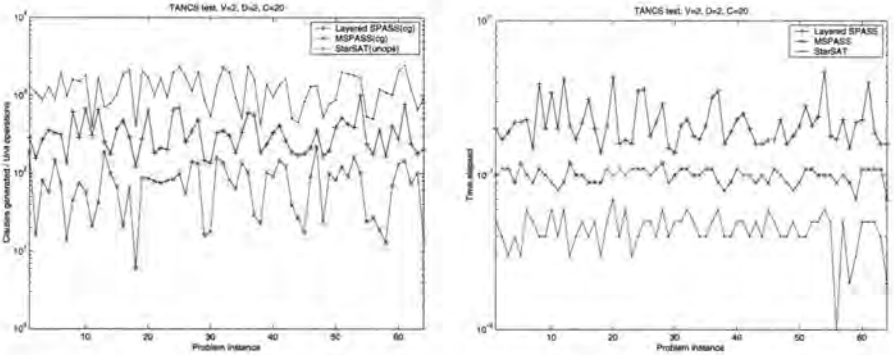


Figure 2.2: The standardized tests provided by TANCS, used for SPASS, MSPASS, and *SAT. (Left): clauses generated/unit propagations per problem instance, log scale. (Right): CPU time (seconds) per problem instance, log scale.

While the number of clauses generated by resolution provers and the number of unit propagations in *SAT are not directly comparable as a performance measurement, they do give an indication of the relative difficulty of a problem (or problem set). As such, we can see that the difficulty of a problem varies with the method used to solve it. The correlation between time elapsed and clauses generated/unit propagations varies widely between the methods. In fact, for this test the *SAT times are so low as to be completely dominated by startup costs and don't really inform us about relative problem difficulty.

Next we ran a number of sweeps, with each of three provers, with $v = 2$ and increasing d from 1 to 4 (and to 5 in the case of *SAT), while increasing

[1] Unit propagations came out to be a less than perfect indicator of resource consumption in the general case, although for this benchmark it was roughly as informative as the number of assignments found.

c from 1 to 100 (or to the maximum number of clauses allowed by the rest of the parameters, whichever was lower). The resulting CPU times and the number of clauses generated/unit propagations are depicted in Figure 2.3; the curves for $d = 1$, $d = 2$ do not extend to the right-hand side of the plots, as the formulas being generated with these settings are simply too small to be be able to accommodate a larger number of different clauses.

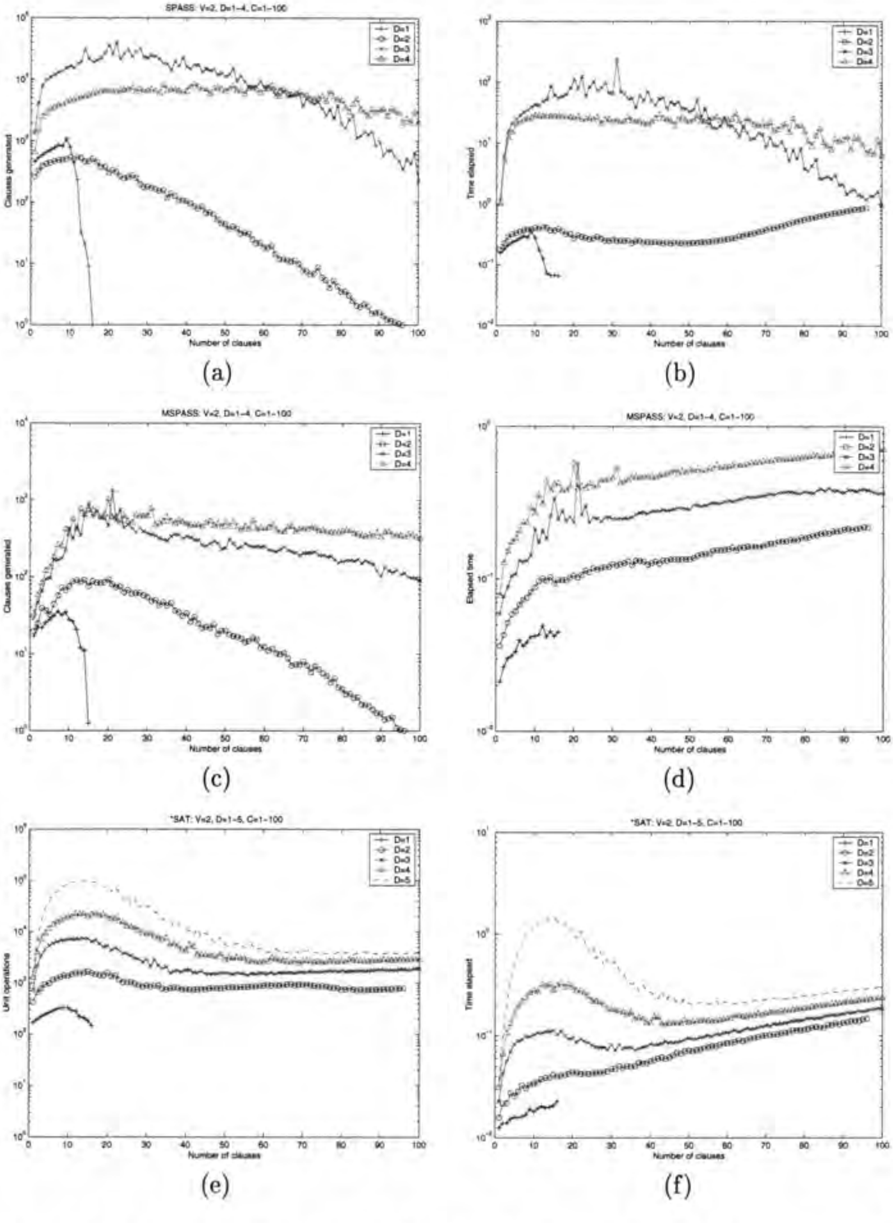
Several things are worth noting about Figure 2.3. First, the sets display an easy-hard-easy pattern familiar from propositional satisfiability testing [GvMW00]. The shape of the curves is strongly dependent on the solver used. Moreover, the patterns seem to vary from not-too-hard-hard-easy in some cases (SPASS, $d = 1$, $d = 2$, $d = 4$) to not-too-hard-hard-hard in others (SPASS, $d = 4$; MSPASS, $d = 3$, $d = 4$) to not-too-hard-hard-not-too-hard in yet others (SPASS, $d = 3$;*SAT, $d = 2$, $d = 3$, $d = 4$, $d = 5$).

Second, for both SPASS and MSPASS we see that curves cross each other; this is most clearly visible in (a), where the number of clauses generated by SPASS are displayed,but it also shows up in (b) where the CPU times for SPASS are shown. Hence, for SPASS (and to a lesser extent for MSPASS) the d parameter does not influence the difficulty of the problems being generated in a monotonic way.

Third, the time elapsed (displayed in (b), (d), and (f)) has a very strong dependence on file size: after the hard region has been crossed and the elapsed time tends to decrease, it actually starts going up again. The impact of input file size and I/O is most noticeable for MSPASS (plot (d)); but even in the case of *SAT, where the number of unit propagations remains more or less constant after the hard region has been traversed, the CPU times start going up: this increase is entirely due to input file size and I/O[2]. In Figure 2.4 we have plotted the growth of the input file size against c, and against d. The file size can be approximated by $11500+c*485$, while the preprocessing performed by the layered translation brings this up to $20000 + c * 930$. Remarkably, the translated file for MSPASS is *smaller* than the original input file. (For the purposes of illustration, we have also indicated what the input file size would be for the first order prover BLIKSEM [Bli].)

When we increased the v parameter, we saw similar curve shapes as for $v = 2$. In Figure 2.5 we have displayed results of running *SAT with $v = 2$ (top) and with $v = 3$ (bottom). Notice that the humps indicating the hard regions are higher for $v = 3$ than for $v = 2$, indicating that the problems are harder; hence, the CPU times are not as strongly dominated by file size and I/O aspects as in the case where $v = 2$. The fact that the hard regions are 'wider' than for $v = 2$ indicates that we are not only getting harder problems, but also that the fraction of hard problems is increasing.

[2] Our filesystem runs over a network: performance in local filesystems is very likely to be much better.



(a) (b)

(c) (d)

(e) (f)

Figure 2.3: SPASS, MSPASS, and *SAT on QBF test sets, $v = 2$, $d = 1 \ldots 4$ (5), 64 samples/point. (Left): clauses generated/unit propagations, log scale. (Right): CPU time in seconds, log scale.

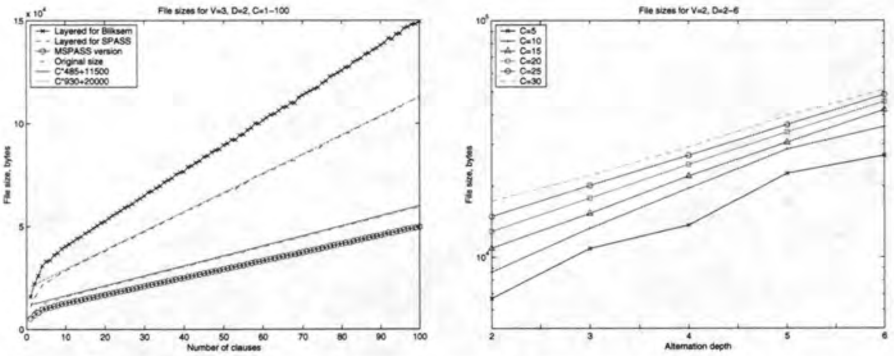
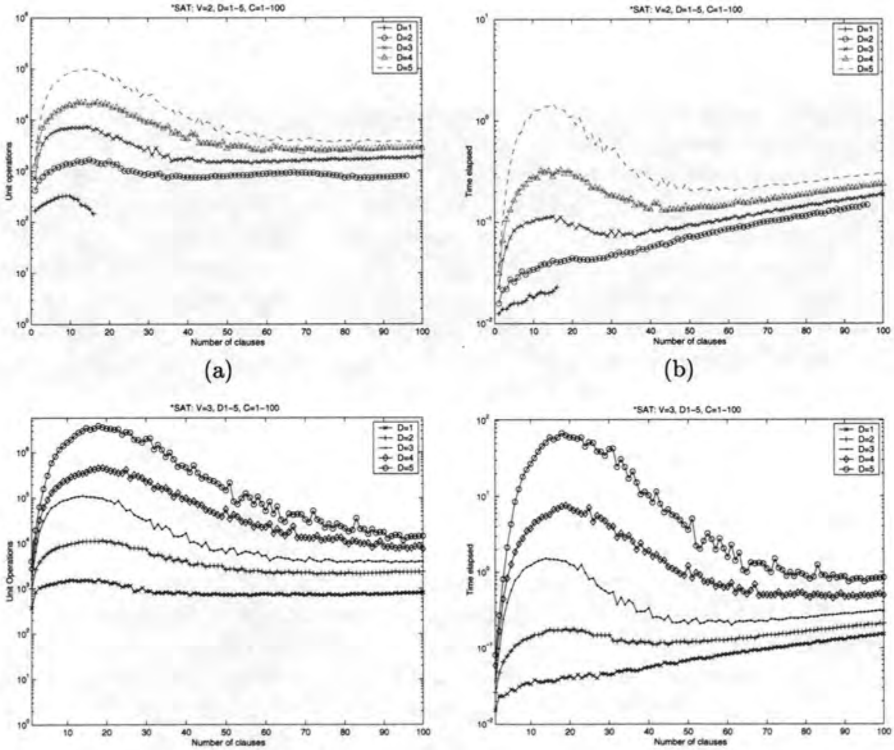


Figure 2.4: Size of the input files. (Left): as a function of the number of clauses. (Right): as a function of the alternation depth.



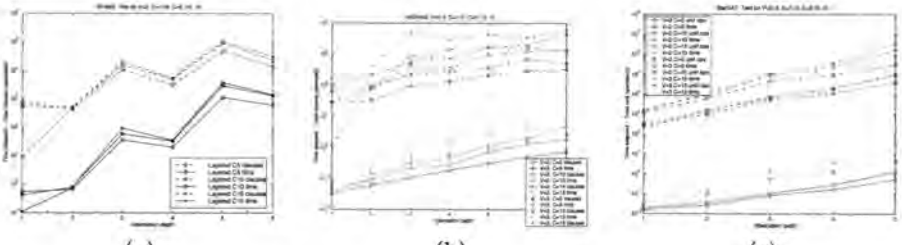
(a)

(b)

(c)

(d)

Figure 2.5: *SAT results for $v = 3$, $d = 1 \dots 5$, 64 samples/point. (Left): unit propagations, log scale. (Right): CPU time in seconds, log scale.



(a) (b) (c)

Figure 2.6: SPASS, MSPASS and *SAT results for $v = 2$ (3), $d = 1 - 6$ (7), 64 samples/point. (a): SPASS with $v = 2$. (b): MSPASS with $v = 2$, 3. (c): *SAT with $v = 2$, 3. Log scales are used in (a), (b) and (c).

Let us return to the phenomenon observed in Figure 2.3, where it was found that the d parameter does not monotonically control difficulty. We can observe this even clearer when we plot d along the x-axis, as in Figure 2.6. Note that the phenomenon is strongly prover dependent: it clearly shows up for SPASS (with the layered translation) as shown in (a); it is somewhat visible with MSPASS (b), but not at all with *SAT (c). Further experimental work has shown that this 'staircase phenomenon' is also present with larger values of v for SPASS. The phenomenon is related to the special way in which QBFs grow: existential quantifiers are added to the original QBF when d is increased from odd to even, universal quantifiers when d is increased from even to odd; see Figure 2.1. The former simplifies matters for SPASS with the layered translation, while the latter makes matters considerably harder for that solver.[3]

One central concern with any test set, synthetic or not, is *parameterization*: to which extent can we choose the difficulty of the problem and of exploring the input space? In the QBF test set the difficulty can easily be controlled: the v parameter controls it monotonically, the d parameter also with some caveats. It seems, however, that v and d do not control truly *independent* dimensions of the problem space. More precisely, combinations of v and d for which the value of $v \cdot (d+1)$ coincides have very similar curves, as can be seen in Figure 2.7. This suggests that $v \cdot (d+1)$ is the dimension along which the QBF problem space should be explored, instead of either v or d independently. (As an aside, it is clear from Figure 2.7 that with increasing values of $v \cdot (d+1)$, the truly hard region for a given setting of parameters moves to the right as we increase the number of clauses.)

An important aspect that we have not discussed so far is the satisfiable vs. non-satisfiable fraction. The parameter c does indeed allow us to control the

[3] Note that the staircase phenomenon will not be observed if one only performs the standardized TANCS test as this test only involves a single value of d.

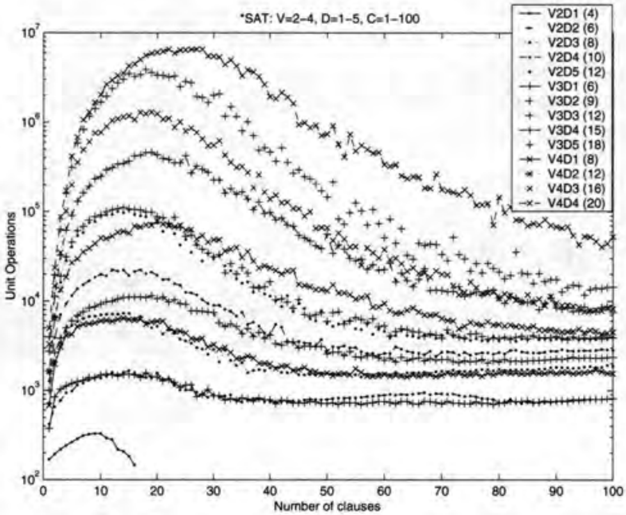


Figure 2.7: *SAT results for $v = 2$–4, $d = 1$–5, and $c = 1$–100, 64 samples/point. The numbers in brackets indicate the value of $v \cdot (d + 1)$.

satisfiability fraction: it goes from 1 to 0 monotonically with c. However, there are remarkably few values of c for which the satisfiable fraction is 1; see Figure 2.8. In line with Figure 2.8 (a), we have found satisfiable fractions of about 20% in many repeated runs of the standardized 20/2/2 TANCS test (see Figure 2.2). Moreover, there is a heavy 'tail' of unsatisfiable problems, as indicated by the curves in Figure 2.7. And contrary to intuition, the constrainedness of problems does not seem to depend very strongly on the d parameter; for a fixed v, increasing d from odd to even doesn't shift the satisfiable fraction graph by any noticeable amount. The constrainedness of the underlying models, then, remains unchanged despite the addition of more variables and the increase in depth.

Finally, recall that a modal formula is *trivially satisfiable* iff it is satisfiable

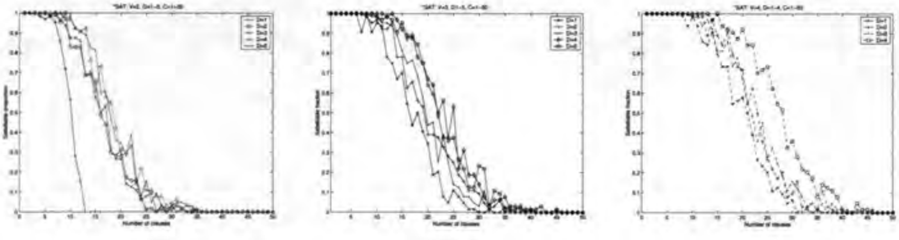


Figure 2.8: *SAT results for $d = 1$–4 (5), $c = 1$–50, 64 samples/point. (Left): Satisfiable fraction for $v = 2$. (Middle): Satisfiable fraction for $v = 3$. (Right): Satisfiable fraction for $v = 4$.

on a model with a single node [HPSS00, HS97]. Clearly, trivial satisfiability is not a problem for modal QBF test sets. Because of the highly structured form of the randomly generated QBFs, the resulting modal formulas always contain $\Diamond$-subformulas, thus avoiding trivial satisfiability.

Evaluating the evaluator

The general criteria for evaluating modal test methodologies put forward in Section 2.1, boil down to demanding a reproducible sample of an interesting portion of the input space with appropriate difficulty. To conclude this section, here's a brief discussion of these criteria as they relate to our setting.

By its very nature, *reproducibility* is guaranteed for the modal QBF test set. The modal QBF test set seems to represent just a restricted area of the whole input space; that is, it scores low on *representativeness.* There are three reasons for this. First, the QBF test set provides poor coverage of the satisfiable region; most of the hard modally encoded QBF-formulas generated with values of v and d that are within reach of today's tools, are unsatisfiable, as suggested by Figure 2.8. Second, the modally encoded QBFs are of a very special shape, which seems to lead to the so-called staircase phenomenon for some solvers. And third, the v and d parameters end up being substantially overlapping and interrelated as part of the translation of QBFs into modal formulas. A strong point in favor of the QBF test set is that it is possible to generate hard problems with a large modal depth which are still within reach of today's modal satisfiability solvers; in this respect the QBF random test methodology fares better than, for instance, the $3\mathrm{CNF}_{\Box_m}$ test methodology, as reported in [PSS03].

The levels of *difficulty* offered by the modal QBF test set are certainly sufficient, as they range from next to trivial to too hard for today's systems. Related to this, the tests *terminate* and provide information in a reasonable amount of time.

In conclusion, then, the random modal QBF test methodology provides useful test sets that should, however, not be used as the sole measure in the evaluation of modal satisfiability solvers.

Random QBF and Hybrid Logic. It would be possible to fit nominals and te @-operator into a random QBF translation framework, but there is no natural way to fit the $\downarrow$ operator. QBF does not actually capture the complexity class of the $\mathcal{H}(@, \downarrow)$ logic, which is undecidable, and we have just seen that the coverage of the input space is not so thorough even in the modal case. We have to keep looking.

2.5 Random problems: Random CNF

2.5.1 3SAT

The satisfiability problem for propositional logic has been widely investigated, since it has many applications such as timetabling, code optimization, or cryptography. It is known that random CNF clauses of three or more literals capture the complexity of the satisfiability problem for the logic [GJ79], and that random CNF problems of more than 3 literals can be linearly encoded into CNF formulas of exactly 3 literals each. Therefore, even though there are many real-world problems and test sets available for propositional logic, one of the best known and most used test sets for propositional logic is Random 3SAT: a conjunction of L clauses of 3 random propositional literals each, chosen from a set of N different propositional variables. Since Propositional 3CNF has become the *de facto* standard random test set for propositional satisfiability testing [GvMW00], developing a modal version of the test set has naturally received a lot of attention. We will see now how propositional 3SAT has been expanded into modal and hybrid CNF formula generation.

2.5.2 Random Modal CNF

In this test set, the formulas to be checked for satisfiability are randomly generated $\mathrm{CNF}_{\Box_m}$ formulas. A $\mathrm{CNF}_{\Box_m}$ formula is a conjunction of $\mathrm{CNF}_{\Box_m}$ clauses, where each clause is a disjunction of a certain number of either propositional or modal literals. A literal is either an atom or its negation, and modal atoms are formulas of the form $\Box_i C$, where C is a $\mathrm{CNF}_{\Box_m}$ clause. A $3\mathrm{CNF}_{\Box_m}$ formula is a $\mathrm{CNF}_{\Box_m}$ formula where all clauses have exactly 3 literals.

The latest version [PSS03] of this generator accepts five main parameters: the maximum modal depth D, the number of propositional variables N, the number of modalities m, the number of clauses L, and the probability p of an atom occurring at depths less than d being purely propositional. Although the usual number of literals per clause is 3, the generator gives a great degree of control over the clause size. In fact, both modal/propositional balance and clause size probability distributions can be specified either as constants or as a function of modal depth.

Given these parameters, a $\mathrm{CNF}_{\Box_m}$ formula of depth D is a set of L clauses, each made up of a number (chosen randomly according to the clause size probability distribution) of distinct modal CNF disjuncts, each consisting of either

- a proposition from the set $\{P_1, \ldots, P_N\}$, or
- (if $D > 0$) a disjunct $\Box_r C$, where $\Box_r \in \{\Box_1, \ldots, \Box_m\}$, and C is a $\mathrm{CNF}_{\Box_m}$ clause of depth $(D - 1)$.

The way this test method works is the following: all the parameters but L are fixed, and then a range for L is selected that covers the transition from 'only satisfiable formulas generated' to 'only unsatisfiable formulas generated'. For values of L covering this range, a tuple of the parameters' values is defined. A fixed number of formulas (usually a hundred or more) is generated and given as input to the prover under test, generally with a time limit. Satisfiability rates, median/90^{th} percentile of CPU time elapsed, and other possible indicators are plotted against either L or L/N.

The original $3\text{CNF}_{\Box_m}$ test set had a series of problems with respect to the fitness criteria we introduced [HS97, HPSS00]. One concerned *redundancy*: as the original generator did not check for repetition of propositional variables inside the same clause, the generated formulas could contain propositional tautologies. This made the effective size of a problem much smaller. The same problem was detected for the modal atoms [GGST98]. The other problem was *triviality*: for certain values of the generator's parameters, the formulas generated contained enough purely propositional clauses that they could be solved without recourse to modal reasoning. This methodology has now gone through a series of improvements, and is believed to be fully compliant with the fitness criteria.

CNF and hybrid logics The random modal CNF generation is very appealing to us as a method for generating hybrid formulas: it is simple to expand, its triviality issues are under control, and (at least for modal logic) it provides the most coverage of the input space. We decided to use the $\text{CNF}_{\Box_m}$ test set as a base for our hybrid test set;

We used the latest version of the generator [PSS03] to develop a test set for hybrid logics to benchmark the HyLoRes prover [AH02b].

2.5.3 Random Hybrid CNF

Why is a new test set necessary? In Chapter 4 we will introduce HyLoRes, a theorem prover for hybrid logics based on direct resolution [AH02b]. We made extensive use of empirical testing to evaluate our development work on the basic algorithm, but the available test sets were not sufficient to evaluate the prover on the aspect that was most distinguishing of HyLoRes, that is, its ability to deal with *hybrid* formulas. We had a few handcrafted hybrid formulas, but in order to do some exploration of the hybrid satisfiability space we needed a more thorough tool. We decided to expand the algorithm presented in the latest version of the modal CNF test set to generate hybrid logic formulas.

Basic Idea. We decided to make as few changes as possible to the algorithm described in [PSS03], and add the $@$, $\downarrow$ and A (universal modality) operators. This requires us to talk not about modal depth, but about *operator* depth, which is defined as the level of nesting of a specific operator, independently of the others. For example, the formula

$$@_{n_1}(p_1 \vee {\downarrow}x_1.(p_2 \vee \Diamond_1(x_1 \vee @_{x_1}n_1) \vee \Diamond_1(p_2 \vee p_3)))$$

has modal depth 1, @-depth 2, and ↓-depth 1. Also, instead of propositions we use *atoms* of hybrid logic, that is, propositions, nominals, and state variables.

Parameters. The program accepts as parameters:

- The maximum nesting of operators, D (generalizes modal depth)
- The number of propositional variables, nominals, and state variables, N_p, N_n and N_x
- The number of modalities, N_m
- The number of clauses, L
- The distribution of probabilities for clause size (a list $[f_1, \ldots, f_n]$, with f_i the relative frequency of clauses of size i)
- The probability for a disjunct of being non-atomic, p_{op}
- The relative frequencies of modalities, @-operators, ↓ operators, and the universal modality as main operator in non-atomic disjuncts, $p_{mod}, p_{down}, p_{at}$, and p_{univ}
- The relative frequencies of propositions, nominals and state variables when the disjunct is an atom, p_{prop}, p_{nom}, and p_{svar}
- The probability for any literal of appearing negated, p_{neg}
- The number of instances to generate, *numinst*

Given these parameters, a hybrid CNF formula of depth D is a set of L clauses, each made up of (a number between 1 and n chosen with relative frequencies $[f_1, \ldots, f_n]$ of) distinct hybrid CNF disjuncts, each consisting of either

- a proposition from the set $\{P_1, \ldots, P_{N_p}\}$, or
- a nominal from the set $\{n_1, \ldots, n_{N_n}\}$, or
- a state variable from the set $\{x_1, \ldots, x_{N_x}\}$, or
- (if $D > 0$)
  - a disjunct $\Box_r C$, where $\Box_r \in \{\Box_1, \ldots, \Box_{N_m}\}$, and C is a random hybrid CNF clause of depth $(D-1)$, or
  - a disjunct $@_n C$, where $n \in \{n_1, \ldots, n_{N_n}\}$, and C is a random hybrid CNF clause of depth $(D-1)$, or
  - a disjunct ${\downarrow}x_r$ *op* C, where $x_r \in \{x_1, \ldots, x_{N_x}\}$, *op* is one of $\{@, \Box, \mathsf{A}\}$ and C is a random hybrid CNF clause with depth $(D-1)$, or
  - a disjunct $\mathsf{A}C$, where C is a random hybrid CNF clause of depth $(D-1)$.

Algorithm Used. The algorithm used to generate the formulas is as follows:

```
gen_clauses(params)
    for i := 1 to L do $Cl_i$ := gen_cl(params);
    return ($\bigwedge_{i=1}^{L} Cl_i$);

gen_cl(params)
    nd := rnd_length(params.C);
    nop := rnd_numops(nd, params);
    Atoms := rnd_atoms(params, nd − nop);
    Ops := rnd_opsd(nop, params);
    OC := {};
    foreach $op_i$ in Ops
        OC := OC ∪ {$op_i$ (gen_cl(params{depth := depth − 1))}
    return($\bigvee$ OC ∨ $\bigvee$ Atoms);

rnd_numops(nd, params)
    if (params.depth = 0) then 0
    else rnd_fc d(nd, params.pop);

rnd_atoms(params, nat)
    if (nat = 0) then {}
    else Atoms := rnd_atoms(params, nat − 1);
          atom := rnd_atom(Atoms, params);
          return(Atoms ∪ atom);

rnd_ops(n, params)
    if (n = 0) then {}
    else Ops := rnd_ops(params, n − 1);
          op := rnd_op(params);
          return(Ops ∪ op);
```

Figure 2.9: Test generation structure

The outline of the algorithm used to generate hybrid CNF formulas is given in Figure 2.9. The function **rnd_atom**(*Atoms*, *params*) returns a random atom not in the set *Atoms*, respecting the relative frequencies of the different types of atom as given in *params*. **rnd_fc**(*nd*, *params.pop*) takes as arguments the number of disjuncts *nd* and the proportion of non-atomic disjuncts in a clause, *params.pop*. If $prop = nd \cdot params.pop$ is an integer, it returns *prop*, otherwise it returns $\lceil prop \rceil$ with probability $prop - \lfloor prop \rfloor$, or $\lfloor prop \rfloor$ otherwise (probability $\lceil prop \rceil - prop$).

This prevents the accidental creation of clauses in which all disjuncts are atomic, which has been a source of triviality in early modal CNF test set generators [HS97, HPSS00, PSS03]. **rnd_op**(*params*) returns an operator according to the relative frequencies stated in *params*, optionally enforcing maximum nesting per operator. A special case is the $\downarrow$ operator, which always precedes another operator; the reason is explained later.

Differences with the random modal CNF generator. In the presence of multiple modalities, the satisfiability operator and the universal modality, the notion of modal-depth becomes rather involved. In hGen, we work instead with a global notion of depth defined as operator nesting (this together with the probabilities for each operator, allows strict control over the generation of formulas for fragments of $\mathcal{H}(@, \downarrow, \mathsf{A})$ defined in terms of operator nesting). Clause size probability distribution is kept constant. This departs from the generator presented in [PSS03]: in that generator, it is possible to select a different clause size distribution and modal/propositional balance for each modal depth; we are not convinced such a feature can be meaningfully generalized to hybrid logic in a practical way. We calculate the maximum nesting per operator from its probability of appearance and the total depth D; whether the calculated depths should be enforced or not can be set from the command line. Since we're generating binders and variables, we ensure that every appearing variable is bound, and we force bound variables to appear.

New redundancy sources. The extended expressivity of the target languages that hGen can handle introduces new redundancy sources; the following cases are handled by hGen.

For all ϕ, $\downarrow x_i.(x_i \vee \phi)$ is a tautology, and conversely for all ϕ, $\downarrow x_i.(\neg x_i \vee \phi)$ is equivalent to $\downarrow x_i.\phi$. Such formulas are never generated by hGen. Moreover, the $\downarrow$ operator does not cause its argument clause to be evaluated at another element in the model, allowing for formulas of operator depth > 0 that still require no model exploration. hGen introduces $\downarrow$ only in expressions of the form $\downarrow x_i(\neg)\Box_j\phi$, $\downarrow x_i(\neg)@_{n_j}\phi$, or $\downarrow x_i(\neg)\mathsf{A}\phi$. Otherwise the clause would be equivalent to one in which all the atomic disjuncts are outside of the scope of the $\downarrow$ (since we're banning the bound variable from appearing at the same level it is bound in), effectively altering the clause size. There are two cases to consider when we want to place the $\downarrow$ operator, the difference being whether we are enforcing maximum nesting to be per operator or global: if global, then no further considerations are necessary, but if the maximum nesting is enforced per operator, then it can happen that all possible occurrences of $@$, $\Box_j$ and A have already appeared when we select the $\downarrow$, in which case it will be replaced by an atom.

With respect to the $@$ operator, for any ϕ, $@_{n_1}(n_1 \vee \phi)$ is a tautology, and $@_{n_1}(\neg n_1 \vee \phi)$ is equivalent to $@_{n_1}\phi$. Again, such formulas are not generated by hGen: when generating the argument clause for an operator of the form $@_n C$, the nominal n is never chosen by **rnd_atoms**.

Implementation. hGen is implemented in Haskell; it can be compiled with GHC 5.04 [GHC]. The use of random generators (an eminently imperative task) in the context of a purely functional language is transparently handled through a `State` monad; we keep the random seed as the state, and all functions that need to generate random numbers are monadic. See [Wad95] for more about monads in functional programming.

Using hGen as a modal test set. Since we are extending the language of the formulas generated with the modal CNF algorithm, it is important to verify that constraining the generator to modal formulas produced similar results to those obtained with the generator from [PSS03]. We decided to run a series of benchmarks and see if the results compared, in terms of mean difficulty, location of the easy-hard-easy pattern, and shape of the satisfiability fraction plot.

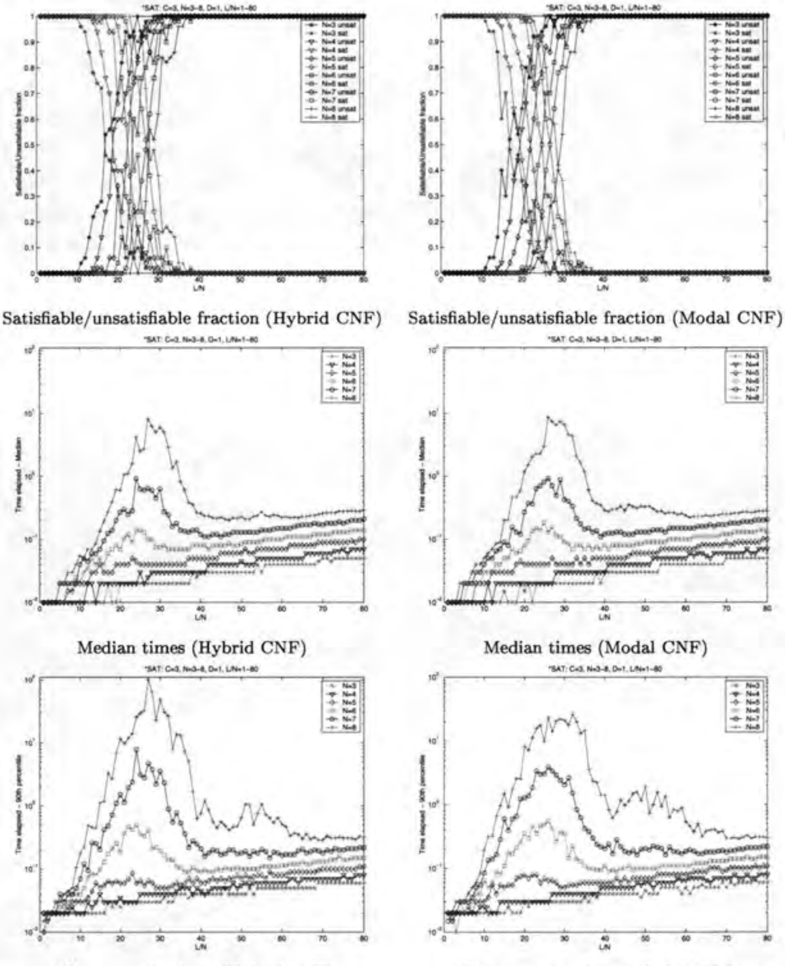
Experimental setting. We used a 1.6 GHz Pentium 4 computer running Linux Red Hat 7.3 for the tests, and fixing all the parameters but L, we ran the tests for L/N going from 1 to 80, with 50 instances per data point, for N going from 3 to 8. Modal depth was fixed at 1. We set the parameters of the generator to only produce modal formulas, and checked whether the runs showed any variations with respect to runs of the Modal CNF test set for equivalent parameter sets. The prover we used for this benchmark was *SAT [*SA]; we ran the tests with a timeout of 300 seconds.

Results. The results are displayed in Figure 2.10. The first row displays the satisfiable/unsatisfiable fractions; the second row shows the median of the CPU time used for every data point, and the third row shows the 90^{th} percentile of the CPU times. The experiment confirmed that, for equivalent parameter sets, the behavior of both test sets was very similar, in terms of location of the satisfiable/unsatisfiable transition and overall difficulty[4]. We are aware that the number of problems per data point (50) is not the best, and maybe 100 samples per data point would give more accurate results and smoother curves; this can be considered preliminary testing.

Of course, the Modal CNF test set allows for specification of clause size probability distribution and modal/propositional balance as a function of modal depth, while the hybrid CNF generator only accepts constant distributions, so the relationship between the test sets is more one of overlap than one of inclusion. One intriguing thing that can be seen in the 90^{th} percentile graphs is a second "hump"

[4] Our filesystem is networked, which means it takes longer for files to load; this accounts for the steady increase in solve times as a function of L/N. We apologize for the inconvenience

in the graph, for $N = 8$, around $L/N = 50$, in both plots. We plan to further investigate the phenomenon.



Satisfiable/unsatisfiable fraction (Hybrid CNF) Satisfiable/unsatisfiable fraction (Modal CNF)

Median times (Hybrid CNF) Median times (Modal CNF)

90^{th} percentile times (Hybrid CNF) 90^{th} percentile times (Modal CNF)

Figure 2.10: Results of the comparison between Hybrid and Modal CNF

2.6 Conclusion

We have given an overview of the different empirical test methodologies for modal theorem provers, and we have seen that since there are many criteria, each test set has its own place. The Heuerding and Schwendimann test set focuses on narrow problems and can be used with developing modal theorem provers, although it cannot distinguish between mature provers; the random modal QBF test set provides nontrivial doable problems of a good modal depth, however the coverage of the input space is poor, and the results obtained with it might not carry over to other areas of the input space; finally, the random $\mathrm{CNF}_{\Box_m}$ generator can produce formulas that range all over the input space, and we think is the one to use for empirical comparison of mature modal theorem provers. We will be using all the modal test sets, however, to estimate the relative merits of the different translations from modal logic into FOL in Chapter 3. We also introduced a new test set generator, based on modal $\mathrm{CNF}_{\Box_m}$ that produces random *hybrid* CNF formulas; this test set will be useful for testing HyLoRes in Chapter 4.

Chapter 3

Modal Theorem Proving: Translations into First Order Logic

Not knowing is like not seeing.
–Old Spanish proverb

3.1 Introduction

For many years, the main logic used in automated theorem proving has been classical logic. However, as we have seen in Chapter 1, for some applications other logics may be more suitable, be it because they express more naturally the concepts at hand, or because the full expressive power of classical logic is not needed, or not sufficient. Sometimes, then, we want to work with other logics; we have in that case the choice of developing tools which are specific to the logic, usually from scratch, or take advantage of the wealth of tools available for classical FOL, if a suitable translation from our logic exists. Broadly speaking, there are three general strategies for modal theorem proving: (1) develop purpose-built calculi and tools [PS98, *SA]; (2) translate modal problems into automata-theoretic problems, and use automata-theoretic methods to obtain answers [PSV02]; and (3) translate modal problems into first order problems, and use general first order tools [MSP]. The advantage of indirect methods such as (2) and (3) is that they allow us to re-use well-developed and well-supported tools instead of having to develop new ones from scratch.

In this chapter we focus on the third option: translation-based theorem proving for modal logic, where modal formulas and reasoning problems are translated into first order formulas and into reasoning problems to be fed to first order theorem provers. Since most of the state of the art first order theorem provers are based on resolution, one aspect we will pay particular attention to is the interaction between the translated formulas and the mechanics of resolution-based theorem proving. The rest of this chapter is organized as follows: first, we will introduce the basics of resolution-based theorem proving; then, we will recall the relational translation from modal logic to FOL [vB83], and we'll show how we run into trouble if we want to perform resolution on the resulting formulas. Then we will discuss an alternative strategy for improving the effectiveness of FO provers on (translated) modal input:the functional translation [ONdRG00], which has been integrated with the SPASS first order theorem prover, resulting in the MSPASS theorem prover [MSP]. Finally, we will introduce an improvement on the relational translation: the layered translation [AGHdR00], and show its effects on resolution-based theorem proving.

3.2 Resolution Theorem Proving in a Nutshell

Resolution theorem proving was invented by Robinson [Rob65]; the basic idea behind it is to derive new formulas from a set of given ones, by applying certain *inference rules*, in the hope of arriving at a contradiction. We refer to [BG01] for a detailed exposition. When no more formulas can be inferred, and a contradiction has not been derived, the conclusion is that the formula is satisfiable; we have then arrived at a *saturation*. When implementing a resolution-based theorem prover, a key problem that has to be solved is finding a good strategy for choosing, at each step, which formulas to process and which inference rules to use in order to minimize the search space.

The resolution rule. The resolution principle for propositional logic is stated as follows:

$$\frac{A \vee B \quad \neg A \vee C}{B \vee C}$$

The rationale for the resolution rule is that for both $A \vee B$ and $\neg A \vee C$ to be true in the same model, either B will have to be true (when A is false) or C will have to be true (when A is true). Since A will be either true or false, we can infer $B \vee C$ (the *resolvent* of $A \vee B$ and $\neg A \vee C$) from these premises. This extends to first order logic in the following way:

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

where σ is the mgu of the atomic formulas A and B, and *factoring*:

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

where σ is the most general unifier (mgu) of the atomic formulas A and B. Formulas are assumed to be in clause form. That is, a conjunction of *clauses*, which are defined as quantifier-free disjunctions of *literals*. A literal is an expression A (*positive* literal) or $\neg A$ (*negative* literal), where A is an atomic formula, or *atom*. An atom is an expression $P(t_1, \ldots, t_n)$, where P is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms. All variables are implicitly universally quantified; any variables originally existentially quantified are replaced by *skolem terms*; that is, terms of the form $f(x_1, \ldots, x_n)$, where f does not otherwise appear in the formula and $x_1, \ldots, x_n$ are universally quantified at the position where the existential quantifier appears. The search for a contradiction consists of systematically applying the inference rules until either a contradiction is found or no further rules can be applied. Resolution with factoring is refutationally complete for first order logic without equality; that is, a contradiction can be inferred from any unsatisfiable set of clauses [BG01].

Reasoning with equality. Improving the behavior of resolution-based methods with respect to the equality predicate has naturally received a lot of attention, given the relevance of reasoning with equality in mathematics, logic and computer science. In principle, a set of formulas can be expanded with a series of axioms about equality to ensure the properties of equality are respected (monotonicity, symmetry, transitivity, reflexivity), but this usually results in the generation of excessive numbers of unnecesary clauses. Robinson and Wos [RW69] discovered another way of dealing with equality: treating it as part of the logical language, and developing dedicated inference rules for first order logic with equality. An example of this is the *paramodulation* rule:

$$\frac{C \vee s \simeq t \quad D}{(C \vee D[t]_p)\sigma}, \quad \text{if } \sigma = mgu(s, D|_p),$$

where $D|_p$ is the subterm of D at position p, and $D[t]_p$ denotes the result of replacing in D this subterm by p. The addition of paramodulation to resolution and factoring has been proved refutation complete, under the presence of the reflexivity axiom $x \simeq x$ [Bra75].

Given that all the rules presented generate new clauses, therefore extending the search space, a very important aspect of resolution theorem proving is doing an efficient search. Also, some of the generated clauses will be redundant, and some will make preexisting clauses redundant. Accordingly, for most refutational provers, a substantial part of the program is devoted to guiding the proof effort and discarding redundant clauses to prune the search space. While the worst-case complexity is not affected, the average case performance gains can be dramatic [BG01, Vor01].

3.3 Translations from Modal Logic to First Order Logic

After our quick reminder of basic facts on first order resolution, we turn to mapping modal logic into FO formulas, keeping in mind that our goal is to use a first order prover to determine their satisfiability status.

3.3.1 The Relational Translation

Our first step will be to define suitable first order languages that we can translate to. Let *Index* be an index set. Consider the language $\mathcal{ML}$ as presented in Definition 1.4.3, with $\mathsf{REL} = \{R\}$, and the multi-modal language $\mathcal{MML}$, with $\mathsf{REL} = \{R_a \mid a \in \mathit{Index}\}$. The vocabulary of the first order language $\mathcal{FO}_1$ has unary relation symbols P corresponding to the proposition letters in PROP, and a single binary relation symbol R. Instead of a single binary relation symbol R, the vocabulary of the first order language $\mathcal{FO}_2$ has binary relation symbols R_a, for every $a \in \mathit{Index}$.

Models for $\mathcal{ML}$ and $\mathcal{MML}$ can also be viewed as models for the corresponding first order languages $\mathcal{FO}_1$ and $\mathcal{FO}_2$, respectively. To interpret the unary relation symbols, we simply look up the values of the corresponding proposition letters in the valuation.

3.3.1. DEFINITION. [Relational Translation] The *relational translation* $ST(\phi)$ of uni-modal formulas ϕ into first order formulas of $\mathcal{FO}_1$, is defined as follows. Let x be an individual variable.

$$\begin{aligned} ST_x(p) &= P(x) && (3.1)\\ ST_x(\neg\phi) &= \neg ST_x(\phi) \\ ST_x(\phi \wedge \psi) &= ST_x(\phi) \wedge ST_x(\psi) \\ ST_x(\Diamond\phi) &= \exists y\,(Rxy \wedge ST_y(\phi)). && (3.2)\end{aligned}$$

In (3.1), P is the unary relation symbol corresponding to the proposition letter p; in (3.2), the variable y is fresh. Observe how (3.2) reflects the truth definition for the modal operator $\Diamond$. The translation ST is easily extended to a translation taking multi-modal formulas into $\mathcal{FO}_2$, by using the relation symbol R_a instead of just R in the translation of the modal operator $\langle a\rangle$.

For example, the modal formula $\Box(p \to \Diamond p)$ translates into the first order formula $\forall y\,(Rxy \to (Py \to \exists z\,(Ryz \wedge Pz)))$.

One can show that a modal formula is satisfiable if and only if its relational translation is [vB83]. This effectively embeds the modal languages considered here into first order languages, and, thus, opens the way to solving modal problems by first order means. The resulting first order fragments can be described as follows.

3.3.2. Definition. [Modal Fragment] Let x be an individual variable. The *modal fragment* MF of $\mathcal{FO}_1$ is built up from unary atoms Px, using negation, conjunction, and guarded quantifications of the form $\exists y\,(Rxy \wedge \alpha[x \mapsto y])$ and $\forall y\,(Rxy \rightarrow \alpha[x \mapsto y])$, where y is fresh, and $\alpha[x \mapsto y]$ is the result of replacing all free occurrences of x in α by y, and $\alpha(x) \in$ MF only has x free. Observe that the relational translation maps modal formulas into MF. The modal fragment of $\mathcal{FO}_2$ is defined analogously.

We have seen a method to verify (un)satisfiability of formulas in FOL, and a way of translating modal formulas into FOL: we will now see how the two interact.

3.3.3. Example. Consider the formula $\Box(p \rightarrow \Diamond p)$ again; it is clearly satisfiable. Proving this in first order logic amounts to showing that the translation of the formula, $\forall y(Rxy \rightarrow (Py \rightarrow \exists z(Ryz \wedge Pz)))$ is satisfiable, or equivalently, that the following set of clauses is satisfiable.

1. $\{\neg R(c,y),\ \neg P(y),\ R(y,f(y))\}$
2. $\{\neg R(c,z),\ \neg P(z),\ P(f(z))\}$.

The clauses have two resolvents (f^n is f applied n times):

3. $\{\neg R(c,c),\ \neg P(c),\ \neg P(f(c)),\ P(f^2(c))\}$
4. $\{\neg R(c,f(z)),\ R(f(z),f^2(z)),\ \neg R(c,z),\ \neg P(z)\}$.

Clauses 2 and 4 resolve to produce

5. $\{\neg R(c,f^2(z)),\ R(f^2(z),f^3(z)),\ \neg R(c,f(z)),\ \neg R(c,z),\ \neg P(z)\}$.

Clauses 2 and 5 resolve again to produce an analogue of 5 with even higher term-complexity, etc. None of the clauses is redundant and can be deleted; in the limit our input set has infinitely many resolvents. This shows that standard resolution does not necessarily terminate for relational translations of satisfiable modal formulas.

What went wrong in Example 3.3.3? First, to obtain the resolvent in step 3, a positive and negative binary literal were resolved; note that these literals (or rather: the modal operators from which they derive) live at different modal depths in the original modal formula $\Box(p \rightarrow \Diamond p)$. This resolution step is useless: the negative R-literal derives from the $\Box$-operator which occurs at modal depth 0, and the positive R-literal comes from the $\Diamond$-operator which occurs at modal depth 1. Unless we explicitly stipulate so (by means of axioms), different modal depths are completely independent and should not resolve. A similar comment can be made about the resolvent obtained in step 4, where a positive and negative unary literal corresponding to the two occurrences of the proposition letter p were resolved upon.

A number of solutions have been proposed for this problem: we'll review here the functional translation [ONdRG00] and the layered relational translation [AGHdR00].

3.3.2 The Functional Translation

The functional translation is based on an alternative semantics of modal logic. The fundamental idea is to represent each binary relation as a set of (partial or total) functions. It appeared simultaneously and independently in a number of publications: see [Ohl88, LFdC88, Her89, Zam89, AE92]. We give a short introduction of the translation as presented in [ONdRG00]

3.3.4. Proposition. *For any binary relation R on a non-empty set W there is a set AF_R of accessibility functions, that is, a set of partial functions $\gamma : W \rightarrow W$, such that*

$$\forall x, y\, (R(x,y) \iff (\exists \gamma \in AF_R\ \gamma(x) = y))).$$

To avoid quantification over function symbols, a list notation is introduced, in which any term $\gamma(x)$ is written as $[x\gamma]$. $[\cdot,\cdot]$ denotes the functional application operation which is defined to be a mapping from a domain W to the set of all partial functions over W. So complex terms of the form $\gamma_m(\cdots(\gamma_2(\gamma_1(x))))$ become terms of the form $[[[[x\gamma_1]\gamma_2]\cdots]\gamma_m]$. Of course, when the accessibility relation R is not serial, it cannot be properly represented by any set of *total* functions. As the target logic for the translation demands total functions ($[x\gamma]$ is a first order term and will always have an interpretation), a special element $\perp$ is adjoined to the domain W of the model at hand. Now, every function γ will map the elements which have no successor under R to the special element $\perp$, and a special 'dead end' predicate, de$_R$, is introduced, defined as follows:

3.3.5. Definition. The *dead-end predicate*, representing the absence of successors, is defined as

$$\forall x (de_R(x) \iff \forall\gamma(\gamma \in AF_R \rightarrow [x\gamma] = \perp)).$$

3.3.6. Theorem. *Let R be a binary relation on a set W, and let $W^\perp = W \cup \{\perp\}$. Then, the following defines R in terms of a set AF_R of total functions $\gamma : W^\perp \rightarrow W^\perp$:*

$$\forall x, y(W\ \ (R(x,y) \iff (\neg de_R(x) \wedge \exists\gamma\,(\gamma \in AF_R \wedge [x\gamma] = y)))),$$

where de_R is defined in 3.3.5

3.3.7. Definition. A *functional frame* is a 4-tuple $\mathcal{F} = (W, de, AF, [\cdot,\cdot])$, where W is a non-empty set, *de* is a subset of W, AF is a set of *total* functions $\gamma : W \rightarrow W$, and $[\cdot,\cdot] : W \times AF \rightarrow W$ the functional application operation.
A *functional model* is a pair $\mathfrak{F} = (\mathcal{F}, P)$, where $\mathcal{F}$ is a functional frame, and P is a valuation. The new truth definition for the diamond operator is

$$\mathfrak{F}, w \models \Diamond A \text{ iff } w \notin de \text{ and } \exists\gamma(AF\ (\mathfrak{F}, [w\gamma] \models A)),$$

and dually for the box operator.

3.3.8. DEFINITION. Following [Sch97], we choose as our target a many-sorted logic with a sort hierarchy and set declarations for function symbols [Wal94]. In this logic, a sort symbol can be viewed as a unary predicate and it denotes a subset of the domain. For the functional translation we introduce the sorts W and AF. The variables $x, y, z, \ldots$, are assumed to be of the sort W; the functional variables are denoted by $\lambda_1, \lambda_2, \ldots$, and are of sort AF. The sort of the operator $[\cdot,\cdot]$ is $W \times AF \to W$. The *functional translation* $FT(t, A)$ is defined as follows:

$$\begin{array}{rcl}
FT(t,p) & = & p(t) \\
FT(t,\neg A) & = & \neg FT(t,A) \\
FT(t,A \vee B) & = & FT(t,A) \vee FT(t,B) \\
FT(t,A \wedge B) & = & FT(t,A) \wedge FT(t,B) \\
FT(t,\Diamond A) & = & \begin{cases} \exists\gamma(AF(FT([t\gamma],A))) & \text{if } R \text{ is serial,} \\ \neg de(t) \wedge \exists\gamma(AF(FT(t\gamma,A))), & \text{otherwise} \end{cases} \\
FT(t,\Box A) & = & \begin{cases} \forall\gamma\,(AF(FT([t\gamma],A))) & \text{if } R \text{ is serial,} \\ \neg de(t) \to \forall\gamma\,(AF(FT(t\gamma,A))), & \text{otherwise} \end{cases}
\end{array}$$

While the functional translation results in great improvements for theorem proving over the relational translation, as seen for example in Figure 4.5(b), we believe we can improve the performance for theorem proving without departing so much from the inspiration behind the relational translation.

We will boost the performance of resolution procedures on the relational translation of modal formulas by making literals living at different modal depths syntactically different. The mathematical justification for these ideas is provided by a strong form of the tree model property, as we will explain in the following section.

3.3.3 The Tree Model Property

To increase the performance of general first order theorem provers on 'modal input', we will feed them with information about its modal character. More precisely, we will aim to encode by syntactic means the fact that basic modal logic enjoys a very strong form of the tree model property. In recent years, the latter has been identified as one of the semantic key features explaining the good logical and computational behavior of many modal logics; see [Grä01, Var97] for two very accessible presentations.

First, by a *tree* $\mathcal{T}$ we mean a relational structure (T, S) where T, the set of nodes, contains a unique $r \in T$ (called the *root*) such that $\forall t \in T\,(S^*rt)$; every element of T distinct from r has a unique S-predecessor; and S^+ is acyclic; that is, $\forall t\,(\neg S^+tt)$. (Here, S^+ and S^* denote the transitive and reflexive, transitive closure of S, respectively.)

A *tree model* (for the uni-modal language $\mathcal{ML}$) is a model $\mathcal{M} = (W, R, V)$, where (W, R) is a tree. A *tree-like model* for the multi-modal language $\mathcal{MML}$ is a model $(W, \{R_a \mid a \in \mathit{Index}\}, V)$ such that $(W, \bigcup_a R_a)$ is a tree. A logic **L** has the *tree model property* if every **L**-satisfiable formula is satisfiable at the root of a tree or tree-like model for **L**. Observe that the tree model property is incomparable to the finite model property; there are modal logics where the former fails but the latter holds, and vice versa. For example, the logic $\mathcal{H}(@)$ has the finite model property but not the tree model property, and the fixed point logic with chop (FLC) has the tree model property but not the finite model property [LS02].

3.3.9. Proposition. *[BdRV01]*

1. *The basic uni-modal logic of the language $\mathcal{ML}$ has the tree model property.*
2. *The basic multi-modal logic of the language $\mathcal{MML}$ has the tree model property.*

Many modal logics, including **K** and $\mathbf{K}_{(m)}$, enjoy stronger versions of the tree model property, where the degree of the tree model can be bounded by the size of the formula [BdRV01]. But **K** and $\mathbf{K}_{(m)}$ enjoy an even stronger version of the tree model property. The key notion here is that of *layering*, both w.r.t. tree models and w.r.t. formulas. Tree (or tree-like) models come with a layering induced by the *depth* of the nodes. Likewise, the parse tree of a modal formula induces a natural formula layering, where new layers begin immediately below nodes labeled by modal operators. For instance, in $\Box(p \rightarrow \Diamond p)$, the $\Box$ occurs in layer 0, while the $\Diamond$ occurs in layer 1, with its argument in layer 2. Next, the *modal depth*, $\mathrm{mdepth}(\phi)$, of a uni-modal or multi-modal formula ϕ is defined as follows. Proposition letters p have $\mathrm{mdepth}(p) = 0$; $\mathrm{mdepth}(\neg\psi) = \mathrm{mdepth}(\psi)$; $\mathrm{mdepth}(\psi \wedge \chi) = \max(\mathrm{mdepth}(\psi), \mathrm{mdepth}(\chi))$, while $\mathrm{mdepth}(\Diamond\psi) = \mathrm{mdepth}(\langle a\rangle\psi) = 1 + \mathrm{mdepth}(\psi)$.

3.3.10. Proposition. *Let ϕ be a modal formula, and $\mathcal{M}$ be a tree (or tree-like) model with root w such that $\mathcal{M}, w \models \phi$.*

Let ψ be a subformula of ϕ which occurs in formula layer l and which has modal depth k. To determine the truth value of ψ we only need to consider nodes at tree depth i, where $l \leq i \leq k + l$.

In words: there is a direct correlation between formula layers and layers in a tree (or tree-like) model; as a consequence, literals occurring at different formula layers should not resolve and need not be combined.

3.3.4 The Layered Translation

From Uni-Modal to First Order. The key idea behind our improved translation of modal formulas into first order formulas is to label unary and binary relations according to the *number* of modal operators nested within a modal formula. For instance, the formula p is translated into P_0x, while the formula $\Diamond p$ becomes $\exists y\,(R_1xy \wedge P_1y)$. The index 1 of the relation symbols R_1 and P_1 measures the modal depth of the modal formula.

To motivate the translation of uni-modal $\mathcal{ML}$ formulas into an intermediate multi-modal language, consider the following examples, where we use new operators and new proposition letters each time we change modal depth:

$$\begin{array}{rcl} \Diamond\Diamond p & \mapsto & \Diamond_1\Diamond_2 p_2 \\ \Box(p \to \Diamond p) & \mapsto & \Box_1(p_1 \to \Diamond_2 p_2). \end{array}$$

If we then apply the relational translation (Definition 3.3.1) to the intermediate multi-modal representations, we obtain $\exists y\,(R_1xy \wedge \exists z\,(R_2yz \wedge P_2z))$ and $\forall y\,(R_1xy \to (P_1y \to \exists z\,(R_2yz \wedge P_2z)))$, respectively. Observe that the problematic derivation from the relational translation of $\Box(p \to \Diamond p)$ in Example 3.3.3 is no longer possible with the new first order translation.

To make things precise, we need an intermediate multi-modal language $\mathcal{MML}$, whose collection of modal operators is $\{\Diamond_i \mid i \geq 0\}$.

3.3.11. Definition. Let ϕ be a uni-modal formula. Let n be a natural number. The translation $Tr(\phi, n)$ of ϕ into the intermediate modal language $\mathcal{MML}$ is defined as follows:

$$\begin{array}{rcl} Tr(p,n) & := & p_n \\ Tr(\neg\psi,n) & := & \neg Tr(\psi,n) \\ Tr(\psi \wedge \chi,n) & := & Tr(\psi,n) \wedge Tr(\chi,n) \\ Tr(\Diamond\psi,n) & := & \Diamond_{n+1} Tr(\psi,n+1). \end{array}$$

Our next aim is to show that the intermediate translation Tr preserves satisfiability.

3.3.12. Proposition. *Let ϕ be a uni-modal formula. If ϕ is satisfiable, then so is its intermediate multi-modal translation $Tr(\phi, 0)$.*

Proof. By Proposition 3.3.9 we may assume that ϕ is satisfiable at the root w of a tree model $\mathcal{M} = (W, R, V)$. Since $\mathcal{M}$ is a tree model, for every state $v \in W$ there exists a unique path of R-steps from the root w to v; let $d(w, v)$ denote the length of this path.

We define a model $\mathcal{N} = (W, \{R_{n+1} \mid n \geq 0\}, V')$ for the intermediate multi-modal language $\mathcal{MML}$ by taking its universe to be W, the universe of $\mathcal{M}$. Its relations are defined by stipulating that $R_{n+1}(u, v)$ holds iff $d(w, u) = n$ and $R(u, v)$ both hold. We complete the definition of $\mathcal{N}$ by defining the valuation V': for every proposition letter p and every state $v \in W$ such that $d(w, v) = n$, we put $v \in V'(Tr(p, n))$ iff $v \in V(p)$.

We leave it to the reader to show that for every uni-modal formula ϕ, every state v and every n such that $d(w,v) = n$, we have $\mathcal{M}, v \models \phi$ iff $\mathcal{N}, v \models \mathit{Tr}(\phi, n)$. From this the lemma follows. $\dashv$

3.3.13. Proposition. *Let ϕ be a uni-modal formula. If its intermediate multi-modal translation $\mathit{Tr}(\phi, 0)$ is satisfiable, then so is ϕ.*

Proof. Let $\mathit{Tr}(\phi, 0)$ be satisfied at some state w in some model $\mathcal{M}$ for the intermediate multi-modal language $\mathcal{MML}$. As before we may assume that $\mathcal{M}$ is a tree-like model with root w. We define a uni-modal model $\mathcal{N}$ which differs from $\mathcal{M}$ in that it has only one relation (R) and in its valuation. The relation R consists of all pairs (u,v) such that $(u,v) \in R_{n+1}$ and $d(w,u) = n$, where $d(w,u)$ is the length of the path from w to u (in $\mathcal{M}$). The valuation V' of our model $\mathcal{N}$ is defined as follows: for every proposition letter p, for every v such that $d(w,v) = n$, we put $v \in V(p)$ iff $v \in V(\mathit{Tr}(p,n))$, where V is $\mathcal{M}$'s valuation. One can then show that if $d(w,v) = n$, then $\mathcal{M}, v \models \mathit{Tr}(\phi, n)$ iff $\mathcal{N}, v \models \phi$. This implies the lemma. $\dashv$

3.3.14. Definition. The *layered relational translation* is the composition of Tr and ST.

3.3.15. Theorem. *Let ϕ be a uni-modal formula. Then ϕ is satisfiable iff its layered relational translation $ST(\mathit{Tr}(\phi, 0))$ is.*

We contend that the layered translation greatly improves the performance of resolution procedures for the satisfiability problem of translated modal formulas.

3.4 Comparing the approaches: Experimental results

We will now see how the different translations compare, in terms of the efficiency for resolution theorem proving. We will compare the layered translation approach with both the relational translation and the functional translation, using the test sets reviewed in Chapter 2. We do the comparisons separately to better appreciate the differences: the formulas that result from the relational translation take so long to solve that showing results for the three translations together would not permit a correct appreciation of the difference between formulas created with the layered and functional translations. Before going into the test results, we comment on the problem sets and theorem provers used in our experiments.

The Problem Sets To evaluate our tree-based heuristics, we have run a series of tests on a number of problem sets. To compare the relational and layered translations, we used the Heuerding and Schwendimann test set and the modal

QBF test set; provers take too long with the relational translation of Modal CNF formulas. For the comparison between the functional and layered translation, we found that easy modal CNF runs were feasible, so we used the modal CNF for that test.

The Theorem Provers. The comparisons between the layered and relational translations were performed on a Sun ULTRA II (300MHz) with 1Gb RAM, under Solaris 5.2.5, with SPASS version 1.0.3 and MSPASS version V 1.0.0t.1.3. SPASS [SPA] is an automated theorem prover for full sorted first order logic with equality that extends superposition by sorts and a splitting rule for case analysis; it has been in development at the Max-Planck-Institut für Informatik for a number of years. MSPASS [MSP] is an enhancement of SPASS (Version 1.0.0t) with a translator of modal formulae, formulae of description logics, and formulae of the relational calculus into sorted first order logic with equality. For the comparison between the layered and functional translations, a Pentium IV PC with 256MB RAM running RedHat Linux 7.3 was used.

SPASS was invoked with the auto mode switched on; no sort constraints were built, and both optimized and strong Skolemization were disabled.

Layered vs Relational: Heuerding and Schwendimann. Table 3.1 displays the maximum number of problems of the Heuerding and Schwendimann test set solved in less than 100 seconds each, the standard timeout for this test, by the layered and relational translations. We see that he layered translation outperformed the relational translation, being able to solve harder instances in almost all categories. Interestingly, categories *k_ph_p* and *k_ph_n* are known to be *propositionally* hard; in these categories, the effect of layering is not expected to be very noticeable, and indeed these are the only categories in which the layered translation does not improve upon the relational translation (apart from *k_lin_p*, which is too easy for both).

Translation	branch		d4		dum		grz		lin		path		ph		poly		t4p	
	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n
relational	3	3	3	1	3	1	5	0	21	4	4	2	5	5	5	4	0	0
layered	8	8	11	7	21	21	21	21	21	5	7	4	5	5	13	14	13	6

Table 3.1: Comparison using the Heuerding and Schwendimann test set.

Layered vs Relational: Modal QBF. To explore the behavior of our heuristics in a larger portion of the landscape of the **K**-satisfiability problem, we generated sets of 10 random modal QBF problems for different sets of parameters. Table 3.2 compares the average time in CPU seconds and number of clauses generated for the two translations: *layered* and *relational*. "C/V/D" in the first column denotes the number of clauses, the number of variables, and the depth used in the generation. Columns labeled by "M" show the orders of magnitude

C/V/D	Average Time		M	Average Clauses		M
	Layered	Relational		Layered	Relational	
5/2/1	0.53469	9.6222	1	726	5695	1
10/2/1	0.41734	3.9909	1	546	2367	1
15/2/1	0.10859	0.13172	0	10	10	0
5/2/2	0.66141	450.44	3	437	27029	2
10/2/2	0.78297	370.09	3	500	22306	2
15/2/2	0.75656	147.38	2	473	11368	1
5/2/3	36.048	N/A	N/A	10714	N/A	N/A
10/2/3	58.996	N/A	N/A	15395	N/A	N/A
15/2/3	94.192	2094.4	1	20786	45798	0
5/2/4	20.362	N/A	N/A	3121	N/A	N/A
10/2/4	33.084	N/A	N/A	4971	N/A	N/A
15/2/4	35.068	N/A	N/A	5358	N/A	N/A
5/2/5	1136.1	N/A	N/A	48546	N/A	N/A
10/2/5	2896	N/A	N/A	91767	N/A	N/A
15/2/5	3758.2	N/A	N/A	106870	N/A	N/A
5/3/1	7.1862	2047.9	2	4372	105960	1
10/3/1	9.752	2324.2	2	5390	108110	1
15/3/1	14.066	1506.8	2	6687	72605	1
5/3/2	7.0931	N/A	N/A	1804	N/A	N/A
10/3/2	8.3192	N/A	N/A	2221	N/A	N/A
15/3/2	9.3902	N/A	N/A	2687	N/A	N/A
5/3/3	1445.2	N/A	N/A	52153	N/A	N/A
10/3/3	4045.1	N/A	N/A	107800	N/A	N/A
15/3/3	4865.4	N/A	N/A	119150	N/A	N/A

Table 3.2: Comparison using the Modal QBF test set.

of the difference between the preceding two columns, i.e., round($(-1 * log(N/N'))$). We used a time out of 3 hours on a shared machine; N/A indicates that a value is not available due to a time out.

As can easily be seen from Table 3.2, our improved translation method outperformed the relational translation in every case, both in computing time (CPU time) and number of clauses generated; this is not only an average behavior but it was observed in each instance. For some configurations the drop in computing time is as much as three orders of magnitude. The average number of clauses generated was nearly always smaller by at least one order of magnitude.

In Figure 3.1 we display a sample from our experimental results: 64 instances of the 10/3/1 configuration. The top curve indicates the CPU time needed by the relational translation, and the bottom one the CPU time needed by the layered translation. Note that the relational translation can be very sensitive to certain hard problems, which results in significant differences between easy and hard instances; the layered method responds in a much more controlled way to hard

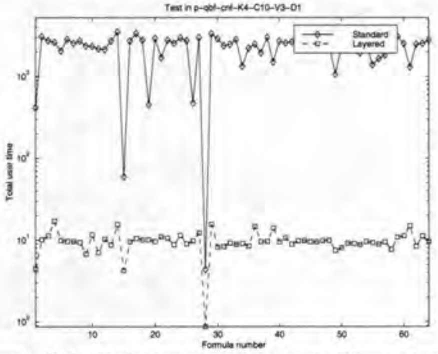
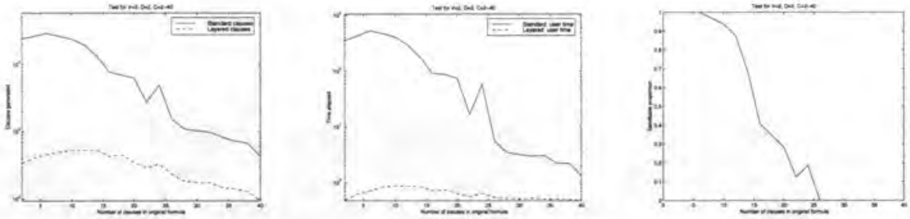


Figure 3.1: Relational vs Layered: Time elapsed.



(a) (b) (c)

Figure 3.2: Easy-hard-easy.

problems. Interestingly, the curves follow each other, even at many orders of magnitude of difference. This shows that our heuristics do not change the nature of the problem: they simply make it much easier for the resolution prover.

The latter phenomenon can also be observed more globally. The plots in Figure 3.2 were obtained with V = D = 2, while C ranged from 2 to 40. Figures 3.2 (a) and (b) show the number of clauses generated and the CPU time needed, respectively, for the relational and layered method, while 3.2 (c) plots the proportion of satisfiable instances as C increases. The curves for the relational and layered methods are very similar, with the layered method lacking the sharp lows and highs that seem to be characteristic for the relational method. Both display a clear easy-hard-easy behavior, but the layered translation improves performance by several orders of magnitude. Note that the biggest improvements are achieved in the satisfiable region, i.e., for $C < 26$.

Once we were confident that the layered method consistently displayed a good behavior and a significant improvement over the relational translation, we ran the standardized tests provided by TANCS (64 instances randomly generated with the 20-clauses/2-variables/2-depth parameters); see Figure 3.3 for the outcomes.

Finally, to obtain the results in Figure 3.4 we generated 64 instances of problems for 2 and 3 variables with depths ranging from 1 to 6, again with a time out of 3 hours. The figure shows the average values we obtained. We ran the same

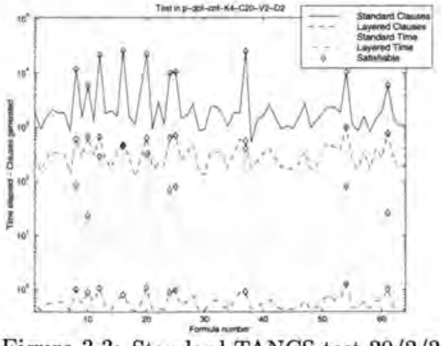


Figure 3.3: Standard TANCS test 20/2/2.

tests with the relational instead of the layered translation, but even for moderate depths the computing time and number of clauses exceeded the available resources.

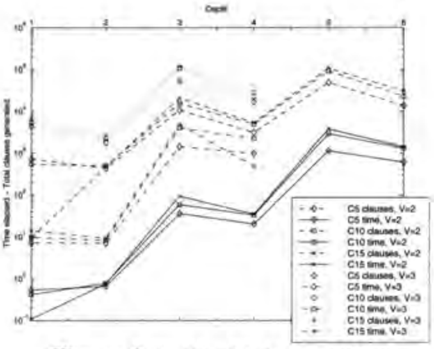
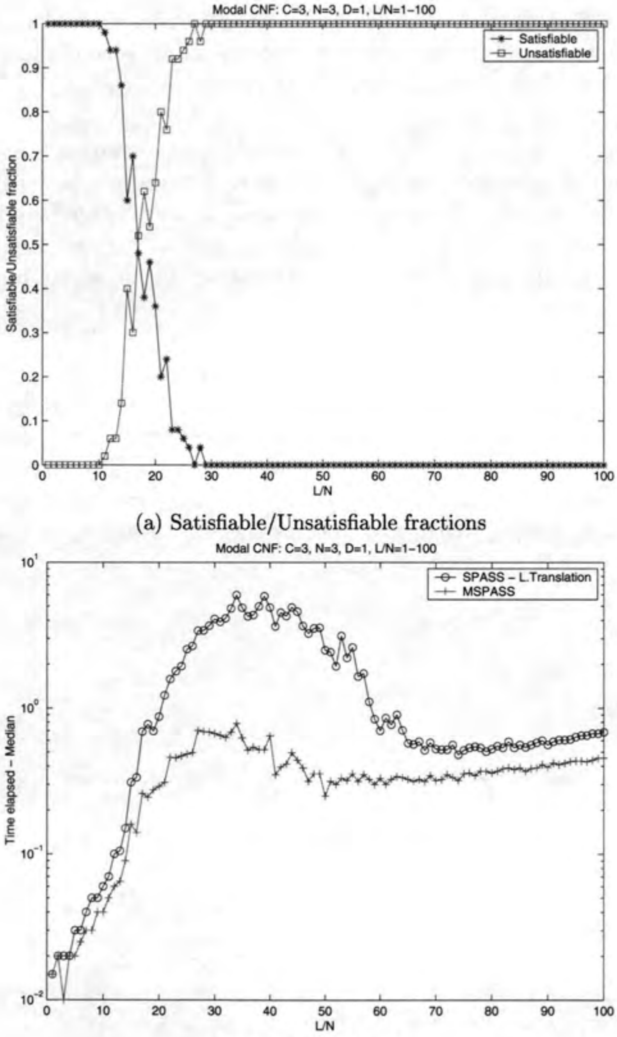


Figure 3.4: The Staircase Effect

Layered vs Functional: Modal CNF. We performed a comparison of the layered and functional translations using the Random Modal CNF test set. We generated a set for $C = 3, D = 1, N = 3, L = 1 - 60$, and the results are shown in Figure 3.5. Figure 3.5 (a) shows the satisfiable/unsatisfiable proportion as a function of L/N. Which was of course identical for both translations, since we feed them the same formulas and there were no timeouts. Figure 3.5 (b) shows the mean CPU time used by each prover on the formulas, also as a function of L/N

One thing which is apparent from this figure is that for both translations the peak difficulty does not correspond to the point of maximum uncertainty with respect to satisfiability; this could be a characteristic of resolution-based modal theorem proving, since it does not happen with other provers [PSS03].



(a) Satisfiable/Unsatisfiable fractions

(b) Mean CPU time

Figure 3.5: Layered - Functional translations comparison.

3.5 Conclusion

We have reviewed different ways of translating modal formulas into first order logic, and seen how the translation method affects the performance of first order theorem provers when checking the satisfiability of those formulas.

Layered vs Relational. Layering proved to be very useful: a simple improvement to the relational translation means that a modal formula will take orders of magnitude less effort to check for satisfiability.

Layered vs Functional. The functional translation enjoys a wider applicability than that of the layered translation; since it does not depend on the strong version of the tree model property we are using, it can be applied to modal logics that do not have it, such as **S4**. The price to pay in this case is the replacement of relation symbols with *functions*: the translated formulas are not in the modal fragment any more.

Other layering inspired techniques. Other variations on the tree model property and layering have been explored. In [PSV02], a very competitive automata-based method of checking modal satisfiability is presented, which is based on the automaton accepting all tree models of the formula. In [BGdR03], the tree model property is used to encode modal satisfiability problems into constraint satisfaction problems, and an algorithm to solve them is proposed; initial experiments show the approach to be promising.

Chapter 4

Modal and Hybrid Theorem Proving – Direct Resolution

"The problem, Mendieta,
is that nature is as wicked
as it is wise."
Roberto Fontanarrosa

4.1 Resolution for Modal-Like Logics

Designing resolution methods that can directly (without translation into large background languages) be applied to modal logics, received quite some attention in the late 1980s and early 1990s; see for example [Min89, EdC89]. Given the simplicity of propositional resolution and the fact that modal languages are sometimes viewed as "simple extensions of propositional logic," we might expect modal resolution to be as simple and elegant. However, direct resolution for modal languages proved to be a difficult task. Intuitively, in basic modal languages the resolution rule has to operate *inside* the box and diamond operators to achieve completeness. This leads to more complex systems, less elegant results, and poorer performance, ruining the "one-dumb-rule" spirit of resolution. In [AdNdR01] a resolution calculus for hybrid logics addressing these problems was introduced: the hybrid machinery is used to "push formulas out of modalities" and in this way, feed them into a simple and standard resolution rule.

In this chapter we describe HyLoRes, an automated theorem prover based on the calculus introduced in [AdNdR01], with special emphasis on implementation details. Indeed, the aim of this chapter is to give a fairly detailed account and assessment of the main optimizations that went into HyLoRes.

The Logic. We will use the language of hybrid logic as introduced in Definitions 1.4.5 and 1.4.6 ; we present the syntax again for ease of reference. The well-formed formulas of the hybrid language $\mathcal{H}(@,\downarrow)$ in the signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM}, \mathsf{SVAR}\rangle$ are

$$\mathsf{FORMS} := \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid [R]\phi \mid @_s\phi \mid {\downarrow}x.\phi,$$

where $a \in \mathsf{ATOM}$, $x \in \mathsf{SVAR}$, $s \in \mathsf{SSYM}$, $R \in \mathsf{REL}$ and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$.

4.2 The Rules

We will now describe the resolution calculus implemented in HyLoRes. We need a normal form that guarantees formulas will have a unique representation with respect to negation, so we define the following rewriting procedure *nf* on formulas of $\mathcal{H}(@,\downarrow)$. Let ϕ be a formula in $\mathcal{H}(@,\downarrow)$, $nf(\phi)$ is obtained by repeated application of the rewrite rules *nf* until none is applicable:

$$\begin{aligned} \neg @_t\psi &\overset{nf}{\rightsquigarrow} @_t\neg\psi \\ \neg{\downarrow}x.\psi &\overset{nf}{\rightsquigarrow} {\downarrow}x.\neg\psi \\ \neg\neg\psi &\overset{nf}{\rightsquigarrow} \psi \end{aligned}$$

Clauses are sets of formulas in this normal form. To determine the satisfiability of a sentence $\phi \in \mathcal{H}(@)$ we first notice that ϕ is satisfiable iff $@_t\phi$ is satisfiable, for a nominal t not appearing in ϕ. Define the clause set *ClSet* corresponding to ϕ to be $ClSet(\phi) = \{\{@_t nf(\phi)\}\}$. Next, let $ClSet^*(\phi)$ – the *saturated* clause set corresponding to ϕ – be the smallest set containing $ClSet(\phi)$ and closed under the rules shown in Figure 4.1.

$$(\wedge)\ \frac{Cl \cup \{@_t(\phi_1 \wedge \phi_2)\}}{\begin{array}{c} Cl \cup \{@_t\phi_1\} \\ Cl \cup \{@_t\phi_2\} \end{array}} \qquad (\vee)\ \frac{Cl \cup \{@_t\neg(\phi_1 \wedge \phi_2)\}}{Cl \cup \{@_t nf(\neg\phi_1), @_t nf(\neg\phi_2)\}}$$

$$(\mathrm{RES})\ \frac{Cl_1 \cup \{@_t\phi\} \quad Cl_2 \cup \{@_t\neg\phi\}}{Cl_1 \cup Cl_2}$$

$$([R])\ \frac{Cl_1 \cup \{@_t[R]\phi\} \quad Cl_2 \cup \{@_t\neg[R]\neg s\}}{Cl_1 \cup Cl_2 \cup \{@_s\phi\}} \qquad (\langle R\rangle)\ \frac{Cl \cup \{@_t\neg[R]\phi\}}{\begin{array}{c} Cl \cup \{@_t\neg[R]\neg n\} \\ Cl \cup \{@_n nf(\neg\phi)\} \end{array}},\ \text{for } n \text{ new.}$$

$$(@)\ \frac{Cl \cup \{@_t@_s\phi\}}{Cl \cup \{@_s\phi\}}$$

$$(\mathrm{SYM})\ \frac{Cl \cup \{@_t s\}}{Cl \cup \{@_s t\}} \qquad (\mathrm{REF})\ \frac{Cl \cup \{@_t\neg t\}}{Cl} \qquad (\mathrm{PARAM})\ \frac{Cl_1 \cup \{@_t s\} \quad Cl_2 \cup \{\phi(t)\}}{Cl_1 \cup Cl_2 \cup \{\phi(t/s)\}}$$

Figure 4.1: Resolution calculus for the logic $\mathcal{H}(@)$

Let us briefly explain the rules. The (RES) rule is the known resolution rule. To understand the ([R]) rule, keep in mind the relational translation of the $\Box$ operator, from Definition 3.3.1:

$$ST_x(\neg(\Diamond\phi)) = \neg(\exists y\,(Rxy \wedge ST_y(\phi)))$$

Or, equivalently,

$$ST_x(\neg(\Diamond\phi)) = \forall y\,(\neg Rxy \vee \neg ST_y(\phi))$$

Here, x plays the role of t. In essence, what happens with this rule is that the "hidden" universally quantified variable y, which should only be unified to R-successors of x, is both created and unified behind the scenes, when an R-successor of x is available, and resolution is applied. R-successors of x are created by the ($\langle R\rangle$) rule, which can be seen as a form of skolemization which only introduces constants. This way, unification is controlled, to the point that free variables are not needed in the calculus. The ($\wedge$) and ($\vee$) rules break down complex formulas into their components; the calculus can resolve on complementary formulas of arbitrary complexity, which can save time but is not in itself a complete method. The (@) rule simplifies formulas into equivalent formulas to achieve a unique representation, much like the transformation into negation normal form does for negation, and the (SYM), (REF) and (PARAM) rules all deal with equality between nominals: since nominals can only be true of one element in the model, whenever we encounter a formula of the form $@_s t$, that can only be true if s and t are true on the *same* element of the model. Hence, (SYM) represents *symmetry* (if s and t denote the same element of the model, formulas true in s will also be true in t), (REF) represents *reflexivity* (every nominal is true in the element of the model it denotes), and (PARAM) is the paramodulation rule, adapted to equality between nominals.

The computation of *ClSet*$^*(\phi)$ is in itself a sound and complete algorithm for checking satisfiability of $\mathcal{H}(@)$, in the sense that ϕ is unsatisfiable if and only if the empty clause $\{\}$ is a member of *ClSet*$^*(\phi)$ [AdNdR01].

The $\downarrow$ operator. To be able to account for hybrid sentences using $\downarrow$ we need only extend the calculus with the rule

$$(\downarrow)\quad \frac{Cl \cup \{@_t{\downarrow}x.\phi\}}{Cl \cup \{@_t\phi(x/t)\}}.$$

The full set of rules is a sound and complete calculus for checking satisfiability of sentences in $\mathcal{H}(@,\downarrow)$ [AdNdR01].

4.2.1. EXAMPLE. We prove that ${\downarrow}x.\langle R\rangle(x \wedge p) \rightarrow p$ is a tautology. Consider the clause set corresponding to the negation of the formula:

1. $\{@_i((\downarrow x.\neg[R]\neg(x \wedge p))\underline{\wedge}\neg p)\}$ by $(\wedge)$
2. $\{@_i\underline{\downarrow x}.\neg[R]\neg(x \wedge p)\}, \{@_i\neg p\}$ by $(\downarrow)$
3. $\{@_i\underline{\neg[R]}\neg(i \wedge p)\}, \{@_i\neg p\}$ by $(\langle R\rangle)$
4. $\{@_i\neg[R]\neg j\}, \{@_j(i\underline{\wedge}p)\}, \{@_i\neg p\}$ by $(\wedge)$
5. $\{@_j\underline{i}\}, \{@_j\underline{p}\}, \{@_i\neg p\}$ by (PARAM)
6. $\{@_i\underline{p}\}, \{@i\underline{\neg p}\}$ by (RES)
7. $\{\}$

Here we see the calculus in action; the underlining reflects the operators or formulas that trigger the rule. In step 2, we see how the variable x is bound to the nominal in which the $\downarrow$ operator is evaluated. In step 3, the $\langle R\rangle$ rule creates a new nominal j, "connects" it to i through R, and creates a clause that states that the argument of $\langle R\rangle$ is true in j. Step 5 shows us the effect of paramodulation: since i and j refer to the same element in the model, formulas satisfied on j must also be satisfied on i, and vice versa.

4.3 The Given Clause Algorithm

HyLoRes implements a version of the "given clause" algorithm [Vor01], which is the underlying framework of many current state of the art resolution-based theorem provers [SPA, Bli, Hil03]; our version is shown in Figure 4.2. A brief explanation of the functions on that figure follows:

- *normalize(A)* applies *nf* to formulas in A and handles trivial tautologies and contradictions.
- *computeComplexity(A)* determines length, modal depth, number of literals, etc. for each of the formulas in A; these values are used by *select* to pick the given clause.
- *infer(*given,*A)* applies the resolution rules to the given clause and each clause in A. If the rules $(\wedge)$, $(\vee)$, $(\langle R\rangle)$ or $(\downarrow)$ are applicable, no other rule is applied as the clauses obtained as conclusions by their application subsume the premises.
- *simplify(A,B)* performs subsumption deletion, returning the subset of A which is not subsumed by any element in B.
- *notRedundant(*given*)* is true if none of the rules $(\wedge)$, $(\vee)$, $(\neg[R])$ or $(\downarrow)$ was applied to given.

4.4 Implementation

HyLoRes is implemented in Haskell (ca. 3500 lines of code), and compiled with the Glasgow Haskell Compiler (GHC) Version 5.04. We use Happy 1.13 to generate the parser. GHC produces fairly efficient C code which is afterward compiled into an executable file. Thus, users need no additional software to use the prover. The

```
input: init: set of clauses
var: new, clauses, inuse: set of clauses
var: given: clause

clauses := {}; inuse := {}; new := normalize(init)
if {} $\in$ new then return "unsatisfiable"
clauses := computeComplexity(new)
while clauses $\neq$ {} do
    {* Selection of given clause *}
    given := select(clauses); clauses := clauses $-$ {given}
    {* Inference *}
    new := infer(given, inuse); new := normalize(new)
    if {} $\in$ new then return "unsatisfiable"
    {* Subsumption deletion *}
    new := simplify(new, inuse $\cup$ clauses)
    inuse := simplify(inuse, new)
    clauses := simplify(clauses, new)
    {* Initialization for next cycle *}
    if notRedundant(given) then
        inuse := inuse $\cup$ {given}
    clauses := clauses $\cup$ computeComplexity(new)
return "satisfiable"
```

Figure 4.2: Structure of the given clause algorithm.

HyLoRes site (`http://www.illc.uva.nl/~juanh/HyLoRes`) provides executables for Solaris (tested under Solaris 8) and Linux (tested under Red Hat 7.0 and Mandrake 8.2). The original Haskell code is also made publicly available under the GPL license [GNU].

We will see now how HyLoRes handles the formula from Example 4.2.1 :

4.4.1. EXAMPLE. **Input file:**

```
begin
!((down (x1 dia (x1 & p1) )) -> p1)
end
```

Execution:

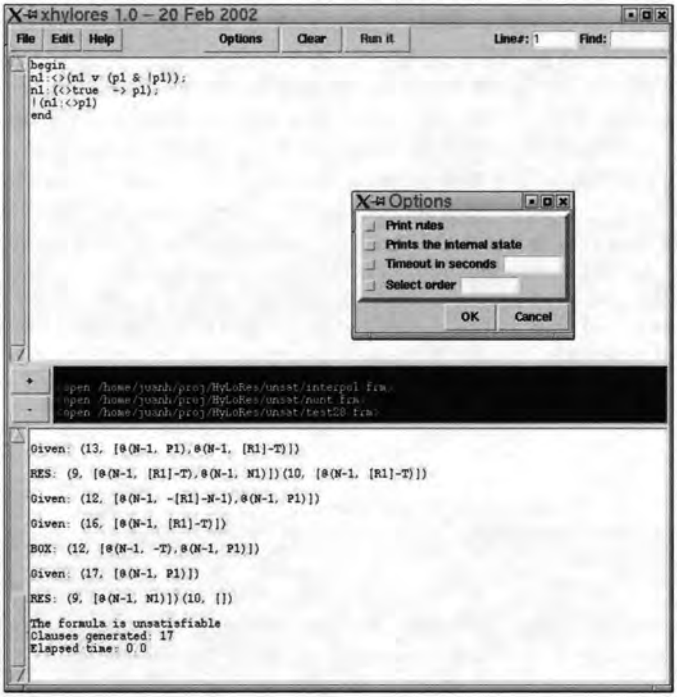
```
(juanh@banaan 149) hylores -f test.frm -r
Input:
   {[@(NO, (-P1 & Down(X1, -[R1]-(P1 & X1))))]}
End of input

Given: (1, [@(NO, (-P1 & Down(X1, -[R1]-(P1 & X1))))])
CON: {[@(NO, -P1)][@(NO, Down(X1, -[R1]-(P1 & X1)))]}
Given: (2, [@(NO, -P1)])
Given: (3, [@(NO, Down(X1, -[R1]-(P1 & X1)))])
ARR: {[@(NO, -[R1]-(P1 & NO))]}
Given: (4, [@(NO, -[R1]-(P1 & NO))])
DIA: {[@(N-2, (P1 & NO))][@(NO, -[R1]-N-2)]}
Given: (5, [@(N-2, (P1 & NO))])
CON: {[@(N-2, P1)][@(N-2, NO)]}
Given: (6, [@(N-2, NO)])
PAR (0,-2): {[@(N-2, (P1 & N-2))][@(N-2, -[R1]-(P1 & N-2))]
[@(N-2, Down(X1, -[R1]-(P1 & X1)))][@(N-2, -P1)]
[@(N-2, (-P1 & Down(X1, -[R1]-(P1 & X1))))]}
Given: (7, [@(N-2, P1)])
Given: (8, [@(N-2, -P1)])
RES: (7, [])

The formula is unsatisfiable
Clauses generated: 11
Elapsed time: 0.0
```

Here we see the prover giving a step by step account of the clause chosen as *given*, the rules applied to it, and the results. Lines starting with CON, ARR, DIA, PAR and RES respectively indicate application of the $(\wedge), (\downarrow), (\langle R\rangle)$, (PARAM) and (RES) rules, with the remainder of the corresponding lines showing the result of applying such rules. A number is assigned to each clause when it becomes the given clause; it is shown when the clause is displayed. In the case of the (PARAM) rule, the nominals involved are shown between brackets, and in the case of the (RES) rule, the numbers of the clauses involved are shown before the corresponding resolvent. We see that the proof follows closely the steps given in Example 4.2.1, except that the paramodulation rule actually generates more clauses than previously shown.

In addition to HyLoRes, a graphical interface called xHyLoRes implemented in Tcl/Tk was developed. It uses HyLoRes in the background and provides full file access and editing capabilities, and a more intuitive control of the command line parameters of the prover, in the manner of Spin/XSpin [X S]. A screenshot of xHyLoRes can be seen in Figure 4.3.



```
begin
n1:<>(n1 v (p1 & !p1));
n1:(<>true  -> p1);
!(n1:<>p1)
end
```

```
<open /home/juanh/proj/HyLoRes/unsat/interpol.frm>
<open /home/juanh/proj/HyLoRes/unsat/nunt.frm>
<open /home/juanh/proj/HyLoRes/unsat/test28.frm>
```

```
Given: (13, [@(N-1, P1),@(N-1, [R1]-T)])

RES: (9, [@(N-1, [R1]-T),@(N-1, N1)])(10, [@(N-1, [R1]-T)])

Given: (12, [@(N-1, -[R1]-N-1),@(N-1, P1)])

Given: (16, [@(N-1, [R1]-T)])

BOX: (12, [@(N-1, -T),@(N-1, P1)])

Given: (17, [@(N-1, P1)])

RES: (9, [@(N-1, N1)])(10, [])

The formula is unsatisfiable
Clauses generated: 17
Elapsed time: 0.0
```

Figure 4.3: A screenshot of xHyLoRes.

4.5 The Gory Details

4.5.1 Data Structures

The design of HyLoRes is modular with respect to the internal representation of the different kinds of data. We have used the Edison package [Oka01] (a library of efficient data types provided with GHC) to implement most of the data types representing sets. The basic data types we created are as follows.

State and Output Monads. Functional programming does not allow for global variables or side effects; in a function, all input must be passed as an argument and all consequences must be part of the returned value. For some applications, this can result in functions having very long and unintuitive lists of arguments, and contrived output types. In Haskell, a particular data type called monad is used to overcome this problem. The internal state of the given clause algorithm (the sets *clauses*, *inuse* and *new*, the data structures used for subsumption checking, the control information, etc) is represented as a combination of a state and an output monads [Wad95]; the former provides transparent access to the internal state of the program from the monadic functions that perform inference, while the latter handles all printing services with no need of further parameters in the function signatures. In addition, the use of monads allows the addition of further structure (hashing functions, etc.) to optimize search, with minimum re-coding. We have already experienced the advantages of the monad architecture as we have been able to test different data structures and improve the performance of some of the most expensive functions with great ease.

Formulas. We took advantage of the possibility of defining recursive data types in Haskell, with the result that the data type definition closely resembles the definition given in Section 1.4.5:

```
data Formula
   = Taut | Nom  Int | Prop Int | Var  Int | Neg  Formula | Con [Formula]
     | At   Int Formula | Atv  Int Formula | Down Int Formula
     | Box  Int Formula
deriving (Ord, Eq)
```

The integers in the definition represent the different elements of their corresponding sets, i.e. `Nom 1` represents the element n_1 in the set NOM, and so on. Conjunctions are stored as the `Con` constructor plus a list of conjuncts, to allow for n-ary conjunctions.

Clauses and Sets of clauses. The given clause algorithm at the heart of HyLoRes deals with three main repositories of clauses: *clauses*, that holds the eligible candidates por processing; *inuse*, that holds the clauses which can interact with the given clause, and *new*, where the clauses that result from the application of the rules go. The different clause sets and their clauses have different access patterns and aggregate information and need a different data type for each. *clauses* uses the `UnbalancedSet` type provided by the Edison library which is specially optimized for search; as in every cycle the given clause has to be selected from this set, the comparison of clause scores is given as the ordering function, so the given clause can be selected without having to examine the whole set. The elements of *clauses* are tuples containing the clause proper (represented also as an `UnbalancedSet`), a complexity measure which depends on the chosen order for clause selection, and the clause number.

In *new*, clauses are stored as `UnbalancedSets` while *new* itself is a list of clauses, as all its elements have to be processed one by one in each cycle. *inuse* is a list of pairs composed of the clause number and a clause represented also as a list, as both clauses and formulas in clauses need to be accessed one by one in every cycle.

4.5.2 Optimizations

The first implementation of HyLoRes was very naïve and as a result was terribly inefficient. We then proceeded to adapt and apply well established first order resolution optimizations to the hybrid environment, with encouraging results.

Ordered resolution with selection. HyLoRes actually implements a version of *ordered resolution with selection* [BG01], where the application of the (RES) and ($[R]$) rules are restricted to certain selected formulas in the clause. Ordered resolution with selection greatly reduces the size of the saturated set, preventing the generation of certain clauses, without compromising the completeness of the calculus. Interestingly, the proof of completeness of ordered resolution with selection for $\mathcal{H}(@, \downarrow)$ [AG03] closely follows the proof in [BG01], based on a step by step construction of a Herbrand model for any consistent input clause set. Once more, hybrid logics seem to provide the appropriate framework to merge first order and modal ideas.

Formula indexing. Formulas are indexed using a mapping between formulas and integers, in which indexes for positive and negative occurrences of the same formula will be equal except for the sign. As the (RES) rule involves searching for complementary formulas, searching for clauses to resolve with is made more efficient by storing the clauses in *inuse* as ordered lists of the indexes. This indexing is much simpler than in the case of first order, as clauses do not have free variables.

Subsumption checking. Whenever a clause A follows from another clause B in the clause set, A is said to be *subsumed* by B, and can be ignored, reducing the search space while maintaining correctness. We consider two main types of subsumption checking: *forward* subsumption (when new clauses are redundant w.r.t. old clauses) and *backward* subsumption (when old clauses are redundant w.r.t. new clauses).Finding out which clauses can be discarded is one of the – or perhaps "the" – most expensive operations in resolution based theorem provers [Vor95]. HyLoRes uses a simple version of subsumption checking where a clause C_1 subsumes a clause C_2 if $C_1 \subset C_2$. Version 0.5 of the prover implemented this test very inefficiently, checking the subset relation element by element, and clause by clause. In the latest prototype, a set-at-a-time subsumption checking algorithm which uses a clause repository structured as a trie [Vor95] was implemented, with dramatic improvements (see Section 4.6). We also noticed that while forward subsumption is essential, many times backward subsumption does not really make a difference. This is also the case for some first order logic provers; see [RSV01].

The clause repository is organized as a list of tries, in the following manner. The clauses are inserted and queried as ordered lists of integers. The repository is a list of tries, in which each node represents a formula and each path that ends in a leaf node represents a clause.

4.5.1. Example. The set of clauses

$$\{ \quad \{1,2,3\}, \quad \{1,2,4\}, \quad \{1,2,8\}, \quad \{1,4,5\}, \quad \{1,4,7,8\}, \quad \{1,4,7,9,10\}, \quad \{2,3,9\}, \quad \{2,7,9\}, \quad \{2,7,8,10\}, \quad \{2,7,8,11\} \quad \}$$

is stored as shown in Figure 4.4.

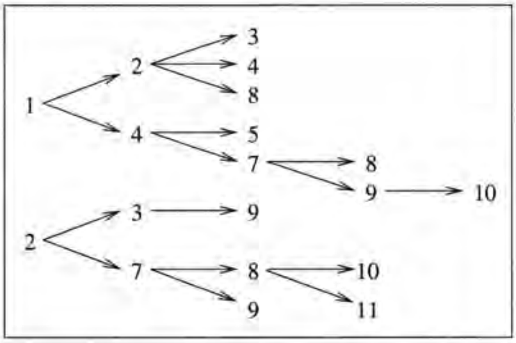


Figure 4.4: Trie representation for a set of clauses

When inserting a clause, if its head is the root of any of the visible tries then we insert its tail into that trie, otherwise we add a branch to the current node and insert the clause there. In this way, all clauses are represented as a path from one of the root nodes to a leaf, so that all the clauses that are extensions of a particular path are stored as branching from it. The fact that the formulas in the clause are ordered gives us the possibility to optimize search, both by having a unique representation and by knowing when it will be useless to keep searching. The clause repository holds both the clauses in *inuse* and the ones in *clauses*, so as to check for (forward or backward) subsumption against just one set of clauses, which also eliminates the cost of transferring clauses from one trie to the other when a clause is moved from *clauses* to *inuse*. Subsumption checking has then become very efficient, and indeed it brought a speed up of about two orders of magnitude to the prover.

In forward subsumption, the clauses in *new* are checked one by one for subsumption by the clauses in *inuse* or *clauses*, as follows: for each clause C in *new*, for each of the visible tries T_i in the repository, if the root of T_i is in the checked clause, all the branches of T_i are successively checked for the elements of the clause that are greater than the root. If we reach the end of any branch, then the clause is subsumed by the repository and the search stops. If we find any element not present in C, none of the clauses represented by the current path subsumes C and we can proceed to the next trie. If the root of the next trie is greater than the maximum element in C, no match will be possible and the search ends.

In backward subsumption, the clauses in *new* are checked one by one for subsumption of the clauses in *inuse* or *clauses*, as follows: for each clause C in *new*, for each of the visible tries T_i in the repository whose root is less than or equal to the head of the clause (the smallest element), if the root of T_i is equal to the head of C, we check the branches of T_i for existence of the elements in the tail of C, and if the root of T_i is less than the head of C we check the branches of T_i for existence of the whole clause. When we find a match for the last element of the clause, we know that all the paths that originate from T_i are subsumed by the clause: we retrieve all of them, and examine the next trie. When we reach a T_i with a root greater than the head of C, the search ends.

Input analysis. At this moment, HyLoRes performs a very simple analysis of its input. It checks for the presence of the $[R], \langle R\rangle, @$ and $\downarrow$ operators and for nominals in order to know which rules will need checking for applicability. For example, if the $\downarrow$ operator does not appear in the input, then the $(\downarrow)$ rule is switched off and never attempted. Most first order provers perform a far more detailed analysis of the input and decide heuristics and settings on account of their findings.

Application of the rules. The rules of the underlying resolution calculus (as shown on Figure 4.1) are applied in such a way as to make the sets of clauses grow as slowly as possible. For example, the $(\neg\wedge)$ rule is checked first of all, and if it's applied then no other rule is applied, and also the given clause is not added to *inuse* (the antecedent and consequent clauses are equivalent, but this does not show in our implementation of subsumption checking). The same is true of $(\wedge)$. Then (RES) is applied, and the empty clause is searched for in the result before proceeding with the rest of the rules.

Another thing that helps pruning the search space is postponing the creation of new nominals (by application of the $\langle R\rangle$ rule) until the clause set is saturated for the current set of prefixes. Whenever the $\langle R\rangle$ rule can be applied, the application is postponed until *clauses* is empty. In a sense, this can be interpreted as exhausting the possibilities of doing propositional reasoning before doing modal reasoning.

Paramodulation. Since we need to do equality reasoning between nominals, we can once more take advantage from experience in first order resolution. In [BG98], Bachmair and Ganzinger develop in detail the modern theory of equational reasoning for first order saturation based provers. Many of the ideas and optimizations discussed there can and should be implemented in HyLoRes. In the current version, paramodulation is done naïvely, the only "optimization" being the orientation of equalities so that we always replace nominals by nominals which are lower in a certain ordering.

4.6 Testing

During the development of HyLoRes, we made extensive use of the modal test sets described in Chapter 2 to evaluate the performance of the prover and guide design decisions. Some results are shown in Figure 4.5.

Hand-tailored tests. Figure 4.5 (a) represents a set of runs of the Balsiger, Heuerding and Schwendimann test set [BHS00], with different criteria for selecting the given clause, and the description logic prover RACER [RAC], version 1-6r2, included as a reference. Even when most of this test set has become trivial for mature modal provers, it still provided a quick way to evaluate the prover in the early stages.

Random tests: Random Modal QBF test set. Figure 4.5 (b) shows a run of several versions of HyLoRes and other provers over a very easy area of the Random Modal QBF test set [Mas99]. The X axis represents the number of clauses in the original QBF formula, and the Y axis represents the average time for solving an instance, with 64 samples/datapoint. The problems range from being all satisfiable at the left, to being all unsatisfiable at the right. We benchmarked HyLoRes 0.5 (no formula indexing, no clause repository), HyLoRes 0.9 (formula indexing, clause repository, backward subsumption still using clause-at-a-time comparison) and HyLoRes 1.0 (now with backward subsumption using set-at-a-time comparison). We also ran SPASS v. 1.0.3 [SPA] with the standard translation to first order logic, MSPASS v. 1.0.0t.1.3 [MSP], *SAT version 1.3 [*SA], and RACER v. 1-6r2 on this test, to compare with more mature provers; in general the times for these provers only reflect start up times, as revealed by the absence of the easy-hard-easy pattern. This test set allowed us to gauge the progress of HyLoRes as we added optimizations to it, although since QBF derived modal formulas have a very rigid structure, as we have seen in Chapter 2, a good performance on this test set was not a guarantee of good performance overall.

Random tests: Random Modal CNF test set. As explained in Section 2.5, this test set [PSS03] generates random modal CNF formulas directly. We ran the test for $C = 2.5$, $V = 3$ and $D = 1$; Figure 4.5 (c) represents median time elapsed as a function of (number of clauses/number of variables). The timeout value was 100 seconds: again, it was too easy for mature provers to compare their performances, while for HyLoRes there were a few timeouts in the hardest area. Figure 4.5 (d) plots the satisfiable/unsatisfiable fractions in the test we just

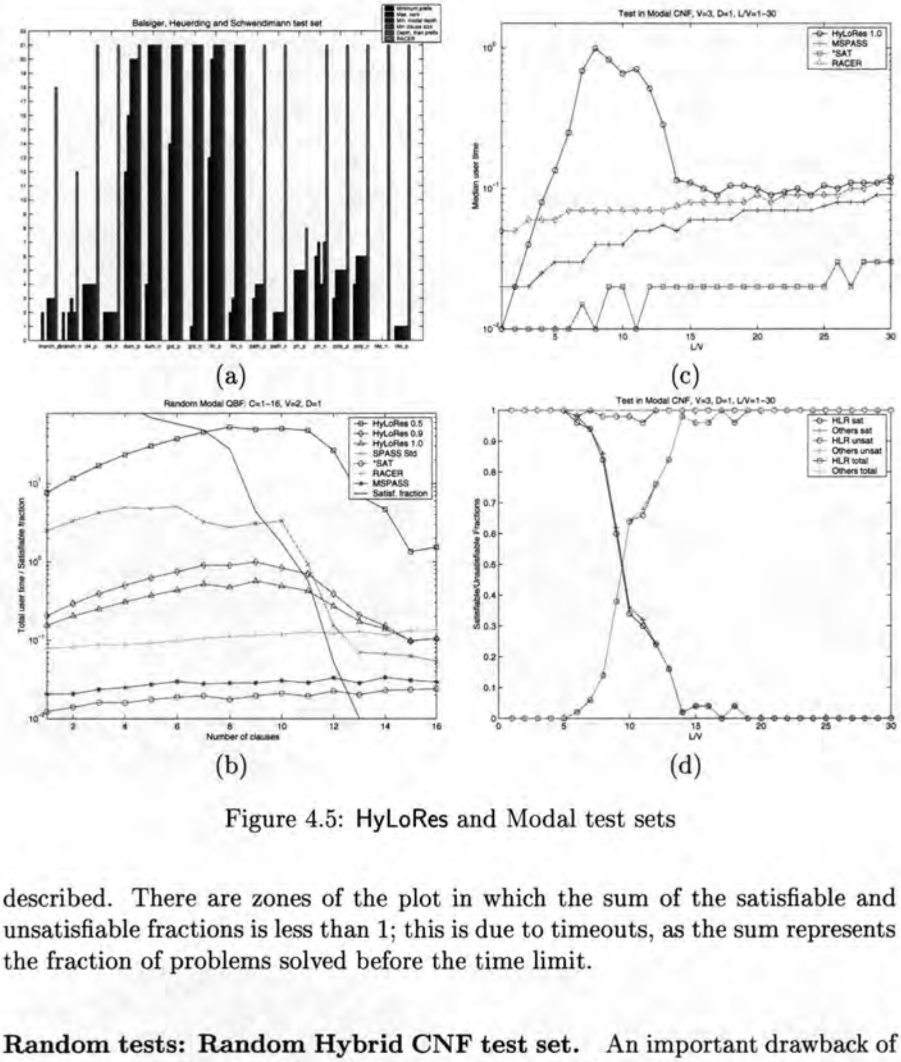
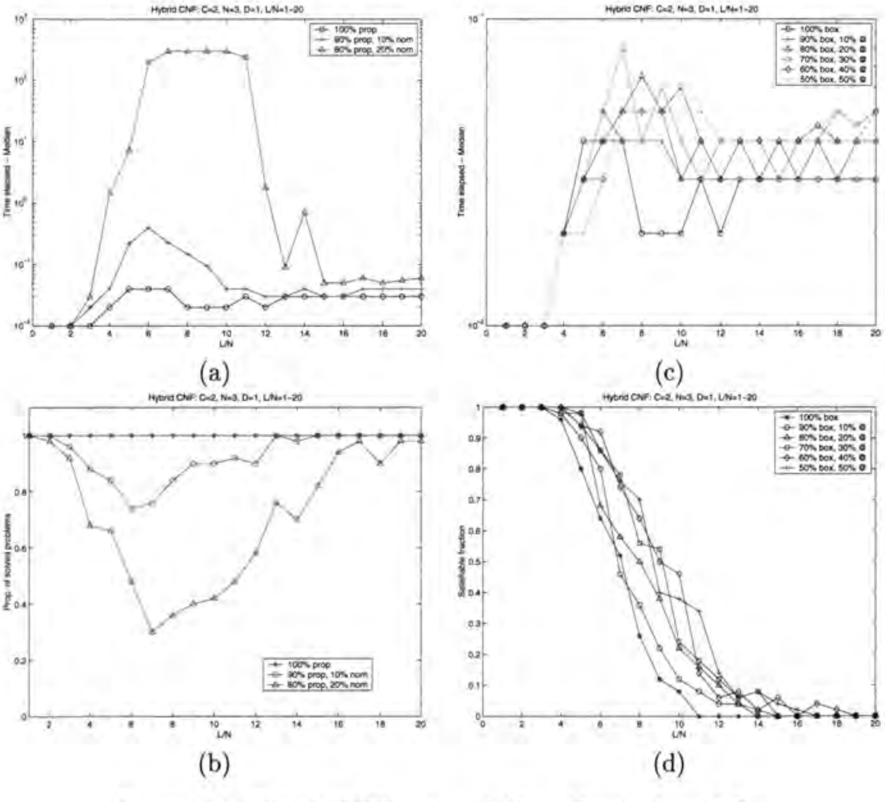


Figure 4.5: HyLoRes and Modal test sets

described. There are zones of the plot in which the sum of the satisfiable and unsatisfiable fractions is less than 1; this is due to timeouts, as the sum represents the fraction of problems solved before the time limit.

Random tests: Random Hybrid CNF test set. An important drawback of these test sets though, is that they only provide *purely modal* input. We present now some preliminary tests of the hybrid capabilities of HyLoRes, evaluated using hGen, the generator introduced in Section 2.5.

In Figure 4.6 (a) and (b) we start with a purely modal base case, with $C = 2$, $N_p = 3$, $D = 1$, and gradually add nominals to the mix; that is, with $N_n = 5$ we keep $p_{svar} = 0$ and do one run with $p_{prop} = 1$, $p_{nom} = 0$, one with $p_{prop} = 9$, $p_{nom} = 1$, and one with $p_{prop} = 8$, $p_{nom} = 2$. The timeout was 300 seconds. Figure 4.6 (a) shows the median time elapsed, while Figure 4.6 (b) shows the proportion of problems solved. Here we see that even slight increases of the quantity of nominals the difficulty rises sharply; this highlights the fact that optimizing paramodulation is crucial. Figure 4.6 (c) and (d) shows the effect of increasing the proportion of @-operators, starting from the same base case. We see that the difficulty changes very little (although the peak moves to the right), and the satisfiable/unsatisfiable transition moves to the right as we increase the proportion of @-operators. This is to be expected, in a sense, since the presence of nominals in a formula triggers the paramodulation rule (which tends to create a state explosion), while the @-operator triggers the much more benign @-rule, which just simplifies the given clause.



(a)

(c)

(b)

(d)

Figure 4.6: Hybrid CNF tests – Adding Nominals and @

4.7 Conclusion

The prototype is not yet meant to be competitive when compared with state of the art provers for modal-like logics like DLP, *SAT, MSPASS or RACER. On the one hand, the system is still in a preliminary stage of development (only very simple optimizations for hybrid logics have been implemented), and on the other hand the hybrid language and the languages handled by the other provers are related but different. $\mathcal{H}(@, \downarrow)$ is undecidable while the target languages of the other provers are decidable. And even when comparing the fragment $\mathcal{H}(@)$ for which HyLoRes implements a decision algorithm, the expressive powers are incomparable ($\mathcal{H}(@)$ permits free Boolean combinations of @ and nominals but lacks, for example, the limited form of universal modality available in the T-Box of DL provers [Are00]).

There certainly remain many things to try and improve in HyLoRes. The next steps in its development include

- a better treatment of paramodulation;
- support for the universal modality A [GP92] (which would allow us to perform inference in full Boolean knowledge bases of the description logic $\mathcal{ALCO}$);
- saving the saturated clause set, if any, for querying;
- and improve input analysis and heuristics.

But the main goal we pursued during the implementation of this prototype has largely been achieved: direct resolution can be used as an interesting, and perhaps even competitive, alternative to tableaux based methods for modal and hybrid logics.

Part II

Programming with Dynamic First Order Logic

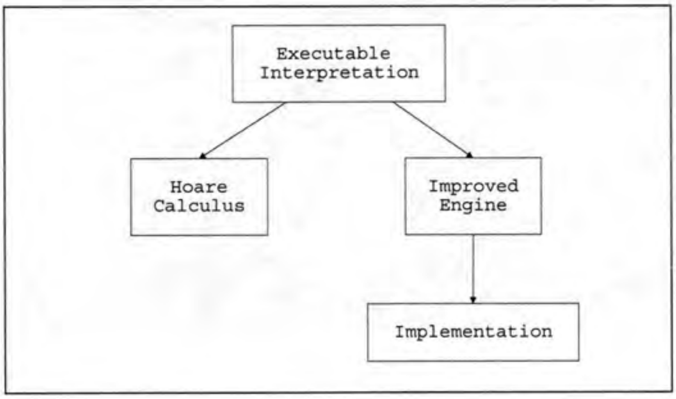


Figure 4.7: Dependency map for Part II

In this second part we will review the concept of formulas-as-programs, and introduce an executable interpretation of DFOL. The interpretation works as a *specification*: the way in which the desired computational effect is to be achieved is not part of the interpretation. This allows us to devote Chapter 6 to define a Hoare calculus for the logic, without worrying about the internal state of the language engine: if the engine is faithful to the executable program interpretation, the calculus applies to it. After introducing the calculus, in Chapter 7 we will introduce a new version of the *Dynamo* engine, which is certainly a departure from the state machine of Chapter 5: since we want to be faithful to DFOL semantics, why not use a tableau prover as the engine? This has proved to have advantages and disadvantages, as we will see in Chapter 8. Figure 4.7 gives a dependency map for this part.

Chapter 5

The Executable Program Interpretation for Dynamic First Order Logic

5.1 Introduction

In this chapter we will introduce the DFOL perspective on the "formulas as programs" paradigm as presented in [AB98]. In essence, by interpreting formulas as actions on a certain data structure, and having such actions respect the semantics of the corresponding formulas, we obtain a programming language that possesses both the power of imperative programming and a declarative semantics. We will give some background on formulas as programs, sketch the computational process approximation to DFOL(⊔) as proposed in [vE98b], and suggest some extensions. This is not how we will ultimately implement *Dynamo*, but it is provided to give some insight on the use of DFOL extensions for programming.

5.2 FOL and Programming

The idea of using FOL as a programming language is not new: a language consisting of formulas in the Horn fragment of FOL was presented in [Kow74], and marked the start of the development of the logic programming field. The benefit of having a declarative semantics for a programming language is that it makes programs easier to understand, modify and verify, since having a dual reading of a program as a logical formula makes it much simpler to reason about its correctness. The problem with the Horn fragment was that it was not expressive enough for programming purposes. Prolog, the first logic programming language, was then extended in order to reach the desired expressivity, but many of the extensions were *extralogical*: arbitrary programs cannot be read as logical formulas anymore, and soundness and completeness results have not yet been conclusively extended to programs including negation. Also, even 'pure' PROLOG programs can be hard and unintuitive to verify in a rigorous way, in part because of the use of *recursion.* To remedy the situation without sacrificing too much expressivity, Apt and Bezem [AB98] proposed a different approach, called *formulas as programs*, where a computation mechanism is suggested that relies exclusively on the basics of first order logic and replaces recursion with (bounded) iteration. The core idea is to consider the expression $v = t$, where v is a variable and t is a term, as an assignment if the value of v is not known, and as a test if it is. If t is not a grounded expression, the procedure returns an error. For any valuation α, we say that a term (or atom) is *α-closed* if all variables appearing in it have a value under α; an expression of the form $v = t$ is called an *α-assignment;* if v is a variable, t is a term, and v is not α-closed but t is.

This approach was extended in a number of ways: non-recursive *procedures*, *sorts* (ie types), *arrays*, and *bounded quantification* (bounded iteration and bounded choice). Recently there has been work on viewing FOL as a *constraint* logic programming language [AV02], introducing the possibility of storing non-grounded atoms as constraints, which greatly reduces the number of cases in which an error is returned.

5.3 Computational Process Approximations to DFOL(∪)

Following this approach, our process approximation to DFOL(∪) results from interpreting identity statements, in suitable contexts, as assignment actions, and existential quantification as *un*assignment actions. That is, when a variable becomes existentially quantified, any value it might have assigned is lost, and it becomes free to be assigned again. The reason for this is given by the semantics of DFOL(∪): let's review what the syntax and semantics of DFOL(∪) were.

5.3.1 DFOL(∪)

Let a first order signature be given. We assume that variables can be built from a set VAR of initial variables by means of appending indices. Let f and P range over the function and relation symbols, with arities n as specified by the signature. We assume that terms range over the natural numbers, and that f and P denote recursive functions and predicates on $\mathbb{N}$. As stated in Definition 1.4.12, the terms and formulas of DFOL(∪) over this signature are given by:

$$\begin{array}{lcll} \mathsf{TERMS} & := & v_{t_1,\dots,t_n} \mid f\bar{t} & \text{(Terms)} \\ \mathsf{FORMS} & := & \exists v \mid P\bar{t} \mid t_1 \doteq t_2 \mid \neg((\phi)) \mid \phi_1; \phi_2 \mid (\phi_1 \cup \phi_2) & \text{(Formulas)} \end{array}$$

where $v \in \mathsf{VAR}$, $t, t_1, t_2 \in \mathsf{TERMS}$, $\bar{t} \in \mathsf{TERMS} \times \cdots \times \mathsf{TERMS}$, and $\phi, \phi_1, \phi_2 \in \mathsf{FORMS}$.

In Definition 1.4.16 we introduced the semantics of DPL and extensions, and hinted at an alternative way to interpret the semantics of a formula ϕ, as a function from assignments to sets of assignments. Let's spell that interpretation out in more detail:

5.3.1. Definition. [Functional Interpretation of DFOL(∪) in a model $\mathcal{M} = (D, I)$] For $N, N_1, N_2 \in \mathbb{N}, t, t_1, t_2 \in \mathsf{TERMS}$,

$$\begin{array}{lcl}
(v[t_1]\cdots[t_n])^s & := & s(v_{t_1^s,\ldots,t_n^s}) \\
(ft_1\cdots t_n)^s & := & I(f)t_1^s\cdots t_n^s \\
[\![\bot]\!]_s & := & \emptyset \\
[\![Pt_1\cdots t_n]\!]_s & := & \begin{cases} \{s\} & \text{if } (t_1^s,\ldots,t_n^s) \in I(P) \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![t_1 \doteq t_2]\!]_s & := & \begin{cases} \{s\} & \text{if } t_1^s = t_2^s \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\exists v]\!]_s & := & \{s' \in D^V \mid s' \sim_v s\} \\
[\![\neg\phi]\!]_s & := & \begin{cases} \{s\} & \text{if } [\![\phi]\!]_s = \emptyset \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\phi_1;\phi_2]\!]_s & := & \bigcup\{[\![\phi_2]\!]_{s'} \mid s' \in [\![\phi_1]\!]_s\} \\
[\![\phi_1 \cup \phi_2]\!]_s & := & [\![\phi_1]\!]_s \cup [\![\phi_2]\!]_s
\end{array}$$

We extend the logic with the following constructs:

$$\begin{array}{lcl}
[\![\phi^0]\!]_s & := & \{s\} \\
[\![\phi^{N+1}]\!]_s & := & [\![\phi;\phi^N]\!]_s \\
[\![\phi^t]\!]_s & := & [\![\phi^{t^s}]\!]_s \\
[\![\cup^v_{N_1..N_2}\phi]\!]_s & := & \begin{cases} [\![v = N_1;\phi]\!]_s \cup\cdots\cup [\![v = N_2;\phi]\!]_s & \text{if } N_1 \leq N_2 \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\cup^v_{t_1..t_2}\phi]\!]_s & := & \begin{cases} [\![v = t_1;\phi]\!]_s \cup\cdots\cup [\![v = t_2;\phi]\!]_s & \text{if } t_1^s \leq t_2^s \\ \emptyset & \text{otherwise.} \end{cases}
\end{array}$$

In this interpretation of formulas as functions from valuations to sets of valuations, existential quantification would require the set $[\![\exists x]\!]_s$ to consist of all the valuations u such that $s \sim_x u$. Since our domain D is usually $\mathcal{N}$, computing this set is not possible. Therefore, by uninitializing the variable, we simply desist from trying all possible values of x, in favor of trying to find those that make the rest of the formula true.

Computation states are partial maps from the set of variables to values in the domain of quantification; if a state s does not have a value for v but does have values for all variables occurring in t, then $v \doteq t$ and $t \doteq v$ can be interpreted as instructions to extend s with the pair (v, t^s).

What we want from an executable process interpretation is the following: (1) if the interpretation computes an answer valuation, then that answer is correct according to the semantics of DFOL, and (2) if the executable process interpretation returns a negative answer then there are no answers according to the semantics of DFOL.

This notion can be formalized as follows. Let $\mathcal{A}$ be the set of all possible valuations, ie $\{s \in D^X \mid X \subseteq \mathsf{VAR}\}$. We introduce the notation $t^s =\downarrow$ when t is s-closed and $t^s =\uparrow$ when it is not. A set of computed states may contain an uninformative state $\bullet$, signifying that at least one computation attempt was given up. We measure the degree of informativeness of an answer by means of a suitable ordering $\sqsubseteq$ on $\mathcal{P}(\mathcal{A} \cup \{\bullet\})$ defined by:

$$A \sqsubseteq B := (\bullet \in A \wedge A - \{\bullet\} \subseteq B) \vee (\bullet \notin A \wedge A = B).$$

This makes $(\mathcal{P}(\mathcal{A} \cup \{\bullet\}), \sqsubseteq)$ into a complete partial order (CPO), with $\{\bullet\}$ as bottom element.

For $s \in \mathcal{A}$, let $s^\circ := \{b \in D^{\mathsf{VAR}} \mid s \subseteq b\}$. Let $\bullet^\circ := \{\bullet\}$. Lift this operation to subsets of $\mathcal{A} \cup \{\bullet\}$ by means of $A^\circ := \bigcup_{s \in A} s^\circ$.

Then, a computation procedure $F : L \to \mathcal{A} \to \mathcal{P}(\mathcal{A} \cup \{\bullet\})$, where L is a language of DFOL, is a *faithful approximation of DFOL* if for all $\phi \in L$, all $s \in \mathcal{A}$:

$$(F_\phi(s))^\circ \sqsubseteq \bigcup_{b \in s^\circ} [\![\phi]\!]_b.$$

The computational strengths of procedures $F, G : L \to \mathcal{A} \to \mathcal{P}(\mathcal{A} \cup \{\bullet\})$, can be compared by lifting our $\sqsubseteq$ ordering to the level of computation maps, as follows:

$$F \sqsubseteq G := \forall \phi \in L\ \forall s \in \mathcal{A} : F_\phi(s) \sqsubseteq G_\phi(s).$$

A computation procedure G is a better approximation to DFOL than F if $F \sqsubseteq G$ and G is faithful to DFOL. In [vE98b], a computation mechanism faithful to DFOL is presented; we will now give a brief review, and present an improvement on it.

5.3.2. Definition. [State] The output $\mathbf{a}$ (alt. $\mathbf{b}, \ldots$) of a computation is represented as a triple (a, g^a, l^a) $((b, g^b, l^b), \ldots)$, where a is a valuation, g^a is a list of *global* variables, meaning those that are not existentially quantified, and l^a is a list of *local* or existentially quantified variables. The reason for this is the interaction between the use of equality as assignment and the treatment of negation: intuitively, we consider the evaluation of $\neg\phi$ to fail if the evaluation of ϕ succeeds *without making global assignments.* We want to distinguish between cases in which extension of the input assignment occurs inside a negated formula from the "normal" case in which it occurs in a positive context. We call *unsafe* those cases in which extension of the input assignment occurs during evaluation of negated formulas, because they result in, well, unsafe conclusions. For example, we want the formula $x \doteq 1$ to succeed on the empty assignment ϵ (and assign 1 to x), but we do not want $\neg(x \doteq 1)$ to fail on ϵ, as it would mean that there is no x that is equal to 1. Note that the formula $\neg(\exists x; x \doteq 1)$ *must* fail on any input; hence the need to distinguish between variables that are local, ie existentially quantified, and global, or free. In cases where a computation would be unsafe or there is insufficient data to perform it, we 'give up' on the computation, and its output will be the $\bullet$ state. Then, our *state* for the executable process interpretation function is of type $((\mathcal{A} \times \mathcal{P}(\mathsf{VAR}) \times \mathcal{P}(\mathsf{VAR})) \cup \{\bullet\})$, and our executable interpretation function proper is of type $\mathsf{FORMS} \times ((\mathcal{A} \times \mathcal{P}(\mathsf{VAR}) \times \mathcal{P}(\mathsf{VAR})) \cup \{\bullet\}) \rightarrow \{\mathcal{A} \times \mathcal{P}(\mathsf{VAR}) \times \mathcal{P}(\mathsf{VAR})\} \cup \{\bullet\}$, where $(T_1 \cup T_2)$ means "either type T_1 or type T_2".

5.3.3. DEFINITION. [Safe states] A state $\mathbf{b}$ is *safe* for (a, g^a, l^a) if $\mathbf{b} \neq \bullet$ and $l^a \cup g^b \subseteq dom(a)$.

5.3.4. DEFINITION. [Risky states] A set of states $\mathbf{B}$ is *risky* for (a, g^a, l^a) if $\mathbf{B} \neq \emptyset$, but no member $\mathbf{b}$ of $\mathbf{B}$ is safe for (a, g^a, l^a).

5.3.5. DEFINITION. [Executable process interpretation for DFOL]

$$
\begin{array}{lcl}
[\![\phi]\!](\bullet) & := & \{\bullet\} \\
[\![\bot]\!](a, g^a, l^a) & := & \emptyset \\
[\![Pt_1 \cdots t_n]\!](a, g^a, l^a) & := & \begin{cases} \{(a, g^a, l^a)\} & \text{if } Pt_1 \cdots t_n \textit{a-closed, } Pt_1 \cdots t_n \in I(P), \\ \emptyset & \text{if } Pt_1 \cdots t_n \textit{a-closed, } Pt_1 \cdots t_n \notin I(P), \\ \bullet & \text{if } Pt_1 \cdots t_n \textit{not a-closed.} \end{cases} \\
[\![\exists v]\!](a, g^a, l^a) & := & \{(a - \{v/v^a\}, g^a, l^a \cup \{v\})\} \\
[\![\neg\phi]\!](a, g^a, l^a) & := & \begin{cases} \{(a, g^a, l^a)\} & \textit{if } [\![\phi]\!](a, g^a, l^a) = \emptyset \\ \emptyset & \textit{if } \exists\ b \in [\![\phi]\!](a, g^a, l^a) \textit{with } b \textit{ safe for} (a, g^a, l^a) \\ \{\bullet\} & \textit{if } [\![\phi]\!](a, g^a, l^a) \textit{ is risky for } (a, g^a, l^a) \end{cases} \\
[\![\phi_1; \phi_2]\!](a, g^a, l^a) & := & \bigcup\{[\![\phi_2]\!](\mathbf{b}) \mid \mathbf{b} \in [\![\phi_1]\!](a, g^a, l^a) \\
[\![\phi_1 \cup \phi_2]\!](a, g^a, l^a) & := & [\![\phi_1]\!](a, g^a, l^a) \cup [\![\phi_2]\!](a, g^a, l^a)
\end{array}
$$

This far, we are simply checking a formula against a valuation; the treatment of $\doteq$ that follows is what makes our system a computation engine.

$$[\![(t_1 \doteq t_2)]\!](a, g^a, l^a) \quad := \quad \begin{cases} \{(a, g^a, l^a)\} & \textit{if} \quad t_1, t_2 \textit{ a-closed, } t_1^a = t_2^a, \\ \emptyset & \textit{if} \quad t_1, t_2 \textit{ a-closed, } t_1^a \neq t_2^a, \\ \{(a \cup \{v/t_2^a\}, g^a, l^a)\} & \textit{if} \quad t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \qquad t_1 \equiv v, v^a =\uparrow, t_2^a =\downarrow, v \in l^a \\ \{(a \cup \{v/t_2^a\}, g^a \cup \{v\}, l^a)\} & \textit{if} \quad t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \qquad t_1 \equiv v, v^a =\uparrow, t_2^a =\downarrow, v \notin l^a \\ \{(a \cup \{v/t_1^a\}, g^a, l^a)\} & \textit{if} \quad t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \qquad t_2 \equiv v, v^a =\uparrow, t_1^a =\downarrow, v \in l^a \\ \{(a \cup \{v/t_1^a\}, g^a \cup \{v\}, l^a)\} & \textit{if} \quad t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \qquad t_2 \equiv v, v^a =\uparrow, t_1^a =\downarrow, v \notin l^a \\ \{\bullet\} & \textit{if} \quad t_1, t_2 \textit{ not a-closed} \\ & \qquad \textit{and not an a-assignment} \end{cases}$$

5.4 DFOL(∪) as a Programming Language

We will give a few examples of DFOL(∪) formulas, viewed as programs. The formula

$$(x \geq y; z \doteq x) \cup (x < y; z \doteq y)$$

will check whether z is equal to $max(x, y)$. If we want to assign the maximum of x and y to z, we first unassign it by existential quantification:

$$\exists z; ((x \geq y; z \doteq x) \cup (x < y; z \doteq y))$$

DFOL, viewed as a programming language, gives a new perspective on a fundamental feature of imperative programming, the destructive assignment command $x := t$. Take the command $x := x+1$ that increments x. This cannot be rendered as identity, for the identity $x = x+1$ either gives an error message (in cases where the input valuation is not defined for x) or it fails, on the natural numbers at least (for there is no $n \in \mathbb{N}$ with $n = n + 1$). But if we implement the use of an auxiliary ('shadow') variable x' and dynamic quantification over both x and x', we can express $x := x + 1$ with the DFOL formula

$$x' = x + 1; \exists x; x = x'; \exists x',$$

where the final $\exists x'$ is used for unassigning x' for future uses.

If we assume that each regular variable v comes with a unique shadow v' we can abbreviate this as $v \blacktriangleleft t$; we call this *safe assignment.*

5.5 Moving Closer to DFOL(∪) Semantics

In the first incarnation of our executable interpretation, the program state is either a triple (a, g^a, l^a) or $\bullet$. This means that if a statement $x \doteq t$ is not a-closed and not an a-assignment, the result has to be the uninformative state $\bullet$. For example, if $x \doteq 2; x \doteq y$ is computed in a state ϵ (undefined for every variable), then the result is $\{\{x/2, y/2\}\}$ (there is a single output state that maps both of x, y to 2). If we interchange the statements, and compute $x \doteq y; x \doteq 2$ for input state ϵ, the result is the completely uninformative set $\{\bullet\}$, while the assignments that satisfy the two formulas are exactly the same. We believe we can produce an interpretation that is a better approximation to the semantics of DFOL, since the same valuation that satisfies the first formula satisfies also the second one. To accomplish this we extend the states with some further components. The fourth component is a list of literals $Pt_1 \cdots t_n$, $t_1 \doteq t_2$, $\neg Pt_1 \cdots t_n$, $\neg t_1 \doteq t_2$. Since assignments to global variables inside negated formulas make the computation path unsafe, we will, when in the appropriate mode, save $v \doteq t$ as a constraint rather than perform a global assignment to v. The two *execution modes* we distinguish are B(uild) and C(onstrain). We will now present the execution mechanism as a set of transition rules; in the rules where the execution mode does not matter but has to remain the same during a given transition step, we will use m as a variable ranging over B, C. Also, we will need to keep track of the set of variables used somewhere in the current list of unresolved literals. If v is used in the list of unresolved literals, a dynamic quantifier action $\exists v$ would sever the literals that include v from the computation path, so we need to keep track of such situations. We will introduce a new register n^a for constraint variables *needed* by state **a**. A state **a** will now look like $(a, g^a, l^a, n^a, L^a, m^a)$, where

- a is a partial valuation,
- g^a is the set of global variables,
- l^a is the set of local variables,
- n^a is the set of variables needed in a stored constraint,
- L^a is the list of literals stored as constraints,
- m^a is either b or c, the execution mode of the state **a**.

The role of these components in the state transitions will become clear when introduce the transition rules.

When, in build mode, we cannot perform an atomic test or cannot execute an equality statement (either as a test or as an assignment) due to missing values in the input, we store the atom after substituting the values of the current state and add the variables that are still needed to the set of needed variables. When, in constrain mode, we cannot perform an atomic test or execute an equality as a test statement due to missing values in the input, we do the same. Then, if a variable in n^a is assigned a value, the corresponding literals in L^a are updated, and evaluated if they become a-closed or a-assignments (a *reduction* step). This can cause yet more variables from n^a to be assigned values, which triggers yet another reduction.

5.5.1 • and 0 propagation

For conceptual clarity, we use an explicit failure state **0**. Computation of ϕ from state **a** fails if all ϕ-computation paths starting from **a** end in **0**. We will sometimes need this to ensure that no further reduction attempts will be made on **a**. Both **0** and the improper state •, for 'I don't know', are treated as a states from which no recovery is possible.

$$\frac{}{\bullet \xrightarrow{\phi} \bullet} \qquad\qquad \frac{}{\mathbf{0} \xrightarrow{\phi} \mathbf{0}}$$

5.5.2 Atomic Predicate Test

In case we cannot perform an atomic test due to missing values in the input, we store the set of needed variables (those for which no values were available), along with the literal. If a is the input valuation and $Pt_1 \cdots t_n$ is the predicate, the a-instance of $Pt_1 \cdots t_n$ is given by $Pt_1^a \cdots t_n^a$, and the set of needed variables by: $var(t_1^a..t_n^a)$. A test that fails produces a transition to the failure state. It makes no difference whether we are in build mode or constrain mode.

$$\frac{Pt_1 \cdots t_n \ a\text{-closed and } (t_1^a, \ldots, t_n^a) \in I(P)}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{Pt_1 \cdots t_n} (a, g^a, l^a, n^a, L^a, m^a)}$$

$$\frac{Pt_1 \cdots t_n \ \text{ not } a\text{-closed}}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{Pt_1 \cdots t_n} (a, g^a, l^a, n^a \cup W, L; Pt_1^a \cdots t_n^a, m^a)} \; W = var(t_1^a..t_n^a)$$

$$\frac{Pt_1 \cdots t_n \ a\text{-closed and } (t_1^a, \ldots, t_n^a) \notin I(P)}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{Pt_1 \cdots t_n} \mathbf{0}}$$

5.5.3 Equality

If an assignment to v makes the valuation grow, then we have to adjust the list of needed variables (by removing v from it), and the list of constraints (by performing the substitution v/d, where d is the computed value for v). However, even if an equality $t_1 \doteq t_2$ is an assignment for the current valuation, we need not always *perform* the assignment: we will only do so when the assignment is to a local variable (dynamically bound in the current context), or to a global variable while we are in *build* mode. When we are in *constrain* mode all identities that are not tests will be put on the constraint list. We will use $L[v/d]$ for the result of performing substitution $[v/d]$ to every member of L.

The simplest case is the case where $t_1 \doteq t_2$ is a test. In this case it makes no difference whether we are in build or constrain mode. Again, we model failure explicitly by means of a transition to $\mathbf{0}$. We get:

$$\frac{t_1 \doteq t_2 \text{ } a\text{-closed and } t_1^a = t_2^a}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{t_1 \doteq t_2} (a, g^a, l^a, n^a, L^a, m^a)}$$

$$\frac{t_1 \doteq t_2 \text{ } a\text{-closed and } t_1^a \neq t_2^a}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{t_1 \doteq t_2} \mathbf{0}}$$

If $t_1 \doteq t_2$ is an assignment to a variable v that is local to the current context, the assignment is performed, the variable v is removed from the list of needed variables, and the relevant substitution $[v/d]$ is performed on the list elements. It makes no difference whether we are in build or constrain mode:

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \in l^a}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{t_1 \doteq t_2} (a \cup \{v/t_2^a\}, g^a, l^a, n^a - \{v\}, L^a[v/t_2^a], m^a)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \in l^a}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{t_1 \doteq t_2} (a \cup \{v/t_1^a\}, g^a, l^a, n^a - \{v\}, L^a[v/t_1^a], m^a)}$$

If $t_1 \doteq t_2$ is an assignment to a variable v that is global to the current context, what we will do depends on the execution mode. In build mode, we perform the assignment, remove v from the list of needed variables, and carry out the relevant substitution $[v/d]$ on the list elements. In constrain mode, we save the identity on the list.

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \notin l^a}{(a, g^a, l^a, n^a, L^a, b) \xrightarrow{t_1 \doteq t_2} (a \cup \{v/t_2^a\}, g^a \cup \{v\}, l^a, n^a - \{v\}, L^a[v/t_2^a], b)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \notin l^a}{(a, g^a, l^a, n^a, L^a, c) \xrightarrow{t_1 \doteq t_2} (a, g^a, l^a, n^a \cup \{v\}, L; t_1^a \doteq t_2^a, c)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \notin l^a}{(a, g^a, l^a, n^a, L^a, b) \xrightarrow{t_1 \doteq t_2} (a \cup \{v/t_1^a\}, g^a \cup \{v\}, l^a, n^a - \{v\}, L^a[v/t_1^a], b)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \notin l^a}{(a, g^a, l^a, n^a, L^a, c) \xrightarrow{t_1 \doteq t_2} (a, g^a, l^a, n^a \cup \{v\}, L; t_1^a \doteq t_2^a, c)}$$

Finally, for input states where $t_1 \doteq t_2$ is neither an assignment nor a test, we save the identity on the list.

$$\frac{t_1 \doteq t_2 \text{ not an } a\text{-assignment and not } a\text{-closed}}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{t_1 \doteq t_2} (a, g^a, l^a, n^a \cup W, L; t_1^a \doteq t_2^a, m^a)} \quad W = var(t_1^a, t_2^a)$$

5.5.4 Predicate Test Reduction

For efficiency reasons, we indicate the results of a reduction resulting in failure by means of a transition to **0**. This prevents the futile application of other rules to the state: we make sure that there are no transitions from **0**.

Due to the fact that identity statements make the valuation grow, test predicates on the unresolved literal list may turn into grounded literals (literals without variables, i.e., $\emptyset$-closed literals), in which case we can perform the test. It does not matter where a test literal occurs on the list. We indicate this with $L(Pt_1 \cdots t_n)$. In the same context, $L()$ indicates the result of removing all occurrences of $Pt_1 \cdots t_n$ from the list L.

$$\frac{Pt_1 \cdots t_n \text{ grounded and } (t_1, \ldots, t_n) \in I(P)}{(a, g^a, l^a, n^a, L(Pt_1 \cdots t_n), m^a) \xrightarrow{r} (a, g^a, l^a, n^a, L(), m^a)}$$

$$\frac{Pt_1 \cdots t_n \text{ grounded and } (t_1, \ldots, t_n) \notin I(P)}{(a, g^a, l^a, n^a, L(Pt_1 \cdots t_n), m^a) \xrightarrow{r} \mathbf{0}}$$

$$\frac{Pt_1 \cdots t_n \text{ grounded and } (t_1, \ldots, t_n) \notin I(P)}{(a, g^a, l^a, n^a, L(\neg Pt_1 \cdots t_n), m^a) \xrightarrow{r} (a, g^a, l^a, n^a, L(), m^a)}$$

$$\frac{Pt_1 \cdots t_n \text{ grounded and } (t_1, \ldots, t_n) \in I(P)}{(a, g^a, l^a, L(\neg Pt_1 \cdots t_n), m^a) \xrightarrow{r} \mathbf{0}}$$

5.5.5 Equality Test Reduction

In case an equality or inequality on the unresolved literal list is a test for the current input, the treatment is as for atomic tests.

$$\frac{t_1, t_2 \text{ grounded and } t_1 = t_2}{(a, g^a, l^a, n^a, L(t_1 \doteq t_2), m^a) \xrightarrow{r} (a, g^a, l^a, n^a, L(), m^a)}$$

$$\frac{t_1, t_2 \text{ grounded and } t_1 \neq t_2}{(a, g^a, l^a, n^a, L(t_1 \doteq t_2), m^a) \xrightarrow{r} \mathbf{0}}$$

$$\frac{t_1, t_2 \text{ grounded and } t_1 \neq t_2}{(a, g^a, l^a, n^a, L(\neg t_1 \doteq t_2), m^a) \xrightarrow{r} (a, g^a, l^a, n^a, L(), m^a)}$$

$$\frac{t_1, t_2 \text{ grounded and } t_1 = t_2}{(a, g^a, l^a, n^a, L(\neg t_1 \doteq t_2), m^a) \xrightarrow{r} \mathbf{0}}$$

5.5.6 Assignment Reduction

If an equality occurs anywhere in the unresolved literal list that is an assignment for the current input, then it can be used to extend the input valuation, provided we are in *build* mode.

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \in l^a}{(a, g^a, l^a, n^a, L(t_1 \doteq t_2), b) \xrightarrow{r} (a \cup \{v/t_2^a\}, g^a, l^a, n^a - \{v\}, L()[v/t_2^a], b)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \notin l^a}{(a, g^a, l^a, n^a, L(t_1 \doteq t_2), b) \xrightarrow{r} (a \cup \{v/t_2^a\}, g^a \cup \{v\}, l^a, n^a - \{v\}, L()[v/t_2^a], b)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \in l^a}{(a, g^a, l^a, n^a, L(t_1 \doteq t_2), b) \xrightarrow{r} (a \cup \{v/t_1^a\}, g^a, l^a, n^a - \{v\}, L()[v/t_1^a], b)}$$

$$\frac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \notin l^a}{(a, g^a, l^a, n^a, L(t_1 \doteq t_2), b) \xrightarrow{r} (a \cup \{v/t_1^a\}, g^a \cup \{v\}, l^a, n^a - \{v\}, L()[v/t_1^a], b)}$$

5.5.7 Quantification

If we encounter a quantifier $\exists v$ in a state with a list L with at least one literal with v occurring in it, then we are in trouble. The bookkeeping device for the set of needed variables for list reduction is n^a. In case $v \in n^a$, there is nothing we can do but go to the state of irrecoverable error. The reason is that there is an unresolved test involving v on the list, and that test cannot be postponed any further.

$$\frac{v \in n^a}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\exists v} \bullet}$$

If $\exists v$ is executed in a state $(a, g^a, l^a, n^a, L^a, m^a)$, and none of the literals in L needs v, we throw away the old a-value of v (if any), and put v in the local variable register.

$$\frac{v \in dom(a), v \notin n^a}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\exists v} (a - \{v/v^a\}, g^a, l^a \cup \{v\}, n^a, L^a, m^a)}$$

5.5.8 Negation

As the treatment of negation is rather involved, some preliminary definitions are useful. Here is a definition of a reduced computation.

$$\begin{aligned}\mathbf{a} \xrightarrow{\phi}_r \mathbf{b} \;:\equiv\; & \text{either } \mathbf{a} \xrightarrow{\phi} \mathbf{b} \text{ and there is no } \mathbf{b}' \text{ with } \mathbf{b} \xrightarrow{r} \mathbf{b}' \\ & \text{or } \mathbf{a} \xrightarrow{\phi} \mathbf{a}_1 \xrightarrow{r} \cdots \xrightarrow{r} \mathbf{b}, \text{ and there is no } \mathbf{b}' \text{ with } \mathbf{b} \xrightarrow{r} \mathbf{b}'.\end{aligned}$$

We define $\mathbf{a}^\circ$ by means of:

$$(a, g^a, l^a, n^a, L^a, m^a)^\circ := (a, dom(a), \emptyset, n^a, L^a, c).$$

In other words, $\mathbf{a}^\circ$ denotes the result of putting the list of global variables of $\mathbf{a}$ equal to the domain of the valuation, making the list of local variables of $\mathbf{a}$ empty, and putting the state in constrain mode.

In terms of $\xrightarrow{\phi}_r$ we define the set of all outcomes of computations ϕ starting from $\mathbf{a}^\circ$, as follows:

$$O_{\mathbf{a}}(\phi) := \{\mathbf{b} \mid \mathbf{a}^\circ \xrightarrow{\phi}_r \mathbf{b}\}.$$

In other words, $O_{\mathbf{a}}(\phi)$ is the set of all fully reduced output states that are the result of executing ϕ in state $\mathbf{a}^\circ$.

Next, note that if we take care to always execute formulas in the scope of negation in constrain mode (this is part of the definition of $\mathbf{a}^\circ$), no global assignment ever takes place (as is easily verified by inspection of the rules). In constrain mode, instead of assigning a new value to a global variable v, we put a constraint on v on the list. In other words, the members $\mathbf{b}$ of $O_{\mathbf{a}}(\phi)$ will all be safe, in the sense that they all will satisfy $g^b = dom(a)$.

The members of $O_{\mathbf{a}}(\phi)$ fall in the following categories:

- $\mathbf{b}$ is simple if $\mathbf{b}$ has the form $(b, g^b, l^b, \emptyset, \emptyset, c)$. A simple state is one with an empty list of literals.
- $\mathbf{b}$ is constrained if $\mathbf{b}$ has the form $(b, g^b, l^b, n^a, L^a, c)$, with $n^a \neq \emptyset$, $L \neq \emptyset$.
- $\mathbf{b} = \mathbf{0}$: the failed state.
- $\mathbf{b} = \bullet$: the *don't know* state.

In cases where the set of outcomes $O_{\mathbf{a}}(\phi)$ contains at least one simple state, we know that the embedded computation has succeeded, so the computation of $\neg\phi$ from $\mathbf{a}$ should fail (we should get an explicit transition to $\mathbf{0}$):

$$\boxed{\dfrac{\mathbf{b} \in O_{\mathbf{a}}(\phi) \text{ with } \mathbf{b} \text{ simple.}}{\mathbf{a} \xrightarrow{\neg\phi} \mathbf{0}}}$$

If $O_{\mathbf{a}}(\phi) = \{\mathbf{0}\}$, we know that the embedded computation has failed, so the computation of $\neg\phi$ from $\mathbf{a}$ should succeed:

$$\boxed{\frac{O_{\mathbf{a}}(\phi) = \{\mathbf{0}\}}{\mathbf{a} \xrightarrow{\neg\phi} \mathbf{a}}}$$

If the set $O_{\mathbf{a}}(\phi) - \{\mathbf{0}\}$ is non-empty and contains only constrained states, we can dualize it. If l is an atom or identity A, then $\bar{l}$ is its negation $\neg A$; if c is a negated atom or identity $\neg A$, then $\bar{c}$ is the unnegated literal A. We call $\bar{c}$ the complement of c.

Dualisation of a list of constrained states means constructing all lists of literals that result from picking the complement of a literal on the constraint list of each of the constrained states, provided no variable in such a literal is in the local variable list.

The reason for the proviso is that negating a constraint with an existentially quantified variable cannot be expressed as a literal constraint on variables. At a later stage we might wish to take such 'universal constraints' on board as well, but here we refrain from doing so, and in such cases we simply admit defeat and give up. The function U from lists of states to $\mathcal{P}(\{\bullet\})$ indicates whether dualizing a list of states gives rise to universal constraints:

$$U(\mathbf{b}_1, \ldots, \mathbf{b}_n) := \begin{cases} \{\bullet\} & \text{if for some } j \text{ with } 1 \leq j \leq n, var(L^{\mathbf{b}_j}) \cap l^{\mathbf{b}_j} \neq \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

Extracting the dual lists of constraints from a list of constrained states is done with D, defined as follows:

$$D(\mathbf{b}_1, \ldots, \mathbf{b}_n) := \{(\bar{c_1}; \cdots; \bar{c_n}) \mid c_j \in L^{\mathbf{b}_j}, var(c_j) \cap l^{\mathbf{b}_j} = \emptyset \text{ for } 1 \leq j \leq n\}.$$

Since a set of states represents the disjunction of all the possible execution paths to the present, and the list of literals in each state is to be read as a conjunction, the dualization is simply the result of negating the whole disjunction while keeping the disjunctive form. E.g., if $\mathbf{b}_1$ has constraint list $L_1 = (Px; Rxy)$ and $\mathbf{b}_2$ has constraint list $L_2 = (Qx; \neg Sxz)$, and $x, y \notin l^{\mathbf{b}_1}$, $x, z \notin l^{\mathbf{b}_2}$, then

$$D(\mathbf{b}_1, \mathbf{b}_2) = \{(\neg Px; \neg Qx), (\neg Px; Sxz), (\neg Rxy; \neg Qx), (\neg Rxy; Sxz)\}.$$

If L is a list of literals, $var(L)$ is its list of needed variables.

We now use dualization to compute the continuations of a state $\mathbf{a}$, given negated constrained states $\mathbf{b}_1, \ldots, \mathbf{b}_n$. Assume that $\mathbf{a}$ has the form $(a, g^a, l^a, n^a, L^a, m^a)$:

$$\mathbf{D_a}(\mathbf{b}_1, \ldots, \mathbf{b}_n) := \{(a, g^a \cup vL', l^a, n^a \cup var(L'), L^a; L', m^a) \mid L' \in D(\mathbf{b}_1, \ldots, \mathbf{b}_n)\} \cup U(\mathbf{b}_1, \ldots, \mathbf{b}_n).$$

where vL' is the set of global variables present on L'.

The next rule uses dualisation to compute appropriate lists of literals.

$$\frac{\mathbf{B} = O_{\mathbf{a}}(\phi) - \{\mathbf{0}\} \neq \emptyset, \text{ all members of } \mathbf{B} \text{ constrained}, \mathbf{b} \in \mathbf{D}_{\mathbf{a}}(\mathbf{B})}{\mathbf{a} \xrightarrow{\neg\phi} \mathbf{b}}$$

Finally, we need a rule to specify the cases where the negation cannot be correctly computed. This happens when $\bullet \in O_{\mathbf{a}}(\phi)$, while $O_{\mathbf{a}}(\phi)$ does not contain any simple states.

$$\frac{\text{No } \mathbf{b} \in \mathbf{B} = O_{\mathbf{a}}(\phi) \text{ simple}, \bullet \in \mathbf{B}}{\mathbf{a} \xrightarrow{\neg\phi} \bullet}$$

5.5.9 Composition, Union, Bounded Search/Choice

Nothing out of the ordinary here. Using $\mathbf{a}, \mathbf{b}, \mathbf{c}$ as shorthand for $(a, g^a, l^a, n^a, L^a, m^a)$ etc, we get:

$$\frac{\mathbf{a} \xrightarrow{\phi_i} \mathbf{b}}{\mathbf{a} \xrightarrow{\phi_1 \cup \phi_2} \mathbf{b}}\; i \in \{1,2\} \qquad \frac{\mathbf{a} \xrightarrow{\phi_1} \mathbf{b} \quad \mathbf{b} \xrightarrow{\phi_2} \mathbf{c}}{\mathbf{a} \xrightarrow{\phi_1;\phi_2} \mathbf{c}}$$

$$\frac{}{\mathbf{a} \xrightarrow{\phi^0} \mathbf{a}} \qquad \frac{\mathbf{a} \xrightarrow{\phi} \mathbf{b} \quad \mathbf{b} \xrightarrow{\phi^N} \mathbf{c}}{\mathbf{a} \xrightarrow{\phi^{N+1}} \mathbf{c}}$$

$$\frac{}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\phi^t} \bullet}\; t^a =\uparrow \qquad \frac{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\phi^N} \mathbf{b}}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\phi^t} \mathbf{b}}\; t^a = N$$

$$\frac{}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\cup^v_{t_1..t_2}\phi} \bullet}\; t^a_1 =\uparrow \qquad \frac{}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\cup^v_{t_1..t_2}\phi} \bullet}\; t^a_2 =\uparrow$$

$$\frac{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{v=j;\phi} \mathbf{b}}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{\cup^v_{t_1..t_2}\phi} \mathbf{b}}\; t^a_1 = M, t^a_2 = N, M \leq j \leq N$$

5.6 Ways of Running the Dynamo Execution Process

It is convenient to define the outcome of a Dynamo computation process as a set of reduced states. Let a valuation a be given. Then the state based on a is the state s_a given by:

$$(a, dom(a), \emptyset, \emptyset, \emptyset, b).$$

The result of running ϕ from state s_a is given by:

$$R_{s_a}(\phi) := \{\mathbf{b} \mid \exists \mathbf{a} \text{ such that } s_a \xrightarrow{\phi} \mathbf{a} \xrightarrow{r} \mathbf{b}\}.$$

In other words, the result will not contain states with grounded literals on their constraint lists, for such grounded literals represent tests that can be applied, and they will be applied during the $\xrightarrow{r}$ steps.

In addition, it may be useful to check the constraint lists for consistency, by means of applying a rule like the following:

$$\boxed{\quad \frac{l \in L \wedge \bar{l} \in L}{(a, g^a, l^a, n^a, L^a, m^a) \xrightarrow{r} \mathbf{0}} \quad}$$

As a special case, we have execution from the state of minimal information $s_\emptyset = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, b)$.

Also special is the case where Dynamo execution starts out from a test state for ϕ, i.e., from an initial valuation a with $var(\phi) \subseteq dom(a)$, where $var(\phi)$ is the list of variables that occur dynamically free in ϕ. One should be able to prove that if execution starts out from a test state for ϕ the states that result from the execution will not be constrained.

A final possibility that should be noted here is execution of ϕ from an initial state that imposes a list of constraints L, say with valuation a:

$$(a, dom(a), \emptyset, var(L^s), L^s, b).$$

This can be useful for putting initial constraints on computed solution sets.

5.7 Faithfulness to DFOL($\cup$)

The new notion of state of a computation requires us to redefine also the notion of faithfulness to DFOL($\cup$): now, a state will represent possible valuations in the following way:

$$F_{\mathbf{a}} := \begin{cases} \{b \in D^V \mid a \subseteq b, \forall l \in L^a(\mathcal{M} \models_b l)\} & \text{if } \mathbf{a} \notin \{\bullet, \mathbf{0}\}, \\ \{\bullet\} & \text{if } \mathbf{a} = \bullet, \\ \emptyset & \text{if } \mathbf{a} = \mathbf{0}. \end{cases}$$

Notice that there is no guarantee that $F_a \neq \emptyset$ even if $\mathbf{a} \neq \mathbf{0}$; there might be no valuation b that simultaneously satisfies all the ungrounded literals.

5.7.1. Theorem (Conditional forward property). *Suppose $\boldsymbol{a} \xrightarrow{\phi} \boldsymbol{b}$, with $l^a \cup g^b \subseteq dom(a)$, $n^a \subseteq n^b$, L^b extends L^a. Then $\boldsymbol{a} \preceq \boldsymbol{a}'$ implies there is a $\boldsymbol{b}' \succeq \boldsymbol{b}$ with $\boldsymbol{a}' \xrightarrow{\phi} \boldsymbol{b}'$.*

5.7.2. Theorem (Faithfulness to DFOL(∪)). *The execution mechanism is faithful to DFOL(∪), in the following sense: for all $\mathcal{M}, \phi$: if $\boldsymbol{a} \xrightarrow{\phi} \boldsymbol{b}$, $\boldsymbol{b} \neq \bullet$, $l^a \cup g^b \subseteq dom(a)$, then either:*

- *$\boldsymbol{b} \neq \boldsymbol{0}$ and for all $a' \in F_a$ there is a $b' \in F_b$ with ${}_{a'}[\phi]^{\mathcal{M}}_{b'}$,*
- *$\boldsymbol{b} = \boldsymbol{0}$ and for all $a' \in F_a$, $[\phi]^{\mathcal{M}}_{a'} = \emptyset$*

Proof. By induction on the structure of ϕ. We will show the proof for the case of negation for illustration.

Assume that there are $\mathbf{a}$, $\mathbf{b}$ such that $\mathbf{a} \xrightarrow{\neg\phi} \mathbf{b}$, $\mathbf{b} \neq \bullet$, $l^a \cup g^b \subseteq dom(a)$. Assume further that $\mathbf{b} \neq \mathbf{0}$. Then, we must prove that for all $a' \in F_a$ there is a $b' \in F_b$ with ${}_{a'}[\phi]^{\mathcal{M}}_{b'}$.

If $\mathbf{b} \neq \mathbf{0}$, there are two main possibilities:

- $\mathbf{b} = \mathbf{a}$. This happens when $O_a(\phi) = \mathbf{0}$. In turn, by inductive hypothesis, we know that if $O_a(\phi) = \mathbf{0}$, then $O_{a'}(\phi) = \mathbf{0}$ for any $a' \in F_a$.
- $\mathbf{b} \in {}_a(O_a(\phi) - \{\mathbf{0}\})$, with all members of $O_a(\phi) - \{\mathbf{0}\}$ constrained. We know that for all members of ${}_a(\mathbf{b}_1, \ldots, \mathbf{b}_n)$, the variables from the newly added constraints are in the global variable list g^b. Since by hypothesis $l^a \cup g^b \subseteq a$, extending a will not produce any simple $\mathbf{b} \in O_{a'}(\phi)$; all the atoms in the literal lists of $O_{a'}(\phi)$ will be already grounded. Otherwise the elements of $O_{a'}(\phi)$ will not be $\mathbf{0}$ or $\bullet$ because of the inductive hypothesis. Then, the lists of literals in the elements of $D_{a'}(O_{a'}(\phi) - \{\mathbf{0}\})$ will be the same as for a, which means that the new dualized states will simply be extensions of the previous ones.

If $\mathbf{b} = \mathbf{0}$, then this means that there is a simple element of $O_a(\phi) - \{\mathbf{0}\}$; by inductive hypothesis, the corresponding element of $O_{a'}(\phi) - \{\mathbf{0}\}$ will also be simple, therefore $[\phi]^{\mathcal{M}}_{a'} = \emptyset$ ⊣

5.8 Conclusion

We have presented an interpretation of DFOL formulas as programs, and an execution mechanism for DFOL(∪). This allows us to write imperative programs which have a declarative semantics, which in turn makes it very simple to verify that programs perform the tasks for which they are written; however, the treatment of negation is a bit involved, and universal quantification usually results in the • state. Seeking a way to solve this problem, we decided that since we were trying to approximate the semantics of the logic, we might as well do it with semantic tableaux [Smu68]. In the next chapter we will present a calculus designed for the verification of DFOL programs; later we will present the tableau engine which went into the latest version of *Dynamo*.

Chapter 6

Hoare Calculus for DFOL

6.1 Hoare Calculus

In Chapter 1 we introduced the concept of using logic for program verification: we will expand on the subject now. If we want to be able to use logic to verify the correctness of a program, we will need a *language* in which properties of the program can be expressed, with a set of rules that allow us to construct *well-formed formulas*. This is called an *assertion language* and its wffs are *assertions*. Of course, we also need a *proof system*: the axioms and rules that let us prove our assertions. This proof system should have the property, naturally, that it only allows us to prove *true* assertions; ideally it should allow us to prove *any* true assertion.

The Hoare calculus deals with a logic (the Hoare logic) in which one can formulate propositions about the correctness of programs. If we call the assignment of values to variables a *state*, and A and B are assertions about a state, a program ϕ satisfies the specification (A, B), if for any state g satisfying A the state reached by executing ϕ satisfies B. However, the possibility that a program does not terminate at all must be taken into account, so we distinguish between *partial* correctness:

$$\{A\}\phi\{B\} \iff \forall g(\mathcal{M} \models_g A \implies \forall h({}_g\llbracket\phi\rrbracket_h^{\mathcal{M}} \implies \mathcal{M} \models_h B))$$

and *total* correctness:

$$[A]\phi[B] \iff \forall g(\mathcal{M} \models_g A \implies \forall h(({}_g\llbracket\phi\rrbracket_h^{\mathcal{M}} \implies \mathcal{M} \models_h B) \wedge \llbracket\phi\rrbracket_g^{\mathcal{M}} \neq \emptyset))$$

Thus given a specification (A,B), we may consider that the job of the programmer is to find a program ϕ such that $\{A\}\phi\{B\}$, or even $[A]\phi[B]$ is true. The Hoare calculus provides us with the means to derive true assertions about atomic statements, and to combine them into true assertions about programs.

6.2 Why the Executable Interpretation of DFOL is particularly adequate for programming

The idea behind the executable interpretation of DFOL is precisely to have a programming language whose semantics are devoid of side effects or control features; the Hoare calculus for such a language would be clean and simple. We presented the interpretation in the previous chapter; now it becomes clear why it is important to have a declarative semantics for the language. If we were to have the calculus deal with the program state as defined in Chapter 5, it would be too cumbersome and impractical. Instead, we can trust the that interpretation is faithful to DFOL and work with the much cleaner semantics of the logic itself.

We will expand on the language presented in Chapter 5 to include *explicit bindings* (σ); we'll also suggest rules for dealing with the *hiding operator* ($\exists_x(\phi)$) and with the Kleene star operator at the end of the chapter.

As we stated on 1.4.12, given a signature of function and predicate symbols, the syntax of DFOL($\cup, \sigma, \exists, *$) is as follows:

$$\begin{array}{lcll} t & ::= & v \mid f\bar{t} & \text{(Terms)} \\ \phi & ::= & \sigma \mid \exists v \mid P\bar{t} \mid t_1 \doteq t_2 \mid \exists_x(\phi) \mid \neg(\phi) \mid \phi_1;\phi_2 \mid (\phi_1 \cup \phi_2) \mid \phi^* & \text{(Formulas)} \end{array}$$

6.3 The Rules

As we said in Chapter 1, the use of negation as failure forces us to adopt a slightly different set of correctness criteria. We have then two kinds of correctness rules: existential and universal. Their meaning is the following:

$$\begin{array}{rcl} \mathcal{M} \models (A)\phi(B) & \iff & \forall g\, (\mathcal{M} \models_g A \implies \exists h\, ({}_g[\![\phi]\!]_h^{\mathcal{M}} \wedge \mathcal{M} \models_h B)) \\ \mathcal{M} \models \{A\}\phi\{B\} & \iff & \forall g\, (\mathcal{M} \models_g A \implies \forall h\, ({}_g[\![\phi]\!]_h^{\mathcal{M}} \implies \mathcal{M} \models_h B)) \end{array}$$

Note that universal correctness is equivalent to the old partial correctness, but existential correctness does not guarantee that all terminating executions of ϕ satisfy the postcondition. We can see, however, that if ϕ satisfies $\{A\}\phi\{B\}$ *and* $(A)\phi(\top)$, then total correctness is achieved. Note also that universal and existential correctness rules are *interdependent* for the case of negation. Now, we enunciate the rules of the calculus. This is an adaptation and expansion of a calculus presented by van Eijck and de Vries [vEdV92] for a different extension of DFOL.

There are also rules for defined *Dynamo* constructs, such as bounded iteration and bounded choice. As the constructs are defined in terms of operators for which there is a rule already, these rules are derived from the basic rules too, and are

Figure 6.1: Universal correctness rules:

Existential quantification:	$\{\forall x A\}\exists x\{A\}$
Substitution:	$\{A\sigma\}\sigma\{A\}$
Equality:	$\{(t_1 = t_2) \to A\}t_1 \doteq t_2\{A\}$
Predicates:	$\{P\bar{t} \to A\}P\bar{t}\{A\}$
Negation:	$\dfrac{(A)\phi(\top)}{\{A \vee B\}\neg(\phi)\{B\}}$
Sequential composition:	$\dfrac{\{A\}\phi_1\{B\} \qquad \{B\}\phi_2\{C\}}{\{A\}\ (\phi_1;\phi_2)\ \{C\}}$
Union:	$\dfrac{\{A\}\phi_1\{C\} \qquad \{B\}\phi_2\{C\}}{\{A \wedge B\}\ (\phi_1 \cup \phi_2)\ \{C\}}$
Rule of consequence:	$\dfrac{\{A\}\phi\{B\}}{\{A'\}\phi\{B'\}}$ if $\mathcal{M} \models (A' \to A)$ and $\mathcal{M} \models (B \to B')$
Filter Rule:	$\dfrac{\{A\}\phi\{B\} \qquad \{C\}\phi\{\bot\}}{\{A \vee C\}\phi\{B\}}$

called *admissible* rules. All that is needed for them is to prove that they follow from the basic rules.

6.3.1. LEMMA (*Dynamo* CONSTRUCTS). *The following rules are admissible:*

	Bounded iteration	*Bounded search*
Universal correctness:	$\dfrac{\{A\}\phi\{A\}}{\{A\}\phi^n\{A\}}$	$\dfrac{\{A\}\phi\{B\}}{\{\forall v \in \{N,\ldots,M\} : A\} \bigcup^{v}_{N..M} \phi\{B\}}$
Existential correctness:	$\dfrac{(A)\phi(A)}{(A)\phi^n(A)}$	$\dfrac{(A)\phi(B)}{(\exists v \in \{N,\ldots,M\} : A) \bigcup^{v}_{N..M} \phi(B)}$

Proof. We first consider universal correctness for the constructs. We show that bounded iteration is admissible by induction on the number of iterations.
For $n = 0$, we have ${}_g[\![\phi^0]\!]^{\mathcal{M}}_h$ iff $g = h$. So, $\{A\}\phi^n\{A\}$ is trivially true.
If $\{A\}\phi^n\{A\} \implies \{A\}\phi^{n+1}\{A\}$, then $\forall n \in \mathbb{N}, \{A\}\phi^n\{A\}$

Figure 6.2: Existential correctness rules:

Existential quantification:	$(\exists x A)\exists x(A)$
Substitution:	$(A\sigma)\sigma(A)$
Equality:	$(A \wedge (t_1 = t_2))t_1 \doteq t_2(A)$
Predicates:	$(A \wedge P\bar{t})P\bar{t}(A)$
Negation:	$\dfrac{\{A\}\phi\{\bot\}}{(A)\neg(\phi)(A)}$
Sequential composition:	$\dfrac{(A)\phi_1(B) \qquad (B)\phi_2(C)}{(A)(\phi_1;\phi_2)(C)}$
Union:	$\dfrac{(A)\phi_1(C) \qquad (B)\phi_2(C)}{(A \vee B)(\phi_1 \cup \phi_2)(C)}$
Rule of consequence:	$\dfrac{(A)\phi(B)}{(A')\phi(B')}$ if $\mathcal{M} \models (A' \to A)$ and $\mathcal{M} \models (B \to B')$
Combination Rule:	$\dfrac{\{A\}\phi\{B\} \qquad (C)\phi(\top)}{(A \wedge C)\phi(B)}$

Now we can apply the sequential composition rule:

$$\frac{\{A\}\overbrace{\phi;\ldots;\phi}^{\text{n}}\{A\} \quad \{A\}\phi\{A\}}{\{A\}\underbrace{\phi;\ldots;\phi}_{\text{n+1}}\{A\}}\ \text{Seq.comp}$$

where the left premise comes from the inductive hypothesis.
For the case of bounded search, we know that

$$\frac{\{A\sigma\}\sigma\{A\} \quad \{A\}\phi\{B\}}{\{A\sigma\}(\sigma;\phi)\{B\}}\ \text{Seq. comp}$$

So, the correctness condition for each disjunct can be expressed as $\{A[i/v]\}([i/v];\phi)\{B\}$, and then the whole expression can be derived by repeated application of the Union rule:

$$\frac{\{A[N/v]\}([N/v];\phi)\{B\}\ldots\{A[M/v]\}([M/v];\phi)\{B\}}{\{\bigwedge_{i=N}^{M} A[i/v]\}\bigcup_{N\cdots M}^{v}(\phi)\{B\}}\ \text{Union}$$

Finally, $\bigwedge_{i=N}^{M} A[i/v]$ is true under g iff for all values of i between N and M, $A[i/v]$

is true under g. That is, we can apply the Consequence rule to prove

$$\frac{\forall v \in \{N,\dots,M\} : A \implies \bigwedge_{i=N}^{M} A[i/v] \qquad \{\bigwedge_{i=N}^{M} A[i/v]\} \bigcup_{N\cdots M}^{v}(\phi)\{B\}}{\{\forall v \in \{N,\dots,M\} : A\} \bigcup_{N\cdots M}^{v}(\phi)\{B\}}\ \text{Cons}$$

Now we show how the existential version of these rules is admissible. We start with bounded iteration, again arguing by induction on the number of iterations:
For $n = 0$, we have ${}_g[\![\phi^0]\!]_h^{\mathcal{M}}$ iff $g = h$. So, $(A)\phi^n(A)$ is trivially true.
If $(A)\phi^n(A) \implies (A)\phi^{n+1}(A)$, then $\forall n \in \mathbb{N}, (A)\phi^n(A)$
Now we can apply the sequential composition rule:

$$\frac{(A)\overbrace{\phi;\dots;\phi}^{\text{n}}(A) \quad (A)\phi(A)}{(A)\underbrace{\phi;\dots;\phi}_{\text{n+1}}(A)}\ \text{Seq.comp}$$

where the left premise comes from the inductive hypothesis.
For the case of bounded search, we know that

$$\frac{(A\sigma)\sigma(A) \quad (A)\phi(B)}{(A\sigma)(\sigma;\phi)(B)}\ \text{Seq. comp}$$

So, the correctness condition for each disjunct can be expressed as $(A[i/v])([i/v];\phi)(B)$, and then the whole can be derived by application of the Union rule:

$$\frac{(A[N/v])([N/v];\phi)(B)\dots(A[M/v])([M/v];\phi)(B)}{(\bigvee_{i=N}^{M} A[i/v]) \bigcup_{N\cdots M}^{v}(\phi)(B)}\ \text{Union}$$

Finally, $\bigvee_{i=N}^{M} A[i/v]$ is true under g iff there is a value of i between N and M such that $A[i/v]$ is true under g. That is,

$$\frac{\exists v \in \{N,\dots,M\} : A \implies \bigvee_{i=N}^{M} A[i/v] \qquad (\bigvee_{i=N}^{M} A[i/v]) \bigcup_{N\cdots M}^{v}(\phi)(B)}{(\exists v \in \{N,\dots,M\} : A) \bigcup_{N\cdots M}^{v}(\phi)(B)}\ \text{Cons}$$

$\dashv$

There are two properties that make a proof calculus useful. The most basic is *soundness*: it should never produce false statements. The other, complementary property, is *completeness*: one should be able to obtain every true statement that is expressible in the calculus. Clearly, while not achieving absolute completeness is bad, not achieving soundness is catastrophic. We will now test our calculus for these two properties.

6.4 Soundness

Soundness (Existential rules):

$$\mathcal{M} \vdash (A)\phi(B) \implies \forall g(\mathcal{M} \models_g A \implies \exists h(\ {}_g[\![\phi]\!]_h^{\mathcal{M}} \wedge \mathcal{M} \models_h B)),$$

that is, if we can derive $(A)\phi(B)$ in a model $\mathcal{M}$, then for all states g satisfying A, there must exist a state h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$ and h satisfies B.

Soundness (Universal rules):

$$\mathcal{M} \vdash \{A\}\phi\{B\} \implies \forall g(\mathcal{M} \models_g A \implies \forall h(\ {}_g[\![\phi]\!]_h^{\mathcal{M}} \implies \mathcal{M} \models_h B)),$$

that is, if we can derive $\{A\}\phi\{B\}$ in a model $\mathcal{M}$, then for all states g satisfying A, all states h satisfying ${}_g[\![\phi]\!]_h^{\mathcal{M}}$ must also satisfy B.

6.4.1. THEOREM. *Soundness of the calculus: The calculus presented in Figures 6.1 and 6.2 is sound.*

Proof. We argue by induction on the structure of ϕ. That is, if all the axioms are logically valid, and for every model $\mathcal{M}$ the application of an inference rule on formulas valid in $\mathcal{M}$ results in a formula valid in $\mathcal{M}$, then the derivations obtained with the calculus will always be valid. We start our proof with the axioms and rules for universal correctness.

- $\exists x$: Suppose we have derived $\{\forall x A\}\exists x\{A\}$. Then we must prove that if there is an assignment g under which $\forall x(A)$ is true, and furthermore there is a successful execution of $\exists x$ terminating in assignment h, then A must be true in h. We know that (1) ${}_g[\![\exists x]\!]_h^{\mathcal{M}}$ iff g and h differ at most in the value of x, and (2) $\mathcal{M} \models_g \forall x A$ iff no matter what the value of $g(x), \mathcal{M} \models_g A$. Therefore, for all $d \in \mathcal{D}$, if $h = g_x^d$, then $\mathcal{M} \models_h A$.
  From (1) and (2), we know that $\forall h(\ {}_g[\![\exists x]\!]_h^{\mathcal{M}}, \mathcal{M} \models_h A)$.

- σ: Suppose we have derived $\{A\sigma\}\sigma\{A\}$. Then we must prove that if there is an assignment g under which $A\sigma$ is true, and furthermore there is a successful execution of σ terminating in assignment h, then A must be true in h. We know that (1) ${}_g[\![\sigma]\!]_h^{\mathcal{M}}$ iff $h = g\sigma$, with $\sigma = [t_1/x_1, \ldots, t_n/x_n]$, and (2) $\mathcal{M} \models_g A\sigma$ iff $\mathcal{M} \models_g A[t_1/x_1, \ldots, t_n/x_n]$, so $\mathcal{M} \models_h A$, for $h = g[t_1/x_1, \ldots, t_n/x_n]$.

- $P\bar{t}$: Suppose we have derived $\{P\bar{t} \implies A\}P\bar{t}\{A\}$. Then we must prove that if there is an assignment g under which $P\bar{t} \implies A$ is true, and furthermore there is a successful execution of $P\bar{t}$ terminating in assignment h, then A must be true in h. We know that ${}_g[\![P\bar{t}]\!]_h^{\mathcal{M}}$ iff $h = g$ and $P\bar{t}$ is true in g. As $h = g, (\mathcal{M} \models_h P\bar{t} \wedge P\bar{t} \rightarrow A) \implies \mathcal{M} \models_h A$.

- $t_1 \doteq t_2$: Analogous to $P\bar{t}$.

- $\neg(\phi)$: Suppose we have derived $\{A \vee B\}\neg(\phi)\{B\}$, and that the premises themselves are sound. Then we must prove that if there is an assignment g under which $A \vee B$ is true, and furthermore there is a successful execution of $\neg(\phi)$ terminating in assignment h, then B must be true in h. We know that (1) ${}_g[\![\neg(\phi)]\!]_h^{\mathcal{M}}$ iff $g = h$ and there is no i such that ${}_g[\![\phi]\!]_i^{\mathcal{M}}$.
(2) $\mathcal{M} \models_g A \vee B$ iff $\mathcal{M} \models_g A$ or $\mathcal{M} \models_g B$.
(3) $(A)\phi(\top)$ iff for all g that satisfy A there is an h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$.
By (3), for those g such that $\mathcal{M} \models_g A$, there is no h such that ${}_g[\![\neg(\phi)]\!]_h^{\mathcal{M}}$.
For those g such that $\mathcal{M} \models_g B$, all h such that ${}_g[\![\neg(\phi)]\!]_h^{\mathcal{M}}$ will be equal to g, so $\mathcal{M} \models_h B$.

- $\phi_1;\phi_2$: Suppose we have derived $\{A\}\phi_1;\phi_2\{C\}$. Then we must prove that if there is an assignment g under which A is true, and furthermore there is a successful execution of $\phi_1;\phi_2$ terminating in assignment h, then C must be true in h.

  We know that (1) $\{A\}\phi_1\{B\}$ iff for all g under which A is true, all h such that ${}_g[\![\phi_1]\!]_h^{\mathcal{M}}$ will make B true.
(2) $\{B\}\phi_2\{C\}$ iff for all g under which B is true, all h such that ${}_g[\![\phi_2]\!]_h^{\mathcal{M}}$ will make C true.
(3) ${}_g[\![\phi_1;\phi_2]\!]_h^{\mathcal{M}}$ iff there is an i such that ${}_g[\![\phi_1]\!]_i^{\mathcal{M}}$ and ${}_i[\![\phi_2]\!]_h^{\mathcal{M}}$.
By (1) and (3), there is an i such that ${}_g[\![\phi_1]\!]_i^{\mathcal{M}}$, ${}_i[\![\phi_2]\!]_h^{\mathcal{M}}$, and $\mathcal{M} \models_i B$.
By (2) and (3), $\mathcal{M} \models_h C$.

- $\phi_1 \cup \phi_2$: Suppose we have derived $\{A\}\phi_1 \cup \phi_2\{C\}$. Then we must prove that if there is an assignment g under which $A \wedge B$ is true, and furthermore there is a successful execution of $\phi_1 \cup \phi_2$ terminating in assignment h, then C must be true in h. We know that (1) $\{A\}\phi_1\{C\}$ iff for all g under which A is true, all h such that ${}_g[\![\phi_1]\!]_h^{\mathcal{M}}$ will make C true.
(2) $\{B\}\phi_2\{C\}$ iff for all g under which B is true, all h such that ${}_g[\![\phi_2]\!]_h^{\mathcal{M}}$ will make C true.
(3) ${}_g[\![\phi_1 \cup \phi_2]\!]_h^{\mathcal{M}}$ iff ${}_g[\![\phi_1]\!]_h^{\mathcal{M}}$ or ${}_g[\![\phi_2]\!]_h^{\mathcal{M}}$.
(4) $\mathcal{M} \models_g A \wedge B$ iff $\mathcal{M} \models_g A$ and $\mathcal{M} \models_g B$.
So, if ${}_g[\![\phi_1]\!]_h^{\mathcal{M}}$, by (1) and (4), $\mathcal{M} \models_g C$.
So, if ${}_g[\![\phi_2]\!]_h^{\mathcal{M}}$, by (2) and (4), $\mathcal{M} \models_g C$.

- Filter Rule: Suppose we know that (1)$\{A\}\phi\{B\}$ and (2)$\{C\}\phi\{\perp\}$. Then we must prove that if there is an assignment g under which $(A \vee C)$ is true, and furthermore there is a successful execution of ϕ terminating in assignment h, then B must be true in h. We know that (3) $(A \vee C)$ is true in g iff A is true in g or C is true in g
Suppose there is a g such that $(A \vee C)$ is true in g, and assume there is

an h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$. By (2) and (3), A is true in g. Then, by (1), and assumption, B is true in h.

Existential rules: We proceed to show the existential correctness rules to be sound, again by induction on the structure of ϕ.

- $\exists x$: Suppose we have derived $(\exists x : A)\exists x(A)$. Then we must prove that if there is an assignment g under which $\exists x : A$ is true, there is a successful execution of $\exists x$ terminating in assignment h under which A is true. Let's assume such a g exists. We know that ${}_g[\![\exists x]\!]_h^{\mathcal{M}}$ iff g and h differ at most in the value of x. We also know that $\mathcal{M} \models_g (\exists x A)$ iff there is a $d_0 \in \mathcal{D}$ such that A is true in $g_x^{d_0}$.
We can assume $h = g_x^{d_0}$, and then we know that ${}_g[\![\exists x]\!]_h^{\mathcal{M}}$, and A is true in h.

- σ: Suppose we have derived $(A\sigma)\sigma(A)$. Then we must prove that if there is an assignment g under which $A\sigma$ is true, then there is a successful execution of σ terminating in assignment h under which A is true. Let's assume then that such a g exists. We know that ${}_g[\![\sigma]\!]_h^{\mathcal{M}}$ iff $h = g\sigma$. That is, $h = g[t_1/v_1, \ldots, t_n/v_n]$ always exists.
Now, $\mathcal{M} \models_g (A\sigma)$ iff $\mathcal{M} \models_{g[t_1/v_1,\ldots,t_n/v_n]} (A)$; that is, A is true in h, for $h = g[t_1/v_1, \ldots, t_n/v_n]$.

- $P\bar{t}$: Suppose we have derived $(A \wedge P\bar{t})P\bar{t}(A)$. Then we must prove that if there is an assignment g under which $(A \wedge P\bar{t})$ is true, then there is a successful execution of $P\bar{t}$ terminating in assignment h under which A is true. Let's assume then that such a g exists. Now, $(A \wedge P\bar{t})$ is true under g iff $\mathcal{M} \models_g A \wedge \mathcal{M} \models_g P\bar{t}$, and ${}_g[\![P\bar{t}]\!]_h^{\mathcal{M}}$ iff $h = g$ and $P\bar{t}$ is true in g. So such an h always exists, and $\mathcal{M} \models_h A$.

- Equality tests: Same as for $P\bar{t}$.

- $\neg(\phi)$: Suppose we have derived $(A)\neg(\phi)(A)$, knowing that $\{A\}\phi\{\bot\}$. Then we must prove that if there is an assignment g under which A is true, then there is a successful execution of $\neg(\phi)$ terminating in assignment h under which A is true. Let's assume then that such a g exists. We have then that
(1) if g makes A true then there is no h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$.
(2) ${}_g[\![\neg(\phi)]\!]_h^{\mathcal{M}}$ iff $g = h$ and there is no i such that ${}_g[\![\phi]\!]_i^{\mathcal{M}}$.
By (1) and (2), we have that $\exists h = g :\ {}_g[\![\neg(\phi)]\!]_h^{\mathcal{M}}$, and as $h = g$, $\mathcal{M} \models_h A$.

- $(\phi_1 \cup \phi_2)$: Suppose we have derived $(A \vee B)(\phi_1 \cup \phi_2)(C)$, knowing that $(A)\phi_1(C)$ and $(B)\phi_2(C)$. Then we must prove that if there is an assignment g under which $A \vee B$ is true, then there is a successful execution of $(\phi_1 \cup \phi_2)$ terminating in assignment h under which C is true. Let's assume then that such a g exists. We have: (1) If g satisfies A, then there is an h such that

${}_g[\![\phi_1]\!]_h$, and h satisfies C.
(2) If g satisfies B, then there is an h such that ${}_g[\![\phi_2]\!]_h$, and h satisfies C.
We know that (3) ${}_g[\![(\phi_1 \cup \phi_2)]\!]_h^{\mathcal{M}}$ iff ${}_g[\![\phi_1]\!]_h^{\mathcal{M}}$ or ${}_g[\![\phi_2]\!]_h^{\mathcal{M}}$.
Now, (4) $\mathcal{M} \models_g (A \vee B)$ iff A is true in g or B is true in g.
Let us assume first that A is true in g. Then, by (1), there is an h_1 such that ${}_g[\![\phi_1]\!]_{h_1}^{\mathcal{M}}$ and C is true.
Now, if B is true in g, by (2), there is an h_2 such that ${}_g[\![\phi_2]\!]_{h_2}^{\mathcal{M}}$ and C is true.
Then, by (3) and (4), there is an h such that ${}_g[\![(\phi_1 \cup \phi_2)]\!]_h^{\mathcal{M}}$ and C is true.

- $\phi_1; \phi_2$: Suppose we have derived $(A)(\phi_1 \cup \phi_2)(C)$, knowing that $(A)\phi_1(B)$ and $(B)\phi_2(C)$. Then we must prove that if there is an assignment g under which A is true, then there is a successful execution of $\phi_1; \phi_2$ terminating in assignment h under which C is true. Let's assume then that such a g exists. We have: (1) If g satisfies A, then there is an h such that ${}_g[\![\phi_1]\!]_h$, and h satisfies B.
  (2) If g satisfies B, then there is an h such that ${}_g[\![\phi_2]\!]_h$, and h satisfies C.
  (3) ${}_g[\![\phi_1; \phi_2]\!]_h^{\mathcal{M}}$ iff there is an i such that ${}_g[\![\phi_1]\!]_i^{\mathcal{M}}$ and ${}_i[\![\phi_2]\!]_h^{\mathcal{M}}$.
  By (1), there is an i such that ${}_g[\![\phi_1]\!]_i^{\mathcal{M}}$ that satisfies B.
  Then, by (2), there is an h such that ${}_i[\![\phi_2]\!]_h^{\mathcal{M}}$ that satisfies C.
  Now, by (3), ${}_g[\![\phi_1; \phi_2]\!]_h^{\mathcal{M}}$.

- Combination Rule: Suppose we have derived $(A \wedge C)\phi(B)$, knowing that $(A)\phi(B)$ and $\{C\}\phi\{\top\}$. Then we must prove that if there is an assignment g under which $A \wedge C$ is true, then there is a successful execution of ϕ terminating in assignment h under which B is true. Let's assume then that such a g exists. We know that (1) for all g that make A true, all h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$ will make B true.
  (2) for all g that make C true, there is an h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$.
  (3) $\mathcal{M} \models_g A \wedge C$ iff $\mathcal{M} \models_g A$ and $\mathcal{M} \models_g C$.
  By (2) and (3), there is an h such that ${}_g[\![\phi]\!]_h^{\mathcal{M}}$.
  By (1) and (3), B is true in h.

$\dashv$

6.5 Completeness

We shall prove completeness of the calculus for formulas in DFOL$(\sigma, \cup)$

6.5.1. THEOREM. *(Completeness of the calculus): For all models $\mathcal{M}$, and all programs $\phi \in$ DFOL$(\sigma, \cup)$,*
if $\mathcal{M} \models \{A\}\phi\{B\}$ and $\mathcal{M} \models (C)\phi(D)$, then $\{A\}\phi\{B\}$ and $(C)\phi(D)$ are derivable in $H + T$, where $T = Th(\mathcal{M})$ (the theorems of $\mathcal{M}$).

Proof. By simultaneous induction on the structure of ϕ. We define the predicates wup(ϕ, B) (weakest universal precondition) and wep(ϕ, B)(weakest existential precondition) as

$$\begin{aligned}
\mathcal{M} \models_g \mathrm{wup}(\phi, B) &\iff \forall h(\ {}_g[\![\phi]\!]^{\mathcal{M}}_h \implies \mathcal{M} \models_h B), \\
\mathcal{M} \models_g \mathrm{wep}(\phi, B) &\iff \exists h(\ {}_g[\![\phi]\!]^{\mathcal{M}}_h \wedge \mathcal{M} \models_h B),
\end{aligned}$$

and we prove that the calculus gives the wlp(ϕ, B) for the universal rules and the wep(ϕ, B) for the existential rules. As these predicates are the *weakest* precondition, they must be implied by any precondition for the triples to hold; we can then use the consequence rule to derive any valid Hoare triple we may encounter.

- $\exists x$:

$$\begin{aligned}
\mathcal{M} \models_g \mathrm{wup}(\exists x, B) &\iff \forall h({}_g[\![\exists x]\!]^{\mathcal{M}}_h \implies \mathcal{M} \models_h B) \\
&\iff \forall h(g \sim_x h \implies \mathcal{M} \models_h B) \\
&\iff \forall d \in \mathcal{D} : (\mathcal{M} \models_{g^d_x} B) \\
&\iff \mathcal{M} \models_g \forall x B
\end{aligned}$$

Since the wup$(\exists x, B)$ is equivalent to $\forall x B$, we know that for all g, A implies wup$(\exists x, B)$. Then, by the consequence rule,

$$\frac{A \rightarrow \forall x B \qquad \{\forall x B\}\exists x\{B\}}{\{A\}\exists x\{B\}}\ \mathrm{Cons}$$

We can limit ourselves then to prove that the antecedent given by the rules is always the weakest precondition for each rule.

$$\begin{aligned}
\mathcal{M} \models_g \mathrm{wep}(\exists x, D) &\iff \exists h({}_g[\![\exists x]\!]^{\mathcal{M}}_h \wedge \mathcal{M} \models_h D) \\
&\iff \exists h(g \sim_x h \wedge \mathcal{M} \models_h D) \\
&\iff \exists d \in \mathcal{D} : (\mathcal{M} \models_{g^d_x} D) \\
&\iff \mathcal{M} \models_g \exists x(D).
\end{aligned}$$

- $x \doteq t$:

$$\begin{aligned}
\mathcal{M} \models_g \mathrm{wup}(x \doteq t, B) &\iff \forall h({}_g[\![x \doteq t]\!]^{\mathcal{M}}_h \implies \mathcal{M} \models_h B) \\
&\iff \forall h((g = h \wedge \mathcal{M} \models_g x = t) \implies \mathcal{M} \models_h B) \\
&\iff \mathcal{M} \models_g (x = t) \rightarrow B
\end{aligned}$$

$$\begin{aligned}\mathcal{M} \models_g \mathrm{wep}(x \doteq t, D) &\iff \exists h({}_g[x \doteq t]_h^{\mathcal{M}} \wedge \mathcal{M} \models_h D)\\ &\iff h = g \wedge \mathcal{M} \models_g x = t \wedge \mathcal{M} \models_h D\\ &\iff \mathcal{M} \models_g x = t \wedge D.\end{aligned}$$

- σ:

$$\begin{aligned}\mathcal{M} \models_g \mathrm{wup}(\sigma, B) &\iff \forall h({}_g[\sigma]_h^{\mathcal{M}} \implies \mathcal{M} \models_h B)\\ &\iff \forall h(h = g\sigma \implies \mathcal{M} \models_h B)\\ &\iff \mathcal{M} \models_g B\sigma\end{aligned}$$

$$\begin{aligned}\mathcal{M} \models_g \mathrm{wep}(\sigma, D) &\iff \exists h({}_g[\sigma]_h^{\mathcal{M}} \wedge \mathcal{M} \models_h D)\\ &\iff \exists h(h = g\sigma \wedge \mathcal{M} \models_h D)\\ &\iff \mathcal{M} \models_g D\sigma.\end{aligned}$$

- $\neg\phi$:

  Assume $\mathcal{M} \models \{A\}\neg(\phi)\{B\}$. To prove: $\mathcal{M} \vdash \{A\}\neg(\phi)\{B\}$.

  By inductive hypothesis, we have that

$$\begin{aligned}\mathcal{M} \models_g (\mathrm{wep}(\phi, \top)) &\implies \mathcal{M} \vdash_g (\mathrm{wep}(\phi, \top))\\ \mathcal{M} \models_g (\mathrm{wup}(\phi, \bot)) &\implies \mathcal{M} \vdash_g (\mathrm{wup}(\phi, \bot))\end{aligned}$$

$$\begin{aligned}\mathcal{M} \models_g (\mathrm{wup}(\neg(\phi), B)) &\iff \forall h({}_g[\neg(\phi)]_h^{\mathcal{M}} \implies \mathcal{M} \models_h B)\\ &\iff \forall h((h = g \;\wedge \neg\exists i({}_g[\phi]_i^{\mathcal{M}})) \implies \mathcal{M} \models_h B)\\ &\iff \forall h((h = g \wedge \neg\mathcal{M} \models_g \mathrm{wep}(\phi, \top)) \implies \mathcal{M} \models_h B)\\ &\iff \forall h(h = g \implies \mathcal{M} \models_g (\mathrm{wep}(\phi, \top) \vee B))\\ &\iff \mathcal{M} \models_g (\mathrm{wep}(\phi, \top) \vee B)\end{aligned}$$

We know that $\mathcal{M} \models \{A\}\neg(\phi)\{B\}$, and that this means $\mathcal{M} \models_g (A \to \mathrm{wup}(\neg(\phi), B))$, and therefore $\mathcal{M} \models_g (A \to (\mathrm{wep}(\phi, \top) \vee B))$ Then, $\mathcal{M} \vdash (A \to (\mathrm{wep}(\phi, \top) \vee B))$.

Existential case: Assume $\mathcal{M} \models (C)\neg(\phi)(D)$. To prove: $\mathcal{M} \vdash (C)\neg(\phi)(D)$

$$\begin{aligned}\mathcal{M} \models_g (\mathrm{wep}(\neg(\phi), D)) &\iff \exists h({}_g[\neg(\phi)]_h^{\mathcal{M}} \wedge \mathcal{M} \models_h D).\\ &\iff \exists h(h = g \;\wedge \neg\exists i({}_g[\phi]_i^{\mathcal{M}} \wedge \mathcal{M} \models_h D))\\ &\iff \mathcal{M} \models_g \mathrm{wup}(\phi, \bot) \wedge \mathcal{M} \models_g D\\ &\iff \mathcal{M} \models_g (\mathrm{wup}(\phi, \bot) \wedge D)\end{aligned}$$

So, by definition of weakest existential precondition and inductive hypothesis, $\mathcal{M} \models_g (C)\neg(\phi)(D) \implies \mathcal{M} \vdash_g (\mathrm{wup}(\phi, \bot) \wedge D)$,
which means we can derive $\mathcal{M} \vdash (C)\neg(\phi)(D)$.

- $\phi_1; \phi_2$: By induction hypothesis, we have that

$$\begin{array}{rcl} \mathcal{M} \vdash \{\mathrm{wup}(\phi_2, B)\} & \phi_2 & \{B\}, \\ \mathcal{M} \vdash \{\mathrm{wup}(\phi_1, \mathrm{wup}(\phi_2, B))\} & \phi_1 & \{\mathrm{wup}(\phi_2, B)\} \end{array}$$

so that the sequential composition rule will result in

$$\{\mathrm{wup}(\phi_1, \mathrm{wup}(\phi_2, B))\} \quad \phi_1; \phi_2 \quad \{B\}$$

We should prove then that $\mathcal{M} \models_g \mathrm{wup}(\phi_1; \phi_2, B) \rightarrow \mathrm{wup}(\phi_1, \mathrm{wup}(\phi_2, B))$.
Assume $\mathcal{M} \models_g \mathrm{wup}(\phi_1; \phi_2, B)$. To prove: $\mathcal{M} \models_g \mathrm{wup}(\phi_1, \mathrm{wup}(\phi_2, B))$

  - If $\nexists h({}_g\llbracket\phi_1\rrbracket_h^{\mathcal{M}})$, the result is trivially true.
  - If $\exists h({}_g\llbracket\phi_1\rrbracket_h^{\mathcal{M}})$, then :
    - if $\nexists i({}_h\llbracket\phi_2\rrbracket_i^{\mathcal{M}}$, the result is again trivially true.
    - if $\exists i({}_h\llbracket\phi_2\rrbracket_i^{\mathcal{M}}$, then $\mathcal{M} \models_i B$, and then $\mathcal{M} \models_g \mathrm{wup}(\phi_1, \mathrm{wup}(\phi_2, B))$

The existential counterpart is analogous: by induction hypothesis,

$$\begin{array}{rcl} (\mathrm{wep}(\phi_2, B)) & \phi_2 & (B), \\ (\mathrm{wep}(\phi_1, \mathrm{wep}(\phi_2, B))) & \phi_1 & (\mathrm{wep}(\phi_2, B)) \end{array}$$

so that the sequential composition rule will result in

$$\{\mathrm{wep}(\phi_1, \mathrm{wep}(\phi_2, B))\} \quad \phi_1; \phi_2 \quad \{B\}.$$

We should prove then that $\mathrm{wep}(\phi_1; \phi_2, B) \implies \mathrm{wep}(\phi_1, \mathrm{wep}(\phi_2, B))$.
Assume $\mathcal{M} \models_g \mathrm{wep}(\phi_1; \phi_2, B)$. To prove: $\mathcal{M} \models_g \mathrm{wep}(\phi_1, \mathrm{wep}(\phi_2, B))$
As $\mathcal{M} \models_g \mathrm{wep}(\phi_1; \phi_2, B)$, we know that $\exists i({}_g\llbracket\phi_1; \phi_2\rrbracket_i^{\mathcal{M}} \wedge \mathcal{M} \models_i B)$.
Now, that means that $\exists h({}_g\llbracket\phi_1\rrbracket_h^{\mathcal{M}} \wedge {}_h\llbracket\phi_2\rrbracket_i^{\mathcal{M}})$, and therefore
$\mathcal{M} \models_g \mathrm{wep}(\phi_1, \mathrm{wep}(\phi_2, B))$

- $\phi_1 \cup \phi_2$: By induction hypothesis, we have that

$$\begin{array}{rcl} \mathcal{M} \vdash \{\mathrm{wup}(\phi_1, B)\} & \phi_1 & \{B\}, \\ \mathcal{M} \vdash \{\mathrm{wup}(\phi_2, B)\} & \phi_2 & \{B\} \end{array}$$

By the Union rule,

$$\mathcal{M} \vdash \{\mathrm{wup}(\phi_1, B) \wedge \mathrm{wup}(\phi_2, B)\} \quad (\phi_1 \cup \phi_2) \quad \{B\}$$

We should prove then that $\mathcal{M} \models_g \mathrm{wup}(\phi_1 \cup \phi_2, B) \rightarrow (\mathrm{wup}(\phi_1, B) \wedge \mathrm{wup}(\phi_2, B))$.
So, assume $\mathcal{M} \models_g \mathrm{wup}(\phi_1 \cup \phi_2, B)$. To prove: $\mathcal{M} \models_g \mathrm{wup}(\phi_1, B) \wedge \mathrm{wup}(\phi_2, B)$

$$\begin{aligned} \mathcal{M} \models_g \mathrm{wup}(\phi_1 \cup \phi_2, B) &\iff \forall h({}_g[\phi_1 \cup \phi_2]^{\mathcal{M}}_h \implies \mathcal{M} \models_h B) \\ &\iff \forall h(({}_g[\phi_1]^{\mathcal{M}}_h \vee_g [\phi_2]^{\mathcal{M}}_h) \implies \mathcal{M} \models_h B) \end{aligned}$$

Assume $\exists h({}_g[\phi_1 \cup \phi_2]^{\mathcal{M}}_h)$. Then, we know that $\mathcal{M} \models_h B$.
Now, assume $\neg\mathcal{M} \models_g \mathrm{wup}(\phi_1, B)$. If ${}_g[\phi_1]^{\mathcal{M}}_h$, then there is an h such that ${}_g[\phi_1 \cup \phi_2]^{\mathcal{M}}_h \wedge \mathcal{M} \models_h \neg B$, which contradicts $\mathcal{M} \models_g \mathrm{wup}(\phi_1 \cup \phi_2, B)$. Similarly for $\mathrm{wup}(\phi_2, B)$, so $\mathcal{M} \models_g \mathrm{wup}(\phi_1 \cup \phi_2, B) \implies \mathcal{M} \models_g (\mathrm{wup}(\phi_1, B) \wedge \mathrm{wup}(\phi_2, B))$, and by inductive hypothesis $\mathcal{M} \vdash_g (\mathrm{wup}(\phi_1, B) \wedge \mathrm{wup}(\phi_2, B))$
So, if $\mathcal{M} \models \{A\}\phi_1 \cup \phi_2\{B\}$, then $\mathcal{M} \vdash (A \rightarrow (\mathrm{wup}(\phi_1, B) \wedge \mathrm{wup}(\phi_2, B)))$.
Now for the existential part: Assume $\mathcal{M} \models (A)(\phi_1 \cup \phi_2)(B)$. To prove: $\mathcal{M} \vdash (A)(\phi_1 \cup \phi_2)(B)$.

$$\begin{aligned} \mathcal{M} \models_g (\mathrm{wep}(\phi_1, B)) &\iff \exists h({}_g[\phi_1]^{\mathcal{M}}_h \wedge \mathcal{M} \models_h B), \\ \mathcal{M} \models_g (\mathrm{wep}(\phi_2, B)) &\iff \exists h({}_g[\phi_2]^{\mathcal{M}}_h \wedge \mathcal{M} \models_h B) \end{aligned}$$

By inductive hypothesis, we have that

$$\begin{aligned} \mathcal{M} \models_g (\mathrm{wep}(\phi_1, B)) &\implies \mathcal{M} \vdash_g (\mathrm{wep}(\phi_1, B)) \\ \mathcal{M} \models_g (\mathrm{wep}(\phi_2, B)) &\implies \mathcal{M} \vdash_g (\mathrm{wep}(\phi_2, B)) \end{aligned}$$

By the Union rule, $\dfrac{(\mathrm{wep}(\phi_1, B))\phi_1(B) \qquad (\mathrm{wep}(\phi_2, B))\phi_2(B)}{(\mathrm{wep}(\phi_1, B) \vee \mathrm{wep}(\phi_2, B))\phi_1 \cup \phi_2(B)}$

To prove: $\mathcal{M} \models (A \implies \mathrm{wep}(\phi_1, B) \vee \mathrm{wep}(\phi_2, B))$. We know that $\mathcal{M} \models (A \rightarrow \mathrm{wep}(\phi_1 \cup \phi_2, B))$; if we assume there is a g such that $\mathcal{M} \models_g A$, we must prove that $\mathcal{M} \models_g (\mathrm{wep}(\phi_1, B) \vee \mathrm{wep}(\phi_2, B))$.

$$\begin{aligned} \mathcal{M} \models_g \mathrm{wep}(\phi_1 \cup \phi_2, B) &\implies \exists h({}_g[\phi_1 \cup \phi_2]^{\mathcal{M}}_h \wedge \mathcal{M} \models_h B) \\ &\implies \exists h(({}_g[\phi_1]^{\mathcal{M}}_h \vee {}_g[\phi_2]^{\mathcal{M}}_h) \wedge \mathcal{M} \models_h B) \\ &\implies \mathcal{M} \models_g \mathrm{wep}(\phi_1, B) \vee \mathcal{M} \models_g \mathrm{wep}(\phi_2, B) \\ \mathcal{M} \models_g \mathrm{wep}(\phi_1 \cup \phi_2, B) &\implies \mathcal{M} \models_g \mathrm{wep}(\phi_1, B) \vee \mathcal{M} \models_g \mathrm{wep}(\phi_2, B) \end{aligned}$$

Then, $\mathcal{M} \models_g A \rightarrow \mathrm{wep}(\phi_1, B) \vee \mathcal{M} \models_g \mathrm{wep}(\phi_2, B)$,

concluding our proof.

$\dashv$

We now have a calculus that allows us to verify both partial and total correctness for DFOL($\sigma, \cup$) formulas under the executable interpretation presented in Chapter 5. If the execution mechanism of *Dynamo* is faithful to the executable interpretation, then our calculus is also useful for verification of *Dynamo* programs. In the next chapter, we will present an execution mechanism that is even closer to the semantics of DFOL($\sigma, \cup$); since we want to respect the semantics of a logic, we will use a theorem prover to run our programs. But first we want to propose two additions to the language that make it much more expressive: local variable declaration and `WHILE` loops, or in logical terms, the $\exists$ operator and the Kleene star operator.

6.6 Extending the Language

The calculus as presented deals with the language of DFOL($\sigma, \cup$). We will now present rules for dealing with two possible extensions to the core language: the $\exists$ operator and the Kleene star operator, which give us the possibility to use local variables and unbounded iteration, respectively.

6.6.1 The Hiding operator

The semantics of $\exists_x(\phi)$ tell us that we can 'hide' the value of the variable x and treat it as if it was unassigned while we execute ϕ, and recover it afterwards. An use for the $\exists$ operator is to have *local* variable declarations; for example the formula $\exists_z(z \doteq x; \exists x; x \doteq y; \exists y; y \doteq x)$ swaps the values of x and y, with z being used as an auxiliary variable *only within the scope of the* $\exists$ *operator*. This means that any value that z might have had prior to the execution of the formula is restored when execution terminates.

Let's see how the Hoare calculus rules for that would look:

$$\text{Universal correctness:} \quad \frac{\{A\}\phi\{B\}}{\{\forall x A\}\exists_x(\phi)\{B\}} \; x \text{ not free in } B$$

$$\text{Existential correctness:} \quad \frac{(A)\phi(B)}{(\exists x A)\exists_x(\phi)(B)} \; x \text{ not free in } B$$

We also need an axiom that states that $\exists_x(\phi)$ does not alter the value of x:

$$\frac{}{\{A\}\exists_x(\phi)\{A\}} \; free(A) \cup change(\phi) \subseteq \{x\}$$

with *change*(ϕ) being the set of variables that can be changed by execution of (ϕ).

Soundness

Soundness - Universal rule:
Suppose we have derived $\{\forall x A\}\exists_x(\phi)\{B\}$, for x not free in B. Then we must prove that if there is an assignment g under which A is true, and furthermore there is a successful execution of $\exists_x(\phi)$ terminating in assignment h, then B must be true in h.

Assume that there exist g, h such that $\mathcal{M} \models_g A$, and ${}_g[\![\exists_x(\phi)]\!]_h$. We must prove that $\mathcal{M} \models_h B$. We know that $\{A\}\phi\{B\}$ and that x is not free in B; now, if $\mathcal{M} \models_g \forall x A$, then for any $g' \sim_x g$, $\mathcal{M} \models_{g'} A$. Recall that ${}_g[\![\exists_x(\phi)]\!]_h$ iff $\exists g', h'(g \sim_x g', {}_{g'}[\![\phi]\!]_{h'}, h' \sim_x h, h(x) = g(x))$ Since we have assumed ${}_g[\![\exists_x(\phi)]\!]_h$, then we know those g', h' exist. Now, as $g' \sim_x g$ and $\mathcal{M} \models_g \forall x A$, we know that indeed $\mathcal{M} \models_{g'} A$. By inductive hypothesis, we know then that $\mathcal{M} \models_{h'} B$, and since $h \sim_x h'$ and x is not free in B, $\mathcal{M} \models_h B$.
Soundness - Existential rule:
Suppose we have derived $(\exists x A)\exists_x\phi(B)$, for x not free in B. Then we must prove that if there is an assignment g under which $\exists x A$ is true, then there is a successful execution of $\exists_x(\phi)$ terminating in assignment h under which B is true. Let's assume then that such a g exists. We know that $(A)\phi(B)$, and that x is not free in B. If $\mathcal{M} \models_g \exists x A$, then there is a $d \in D$ such that $\mathcal{M} \models_{g_1} A$, for $g_1 = g^d_x$. We also know that ${}_g[\![\exists_x(\phi)]\!]_h$ iff $\exists g', h'(g \sim_x g', {}_{g'}[\![\phi]\!]_{h'}, h' \sim_x h, h(x) = g(x))$. Then, we can set $g' = g_1$, and by inductive hypothesis we know there is an h_1 such that ${}_{g'}[\![\phi]\!]_{h'}$, with $\mathcal{M} \models_{h'} B$, and furthermore $g' \sim_x g$. We only need to take $h(x) = g(x)$ and $h \sim_x h'$ to have an h such that ${}_g[\![\exists_x(\phi)]\!]_h$, and since x is not free in B, $\mathcal{M} \models_h B$.

Completeness

To prove completeness of the calculus including the $\exists_x(\phi)$ rule, we simply expand the proof for the core language with the following:

Completeness - Universal rule:
Under the condition that x not free in B, by induction hypothesis, we have that

$$\mathcal{M} \models \{\mathrm{wup}(\phi, B)\}\phi\{B\} \quad \Longrightarrow \quad \mathcal{M} \vdash \{\mathrm{wup}(\phi, B)\}\phi\{B\}$$

We want to prove that $\mathcal{M} \models \{A\}\exists_x(\phi)\{B\}$ implies $\mathcal{M} \models (A \to (\forall x(\mathrm{wup}(\phi, B))))$. Assume that $\mathcal{M} \models \{A\}\exists_x(\phi)\{B\}$, and that we have a g such that $\mathcal{M} \models_g A$. We want to prove that for any $d \in D$, $\mathcal{M} \models_{g^d_x} \mathrm{wup}(\phi, B)$. By the semantics of $\exists$ and the definition of $\mathrm{wup}(\phi, B)$, we know that for any $d \in D$, either there exist h, h' such that ${}_{g^d_x}[\![\phi]\!]_{h'} \wedge h \sim_x h' \wedge \mathcal{M} \models_h B$ (and since x is not free in B, also $\mathcal{M} \models_{h'} B$), or there is no h' such that ${}_{g^d_x}[\![\phi]\!]_{h'}$. Now, by definition of $\mathrm{wup}(\phi, B)$, we have that $\mathcal{M} \models_{g^d_x} \mathrm{wup}(\phi, B)$, for arbitrary $d \in D$, and therefore $\mathcal{M} \models_g \forall x(\mathrm{wup}(\phi, B))$.

Now, the existential half is the same; we know that

$$\begin{array}{lcl} \mathcal{M} & \models & (A)\exists_x(\phi)(B) \\ \mathcal{M} & \vdash & (\text{wep}(\phi, B))\phi(B) \end{array}$$

Now, we should prove that $\mathcal{M} \models (A \rightarrow (\exists x(\text{wep}(\phi, B))))$, for x free in B. Assume there is a g such that $\mathcal{M} \models_g A$. This means there exists an h such that ${}_g[\![\exists_x\phi]\!]_h$, and that $\mathcal{M} \models_h B$. This again means there exist $d \in D$ and assignments h, h' such that ${}_{g^d_x}[\![\phi]\!]_{h'} \wedge h \sim_x h' \wedge \mathcal{M} \models_h B$ (and since x is not free in B, also $\mathcal{M} \models_{h'} B$). In other words, $\mathcal{M} \models_g (\exists x(\text{wep}(\phi, B))$, and therefore $\mathcal{M} \models (A \rightarrow (\exists x(wep(\phi, B))))$.

6.6.2 The Kleene star

We have presented a set of rules that allow us to reason about correctness of programs in DFOL$(\sigma, \cup)$. While this is already a powerful language, it is still missing ***unbounded*** iteration. If we add the Kleene star operator, we become able to express the WHILE statement, achieving Turing completeness. As with explicit bindings, since the Kleene star operator semantics have been defined in this framework, we can already talk about correctness of programs that include it; we can add it to the executable interpretation later. The *Dynamo* version of WHILE would be the following:

$$(\texttt{while}\ S_1\ S_2)^\circ := (\neg\neg S_1^\circ; S_2^\circ)^*; \neg S_1^\circ$$

$$\text{Universal correctness:} \qquad \frac{\{A\}\phi\{A\}}{\{A\}\phi^*\{A\}}$$

$$\text{Existential correctness:} \qquad \frac{\{A\}\phi\{A\} \qquad (t = i)\phi(t < i)}{(A)\phi^*(A \wedge t < N)}$$

Soundness – Universal:
Suppose we have derived $\{A\}\phi^*\{A\}$. Then we must prove that if there is an assignment g under which A is true, and furthermore there is a successful execution of ϕ^* terminating in assignment h, then A must be true in h.
We know that: (1) ${}_g[\phi^*]_h^{\mathcal{M}}$ iff $g = h$ or there is an i such that ${}_g[\phi]_i^{\mathcal{M}}$ and ${}_i[\phi^*]_h^{\mathcal{M}}$; (2)$\{A\}\phi\{A\}$means that for all g under which A is true, all h that verify ${}_g[\phi]_h^{\mathcal{M}}$ also make A true.
Proof by induction on the number of iterations of ϕ:

- 0 iterations : $g = h$, so trivially h makes A true whenever g does.
- $n + 1$ iterations: We assume that A is true under g and that there exist $g_1 \ldots g_n$ such that ${}_g[\phi]_{g_1}^{\mathcal{M}} \wedge \ldots \wedge {}_{g_{n-1}}[\phi]_{g_n}^{\mathcal{M}}$ and A is true under g_n , and

we must prove that if there are $g_1 \dots g_{n+1}$such that ${}_{g}[\phi]^{\mathcal{M}}_{g_1} \wedge \dots \wedge_{g_n} [\phi]^{\mathcal{M}}_{g_{n+1}}$, then A is true under g_{n+1}.

We have that A is true under g_n, and ${}_{g_n}[\phi]^{\mathcal{M}}_{g_{n+1}}$. By (2), A is true under g_{n+1}.

Soundness – Existential:

We have that (1) $\forall g : \mathcal{M} \models_g A, \exists h : {}_{g}[\phi]^{\mathcal{M}}_{h} \implies \mathcal{M} \models_h A$.

(2) $\forall g : \mathcal{M} \models_g (t = i), \exists h :_g [\phi]^{\mathcal{M}}_{h} \wedge \mathcal{M} \models (t < i)$.

(3) ${}_{g}[\phi^*]^{\mathcal{M}}_{h} \iff g = h \vee \exists i : {}_{g}[\phi]^{\mathcal{M}}_{i} \wedge {}_{i}[\phi^*]^{\mathcal{M}}_{h}$.

Assume $\exists g : \mathcal{M} \models_g A$. To prove: $\exists h : {}_{g}[\phi^*]^{\mathcal{M}}_{h} \wedge \mathcal{M} \models_h A \wedge (t < N)$.

We can use strong induction as follows:

$\forall i \in \mathbb{N}$,

if for all j lesser than i, that from any g that satisfies $(A \wedge t = j)$ we can execute ϕ^* and reach an h that satisfies $(A \wedge t < N)$ means that for any g that satisfies $(A \wedge t = i)$ we can also execute ϕ^* and reach an h that satisfies $(A \wedge t < N)$, then for all $i \in \mathbb{N}$, for any g that satisfies $(A \wedge t = i)$ there is an h such that ${}_{g}[\phi^*]^{\mathcal{M}}_{h}$ and which satisfies $(A \wedge t < N)$.

So, assume (4) $\forall i \in \mathbb{N}, (\forall j < i \ \forall g : \mathcal{M} \models_g (A \wedge t = j) \implies \exists h : {}_{g}[\phi^*]^{\mathcal{M}}_{h} \wedge \mathcal{M} \models_h (A \wedge t < N))$.

To prove: $\forall g : \mathcal{M} \models_g (A \wedge t = i) \implies \exists h : {}_{g}[\phi^*]^{\mathcal{M}}_{h} \wedge \mathcal{M} \models_h (A \wedge t < N)$

From (1) and (2), we know that $\exists h_1 : {}_{g}[\phi]^{\mathcal{M}}_{h_1} \wedge \mathcal{M} \models_{h_1} (A \wedge t < i)$.

Now, h_1 satisfies (3), and by (4), ${}_{h_1}[\phi^*]^{\mathcal{M}}_{h} \wedge \mathcal{M} \models_h (A \wedge t < N)$.

Completeness of the Kleene star rules Completeness of rules for unbounded iteration has of course the problem that preconditions might not be first order definable. We introduce the notion of *expressive* models [vBV92]:

6.6.1. Definition. A model (D, I) is expressive if weakest preconditions (ϕ, B) are first order definable in it, for every ϕ, and B.

Examples of expressive models are all finite models, as well as the natural numbers. Note however that this is not a common property of models.

Even then, proving completeness of the rules for the Kleene star was possible only for the universal correctness rule:

For every expressive model $\mathcal{M}$, assuming $\mathcal{M} \models \{A\}\phi^*\{B\}$, we must prove $\mathcal{M} \vdash \{A\}\phi^*\{B\}$, under the assumption that $\mathcal{M} \models \{A\}\phi\{B\} \implies \mathcal{M} \vdash \{A\}\phi\{B\}$.

We assume there exists an assignment g such that $\mathcal{M} \models_g A$, and an assignment h such that ${}_{g}[\phi^*]_{h}(1)$.

If $\mathcal{M} \models \{A\}\phi^*\{B\}$, by semantics of *, we have that $\mathcal{M} \models (A \rightarrow B)$. Consider $C = \mathrm{wup}(\phi^*, B)$. It is clear from the premises that $M \models (A \rightarrow C)$. Now, there is the trivial case in which $\mathcal{M} \models \{A\}\phi\{\bot\}$. In this case, the only h that satisfies (1) is $h = g$. Also, by inductive hypothesis, $\mathcal{M} \models \{A\}\phi\{\bot\} \implies \mathcal{M} \vdash \{A\}\phi\{\bot\}$. Then, we can derive $\mathcal{M} \vdash \{A\}\phi^*\{B\}$:

$$\dfrac{\dfrac{\dfrac{\{A\}\phi\{\bot\} \quad \bot \rightarrow A}{\{A\}\phi\{A\}}}{\{A\}\phi^*\{A\}} \qquad A \rightarrow B}{\{A\}\phi^*\{B\}}$$

In the nontrivial case, there is an assignment h such that ${}_g[\![\phi]\!]_h$. Now,

$$\begin{array}{rcl}
\mathcal{M} \models_g A & \Longrightarrow & \mathcal{M} \models_g \mathrm{wup}(\phi^*, B) \\
\mathcal{M} \models_g \mathrm{wup}(\phi^*, B) & \Longleftrightarrow & \mathcal{M} \models_g B \wedge \mathrm{wup}((\phi; \phi^*), B) \\
 & \Longleftrightarrow & \mathcal{M} \models_g B \wedge \mathrm{wup}(\phi, \mathrm{wup}(\phi^*, B)) \\
 & \Longleftrightarrow & \mathcal{M} \models_g B \wedge \mathrm{wup}(\phi, C)
\end{array}$$

In particular, $\mathcal{M} \models (C \rightarrow B)$. As per definition of wup, $\mathcal{M} \models \{C\}\phi\{C\}$. Then, by induction hypothesis, we have that $\mathcal{M} \vdash \{C\}\phi\{C\}$. Then,

$$\dfrac{A \rightarrow C \quad \dfrac{\{C\}\phi\{C\}}{\{C\}\phi^*\{C\}} \quad C \rightarrow B}{\{A\}\phi^*\{B\}}$$

The main problem for proving the completeness of the existential correctness rules lies in that one of the antecedents is an universal correctness statement; we can't switch focus from universal to existential correctness without going through negation.

6.7 Conclusion

We have now presented a way to verify *Dynamo* programs; the calculus has been proved sound and complete for the core language, and rules for extensions have been proposed. In the next chapter, we will see how to go even closer to the semantics of DFOL, and also propose an executable interpretation for both the Kleene star and the $\exists$ operator. We will also see how infinitary logic may be used for reasoning about DFOL($*$).

Chapter 7

Tableau Reasoning with DFOL

7.1 Introduction

We have shown how to make sure the semantics of a formula in DFOL($\sigma, \cup$) follow a given specification. In Chapter 5 we introduced an executable program interpretation for formulas in DFOL($\cup$), but we were not quite happy with the result; it gave up all too often, and we had to simulate negation-as-failure, which was a bit involved and did not make it particularly easy to deal with universal quantification.

Our plan now is to introduce an executable interpretation to DFOL($\sigma, \cup$), which is more faithful to the semantics, and works in a completely different way: it is a tableau calculus. We start by describing a tableau calculus for DFOL($\sigma, \cup$) making intensive use of our theory of explicit binding. The explicit bindings represent the intermediate results of calculation that get carried along in the computation process. We illustrate with examples from standard first order reasoning, imperative programming, and derivation of postconditions for imperative programs. Later, we develop an infinitary calculus for DFOL($\cup, \sigma, \exists, *$), and provide a completeness proof, and finally we enunciate some of the relationships with existing calculi. The first calculus that are the subject of this chapter forms the computation and inference engine of *Dynamo*, our toy programming language for theorem proving and computing with DFOL.
Let us consider a signature Σ; we will call $\mathcal{L}_\Sigma$ the DFOL($\sigma, \cup$) language over Σ. The key relation we want to get to grips with in this chapter is the dynamic entailment relation that is due to [GS91]:

7.1.1. Definition. [Entailment in DFOL] ϕ dynamically entails ψ, notation $\phi \models \psi$, if and only if:

> for all $\mathcal{L}_\Sigma$ models $\mathcal{M}$, all valuations s, u for $\mathcal{M}$, if ${}_s[\phi]^{\mathcal{M}}_u$ then there is a variable state u' for which ${}_u[\psi]^{\mathcal{M}}_{u'}$.

In the calculus we will need the function $input(\phi)$, the set of variables that have an input constraining occurrence in ϕ (with $\phi \in \mathcal{L}_\Sigma$), Let $var(\bar{t})$ be the variables occurring in $\bar{t}$. The definition of $input(\phi)$ is as follows:

7.1.2. Definition. [Input constrained variables of $\mathcal{L}_\Sigma$ formulas]

$$\begin{array}{rcl} input(\theta) & := & var(rng(\theta)) \\ input(\theta;\phi) & := & var(rng(\theta)) \cup (input(\phi) \setminus dom(\theta)) \\ input(\exists v;\phi) & := & input(\phi) \setminus \{v\} \\ input(P\bar{t};\phi) & := & var(\bar{t}) \cup input(\phi) \\ input(t_1 \doteq t_2;\phi) & := & var\{t_1,t_2\} \cup input(\phi) \\ input(\neg(\phi_1);\phi_2) & := & input(\phi_1) \cup input(\phi_2) \\ input((\phi_1 \cup \phi_2);\phi_3) & := & input(\phi_1;\phi_3) \cup input(\phi_2;\phi_3). \end{array}$$

The following proposition (the DFOL counterpart to the finiteness lemma from classical FOL) can be proved by induction on formula structure:

7.1.3. Proposition. *For all $\mathcal{L}_\Sigma$ models $\mathcal{M}$, all valuations s, s', u, u' for $\mathcal{M}$, all $\mathcal{L}_\Sigma$ formulas ϕ:*

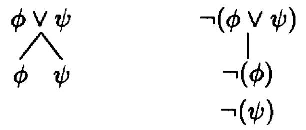
$$_s[\![\phi]\!]^{\mathcal{M}}_u \text{ and } s \sim_{\mathsf{VAR}\setminus input(\phi)} s' \text{ imply } \exists u' \text{ with } {}_{s'}[\![\phi]\!]^{\mathcal{M}}_{u'}.$$

7.2 Tableaux for DFOL($\sigma, \cup$)

7.2.1 Adaptation of Tableaux to Dynamic Reasoning

In classical tableau theorem proving, we want to check the entailment relation by looking for a possibility of making the antecedent ϕ true and the consequent ψ false. If that fails, then we conclude that ψ does follow from ϕ; and if it succeeds we can build a counterexample from any tableau branch that remains open; see [vB86].

Instead of the original method of keeping a formula we want to make true and one we want to make false, and two rules for each operator (one for the false side and one for the true side), we have one formula Φ we want to make true, and two (types of) rules for each operator; one for positive and one for negative occurrences. Consider for example the tableau rule for disjunction in classical logic; a tableau splitting rule like $\vee$ has the node with the disjunction $\phi \vee \psi$ above the two branches with the disjuncts ϕ and ψ. The rule $\vee$ serves as the 'true side rule', and is matched by a rule $\neg\vee$ for dealing with the 'false side'.



In the dynamic versiòn of FOL, order matters: the sequencing operator ';' is not commutative in general. Suppose Φ were to consist of $\exists x; Px$ and $\neg Px$. Then if we read Φ as $\exists x; Px; \neg Px$, we should get a contradiction, but if we read Φ as $\neg Px; \exists x; Px$ then the formula has a model that contains both Ps and non-Ps.

Suppose Φ were to consist of just $\exists x; Px; \neg(Qx \cup Sx)$. Then we can apply the $\neg\cup$ analogue of $\neg\vee$ to Φ, but we should make sure that the results of this application, $\neg Qx$ and $\neg Sx$, remain in the scope of $\exists x; Px$. In other words, the result should be: $\exists x; Px; \neg Qx; \neg Sx$ (or $\exists x; Px; \neg Sx; \neg Qx$: being negated formulas, $\neg Qx$ and $\neg Sx$ are interchangeable), with both $\neg Qx$ and $\neg Sx$ in the dynamic scope of the quantifier $\exists x$. In the tableau calculus to be presented, we will ensure that negation rules $\neg o$ take dynamic context into account, and that all formulas come with an appropriate binding context, to be supplied by explicit bindings.

Local Bindings Versus Global Substitutions. As a rule, we don't apply bindings to formulas unless it is needed; in fact, when processing a formula ϕ with a binding θ, we store the formula $\theta; \phi$ and apply the binding only as needed, for example when processing an atom. We can see tableau theorem proving as the process of building a domain D and finding out whether the requirements imposed on D by Φ are consistent, by decomposing the formulas into positive and negative facts and seeing that there is no contradiction between them. We will employ an infinite set $\boldsymbol{F}_{\mathbf{sko}}$ of skolem functions, with $\boldsymbol{F}_{\mathbf{sko}} \cap \mathsf{FUN} = \emptyset$, plus a set of fresh variables $\boldsymbol{X}$, with $\mathsf{VAR} \cap \boldsymbol{X} = \emptyset$. Call the extended signature Σ^*, and the extended language $\mathcal{L}_{\Sigma^*}$. Let T_{Σ^*} be the terms of the extended language, and $T_{\Sigma^*}^{\mathsf{VAR}}$ the terms of the extended language without occurrences of members of $\boldsymbol{X}$ (the *frozen* terms of $\mathcal{L}_{\Sigma^*}$). We have then two instances of grounding: ground terms, those without any variables, and frozen terms, without variables from $\boldsymbol{X}$. We extend the notion to literals, and call an $\mathcal{L}_{\Sigma^*}$ literal *frozen* if it contains only frozen terms.

The variables in $\boldsymbol{X}$ will function as universal tableau variables [Fit96]. Where the bindings of the variables from VAR are local to a tableau branch, the bindings of the variables from $\boldsymbol{X}$ are global to the whole tableau. Next to the (local) bindings for the variables VAR of $\mathcal{L}_\Sigma$, we introduce (global) substitutions $\boldsymbol{\sigma}$ for the fresh variables $\boldsymbol{X}$ in $\mathcal{L}_{\Sigma^*}$, and extend these to (sequences of) terms and (sets of) formulas in the manner of Definition 1.4.14. A substitution $\boldsymbol{\sigma}$ is a unifier of a set of (sequences of) terms T if $\boldsymbol{\sigma}T$ contains a single term (sequence of terms). It is a most general unifier (mgu) of T if $\boldsymbol{\sigma}$ is a unifier of T, and for all unifiers $\boldsymbol{\rho}$ of T there is a $\boldsymbol{\theta}$ with $\boldsymbol{\sigma} = \boldsymbol{\theta} \cdot \boldsymbol{\rho}$. Similarly for formulas. Note that only unifiers for global *substitutions* (the term maps for the global tableau variables from $\boldsymbol{X}$) will ever be computed.

The definitions and results on binding extend to bindings with values in T_{Σ^*}, and to substitutions (domain $\subset \boldsymbol{X}$, values in T_{Σ^*}). Still, the global substitutions play an altogether different role in the tableau construction process, so we use a different notation for them, and write (representations for) global substitutions as

$$\{\boldsymbol{x}_1 \mapsto t_1, \ldots, \boldsymbol{x}_n \mapsto t_n\}.$$

7.2.2 Tableaux for DFOL($\sigma, \cup$) Formula Sets

If Σ is a first order signature, a DFOL($\sigma, \cup$) tableau over Σ is a finitely branching tree with nodes consisting of (sets of) $\mathcal{L}_{\Sigma^*}$ formulas. A branch in a tableau $\boldsymbol{T}$ is a maximal path in $\boldsymbol{T}$. We will follow custom in occasionally identifying a branch B with the set of its formulas.

Let Φ be a set of $\mathcal{L}_\Sigma$ formulas. A DFOL($\sigma, \cup$) tableau for Φ is constructed by a (possibly infinite) sequence of applications of the following rules:

Initialization The tree consisting of a single node [] is a tableau for Φ.

Binding Composition Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$. Let $\phi \in B \cup \Phi$, let $\theta; \rho$ occur in ϕ, and let ϕ' be the result of replacing $\theta; \rho$ in ϕ by $\theta \circ \rho$. Then the tree $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by extending B by ϕ' is a tableau for Φ.

Expansion Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$. Let $\phi \in B \cup \Phi$. Then the tree $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by extending B according to one of the tableau expansion rules presented in subsection 7.2.3, applied to ϕ, is a tableau for Φ.

Equality Replacement Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$. Let $t_1 \doteq t_2 \in B \cup \Phi$ or $t_2 \doteq t_1 \in B \cup \Phi$, and $L(t_3) \in B \cup \Phi$, where L is a literal. Suppose t_1, t_3 are unifiable with MGU $\boldsymbol{\sigma}$. Then $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by applying $\boldsymbol{\sigma}$ to all formulas in $\boldsymbol{T}$, and extending branch $\boldsymbol{\sigma}B$ with $L(\boldsymbol{\sigma}t_2)$ is a tableau for Φ.

Closure Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$, and $L, \overline{L'}$ are literals in $B \cup \Phi$. If $L, \overline{L'}$ are unifiable with MGU $\boldsymbol{\sigma}$ then $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by applying $\boldsymbol{\sigma}$ to all formulas in $\boldsymbol{T}$ is a tableau for Φ.

A tableau branch can be considered a conjunction of formulas: all of them have to be true for that particular branch to remain open. Since we want to include treatment of identities, the closure of a branch is more involved than in classical free variable tableaux. When checking for closure, we can consider variables from VAR as existentially quantified: occurrence of Pv along branch B does *not* mean that everything has property P, but rather that the element called v has P. We can *freeze* the parameters from $\boldsymbol{X}$ by mapping them to fresh parameters from VAR. Applying a freezing substitution to a tableau replaces references to 'arbitrary objects' $\boldsymbol{x}, \boldsymbol{y}, \ldots$, by 'arbitrary names.' We can then determine closure of a branch B in terms of the *congruence closure* of the set of equalities occurring in a frozen image σB of the branch. See [BN98], Chapter 4, for what follows about congruence closures.

If Φ is a set of $\mathcal{L}_{\Sigma^*}$ formulas without parameters from $\boldsymbol{X}$, the congruence closure of Φ, notation $\approx_\Phi$, is the smallest congruence on T that contains all the equalities in Φ. In general, $\approx_\Phi$ will be infinite: if $a \doteq b$ is an equality in Φ, and f is a one-placed function symbol in the language, then $\approx_\Phi$ will contain $fa \doteq fb, ffa \doteq ffb, fffa \doteq fffb, \ldots$. Therefore, one uses congruence closure modulo some finite set instead.

Let S be the set of all sub-terms (not necessarily proper) of terms occurring in a literal in Φ. Then the congruence closure of Φ modulo S, notation $\mathrm{CC}_S(\Phi)$, is the finite set of equalities $\approx_\Phi \cap (S \times S)$. We can decide whether $t \doteq t'$ in $\mathrm{CC}_S(\Phi)$; [BN98] gives an algorithm for computing $\mathrm{CC}_S(G)$, for finite sets of equalities G and terms S, in polynomial time.

7.2.1. Definition. $t \approx t'$ is suspended in a frozen $\mathcal{L}_{\Sigma^*}$ formula set Φ if $t \doteq t' \in \mathrm{CC}_S(\Phi)$. We extend this notation to sequences: $\bar{t} \approx \bar{t'}$ is suspended in Φ if $t_1 \approx t'_1, \ldots, t_n \approx t'_n$ are suspended in Φ.

A frozen $\mathcal{L}_{\Sigma^*}$ formula set Φ is closed if either $\neg(\theta) \in \Phi$ (recall that $\bot$ is an abbreviation for $\neg([]))$, or for some $\bar{t} \approx \bar{t'}$ suspended in Φ we have $P\bar{t} \in \Phi$, $\neg P\bar{t'} \in \Phi$, or for a pair of terms t_1, t_2 with $t_1 \approx t_2$ suspended in Φ we have $t_1 \neq t_2 \in \Phi$.

A tableau $\boldsymbol{T}$ is *closed* if there is a freezing substitution σ of $\boldsymbol{T}$ such that each of its branches σB is closed.

7.2.3 Tableau Expansion Rules

Note that we can take the form of any $\mathcal{L}_{\Sigma^*}$ formula to be $\theta; \phi$, by prefixing or suffixing [] if necessary. The tableau rules have the effect that bindings get pushed from left to right in the tableaux, and appear as computed results at the open end nodes.

Conjunctive Type. Here are the rules for formulas of conjunctive type (type α in the Smullyan typology):

$$\begin{array}{c} \theta;\ P\bar{t};\ \phi \\ | \\ P\theta\bar{t} \\ \theta;\ \phi \end{array} \qquad \begin{array}{c} \theta;\ t_1 \doteq t_2;\ \phi \\ | \\ \theta t_1 \doteq \theta t_2 \\ \theta \circ [\theta t_i/v];\ \phi \\ \\ \text{where } \theta t_i = v \in \mathsf{VAR}, i \in \{1,2\} \end{array} \qquad \begin{array}{c} \theta;\ t_1 \doteq t_2;\ \phi \\ | \\ \theta t_1 \doteq \theta t_2 \\ \theta;\ \phi \\ \\ \text{where } \theta t_i \notin \mathsf{VAR}, i \in \{1,2\} \end{array}$$

$$\begin{array}{c} \neg(\theta;\ (\phi_1 \cup \phi_2);\ \phi_3) \\ | \\ \neg(\theta;\ \phi_1;\ \phi_3) \\ \neg(\theta;\ \phi_2;\ \phi_3) \end{array} \qquad \begin{array}{c} \theta;\ (\!(\phi_1)\!);\ \phi_2 \\ | \\ (\!(\theta;\ \phi_1)\!) \\ \theta;\ \phi_2 \end{array} \qquad \begin{array}{c} \theta;\ \neg(\phi_1);\ \phi_2 \\ | \\ \neg(\theta;\ \phi_1) \\ \theta;\ \phi_2 \end{array}$$

Call the formula at the top node of a rule of this kind α and the formulas at the leaves α_1, α_2. To expand a tableau branch B by an α rule, extend B with both α_1 and α_2.

Disjunctive Type. The rules for formulas of disjunctive type (Smullyan's type β):

$$\begin{array}{cc} \multicolumn{2}{c}{\neg(\theta;\ P\bar{t};\ \phi)} \\ \diagup & \diagdown \\ \neg P\theta\bar{t} & \neg(\theta;\ \phi) \end{array} \quad \begin{array}{cc} \multicolumn{2}{c}{\neg(\theta;\ t_1 \doteq t_2;\ \phi)} \\ \diagup & \diagdown \\ \theta t_1 \neq \theta t_2 & \neg(\theta;\ \phi) \end{array} \quad \begin{array}{cc} \multicolumn{2}{c}{\theta;\ (\phi_1 \cup \phi_2);\ \phi_3} \\ \diagup & \diagdown \\ \theta; \phi_1; \phi_3 & \theta; \phi_2; \phi_3 \end{array} \quad \begin{array}{cc} \multicolumn{2}{c}{\neg(\theta;\ \neg(\phi_1);\ \phi_2)} \\ \diagup & \diagdown \\ (\!(\theta;\ \phi_1)\!) & (\!(\theta;\ \neg\phi_2)\!) \end{array}$$

Call the formula at the top node of a rule of this kind β, the formula at the left leaf β_1 and the formula at the right leaf β_2. To expand a tableau branch B by a β rule, either extend B with β_1 or with β_2.

Universal Type. Rule for universal formulas (Smullyan's type γ):

$$\begin{array}{c} \neg(\theta;\ \exists v;\ \phi) \\ | \\ \neg(\theta \circ [\boldsymbol{x}/v];\ \phi) \end{array}$$

Here $\boldsymbol{x}$ is a universal variable taken from $\boldsymbol{X}$ that is new to the tableau. Call the formula at the top node of a rule of this kind $\gamma(v)$, and the formula at the leaf γ_1. To expand a tableau branch B by a γ rule, extend B with γ_1.

Existential Type. Rule for existential formulas (Smullyan's type δ):

$$\boxed{\begin{array}{c} \theta;\ \exists v;\ \phi \\ | \\ \theta \circ [\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)/v];\ \phi \end{array}}$$

Here $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$ are the universal parameters upon which interpretation of $\exists v; \phi$ depends, and $\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)$ is a skolem constant that is new to the tableau branch.[1]

By Proposition 7.1.3, $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ is a subset of $\mathit{input}(\theta; \exists v; \phi)$, or, since no members of $\boldsymbol{X}$ occur in ϕ or in $\mathit{dom}(\theta)$, a subset of $\boldsymbol{X} \cap \mathit{input}(\theta) = \boldsymbol{X} \cap \mathit{var}(\mathit{rng}(\theta))$. From this set, we only need[2]

$$\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} := \boldsymbol{X} \cap \mathit{var}(\mathit{rng}(\theta \restriction (\mathit{input}(\phi) \backslash \{v\}))).$$

Call the formula at the top node of a rule of this kind $\delta(v)$, and the formula at the leaf δ_1. To expand a tableau branch B by a δ rule, extend B with δ_1.

The tableau calculus specifies guidelines for extending a tableau tree with new leaf nodes. If one starts out from a single formula, at each stage only a finite number of rules can be applied. Breadth first search will get us all the possible tableau developments for a given initial formula, but this procedure is not an *algorithm* for tableau proof construction; it doesn't tell us how to choose which branch to expand or what to freeze variables from $\boldsymbol{X}$ to. We'll see the algorithm implementing this calculus in chapter 8.

7.3 Soundness of the Calculus

Valuations for Σ^* models $\mathcal{M} = (D, I)$ are functions in $\mathsf{VAR} \cup \boldsymbol{X} \to D$. Any such function g can be viewed as a union $s \cup h$ of a function $s \in \mathsf{VAR} \to D$ and a function $h \in \boldsymbol{X} \to D$ (take $s = g \restriction \mathsf{VAR}$ and $h = g \restriction \boldsymbol{X}$). For satisfaction in Σ^* models we use the notation ${}_{s\cup h}[\phi]^{M}_{u}$, to be understood in the obvious way. In terms of this we define the notion that we need to account for the universal nature of the $\boldsymbol{X}$ variables.

7.3.1. DEFINITION. Let $\phi \in \mathcal{L}_{\Sigma^*}$, $\mathcal{M} = (D, I)$ a Σ^* model, $s, u \in \mathsf{VAR} \to D$.

[1] It is well-known that this can be optimized so that the choice of skolem constant only depends on $\theta;\ \exists v;\ \phi$.

[2] In an implementation, it may be more efficient to not bother about computing $\mathit{input}(\phi)$, and instead work with $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} := \boldsymbol{X} \cap \mathit{var}(\mathit{rng}(\theta))$.

Then ${}^{\forall}_{s}[\phi]^{\mathcal{M}}$ iff for every $h : \boldsymbol{X} \to D$ there is a $u : \mathsf{VAR} \cup \boldsymbol{X} \to D$ with ${}_{s\cup h}[\phi]^{\mathcal{M}}_{u}$. We say: s universally satisfies ϕ in $\mathcal{M}$.

For any tableau $\boldsymbol{T}$ we say that $\mathbf{C}(\boldsymbol{T})$ if there is an Σ^* model $\mathcal{M}$, a branch B of $\boldsymbol{T}$ and a VAR valuation s for $\mathcal{M}$ such that every formula ϕ of B is universally satisfied by s in $\mathcal{M}$.

7.3.2. Lemma. *If s universally satisfies ϕ in $\mathcal{M}$, and $\boldsymbol{\sigma}$ is a substitution on $\boldsymbol{X}$ that is safe for ϕ, then s universally satisfies $\boldsymbol{\sigma}\phi$ in $\mathcal{M}$.*

Proof. If ${}^{\forall}_{s}\llbracket\phi\rrbracket^{\mathcal{M}}$ then for every $\boldsymbol{X}$ valuation h in $\mathcal{M}$ there is a $\mathsf{VAR} \cup \boldsymbol{X}$ valuation u in $\mathcal{M}$ with ${}_{s\cup h}\llbracket\phi\rrbracket^{\mathcal{M}}_{u}$. Thus for every h in $\mathcal{M}$ there is a $\mathsf{VAR} \cup \boldsymbol{X}$ valuation u in $\mathcal{M}$ with

$$ {}_{s\cup h_{\boldsymbol{\sigma}}}\llbracket\phi\rrbracket^{\mathcal{M}}_{u}, $$

and therefore for every h in $\mathcal{M}$ there is a $\mathsf{VAR} \cup \boldsymbol{X}$ valuation u in $\mathcal{M}$ with

$$ {}_{s\cup h}\llbracket\boldsymbol{\sigma};\phi\rrbracket^{\mathcal{M}}_{u}. $$

Since $\boldsymbol{\sigma}$ is safe for ϕ we have by the binding lemma that $[\boldsymbol{\sigma}\phi]^{\mathcal{M}} = [\boldsymbol{\sigma};\phi]^{\mathcal{M}}$, and it follows that s universally satisfies $\boldsymbol{\sigma}\phi$ in $\mathcal{M}$. ⊣

With this, we can show that the tableau building rules preserve the $\mathbf{C}(\boldsymbol{T})$ relation.

7.3.3. Lemma (Tableau Expansion Lemma).

1. *If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of binding composition, then* $\mathbf{C}(\boldsymbol{T})$ *implies* $\mathbf{C}(\boldsymbol{T}')$.

2. *If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of a tableau expansion rule, then* $\mathbf{C}(\boldsymbol{T})$ *implies* $\mathbf{C}(\boldsymbol{T}')$.

3. *If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of equality replacement, then* $\mathbf{C}(\boldsymbol{T})$ *implies* $\mathbf{C}(\boldsymbol{T}')$.

4. *If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of closure, then* $\mathbf{C}(\boldsymbol{T})$ *implies* $\mathbf{C}(\boldsymbol{T}')$.

Proof. 1. Immediate from the fact that $\theta;\rho$ and $\theta\circ\rho$ have the same interpretation.

2. All of the α and β rules are straightforward, except perhaps for the α equality rules. The change of θ to $\theta\circ[\theta t_i/v]$, where $\theta t_j = v$ $(i, j \in \{1,2\}, i \neq j,)$ reflects the fact that $\theta t_1 \doteq \theta t_2$ gives us the information to instantiate v.

The γ rule. Assume $\neg(\theta;\exists v;\phi)$ is universally satisfied by s in $\mathcal{M}$. We may assume that θ is safe for $\exists v;\phi$. If $\boldsymbol{x} \in \boldsymbol{X}$, $\boldsymbol{x}$ fresh to the tableau, then $\theta\circ[\boldsymbol{x}/v]$ will be safe for ϕ, and $\neg(\theta\circ[\boldsymbol{x}/v];\phi)$ will be universally satisfied by s in $\mathcal{M}$.

The δ rule. Assume s universally satisfies $\theta; \exists v; \phi$ in $\mathcal{M}$. By induction on tableau structure, $dom(\theta) \subset \mathsf{VAR}$. Define a new model $\mathcal{M}'$ where $\mathrm{sk}_{\theta;\exists v;\phi}$ is interpreted as the function $f : D^n \to D$ given by

$$\begin{array}{lll} f(d_1, \ldots, d_n) & := & \text{some } d \text{ for which } \phi \text{ succeeds in } \mathcal{M} \\ & & \text{for input state } s_\theta[d_1/\boldsymbol{x}_1, \ldots, d_n/\boldsymbol{x}_n, d/v]. \end{array}$$

By the fact that s universally satisfies $\theta; \exists v; \phi$ in $\mathcal{M}$ and by the way we have picked $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, such a d must exist. Then s will universally satisfy $\theta \circ [\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)/v]; \phi$ in $\mathcal{M}'$, while universal satisfaction of other formulas on the branch is not affected by the switch from $\mathcal{M}$ to $\mathcal{M}'$.

3 and 4 follow immediately from Lemma 7.3.2. $\dashv$

7.3.4. THEOREM (SOUNDNESS). *If $\phi, \psi \in \mathcal{L}_\Sigma$, and the tableau for $\phi; \neg(\psi)$ closes, then $\phi \models \psi$.*

Proof. If the tableau for $\phi; \neg(\psi)$ closes, then by the Tableau Expansion Lemma, there are no $\mathcal{M}, s$ such that ${}^{\forall}_{s}[\![\phi; \neg(\psi)]\!]^{\mathcal{M}}$. Since $\phi, \psi \in \mathcal{L}_\Sigma$, there are no $\mathcal{M}, s, u$ with ${}_s[\![\phi; \neg(\psi)]\!]_u^{\mathcal{M}}$. In other words, for every Σ model $\mathcal{M}$ and every pair of variable states s, u for $\mathcal{M}$ with ${}_s[\![\phi]\!]_u^{\mathcal{M}}$ there has to be a variable state u' with ${}_u[\![\psi]\!]_{u'}^{\mathcal{M}}$. Thus, we have $\phi \models \psi$ in the sense of Definition 7.1.1. $\dashv$

7.4 Derived Principles

Universal Quantification. Immediately from the definition of $\forall v(\phi)$ we get:

$$\begin{array}{c} \theta; \forall v(\phi_1); \phi_2 \\ | \\ ((\theta \circ [\boldsymbol{x}/v]; \phi_1)) \\ \theta; \phi_2 \end{array}$$

where $\boldsymbol{x} \in \boldsymbol{X}$ new to the tableau

Blocks Detachment. A sequence of blocks $\pm(\phi_1); \ldots; \pm(\phi_n)$, where $\pm(\phi_i)$ is either $((\phi_i))$ or $\neg(\phi_i)$, yields the set of its components, by a series of applications of distribution of the empty substitution over block or negation. This is useful, as the formulas $\pm(\phi_1), \ldots, \pm(\phi_n)$ can be processed in any order. In a schema:

$$\begin{array}{c} \pm(\phi_1); \ldots; \pm(\phi_n) \\ | \\ \pm(\phi_1) \\ \vdots \\ \pm(\phi_n) \end{array}$$

Negation Splitting. The following rules are admissible in the calculus:

$$\begin{array}{c} \neg(\phi;\neg(\psi);\chi) \\ \diagup \quad \diagdown \\ ((\phi;\psi)) \qquad \neg(\phi;\chi) \end{array} \qquad\qquad \begin{array}{c} \neg(\phi;((\psi));\chi) \\ \diagup \quad \diagdown \\ ((\phi;\neg(\psi))) \qquad \neg(\phi;\chi) \end{array}$$

Negation splitting can be viewed as the DFOL guise of a well known principle from modal logic: $\Box(A \vee B) \rightarrow (\Diamond A \vee \Box B)$. To see the connection, note that $\neg(\phi;\neg(\psi);\chi)$ is semantically equivalent to $\neg(\phi;\neg(\psi \cup \neg(\chi)))$, where $\neg(\phi;\neg\cdots)$ behaves as a $\Box$ modality.

7.5 Some Examples

In the examples we will use $v_0, v_1, \ldots$ as 0-ary skolem terms for v, etcetera.

Syllogistic Reasoning. Consider the syllogism:

$$\forall x(Ax \rightarrow Bx), \forall x(Bx \rightarrow Cx) \models \forall x(Ax \rightarrow Cx).$$

This is an abbreviation of (7.1).

$$\neg(\exists x; Ax; \neg Bx), \neg(\exists x; Bx; \neg Cx) \models \neg(\exists x; Ax; \neg Cx) \tag{7.1}$$

The DFOL$(\sigma, \cup)$ tableau for this example, a tableau refutation of

$$\neg(\exists x; Ax; \neg Bx); \neg(\exists x; Bx; \neg Cx); ((\exists x; Ax; \neg Cx))$$

is in Figure 7.1.

Reasoning about '<'. Consider example (7.2).

$$y < x; \neg(\exists x; \exists y; x < y). \tag{7.2}$$

This is contradictory, for first two objects of different size are introduced, and next we are told that all objects have the same size. The contradiction is derived as follows:

$$\begin{array}{c} y < x; \neg(\exists x; \exists y; x < y) \\ | \\ y < x \\ \neg(\exists x; \exists y; x < y) \\ | \\ \neg([\boldsymbol{x}_1/x, \boldsymbol{x}_2/y]; x < y) \\ | \\ \neg\boldsymbol{x}_1 < \boldsymbol{x}_2 \\ | \\ \{\boldsymbol{x}_1 \mapsto y, \boldsymbol{x}_2 \mapsto x\} \\ \times \end{array}$$

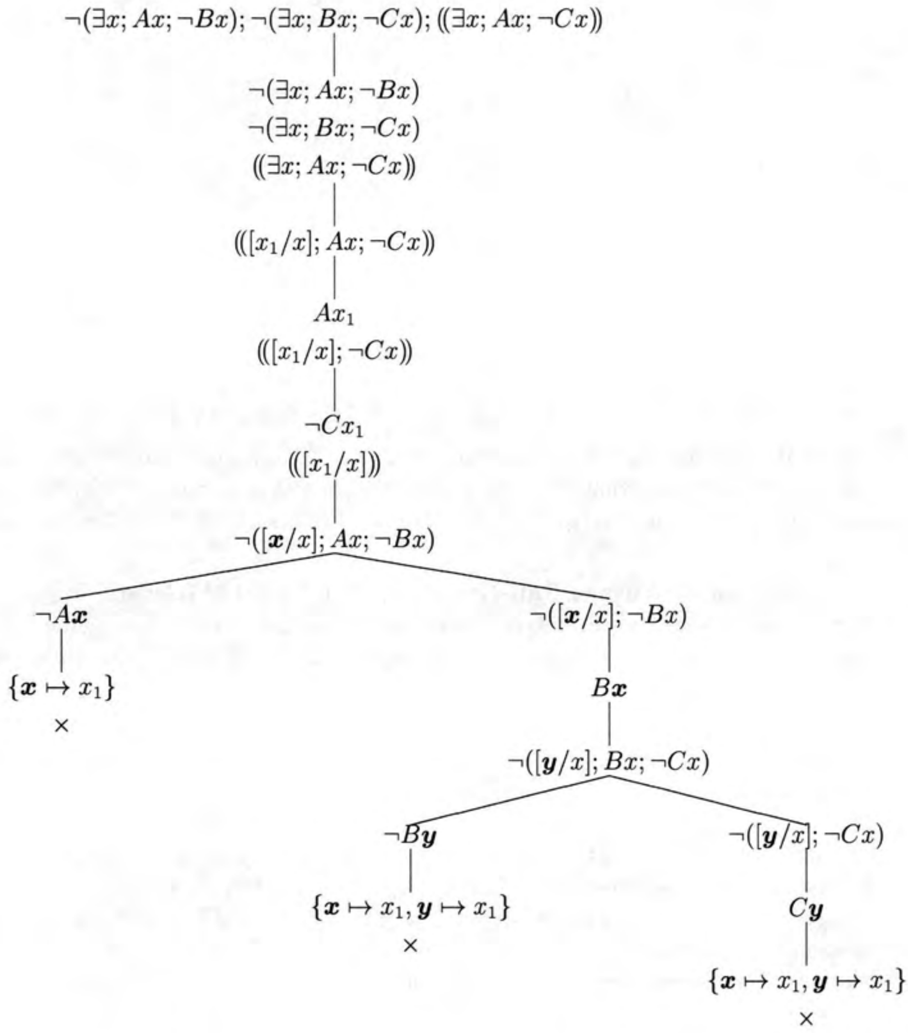
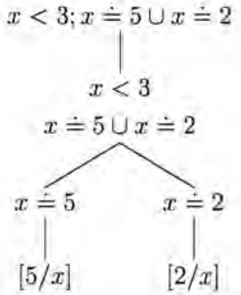

$\neg(\exists x; Ax; \neg Bx); \neg(\exists x; Bx; \neg Cx); (\!(\exists x; Ax; \neg Cx)\!)$
$\neg(\exists x; Ax; \neg Bx)$
$\neg(\exists x; Bx; \neg Cx)$
$(\!(\exists x; Ax; \neg Cx)\!)$
$(\!([x_1/x]; Ax; \neg Cx)\!)$
Ax_1
$(\!([x_1/x]; \neg Cx)\!)$
$\neg Cx_1$
$(\!([x_1/x])\!)$
$\neg([\boldsymbol{x}/x]; Ax; \neg Bx)$
$\neg A\boldsymbol{x}$
$\{\boldsymbol{x} \mapsto x_1\}$
×
$\neg([\boldsymbol{x}/x]; \neg Bx)$
$B\boldsymbol{x}$
$\neg([\boldsymbol{y}/x]; Bx; \neg Cx)$
$\neg B\boldsymbol{y}$
$\{\boldsymbol{x} \mapsto x_1, \boldsymbol{y} \mapsto x_1\}$
×
$\neg([\boldsymbol{y}/x]; \neg Cx)$
$C\boldsymbol{y}$
$\{\boldsymbol{x} \mapsto x_1, \boldsymbol{y} \mapsto x_1\}$
×

Figure 7.1: DFOL($\sigma, \cup$) Tableau for Syllogistic Reasoning (7.1).

Computation of Answer Substitutions. The following example illustrates how the tableau calculus can be used to compute answer substitutions for a query.


$x < 3; x \doteq 5 \cup x \doteq 2$
$x < 3$
$x \doteq 5 \cup x \doteq 2$
$x \doteq 5$
$x \doteq 2$
$[5/x]$
$[2/x]$

A combination with model checking or term rewriting (see [DHK98]) can be used to get rid of the left branch. Adding the relevant axioms for $<$ would achieve the same. See the next example.

More Reasoning about $<$. Assume that $1, 2, 3, \ldots$ are shorthand for $s0, ss0, \ldots$. We derive a contradiction from the assumption that $5 < 2$ together with two axioms for $<$. See Figure 7.2, with arrows connecting the literals that effect closure.

Computation of Answer Substitutions, with Variable Reuse. Figure 7.3 demonstrates how the computed answer substitution stores the final value for x, under the renaming x_1. Because of the renaming, the database information for x_1 does not conflict with that for x.

Closure by Equality Replacement. This example illustrates closure by means of equality replacement, in reasoning about $\exists x; \exists y; x \neq y; \exists x; \neg(\exists y; x \neq y)$. Note that x_1, y_1, x_2 serve as names for objects in the domain under construction. What the argument boils down to is: if the name x_2 applies to everything, then it cannot be the case that there are two different objects x_1, y_1. See Figure 7.4.

The first application of equality replacement in Figure 7.4 unifies $\boldsymbol{x}$ with x_1 and concludes from $x_2 \doteq \boldsymbol{x}, x_1 \neq y_1$ that $x_2 \neq y_1$. The second application of equality replacement unifies $\boldsymbol{y}$ with y_1 and concludes from $x_2 \doteq \boldsymbol{y}, x_2 \neq y_1$ that $x_2 \neq x_2$.

Loop Invariant Checking. To check that $x = y!$ is a loop invariant for $y := y + 1; x := x * y$, assume it is not, and use the calculus to derive a contradiction with the definition of !. Note that $y := y + 1; x := x * y$ appears in our notation as $[y + 1/y]; [x * y/x]$. See Figure 7.5. A more detailed account would of course have to use the DFOL definitions of $+$, $*$ and !.

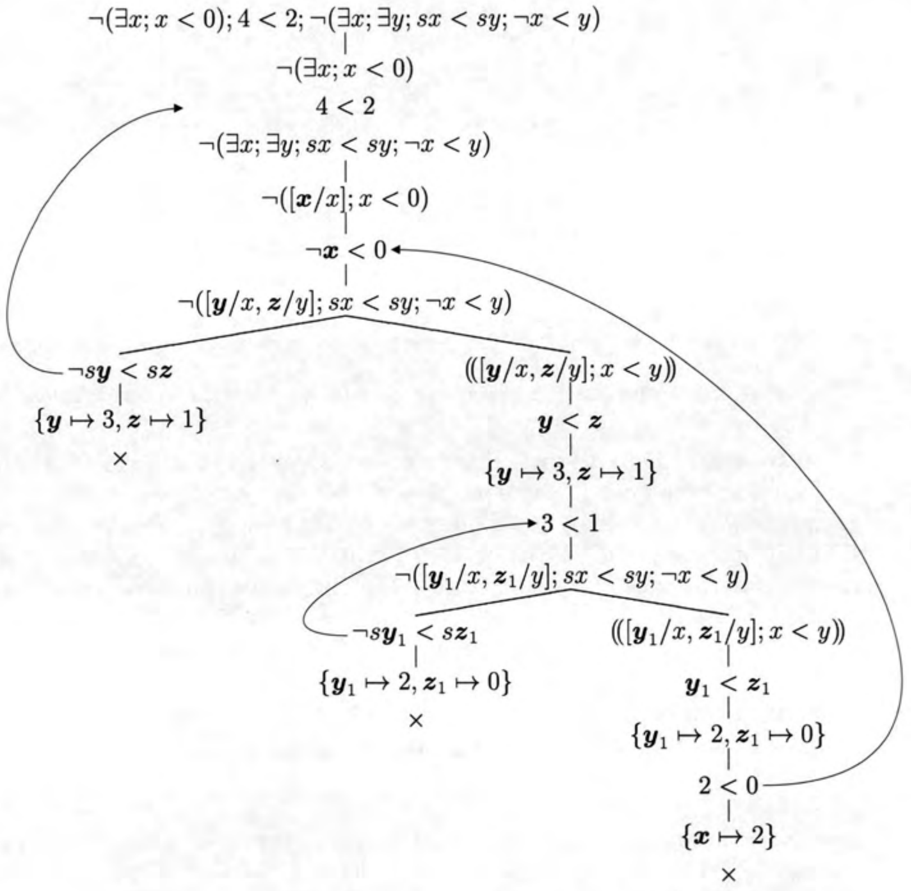

$\neg(\exists x; x < 0); 4 < 2; \neg(\exists x; \exists y; sx < sy; \neg x < y)$
$\neg(\exists x; x < 0)$
$4 < 2$
$\neg(\exists x; \exists y; sx < sy; \neg x < y)$
$\neg([\boldsymbol{x}/x]; x < 0)$
$\neg\boldsymbol{x} < 0$
$\neg([\boldsymbol{y}/x, \boldsymbol{z}/y]; sx < sy; \neg x < y)$
$\neg s\boldsymbol{y} < s\boldsymbol{z}$
$\{\boldsymbol{y} \mapsto 3, \boldsymbol{z} \mapsto 1\}$
$\times$
$(([\boldsymbol{y}/x, \boldsymbol{z}/y]; x < y))$
$\boldsymbol{y} < \boldsymbol{z}$
$\{\boldsymbol{y} \mapsto 3, \boldsymbol{z} \mapsto 1\}$
$3 < 1$
$\neg([\boldsymbol{y}_1/x, \boldsymbol{z}_1/y]; sx < sy; \neg x < y)$
$\neg s\boldsymbol{y}_1 < s\boldsymbol{z}_1$
$\{\boldsymbol{y}_1 \mapsto 2, \boldsymbol{z}_1 \mapsto 0\}$
$\times$
$(([\boldsymbol{y}_1/x, \boldsymbol{z}_1/y]; x < y))$
$\boldsymbol{y}_1 < \boldsymbol{z}_1$
$\{\boldsymbol{y}_1 \mapsto 2, \boldsymbol{z}_1 \mapsto 0\}$
$2 < 0$
$\{\boldsymbol{x} \mapsto 2\}$
$\times$

Figure 7.2: More Reasoning about $<$.

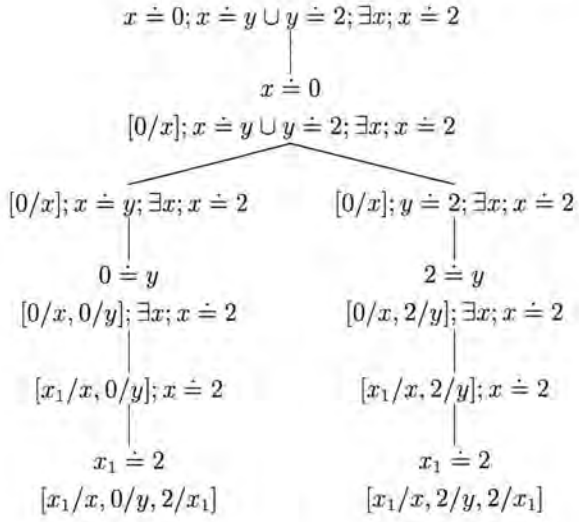

$x \doteq 0; x \doteq y \cup y \doteq 2; \exists x; x \doteq 2$
$x \doteq 0$
$[0/x]; x \doteq y \cup y \doteq 2; \exists x; x \doteq 2$
$[0/x]; x \doteq y; \exists x; x \doteq 2$
$[0/x]; y \doteq 2; \exists x; x \doteq 2$
$0 \doteq y$
$2 \doteq y$
$[0/x, 0/y]; \exists x; x \doteq 2$
$[0/x, 2/y]; \exists x; x \doteq 2$
$[x_1/x, 0/y]; x \doteq 2$
$[x_1/x, 2/y]; x \doteq 2$
$x_1 \doteq 2$
$x_1 \doteq 2$
$[x_1/x, 0/y, 2/x_1]$
$[x_1/x, 2/y, 2/x_1]$

Figure 7.3: Computation of Answer Substitutions, with Variable Reuse

Loop Invariant Detection. This time, we inspect the code $[x*(y+1)/x]; [y+1/y]$ starting from scratch. Since y is the variable that gets incremented, we may assume that x depends on y via an unknown function f. Thus, we start in a situation where $fy = x$. We check what has happened to this dependency after execution of the code $[x*(y+1)/x]; [y+1/y]$, by means of a tableau calculation for $fy \doteq x; [x*(y+1)/x]; [y+1/y]; fy \doteq x$. See Figure 7.6. The tableau shows that $[x*(y+1)/x]; [y+1/y]$ is a loop for the factorial function.

Postcondition Reasoning for 'If Then Else'. For another example of this, consider a loop through the following programming code:

$$i := i + 1; \text{if } x < a[i] \text{ then } x := a[i] \text{ else skip.} \qquad (7.3)$$

Assume we know that before the loop x is the maximum of array elements $a[0]$ through $a[i]$. Then our calculus allows us to derive a characterization of the value of x at the end of the loop. Note that the loop code appears in DFOL$(\sigma, \cup)$ under the following guise:

$$[i+1/i]; (x < a[i]; [a[i]/x] \cup \neg x < a[i]).$$

The situation of x at the start of the loop can be given by an identity $x = m_i^0$, where m is a two-placed function. To get a characterization of x at the end, we just put $X = x$ (X a constant) at the end, and see what we get (Figure 7.7). What the leaf nodes tell us is that in any case, X is the maximum of $a[0], .., a[i+1]$, and this maximum gets computed in x.

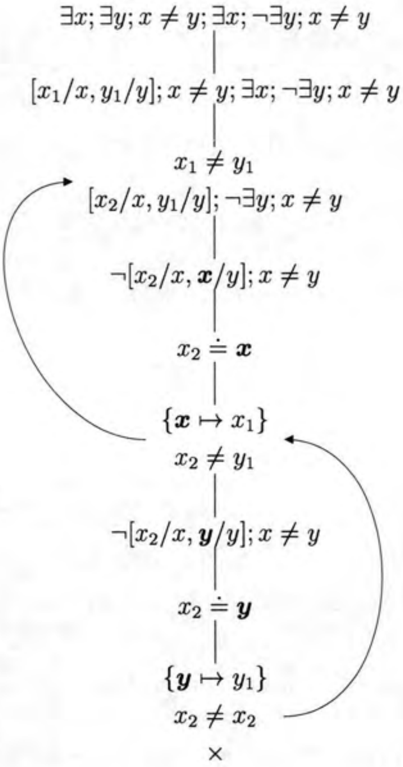

$\exists x; \exists y; x \neq y; \exists x; \neg\exists y; x \neq y$
$[x_1/x, y_1/y]; x \neq y; \exists x; \neg\exists y; x \neq y$
$x_1 \neq y_1$
$[x_2/x, y_1/y]; \neg\exists y; x \neq y$
$\neg[x_2/x, \boldsymbol{x}/y]; x \neq y$
$x_2 \doteq \boldsymbol{x}$
$\{\boldsymbol{x} \mapsto x_1\}$
$x_2 \neq y_1$
$\neg[x_2/x, \boldsymbol{y}/y]; x \neq y$
$x_2 \doteq \boldsymbol{y}$
$\{\boldsymbol{y} \mapsto y_1\}$
$x_2 \neq x_2$
$\times$

Figure 7.4: Reasoning With Equality

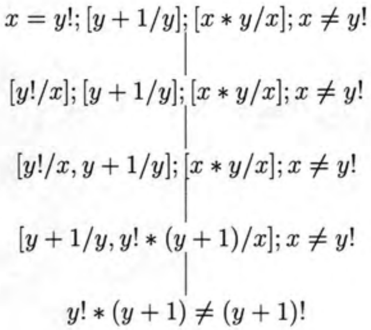

$x = y!; [y+1/y]; [x * y/x]; x \neq y!$
$[y!/x]; [y+1/y]; [x * y/x]; x \neq y!$
$[y!/x, y+1/y]; [x * y/x]; x \neq y!$
$[y+1/y, y! * (y+1)/x]; x \neq y!$
$y! * (y+1) \neq (y+1)!$

Figure 7.5: Loop Invariant Checking.

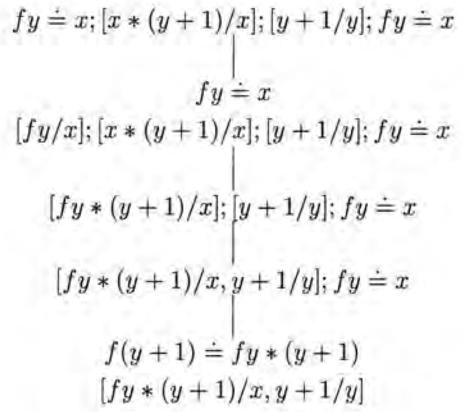

$fy \doteq x; [x * (y+1)/x]; [y+1/y]; fy \doteq x$
$fy \doteq x$
$[fy/x]; [x * (y+1)/x]; [y+1/y]; fy \doteq x$
$[fy * (y+1)/x]; [y+1/y]; fy \doteq x$
$[fy * (y+1)/x, y+1/y]; fy \doteq x$
$f(y+1) \doteq fy * (y+1)$
$[fy * (y+1)/x, y+1/y]$

Figure 7.6: Loop Invariant Detection.

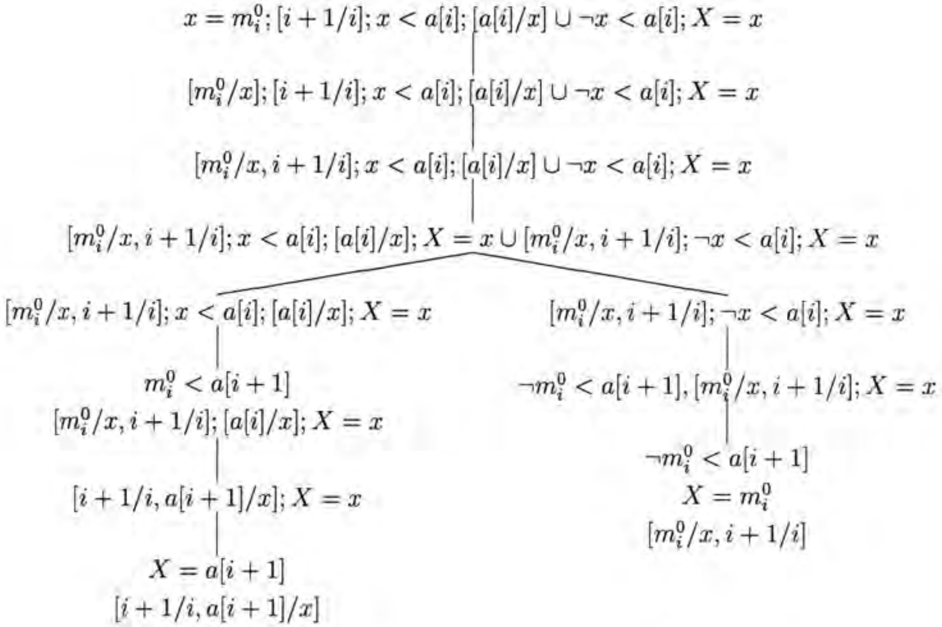

$x = m_i^0; [i+1/i]; x < a[i]; [a[i]/x] \cup \neg x < a[i]; X = x$
$[m_i^0/x]; [i+1/i]; x < a[i]; [a[i]/x] \cup \neg x < a[i]; X = x$
$[m_i^0/x, i+1/i]; x < a[i]; [a[i]/x] \cup \neg x < a[i]; X = x$
$[m_i^0/x, i+1/i]; x < a[i]; [a[i]/x]; X = x \cup [m_i^0/x, i+1/i]; \neg x < a[i]; X = x$
$[m_i^0/x, i+1/i]; x < a[i]; [a[i]/x]; X = x$
$[m_i^0/x, i+1/i]; \neg x < a[i]; X = x$
$m_i^0 < a[i+1]$
$[m_i^0/x, i+1/i]; [a[i]/x]; X = x$
$\neg m_i^0 < a[i+1], [m_i^0/x, i+1/i]; X = x$
$[i+1/i, a[i+1]/x]; X = x$
$\neg m_i^0 < a[i+1]$
$X = m_i^0$
$[m_i^0/x, i+1/i]$
$X = a[i+1]$
$[i+1/i, a[i+1]/x]$

Figure 7.7: Postcondition Reasoning For (7.3).

7.6 Completeness

Completeness for this calculus can be proved by a variation on completeness proofs for tableau calculi in classical FOL. First we define *trace sets* for DFOL($\sigma, \cup$) as an analogue to Hintikka sets for FOL. A trace set is a set of DFOL($\sigma, \cup$) formulas satisfying the closure conditions that can be read off from the tableau rules. Trace sets can be viewed as blow-by-blow accounts of particular consistent DFOL($\sigma, \cup$) computation paths (i.e., paths that do not close).

7.6.1. DEFINITION. A set Ψ of $\mathcal{L}_{\Sigma^*}$ formulas is a **trace set** if the following hold:

1. $\neg(\theta) \notin \Psi$.
2. If $\phi \in \Psi$, then $\overline{\phi} \notin \Psi$.
3. If $\theta; \phi \in \Psi$, then $\theta\phi \in \Psi$.
4. If $\alpha \in \Psi$ then all $\alpha_i \in \Psi$.
5. If $\beta \in \Psi$ then at least one $\beta_i \in \Psi$.
6. If $\gamma(v) \in \Psi$, then $\gamma_1(t) \in \Psi$ for all $t \in T_{\Sigma^*}^{\mathsf{VAR}}$ (all terms that do not contain variables from $\boldsymbol{X}$).
7. If $\delta(v) \in \Psi$, then $\delta_1(t) \in \Psi$ for some $t \in T_{\Sigma^*}^{\mathsf{VAR}}$ (some term t that does not contain variables from $\boldsymbol{X}$).

This definition is motivated by the Trace Lemma:

7.6.2. LEMMA (TRACE LEMMA). *The elements of every trace set Ψ are simultaneously satisfiable.*

Proof. Define a canonical model $\mathcal{M}_0$ in the standard fashion, using congruence closure on the trace set Ψ over the set of terms occurring in Φ, to get a suitable congruence $\equiv$ on terms. Next, define a canonical valuation s_0 by means of $s_0(v) := [v]_{\equiv}$ for members of VAR and $s_0(\mathrm{sk}_i^0) = [\mathrm{sk}_i^0]_{\equiv}$ for 0-ary skolem terms. Verify that s_o satisfies every member of Φ in $\mathcal{M}_0$. $\dashv$

To employ the lemma, we need the standard notion of a fair computation rule. A computation rule is a function F that for any set of formulas Φ and any tableau $\boldsymbol{T}$, computes the next rule to be applied on $\boldsymbol{T}$. This defines a partial order on the set of tableaux for Φ, with the successor of $\boldsymbol{T}$ given by F. Then there is a (possibly infinite) sequence of tableaux for Φ starting from the initial tableau, and with supremum $\boldsymbol{T}_\infty$. A computation rule F is fair if the following holds for all branches B in $\boldsymbol{T}_\infty$:

1. All formulas of type α, β, δ occurring on B or in Φ were used to expand B,

2. All formulas of type γ occurring on B or in Φ were used infinitely often to expand B.

7.6.3. Theorem (Completeness). *For all $\phi, \psi \in \mathcal{L}_\Sigma$: if $\phi \models \psi$ then there is a tableau refutation of $\phi; \neg(\psi)$.*

Proof. Let $\boldsymbol{T}_0, \ldots$ be a sequence of tableaux for $\phi; \neg(\psi)$ constructed with a fair computation rule, without closure rule applications, and with supremum $\boldsymbol{T}_\infty$. Define a freezing map $\boldsymbol{\sigma}_\infty$ on $\boldsymbol{T}_\infty$ in the standard fashion (see, e.g., [Häh01]). In particular, let $(B_k)_{k\geq 0}$ be an enumeration of the branches of $\boldsymbol{T}_\infty$, let $(\phi_i)_{i\geq 0}$ be an enumeration of the type γ formulas of $\boldsymbol{T}_\infty$, and let $\boldsymbol{x}_{ijk}$ be the variable introduced for the j-th application of γ formula ϕ_i along branch B_k. If $(t_j)_{j\geq 0}$ is an enumeration of all the frozen terms of $\boldsymbol{T}_\infty$, we can set $\boldsymbol{\sigma}_\infty(\boldsymbol{x}_{ijk}) := t_j$ for all $i, j, k \geq 0$. Note that $\boldsymbol{\sigma}_\infty$ is not, strictly speaking, a substitution since $dom(\boldsymbol{\sigma}_\infty)$ is not finite.

Suppose $\boldsymbol{\sigma}_\infty\boldsymbol{T}_\infty$ contains an open branch. Then from this branch we would get a trace set, which in turn would give a canonical model and a canonical valuation for $\phi; \neg(\psi)$, and contradiction with the assumption that $\phi \models \psi$. Therefore, $\boldsymbol{\sigma}_\infty\boldsymbol{T}_\infty$ must be closed.

Since the tree $\boldsymbol{T}_\infty$ is finitely branching and all formulas having an effect on closure are at finite distance from the root, there is a finite $\boldsymbol{T}_n$ with $\boldsymbol{\sigma}_\infty\boldsymbol{T}_n$ closed. Finally, construct an MGU $\boldsymbol{\sigma}$ for $\boldsymbol{T}_n$ on the basis of the part of $\boldsymbol{\sigma}_\infty$ that is actually used in the closure of $\boldsymbol{T}_n$, and we are done. $\dashv$

7.6.4. Theorem (Computation Theorem). *If ϕ is satisfiable, then all bindings θ produced by open tableau branches B satisfy ${}_s[\phi]^{\mathcal{M}}_{s\theta}$, where $\mathcal{M}$ is the canonical model constructed from B, and s the canonical valuation.*

Proof. Let $\boldsymbol{T}_0, \ldots$ be a sequence of tableaux for ϕ constructed with a fair computation rule, without closure rule applications, and with supremum $\boldsymbol{T}_\infty$. Consider $\boldsymbol{\sigma}_\infty\boldsymbol{T}_\infty$, where $\boldsymbol{\sigma}_\infty$ is the canonical freezing substitution. Then since ϕ is satisfiable, $\boldsymbol{\sigma}_\infty\boldsymbol{T}_\infty$ will have open branches $(B_k)_{k\geq 0}$ (the number need not be finite). It follows from the format of the tableau expansion rules that every open branch will develop one binding.

A binding $\theta \neq []$ occurs non-protected in a formula of the form $\theta; \psi$. Check that the tableau expansion rules on formulas of the forms $((\psi))$ or $\neg(\psi)$ never yield (nontrivial) non-protected bindings. Check that each application of an α, β, γ or δ rule to a formula with a non-protected binding extends a branch with exactly one non-protected binding. It follows that every tableau branch B_k has a highest node where a formula of the form θ appears. This θ can be thought of as the result of pulling the initial binding $[]$ through the initial formula ϕ. For every such B_k and θ there is a finite $\boldsymbol{T}_n$ with a branch $B_{k'}$ that already contains (a generalization of) θ.

It can be proved by induction on the length of $B_{k'}$ that ${}_s[\phi]^{\mathcal{M}}_{s\theta}$, for $\mathcal{M}$ the canonical model and s the canonical valuation for that branch. $\dashv$

Note that the computation theorem gives no recipe for generating all correct bindings for a given ϕ. Specifying appropriate computation rules for generating these bindings for specific sets of DFOL$(\sigma, \cup)$ formulas remains a topic for future research.

Variation: Using the Calculus with a Fixed Model. Computing with respect to a fixed model is just a slight variation on the general scheme. The technique of using tableau rules for model checking is well known. Assume that a model $\mathcal{M} = (D, I)$ is given. Then instead of storing ground predicates $P\theta\bar{t}$ (ground equalities $\theta t_1 \doteq \theta t_2$), we check the model for $\mathcal{M} \models P\theta\bar{t}$ (for $[\![\theta t_1]\!]^{\mathcal{M}} = [\![\theta t_2]\!]^{\mathcal{M}}$), and close the branch if the test fails, continue otherwise. Similarly, instead of storing ground predicates $P\theta\bar{t}$ (ground equalities $\theta t_1 \doteq \theta t_2$) under negation, we check the model for $\mathcal{M} \not\models P\theta\bar{t}$ (for $[\![\theta t_1]\!]^{\mathcal{M}} \neq [\![\theta t_2]\!]^{\mathcal{M}}$), and close the branch if the test fails, continue otherwise.

7.7 Extending the Language

7.7.1 Local variables: the Hiding operator

Consider the language of DFOL$(\sigma, \cup, \exists)$, that is, the extension of the logic we have been using with the $\exists$ operator. This extension gives a 'classical' existential quantifier to DFOL, and it is therefore quite straightforward to state tableau rules to handle it:

$$\begin{array}{c} \theta;\ \exists v(\phi_1);\ \phi_2 \\ | \\ \theta; \phi_1[\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)/v];\ \phi_2 \end{array}$$

where again $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are the universal parameters upon which interpretation of $\exists v; \phi$ depends, and $\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is a skolem constant that is new to the tableau branch. The rule for the *negated* hiding operator would be just the same as the γ-rule:

$$\begin{array}{c} \theta;\ \neg(\exists v(\phi_1));\ \phi_2 \\ | \\ \neg(\theta \circ [\boldsymbol{x}/v]; \phi_1) \\ | \\ \theta; \phi_2 \end{array}$$

where $\boldsymbol{x}$ is a universal variable taken from $\boldsymbol{X}$ that is new to the tableau. In fact, since the difference between the classical and dynamic interpretations of the existential quantifier lies in this context in the scope of the quantifier, negation brings them to the same form, and the rule for negated $\exists$ operator is the same as the γ rule. Soundness and completeness for rules involving this operator follow from soundness and completeness of the δ and γ rules.

7.7.2 Iteration: the Kleene star

Let us now add the Kleene star operator, making our language DFOL($\sigma, \cup, *$); The intended relational meaning of ϕ^* is that ϕ gets executed a finite (≥ 0) number of times. This extension is then a full-fledged programming language.

The semantic clause for ϕ^* runs as follows:

$$\begin{array}{lll} {}_s[\![\phi^*]\!]_u^{\mathcal{M}} & \text{iff} & \text{either } s = u \\ & & \text{or } \exists s_1, \ldots, s_n (n \geq 1) \text{ with } {}_s[\![\phi]\!]_{s_1}^{\mathcal{M}}, \ldots, {}_{s_n}[\![\phi]\!]_u^{\mathcal{M}}. \end{array}$$

It is easy to see that it follows from this definition that:

$${}_s[\![\phi^*]\!]_u^{\mathcal{M}} \text{ iff either } s = u \text{ or } \exists s_1 \text{ with } {}_s[\![\phi]\!]_{s_1}^{\mathcal{M}} \text{ and } {}_{s_1}[\![\phi^*]\!]_u^{\mathcal{M}}. \tag{7.4}$$

Note, however, that (7.4) is not equivalent to the definition of ${}_s[\![\phi^*]\!]_u^{\mathcal{M}}$, for (7.4) does not rule out infinite ϕ paths.

Let ϕ^n be given by: $\phi^0 := []$ and $\phi^{n+1} := \phi; \phi^n$. Now ϕ^* is equivalent to 'for some $n \in \mathbb{N} : \phi^n$'.

What we will do in our calculus for DFOL($\sigma, \cup, *$) is take (7.4) as the cue to the star rules. This will allow star computations to loop, which does not pose any problem, given that we extend our notion of closure to 'closure in the limit' (see below).

The calculus for DFOL($\sigma, \cup, *$) has all expansion rules of the DFOL($\sigma, \cup$) calculus, plus the following α^* and β^* rules.

α^* expansion rule. Call ψ^* the star formula of the rule.

$$\boxed{\begin{array}{c} \neg(\phi;\psi^*;\chi) \\ | \\ \neg(\phi;\chi) \\ \neg(\phi;\psi;\psi^*;\chi) \end{array}}$$

β^* **expansion rule.** Call ψ^* the star formula of the rule.

$$\begin{array}{c} \phi;\psi^*;\chi \\ \hline \phi;\chi \qquad\qquad \phi;\psi;\psi^*;\chi \end{array}$$

To see that the α^* rule is sound, assume that s universally satisfies $\neg(\phi;\psi^*;\chi)$ in $\mathcal{M} = (D, I)$. By (7.4), this means that there is at least one $h : \boldsymbol{X} \to D$ for which there is no u with ${}_{s\cup h}[\![\phi;\chi]\!]_u^{\mathcal{M}}$ and no u with ${}_{s\cup h}[\![\phi;\psi;\psi^*;\chi]\!]_u^{\mathcal{M}}$. Thus, s universally satisfies $\neg(\phi;\chi)$ and $\neg(\phi;\psi;\psi^*;\chi)$ in $\mathcal{M}$.

For the β^* rule, assume that s universally satisfies $\phi;\psi^*;\chi$ in $\mathcal{M}$. Then for every $h : \boldsymbol{X} \to D$ there are u, u' with ${}_{s\cup h}[\![\phi]\!]_u^{\mathcal{M}}$ and ${}_u[\![\psi^*;\chi]\!]_{u'}^{\mathcal{M}}$. Then, by (7.4), either ${}_u[\![\chi]\!]_{u'}^{\mathcal{M}}$ or there is a u_1 with ${}_u[\![\psi]\!]_{u_1}^{\mathcal{M}}$ and ${}_{u_1}[\![\phi_1^*;\chi]\!]_{u'}^{\mathcal{M}}$. Thus, s universally satisfies either $\phi;\chi$ or $\phi;\psi;\psi^*;\chi$ in $\mathcal{M}$.

Closure in the Limit. To deal with the inflationary nature of the α^* and β^* rules (the star formula of the rule reappears at a leaf node), we need a modification of our notion of tableau closure. We allow closure in the limit, as follows.

7.7.1. DEFINITION. An infinite tableau branch closes in the limit if it contains an infinite star development, i.e., an infinite number of α^* or β^* applications to the same star formula.

Example of Closure in the Limit. We will give an example of an infinite star development. Consider formula (7.5):

$$\neg\exists w \neg(\exists v; v = 0; (v \neq w; [v + 1/v])^*; v = w). \tag{7.5}$$

What (7.5) says is that there is no object w that cannot be reached in a finite number of steps from $v = 0$, or in other words that the successor relation $v \mapsto v+1$, considered as a graph, is well-founded. This is the Peano induction axiom: it characterizes the natural numbers up to isomorphism. What it says is that any set A that contains 0 and is closed under successor contains all the natural numbers. The fact that Peano induction is expressible as an $\mathcal{L}_\Sigma^*$ formula is evidence that $\mathcal{L}_\Sigma^*$ has greater expressive power than FOL. In FOL no single formula can express Peano induction: no formula can distinguish the standard model $(\mathbb{N}, s)$ from the non-standard models. In a non-standard model of the natural numbers it may take an infinite number of s-steps to get from one natural number n to a larger number m.

The expressive power of $\mathcal{L}_\Sigma{}^*$ is the same as that of quantified dynamic logic [Pra76, Gol92]. Arithmetical truth is undecidable, so there can be no finitary refutation

$$\begin{array}{c}
\exists w \neg(\exists v; v \doteq 0; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
[w_1/w]\neg(\exists v; v \doteq 0; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 0/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 0/v]; v \doteq w) \\
\neg([w_1/w, 0/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
0 \neq w_1 \\
| \\
\neg([w_1/w, 1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 1/v]; v \doteq w) \\
\neg([w_1/w, 1/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
1 \neq w_1 \\
| \\
\neg([w_1/w, 2/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 2/v]; v \doteq w) \\
\neg([w_1/w, 2/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
2 \neq w_1 \\
| \\
\neg([w_1/w, 3/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 3/v]; v \doteq w) \\
\neg([w_1/w, 3/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
3 \neq w_1 \\
| \\
\neg([w_1/w, 4/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 4/v]; v \doteq w) \\
\neg([w_1/w, 4/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
4 \neq w_1 \\
| \\
\neg([w_1/w, 5/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\vdots \\
\times
\end{array}$$

Figure 7.8: 'Infinite Proof' of the Peano Induction Axiom.

system for $\mathcal{L}^*_\Sigma$. The finitary tableau system for $\mathcal{L}_\Sigma$ is evidence for the fact that DFOL($\sigma, \cup$) validity is recursively enumerable: all non-validities are detected by a finite tableau refutation. This property is lost in the case of $\mathcal{L}^*_\Sigma$: the language is just too expressive to admit of finitary tableau refutations.

Therefore, some tableau refutations must be infinitary, and the tableau development for the negation of (7.5) is a case in point. Let us see what happens if we attempt to refute the negation of (7.5). A successful refutation will identify the natural numbers up to isomorphism. See Figure 7.8. This is indeed a successful refutation, for the tree closes in the limit. But the refutation tree is infinite: it takes an infinite amount of time to do all the checks.

7.7.2. THEOREM (SOUNDNESS THEOREM FOR $\mathcal{L}^*_\Sigma$). *The calculus for DFOL($\sigma, \cup, *$) is sound:*

For all $\phi, \psi \in \mathcal{L}^*_\Sigma$: if the tableau for $\phi; \neg(\psi)$ closes then $\phi \models \psi$.

The modified tableau method does not always give finite refutations. Still, it is a very useful reasoning tool, more powerful than Hoare reasoning, and more practical than the infinitary calculus for quantified dynamic logic developed in [Gol82, Gol92]. Dynamic logic itself has been put to practical use, e.g. in KIV, a system for interactive software verification [Rei95]. It is our hope that the present calculus can be used to further automate the software verification process.

Precondition/postcondition Reasoning. For a further example of reasoning with the calculus, consider formula (7.6). This gives an $\mathcal{L}^*_\Sigma$ version of Euclid's GCD algorithm.

$$(x \neq y; (x > y; [x - y/x] \cup y > x; [y - x/y]))^*; x \doteq y. \tag{7.6}$$

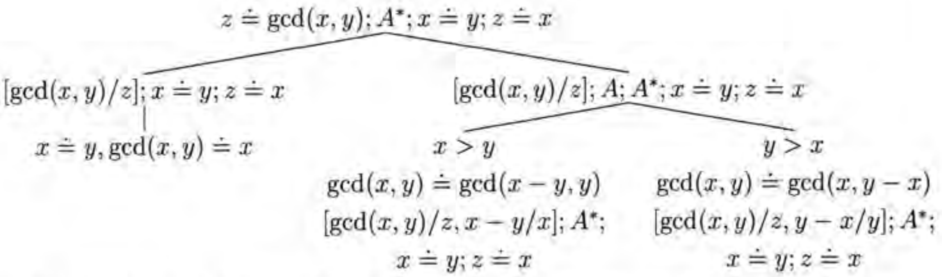
To do automated precondition-postcondition reasoning on this, we must find a trivial correctness statement. Even if we don't know what $\gcd(x, y)$ is, we know that its value should not change during the program. So putting $\gcd(x, y)$ equal to some arbitrary value and see what happens would seem to be a good start. We will use the correctness statement $z \doteq \gcd(x, y)$. The statement that the result gets computed in x can then take the form $z \doteq x$. The program with these trivial correctness statements included becomes:

$$\begin{aligned}&z \doteq \gcd(x, y);\\ &(x \neq y; (x > y; [x - y/x]; z \doteq \gcd(x, y) \cup y > x; [y - x/y]; z \doteq \gcd(x, y)))^*;\\ &x \doteq y; z \doteq x.\end{aligned} \tag{7.7}$$

We can now put the calculus to work. Abbreviating

$$(x \neq y; (x > y; [x - y/x]; z \doteq \gcd(x, y) \cup y > x; [y - x/y]; z \doteq \gcd(x, y)))^*$$

as A^*, we get:


$z \doteq \mathrm{gcd}(x,y); A^*; x \doteq y; z \doteq x$
$[\mathrm{gcd}(x,y)/z]; x \doteq y; z \doteq x$
$x \doteq y, \mathrm{gcd}(x,y) \doteq x$
$[\mathrm{gcd}(x,y)/z]; A; A^*; x \doteq y; z \doteq x$
$x > y$
$\mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x-y,y)$
$[\mathrm{gcd}(x,y)/z, x-y/x]; A^*;$
$x \doteq y; z \doteq x$
$y > x$
$\mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x,y-x)$
$[\mathrm{gcd}(x,y)/z, y-x/y]; A^*;$
$x \doteq y; z \doteq x$

The second split is caused by an application of the rule for $\cup$. By the soundness of the calculus any model satisfying the annotated program (7.7) will satisfy one of the branches. This shows that if the program succeeds (computes an answer), the following disjunction will be true:

$$\begin{aligned}&(x \doteq y \wedge \mathrm{gcd}(x,y) \doteq x)\\ \vee\ &(x > y \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x-y,y) \wedge \phi)\\ \vee\ &(y > x \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x,y-x) \wedge \psi)\end{aligned} \tag{7.8}$$

Here ϕ abbreviates $[\mathrm{gcd}(x,y)/z, x-y/x]; A^*; x \doteq y; z \doteq x$ and ψ abbreviates $[\mathrm{gcd}(x,y)/z, y-x/y]; A^*; x \doteq y; z \doteq x$. From this it follows that the following weaker disjunction is also true:

$$\begin{aligned}&(x \doteq y \wedge \mathrm{gcd}(x,y) \doteq x)\\ \vee\ &(x > y \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x-y,y))\\ \vee\ &(y > x \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x,y-x))\end{aligned} \tag{7.9}$$

Note that (7.9) looks remarkably like a functional program for GCD.

7.8 Completeness for DFOL$(\sigma, \cup, *)$

The method of trace sets for proving completeness from Section 7.6 still applies. Trace sets for DFOL$(\sigma, \cup, *)$ will have to satisfy the obvious extra conditions. In order to preserve the correspondence between trace sets and open tableau branches, we must adapt the definition of a fair computation rule. A computation rule F for $\mathcal{L}^*_\Sigma$ is fair if it is fair for $\mathcal{L}_\Sigma$, and in addition, the following holds for all branches B in $\boldsymbol{T}_\infty$:

- All formulas of type α^*, β^* occurring on B or in Φ were used to expand B.

We can again prove a trace lemma for DFOL$(\sigma, \cup, *)$, in the same manner as before: Again, open branches in the supremum of a fair tableau sequence will correspond to trace sets, and we can satisfy these trace sets in canonical models. The definition of trace sets is extended as follows:

7.8.1. Definition. A set Ψ of $\mathcal{L}^{*}_{\Sigma}$ formulas is a **$*$-trace set** if the following hold:

- Ψ is a trace set,
- If $\beta^{*} \in \Psi$ then at least one $\beta^{*}_{i} \in \Psi$.
- If $\phi; \psi^{*}; \chi \in \Psi$, then there is some $n \geq 0$ with $\phi; \psi^{m}; \chi \notin \Psi$ for all $m > n$. Similarly for $((\phi; \psi^{*}; \chi))$.
- For all ϕ, ψ, χ it holds that $\neg(\phi; \psi^{*}; \chi) \notin \Psi$.

Note that the final two requirements are met thanks to our stipulation about closure in the limit. In the same manner as before, we get:

7.8.2. Theorem (Completeness for $\mathcal{L}^{*}$). *For all $\phi, \psi \in \mathcal{L}^{*}$: if $\phi \models \psi$ then the tableau for $\phi; \neg(\psi)$ closes.*

So we have a complete logic for DFOL($\sigma, \cup, *$), but of course it comes at a price: we may occasionally get in a refutation loop. However, as our tableau construction examples illustrate, this hardly affects the usefulness of the calculus.

7.9 Related Work

Comparison with tableau reasoning for (fragments of) FOL. The present calculus for DFOL can be viewed as a more dynamic version of tableau style reasoning for FOL and for modal fragments of FOL. Instead of just checking for valid consequence and constructing counterexamples from open tableau branches, our open tableau branches yield computed answer bindings as an extra. The connection with tableau reasoning for FOL is also evident in the proof method of our completeness theorems. Our calculus can be used for FOL reasoning via the following translation of FOL into DFOL:

$$\begin{array}{rcl}
(P\bar{t})^{\bullet} & := & P\bar{t} \\
(\neg\phi)^{\bullet} & := & \neg\phi^{\bullet} \\
(\phi \wedge \psi)^{\bullet} & := & \phi^{\bullet}; \psi^{\bullet} \\
(\phi \vee \psi)^{\bullet} & := & \phi^{\bullet} \cup \psi^{\bullet} \\
(\exists x\phi)^{\bullet} & := & ((\exists x; \phi^{\bullet})) \\
(\forall x\phi)^{\bullet} & := & \neg(\exists x; \neg\phi^{\bullet})
\end{array}$$

It is easy to check that for every FOL formula ϕ it holds that $\phi^{\bullet} = ((\phi^{\bullet}))$, i.e., all FOL translations are DFOL tests. Moreover, the translation is adequate in the sense that for every FOL formula ϕ over signature Σ, every Σ-model $\mathcal{M}$, every valuation s for $\mathcal{M}$ it holds that $\mathcal{M} \models_{s} \phi$ iff ${}_{s}[\phi^{\bullet}]^{\mathcal{M}}_{s}$.

Connection with Logic Programming. The close connection between tableau reasoning for DFOL and Logic Programming can be seen by developing a DFOL tableau for the following formula set:

$$\forall x A([], x, x), \forall x \forall y \forall z \forall i (A(x, y, z) \rightarrow A([i|x], y, [i|z])), \neg \exists x A([a|[b|[]], [c|[]], x).$$

This will give a tableau for the append relation, with a MGU substitution $\{\boldsymbol{x} \mapsto [a|[b|c|[]]]\}$ that closes the tableau, where $\boldsymbol{x}$ is the universal tableau variable used in the application of the γ rule to $\neg \exists x A([a|[b|[]], [c|[]], x)$. The example may serve as a hint to the unifying perspective on logic programming and imperative programming provided by tableau reasoning for DFOL. We hope to elaborate this theme in future work.

Comparison with other Calculi for DFOL and for DRT. The calculus developed in [vE99a] uses swap rules for moving quantifiers to the front of formulas. The key idea of the present calculus is entirely different: encode dynamic binding in explicit bindings and protect outside environments from dynamic side effects by means of block operations. In a sense, the present calculus offers a full account of the phenomenon of local variable use in DFOL.

Kohlhase [Koh00] gives a tableau calculus for DRT (Discourse Representation Theory, see [Kam81]) that has essentially the same scope as the [vE99a] calculus for DPL: the version of DRT disjunction that is treated is externally static, and the DRT analogue of ∪ is not treated.

Kohlhase's calculus follows an old DRT tradition in relying on an implicit translation to standard FOL: see [SE88] for an earlier example of this. Kohlhase motivates his calculus with the need for (minimal) model generation in dynamic NL semantics. In order to make his calculus generate minimal models, he replaces the rule for existential quantification by a 'scratchpaper' version (well-known from textbook treatments of tableau reasoning; see [Hin88] for further background, and for discussion of non-monotonic consequence based on minimal models generated with this rule). First try out if you can avoid closure with a term already available at the node. If all these attempts result in closure, it does not follow from this that the information at the node is inconsistent, for it may just be that we have 'overburdened' the available terms with demands. So in this case, and only in this case, introduce a new individual.

This 'exhaustion of existing terms' approach has the virtue that it generates 'small' models when they exist, whereas the more general procedure 'always introduce a fresh variable and postpone instantiation' may generate infinite models where finite models exist. Note, however, that the strategy only makes sense for a signature without function symbols, and for a tableau calculus without free tableau variables.

Kohlhase discusses applications in NL processing, where it often makes sense to construct a minimal model for a text, and where the assumption of minimality can be used to facilitate issues of anaphora resolution and presupposition handling.

Comparison with Apt and Bezem's Executable FOL. Apt and Bezem present what can be viewed as an exciting new mix of tableau style reasoning and model checking for FOL. Our treatment of equality uses a generalization of a stratagem from their [AB98]: in the context of a partial variable map θ, they call $v \doteq t$ a θ *assignment* if $v \notin dom(\theta)$, and all variables occurring in t are in $dom(\theta)$. We generalize this on two counts:

- Because our computation results are bindings (term maps) rather than maps to objects in the domain of some model, we allow computation of non-ground terms as values.
- Because our bindings are total, in our calculus execution of $t_1 \doteq t_2$ atoms never gives rise to an error condition.

It should be noted for the record that the first of these points is addressed in [Apt00]. Apt and Bezem present their work as an underpinning for Alma-0, a language that infuses Modula style imperative programming with features from logic programming (see [ABPS98]). In a similar way, the present calculus provides logical underpinnings for Dynamo, a language for programming with an extension of DFOL. For a detailed comparison of Alma-0 and Dynamo we refer the reader to [vE98b].

Connection with WHILE, GCL. It is easy to give an explicit binding semantics for WHILE, the favorite toy language of imperative programming from the textbooks (see e.g., [NN92]), or for GCL, the non-deterministic variation on this proposed by Dijkstra (see, e.g. [DS94]). DFOL is in fact quite closely related to these, and it is not hard to see that DFOL$(\sigma, \cup, *)$ has the same expressive power as GCL. Our tableau calculus for DFOL$(\sigma, \cup, *)$ can therefore be regarded as an execution engine *cum* reasoning engine for WHILE or GCL.

Connection with PDL, QDL. We can see that there is also a close connection between DFOL$(\sigma, \cup, *)$ on one hand and propositional dynamic logic (PDL) and quantified dynamic logic (QDL) on the other. QDL is a language proposed in [Pra76] to analyze imperative programming, and PDL is its propositional version. See [Seg82, Par78] for complete axiomatizations of PDL, [Gol92] for an exposition of both PDL and QDL, and for a complete (but infinitary) axiomatization of QDL, [HKT84] for an overview, and [Har79] for a a study of QDL and various extensions. In PDL/QDL, programs are treated as modalities and assertions about programs are formulas in which the programs occur as modal operators. Thus, if A is a program, $\langle A\rangle\phi$ asserts that A has a successful termination ending in a state satisfying ϕ. As is well-known, this cannot be expressed without further ado in Hoare logic.

The main difference between DFOL$(\sigma, \cup, *)$ and PDL/QDL is that in the former the distinction between formulas and programs is abolished. Everything is a program, and assertions about programs are test programs that are executed along the way, but with their dynamic effects blocked. To express that A has a successful termination ending in a ϕ state, we can just say $(\!(A; \phi)\!)$. To check whether A has a successful termination ending in a ϕ state, try to refute the statement by constructing a tableau for $\neg(A; \phi)$.

To illustrate the connection with QDL and PDL, consider MIX, the first of the two PDL axioms for $*$:

$$[A^*]\phi \to \phi \wedge [A][A^*]\phi. \tag{7.10}$$

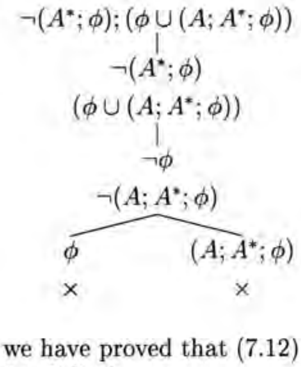
Writing this with $\langle A\rangle, \neg, \wedge, \vee$, and replacing $\neg\phi$ by ϕ, we get:

$$\neg(\neg\langle A^*\rangle\phi \wedge (\phi \vee \langle A\rangle\langle A^*\rangle\phi)). \tag{7.11}$$

This has the following DFOL$(\sigma, \cup, *)$ counterpart:

$$\neg(\neg(A^*; \phi); (\phi \cup (A; A^*; \phi))). \tag{7.12}$$

For a refutation proof of (7.12), we leave out the outermost negation.


$\neg(A^*; \phi); (\phi \cup (A; A^*; \phi))$
$\neg(A^*; \phi)$
$(\phi \cup (A; A^*; \phi))$
$\neg\phi$
$\neg(A; A^*; \phi)$
ϕ
$(A; A^*; \phi)$
×
×

The tableau closes, so we have proved that (7.12) is a DFOL$(\sigma, \cup, *)$ theorem (and thus, a DFOL$(\sigma, \cup, *)$ validity).

We will also derive the validity of the DFOL$(\sigma, \cup, *)$ counterpart to IND, the other PDL axiom for $*$:

$$(\phi \wedge [A^*](\phi \to [A]\phi)) \to [A^*]\phi. \tag{7.13}$$

Equivalently, this can be written with only $\langle A\rangle, \neg, \wedge, \vee$, as follows:

$$\neg(\phi \wedge \neg\langle A^*\rangle(\phi \wedge \langle A\rangle\neg\phi) \wedge \langle A^*\rangle\neg\phi). \tag{7.14}$$

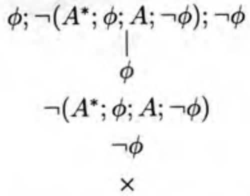
The DFOL($\sigma, \cup, *$) counterpart of (7.14) is:

$$\neg(\phi; \neg(A^*; \phi; A; \neg\phi); A^*; \neg\phi). \tag{7.15}$$

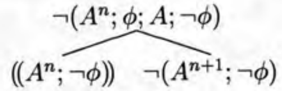
We will give a refutation proof of (7.15) in two stages. First, we show that (7.16) can be refuted for any $n \geq 0$, and next, we use this for the proof of (7.15).

$$\phi; \neg(A^*; \phi; A; \neg\phi); A^n; \neg\phi. \tag{7.16}$$

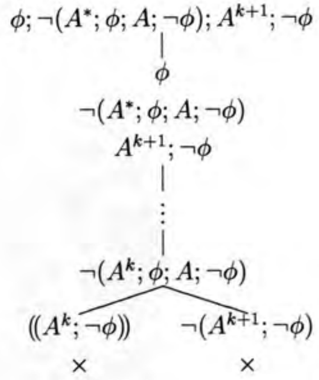
Here is the case of (7.16) with $n = 0$:


$\phi; \neg(A^*; \phi; A; \neg\phi); \neg\phi$
ϕ
$\neg(A^*; \phi; A; \neg\phi)$
$\neg\phi$
×

Bearing in mind that A is a dynamic action and ϕ is a test, we can apply the rule of Negation Splitting (Section 7.4) to formulas of the form $\neg(A^n; \phi; A; \neg\phi)$, as follows:

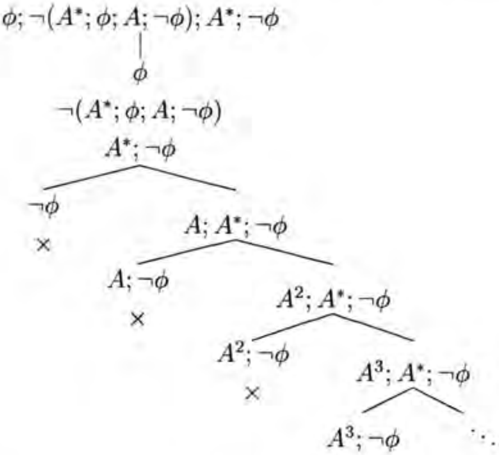


Note that $\neg(A^n; \phi; A; \neg\phi)$ can be derived from $\neg(A^*; \phi; A; \neg\phi)$ by n applications of the α^* rule. Using this, we get the following refutation tableau for the case of (7.16) with $n = k + 1$:


$\phi; \neg(A^*; \phi; A; \neg\phi); A^{k+1}; \neg\phi$
ϕ
$\neg(A^*; \phi; A; \neg\phi)$
$A^{k+1}; \neg\phi$
$\vdots$
$\neg(A^k; \phi; A; \neg\phi)$
$((A^k; \neg\phi))$
$\neg(A^{k+1}; \neg\phi)$
×
×

The left-hand branch closes because of the refutation of $\phi; \neg(A^*; \phi; A; \neg\phi); A^k; \neg\phi$, which is given by the induction hypothesis.

Next, use these refutations of $\neg\phi$, $A; \neg\phi$, $A^2; \neg\phi$, ..., to prove (7.15) by means of a refutation in the limit, as follows:

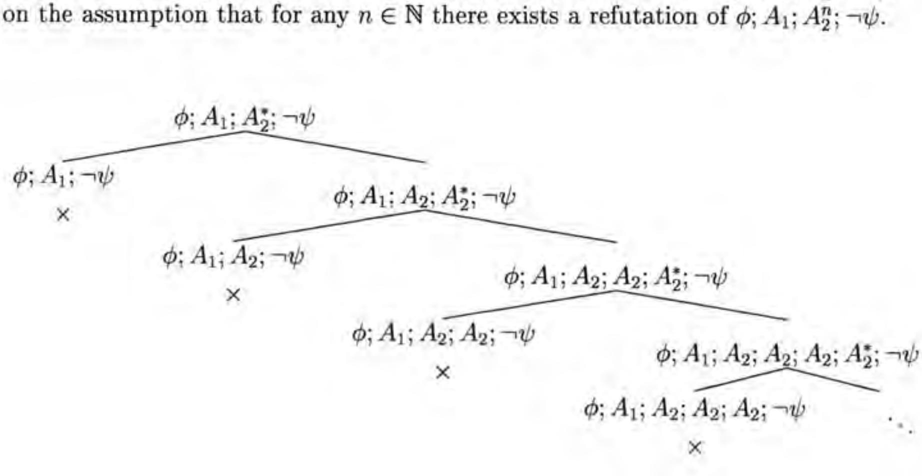


This closed tableau establishes (7.15) as a DFOL($\sigma, \cup, *$) theorem. That closure in the limit is needed to establish the DFOL($\sigma, \cup, *$) induction principle is not surprising. The DFOL *-rules express that * computes a fix-point, while the fact that this fix-point is a *least* fix-point is captured by the stipulation about closure in the limit. The induction principle (7.15) hinges on the fact that * computes a least fix-point.

Goldblatt [Gol82, Gol92] develops an infinitary proof system for QDL with the following key rule of inference:

$$\text{If } \phi \rightarrow [A_1; A_2^n]\psi \text{ is a theorem for every } n \in \mathbb{N}, \text{ then } \phi \rightarrow [A_1; A_2^*]\psi \text{ is a theorem.} \tag{7.17}$$

To see how this is related to the present calculus, assume that one attempts to refute $\phi \rightarrow [A_1; A_2^*]\psi$, or rather, its DFOL($\sigma, \cup, *$) counterpart $\neg(\phi; A_1; A_2^*; \neg\psi)$, on the assumption that for any $n \in \mathbb{N}$ there exists a refutation of $\phi; A_1; A_2^n; \neg\psi$.


$\phi; A_1; A_2^*; \neg\psi$
$\phi; A_1; \neg\psi$
×
$\phi; A_1; A_2; A_2^*; \neg\psi$
$\phi; A_1; A_2; \neg\psi$
×
$\phi; A_1; A_2; A_2; A_2^*; \neg\psi$
$\phi; A_1; A_2; A_2; \neg\psi$
×
$\phi; A_1; A_2; A_2; A_2; A_2^*; \neg\psi$
$\phi; A_1; A_2; A_2; A_2; \neg\psi$
×
$\ddots$

We can close off the $\phi; A_1; A_2^n; \neg\psi$ branches by the assumption that there exist refutations for these, for every $n \in \mathbb{N}$. The whole tableau gives an infinite β^* development, and the infinite branch closes in the limit, so the tableau closes, thus establishing that in the DFOL$(\sigma, \cup, *)$ calculus validity of $\neg(\phi; A_1; A_2^*; \neg\psi)$ follows from the fact that $\neg(\phi; A_1; A_2^n; \neg\psi)$ is valid for every $n \in \mathbb{N}$.

7.10 Conclusion

Starting out from an analysis of binding in dynamic FOL, we have given a tableau calculus for reasoning with DFOL. The format for the calculus and the role of explicit bindings for computing answers to queries were motivated by our search for logical underpinnings for programming with (extensions of) DFOL. The DFOL tableau calculus presented here constitutes the theoretical basis for *Dynamo*, a toy programming language based on DFOL. To find the answer to a query, given a formula ϕ considered as *Dynamo* program data, *Dynamo* essentially puts the tableau calculus to work on a formula ϕ, all the while checking predicates with respect to the fixed model of the natural numbers, and storing values for variables from the inspection of equality statements. If the tableau closes, this means that ϕ is inconsistent (with the information obtained from testing on the natural numbers), and *Dynamo* reports 'false'. If the tableau remains open, *Dynamo* reports that ϕ is consistent (again with the information obtained from inspecting predicates on the natural numbers), and lists the computed bindings for the output variables at the end of the open branches. But the *Dynamo* engine also works for general tableau reasoning, and for general queries. Literals collected along the open branches together with the explicit bindings at the trail ends constitute the computed answers. We report on the development of *Dynamo* in the next chapter.

Chapter 8

Implementing *Dynamo*

Before enlightenment, the mountain is a mountain.
While seeking enlightenment, the mountain is a floating mirage,
at once real and ephemeral, at once there and not there.
After enlightenment, the mountain is a mountain.

- *Zen folklore*

8.1 Introduction

We have so far presented a way of interpreting DFOL formulas as programs, a method of verifying correctness of such programs, and a tableau calculus that can be implemented as an engine for the language. What we need now is to do the implementation, so we can use the language for programming and exploring the concepts so far introduced.

Before developing the tableau calculus, we had other implementations of *Dynamo*, based on the state machines described in Chapter 5 but we were having a bit of trouble with negation; when a negated formula succeeded with 'complex states', we put the result back into state form by dualizing the result, which was basically applying De Morgan's laws to the set of states (taking into account that the set represents a disjunction and the states conjunctions), so that its negation would again be a set of states. This gave us the idea of actually using a tableau calculus to carry out the computations. Now we have a calculus that deals with these matters in a much more natural way, but knows nothing about simple arithmetic; still, we are closer now to DFOL semantics.

We report here on our efforts to bring the two capabilities together; the purpose of this implementation is to test the appropriateness and efficiency of the tableau method for implementation of a programming language.

Free Variable Tableaux

We follow the guidelines given in [Fit96] for the handling of universally quantified variables: the formulas in each branch are organized as a list, and we always process the fomula at the head of the list. When the formula is not of type γ, the formula itself is removed from the list, and is replaced by the formulas that result from applying the corresponding rule. When the γ-rule is applied, the formula can not be discarded, but is then moved to the end of the list, to enable the other formulas in the branch to be processed. The branches themselves are also organized as a list; whenever the γ-rule is applied, the algorithm leaves the branch and goes on to the next one in the list.

Closing the Tableau. In its original version, when the algorithm expands the tableau until the γ rule has been applied a predetermined number of times, the program tries to close all branches, by finding the substitution that will close all of them simultaneously. Essentially, it will sequentially, starting from the empty substitution, find all extensions of the current substitution that close the branch, and try to close the rest of the branches starting from each of the extensions. If no substitution that closes the entire tableau is found, the tableau is considered 'not solved'; either the formula is satisfiable, or the γ rule will have to be applied a higher number of times, usually starting from scratch. In the *Dynamo* implementation, the γ rule is applied at most once per branch (it might not be needed), and then closure is attempted. If closure is not reached, and at least one branch consists only of atoms, the tableau is open, otherwise the algorithm does another pass through the already expanded tableau and tries again until closure is reached.

8.2 The *Dynamo* Engine

8.2.1 The Programming Language

Dynamo is implemented in Haskell; compiles under GHC version 5.04. As before, we chose Haskell over other programming languages because of its small semantic gap between the program and its task, its being strongly typed, and the fact that it compiles into an executable program instead of requiring an interpreter.

8.2.2 The Algorithm

The way we implemented the free variable tableaux is shown in Figure 8.1.

- The function **init_branch**(form) initializes the branch data structure, with the input formula as the only element of the formula list.

```
input: form: formula;
input: query: list of variable schemes
var: Branches, New_branches: list of branches
var: current: branch

Branches := {init_branch(form)};
{* Main loop *}
while (close_branches(Branches)= False
   and all_atomic_branch(Branches) = False) do
  New_branches := $\emptyset$;
  {* Single step *}
  foreach current in Branches do
    {* Infer until univ. quant. or out of formulas *}
    New_branches := New_branches $\cup$ single_step(current);
  Branches := New_branches;

if (close_branches(Branches)= True
   then return "unsatisfiable"
   else return (extract_values(query, Branches))
```

Figure 8.1: Structure of the *Dynamo* engine

- The function **single_step**(current) applies the rule that corresponds to the type of formula at the head of the list in the branch; if the type is γ, after applying the corresponding rule the formula is copied at the end of the formula list and the function returns. Otherwise, **single_step** is applied to all the branches that result from application of the rule, until the formula list is empty or the γ-rule is processed. If the result of applying a rule is a new atom, (ground) closure is checked for. If the branch is found to close, it is removed from the result list.
- The function **close_branches** attempts to close all branches, one by one; it calls **close_branch** for each branch, and carries a list of all the freezing substitutions that close all processed branches. If a branch can not be closed by any of the existing freezing substitutions (or an extension of one), the procedure returns False.
- The function **close_branch**(current) attempts to find complementary atoms in the branch, first through the congruence closure and failing that through unification with a universally quantified variable. It accepts as parameters the list of atoms in the branch, the computed congruence closure, and a freezing substitution, and returns whether the branch can be closed or not, and the substitution that closes it if possible.
- The function **all_atomic_branch**(branches) looks for a branch made up

entirely of atoms. Since it is called after a test for closure, the presence of such a branch always indicates that the tableau will never close.

- The function **extract_values**(query, Branches) extracts the values of the required variables from the open tableau branches.

8.3 Tableau reasoning for DFOL

Here we review the main departures from a tableau prover for FOL; the data structure, which had to take into account the handling of equality, and the implementation of the rules.

8.3.1 Data Structures

The main data structure in this implementation is the *branch*. A branch is a tuple consisting of a list of formulas, a list of universally quantified variables present in the branch, a congruence, and a list of atoms. The list of formulas contains the formulas to be processed, the list of universally quantified variables holds the variables that would serve as arguments for skolem functions, the congruence keeps track of equalities between terms, and the list of atoms carries the list of atomic facts that is searched for complementary assertions. The program state is a tuple containing the list of branches, and the indexes of the last universal variable and the last skolem function instantiated. The program state is reached through a state monad, and is therefore transparent to most functions in the program.

8.3.2 Rules

Rule Extensions. Many of the tableau rules of Chapter 7 included both leading substitutions and trailing formulas, which were actually optional and could be replaced by a tautology; that had to be made more explicit in the program, which multiplied the number of rule instances. For example, one of the cases of the β-rule is:

$$\begin{tikzcd} & \theta;\ (\phi_1 \cup \phi_2);\ \phi_3 \arrow[dl, no head] \arrow[dr, no head] & \\ \theta;\phi_1;\phi_3 & & \theta;\phi_2;\phi_3 \end{tikzcd}$$

In this case, both θ and ϕ_3 are optional; only $(\phi_1 \cup \phi_2)$ is required to be non-empty, so the rule has to fire on $(\theta; (\phi_1 \cup \phi_2); \phi_3)$, $(\theta; (\phi_1 \cup \phi_2))$, $((\phi_1 \cup \phi_2); \phi_3)$ and $(\phi_1 \cup \phi_2)$. We solved the problem in part by placing the empty substitution at the start of the input formula, since the rules themselves ensure all resulting formulas will start with a substitution, but the trailing formulas still force us to duplicate the rules. The reason for the trailing formulas is precisely that the rules push a substitution through the whole: order is not important in FOL, but it matters a lot in DFOL.

Blocking. Block formulas, given their nature as assertions, were given special treatment: when a block formula is found, a new tableau is created for it, with the same data as the current branch but only the blocked formula in the formula list, and evaluated. If it results in many branches, the rule creates branches incorporating the list of atoms of each new branch to copies of the current branch. In this way we block the dynamic effects that would result from existential quantification inside the blocked formula, but keep any atomic checks that were not grounded at the time of evaluation.

Universal Quantification. A problem with universal quantification is that it represents a 'standing order': unlike the other rules, the γ-rule does not consume the formula it processes, and it can be processed again. If a limit to the number of times the γ-rule can be applied were known, the logic would be decidable. The engine will run forever on satisfiable problems in which all branches have a γ-type formula; we are looking for ways in which it can be detected that a new application of the γ rule will be redundant. We also need to ensure *fairness*, as stated in Section 7.6; ensuring that for all branches B of the fully expanded (possibly infinite) tableau T_{inf}, all the α-, β-, and δ-type formulas present either in B or in the original formula Φ are used to expand B, and that all γ-type formulas present in ϕ or B are used to expand B infinitely often. Our computation rule makes sure this happens: formulas are kept in a list, α-, β-, and δ-type formulas are always discarded and replaced by the resulting formula(s), while γ-type formulas are put at the end of the list while the formulas resulting from applying the rule are still placed in the head of the list.

8.4 Extensions to the Calculus

After implementing the free variable tableau, there was still something to be done: we want the engine to do some computation, and maybe even support equational reasoning. The following is an account of our efforts.

8.4.1 Indexed Variables

Having indexed variables is very useful for programming, since it enables us to write a program for the general case of a problem; we don't need to write n programs to sort arrays of size $2 \ldots n$. But then we have a problem:

$$[4/k] \circ [8/l[k], 6/l[4]] = ?$$

We must either make sure that the indexes are grounded or allow composition of substitutions to fail when the result would be inconsistent. Since we find it hard to figure out why someone would want to assign values to unspecified elements of an array, we chose the first option. In [AB98], the corresponding requirement is that all indexes have to be grounded, and the term to be substituted for the index variable must be also grounded; we relax the requirement in that the substituting term can be non-grounded. Another idea to consider is to treat bindings of the form $[t/v[k]]$ and formulas of the form $v[k] \doteq t$, where k is not grounded, as instances of $t_1 \doteq t_2$, that is, add the equality to the list of atoms.

8.4.2 Teaching *Dynamo* to Add

Something that might not be apparent in the description of Chapter 7 is that the handling of terms does not contemplate interpreted function symbols other than $\doteq$. While this is a specialized behavior for theorem provers, it is crucial for a programming language: we want the language to be able to do basic arithmetic, such as necessary for incrementing a counter or specifying a range. We added then interpretation of $+$, $-$, $*$ and `div` (integer division) to the language. Still, something was missing, since expressions could be indirectly ground (as in $x \doteq y; y \doteq 4; z \doteq x + 2$), so we added a lookup to the congruence closure in the term evaluation. Also, formulas of the type ϕ^t and $\bigcup^{v}_{t_1,t_2}$ required evaluation of the terms in the rule body. We also included interpretation of $<,\leq,\geq,<$ for ground terms.

8.4.3 What to do with the Equations

After a formula is determined to be satisfiable, there are two possible outcomes for each branch: either the complete list of atoms has been grounded, found consistent, and discarded, or some values are still to be computed and we are left with a set of ungrounded atoms. Now, this set itself could be unsatisfiable (consider the atoms $\{a > b,\ b > c,\ c > a\}$, or a set of equations). We have not included equation solving in *Dynamo*; possible solutions include coupling an algorithm for equation solving to the tableau algorithm [ABC+02], calling an external program to solve the equation system, and encouraging potential programmers to try and make their programs give values to their variables; we're aiming for a language with an imperative flavor after all.

8.5 Example runs

We will show now some examples that highlight improvements of the implementation over previous versions of the engine and over the calculus as presented.

8.5.1. Example. [Blocks puzzle] Let's consider a classical AI puzzle [Ram87]: we have a pile of three blocks, which are either green or red. The bottom block is red, and the top block is green; we don't know the color of the middle brick. The question is: Is there a green brick on top of a red brick? The *Dynamo* version of the puzzle is as follows:

```
[juanh@banaan dynamo]$ cat tests/aipuzzle
/* program puzzle ;*/

begin
  G 1;R 3 ;
  O 1 2; O 2 3;
  not (some k;!(G k);!(R k ));
  not (some x; G x ;some y; R y ; O x y );
end

? () true
```

Here, the Gx and Rx predicate represents being green and red, respectively, and Oxy represents "block x is over block y". We state the facts about the disposition and coloring of the blocks, and that a block is either green or red. Then, we state that there never is a green brick on top of a red brick, and call *Dynamo*:

```
[juanh@banaan dynamo]$ ./dynamo tests/aipuzzle

Input:
 G{[1]};R{[3]};O{[1,2]};O{[2,3]};
 !(Ex k;!(G{[k]});!(R{[k]}));
 !(Ex x;G{[x]};Ex y;R{[y]};O{[x,y]})
End of input

Formula is False
Elapsed time: 3.0e-2
```

Previous versions of *Dynamo* would just return the • state, since by design they don't deal with universal quantification.

8.5.2. Example. [Computation of Answer Substitutions] In the example on Chapter 7, we hinted that while the tableau engine itself did not know about the semantics of <, the left branch could be eliminated by model checking or term rewriting, or adding the relevant axioms for <. We do a limited form of model checking (interpretation of <,<=,<=,< for ground terms), so here is how the problem looks like in *Dynamo*:

```
[juanh@banaan dynamo]$ cat tests/union
/* program union;*/

begin
x<3;
  begin
  x=5 or x=2
  end
end

? (x) true
```

and how *Dynamo* reacts:

```
[juanh@banaan dynamo]$ ./dynamo tests/union

Input:
 <{[x,3]};x==5 U x==2
End of input

Formula is True
"x=2"
[<{[x,3]}]

Elapsed time: 0.0
```

The line under "$x = 2$" tells us that the condition $x < 3$ is still active: we probably need to remove 'constraint' atoms once the constraint is fulfilled by the model.

8.5.3. Example. [More computed answers: the Eight Queens Problem] This is actually a classical *Dynamo* example program: all versions have been able to solve it. What makes it special in this case is that the engine had to be able to add and substract, address values in an array, and evaluate terms in the rule body. We found that there are a lot of things to improve on the new engine: the old *Dynamo* solved the problem in 5.3 seconds, while the current *Dynamo* took 37.3 seconds. We take comfort that the new engine, while more ponderous, can tackle many more problems than the previous ones, and that this is but a proof-of-concept implementation.

```
[juanh@banaan dynamo]$ cat tests/8queens
/* program Nqueens(f[]) ;*/
```

```
begin
  n = 8;
  some k; k := 0;
  do n times
  begin
    k := k + 1;
    find r in [1 .. n] with
    begin
      r = f[k];
      not (find i in [1 .. k-1] with
        (f[i] = r or  f[i] = r + (k - i) or f[i] = r - (k - i)) )
    end
  end
end

? (f[]) true
```

```
[juanh@banaan dynamo]$ ./dynamo tests/8queens

Input:
 n==8;Ex k;[(k,0)];Do n times ([(k,+([k,1]))];Choose (r:=1..n) with r==f[k];
!(Choose (i:=1..-([k,1]))
  with [[[[f[i]==r]] U [[f[i]==+([r,-([k,i])])]] U [[f[i]==-([r,-([k,i])])]]]]))
End of input

Formula is True
f[8]=3; f[7]=6; f[6]=4; f[5]=2; f[4]=8; f[3]=5; f[2]=7; f[1]=1;
f[8]=3; f[7]=5; f[6]=2; f[5]=8; f[4]=6; f[3]=4; f[2]=7; f[1]=1;
f[8]=5; f[7]=2; f[6]=4; f[5]=7; f[4]=3; f[3]=8; f[2]=6; f[1]=1;
f[8]=4; f[7]=2; f[6]=7; f[5]=3; f[4]=6; f[3]=8; f[2]=5; f[1]=1;
f[8]=5; f[7]=7; f[6]=2; f[5]=6; f[4]=3; f[3]=1; f[2]=4; f[1]=8;
f[8]=4; f[7]=7; f[6]=5; f[5]=2; f[4]=6; f[3]=1; f[2]=3; f[1]=8;
f[8]=6; f[7]=4; f[6]=7; f[5]=1; f[4]=3; f[3]=5; f[2]=2; f[1]=8;
f[8]=6; f[7]=3; f[6]=5; f[5]=7; f[4]=1; f[3]=4; f[2]=2; f[1]=8;
f[8]=5; f[7]=2; f[6]=4; f[5]=6; f[4]=8; f[3]=3; f[2]=1; f[1]=7;
f[8]=4; f[7]=2; f[6]=8; f[5]=6; f[4]=1; f[3]=3; f[2]=5; f[1]=7;
f[8]=5; f[7]=3; f[6]=1; f[5]=6; f[4]=8; f[3]=2; f[2]=4; f[1]=7;
f[8]=6; f[7]=3; f[6]=1; f[5]=8; f[4]=5; f[3]=2; f[2]=4; f[1]=7;
f[8]=4; f[7]=2; f[6]=5; f[5]=8; f[4]=6; f[3]=1; f[2]=3; f[1]=7;
f[8]=4; f[7]=6; f[6]=1; f[5]=5; f[4]=2; f[3]=8; f[2]=3; f[1]=7;
f[8]=5; f[7]=8; f[6]=4; f[5]=1; f[4]=3; f[3]=6; f[2]=2; f[1]=7;
f[8]=6; f[7]=3; f[6]=5; f[5]=8; f[4]=1; f[3]=4; f[2]=2; f[1]=7;
f[8]=4; f[7]=7; f[6]=3; f[5]=8; f[4]=2; f[3]=5; f[2]=1; f[1]=6;
f[8]=3; f[7]=5; f[6]=7; f[5]=1; f[4]=4; f[3]=2; f[2]=8; f[1]=6;
f[8]=3; f[7]=7; f[6]=2; f[5]=8; f[4]=5; f[3]=1; f[2]=4; f[1]=6;
f[8]=3; f[7]=5; f[6]=2; f[5]=8; f[4]=1; f[3]=7; f[2]=4; f[1]=6;
f[8]=8; f[7]=2; f[6]=5; f[5]=3; f[4]=1; f[3]=7; f[2]=4; f[1]=6;
f[8]=3; f[7]=1; f[6]=7; f[5]=5; f[4]=8; f[3]=2; f[2]=4; f[1]=6;
f[8]=7; f[7]=4; f[6]=2; f[5]=5; f[4]=8; f[3]=1; f[2]=3; f[1]=6;
f[8]=5; f[7]=7; f[6]=2; f[5]=4; f[4]=8; f[3]=1; f[2]=3; f[1]=6;
f[8]=4; f[7]=2; f[6]=8; f[5]=5; f[4]=7; f[3]=1; f[2]=3; f[1]=6;
f[8]=5; f[7]=2; f[6]=8; f[5]=1; f[4]=4; f[3]=7; f[2]=3; f[1]=6;
f[8]=4; f[7]=1; f[6]=5; f[5]=8; f[4]=2; f[3]=7; f[2]=3; f[1]=6;
f[8]=5; f[7]=1; f[6]=8; f[5]=4; f[4]=2; f[3]=7; f[2]=3; f[1]=6;
f[8]=7; f[7]=2; f[6]=4; f[5]=1; f[4]=8; f[3]=5; f[2]=3; f[1]=6;
f[8]=8; f[7]=2; f[6]=4; f[5]=1; f[4]=7; f[3]=5; f[2]=3; f[1]=6;
f[8]=3; f[7]=5; f[6]=8; f[5]=4; f[4]=1; f[3]=7; f[2]=2; f[1]=6;
```

```
f[8]=4; f[7]=8; f[6]=5; f[5]=3; f[4]=1; f[3]=7; f[2]=2; f[1]=6;
f[8]=4; f[7]=2; f[6]=7; f[5]=3; f[4]=6; f[3]=8; f[2]=1; f[1]=5;
f[8]=6; f[7]=3; f[6]=7; f[5]=2; f[4]=4; f[3]=8; f[2]=1; f[1]=5;
f[8]=3; f[7]=7; f[6]=2; f[5]=8; f[4]=6; f[3]=4; f[2]=1; f[1]=5;
f[8]=3; f[7]=6; f[6]=2; f[5]=7; f[4]=1; f[3]=4; f[2]=8; f[1]=5;
f[8]=7; f[7]=2; f[6]=6; f[5]=3; f[4]=1; f[3]=4; f[2]=8; f[1]=5;
f[8]=3; f[7]=6; f[6]=8; f[5]=2; f[4]=4; f[3]=1; f[2]=7; f[1]=5;
f[8]=2; f[7]=4; f[6]=6; f[5]=8; f[4]=3; f[3]=1; f[2]=7; f[1]=5;
f[8]=2; f[7]=6; f[6]=8; f[5]=3; f[4]=1; f[3]=4; f[2]=7; f[1]=5;
f[8]=4; f[7]=8; f[6]=1; f[5]=3; f[4]=6; f[3]=2; f[2]=7; f[1]=5;
f[8]=8; f[7]=4; f[6]=1; f[5]=3; f[4]=6; f[3]=2; f[2]=7; f[1]=5;
f[8]=6; f[7]=3; f[6]=1; f[5]=8; f[4]=4; f[3]=2; f[2]=7; f[1]=5;
f[8]=4; f[7]=6; f[6]=8; f[5]=2; f[4]=7; f[3]=1; f[2]=3; f[1]=5;
f[8]=7; f[7]=4; f[6]=2; f[5]=8; f[4]=6; f[3]=1; f[2]=3; f[1]=5;
f[8]=2; f[7]=6; f[6]=1; f[5]=7; f[4]=4; f[3]=8; f[2]=3; f[1]=5;
f[8]=6; f[7]=3; f[6]=7; f[5]=4; f[4]=1; f[3]=8; f[2]=2; f[1]=5;
f[8]=3; f[7]=8; f[6]=4; f[5]=7; f[4]=1; f[3]=6; f[2]=2; f[1]=5;
f[8]=1; f[7]=6; f[6]=8; f[5]=3; f[4]=7; f[3]=4; f[2]=2; f[1]=5;
f[8]=7; f[7]=1; f[6]=3; f[5]=8; f[4]=6; f[3]=4; f[2]=2; f[1]=5;
f[8]=2; f[7]=7; f[6]=3; f[5]=6; f[4]=8; f[3]=5; f[2]=1; f[1]=4;
f[8]=6; f[7]=3; f[6]=7; f[5]=2; f[4]=8; f[3]=5; f[2]=1; f[1]=4;
f[8]=3; f[7]=6; f[6]=2; f[5]=7; f[4]=5; f[3]=1; f[2]=8; f[1]=4;
f[8]=5; f[7]=7; f[6]=2; f[5]=6; f[4]=3; f[3]=1; f[2]=8; f[1]=4;
f[8]=6; f[7]=2; f[6]=7; f[5]=1; f[4]=3; f[3]=5; f[2]=8; f[1]=4;
f[8]=3; f[7]=6; f[6]=2; f[5]=5; f[4]=8; f[3]=1; f[2]=7; f[1]=4;
f[8]=2; f[7]=8; f[6]=6; f[5]=1; f[4]=3; f[3]=5; f[2]=7; f[1]=4;
f[8]=8; f[7]=3; f[6]=1; f[5]=6; f[4]=2; f[3]=5; f[2]=7; f[1]=4;
f[8]=6; f[7]=1; f[6]=5; f[5]=2; f[4]=8; f[3]=3; f[2]=7; f[1]=4;
f[8]=7; f[7]=3; f[6]=8; f[5]=2; f[4]=5; f[3]=1; f[2]=6; f[1]=4;
f[8]=2; f[7]=5; f[6]=7; f[5]=1; f[4]=3; f[3]=8; f[2]=6; f[1]=4;
f[8]=5; f[7]=3; f[6]=1; f[5]=7; f[4]=2; f[3]=8; f[2]=6; f[1]=4;
f[8]=7; f[7]=5; f[6]=3; f[5]=1; f[4]=6; f[3]=8; f[2]=2; f[1]=4;
f[8]=6; f[7]=3; f[6]=1; f[5]=7; f[4]=5; f[3]=8; f[2]=2; f[1]=4;
f[8]=3; f[7]=6; f[6]=8; f[5]=1; f[4]=5; f[3]=7; f[2]=2; f[1]=4;
f[8]=5; f[7]=1; f[6]=8; f[5]=6; f[4]=3; f[3]=7; f[2]=2; f[1]=4;
f[8]=1; f[7]=5; f[6]=8; f[5]=6; f[4]=3; f[3]=7; f[2]=2; f[1]=4;
f[8]=7; f[7]=3; f[6]=1; f[5]=6; f[4]=8; f[3]=5; f[2]=2; f[1]=4;
f[8]=6; f[7]=4; f[6]=2; f[5]=8; f[4]=5; f[3]=7; f[2]=1; f[1]=3;
f[8]=5; f[7]=2; f[6]=6; f[5]=1; f[4]=7; f[3]=4; f[2]=8; f[1]=3;
f[8]=5; f[7]=1; f[6]=4; f[5]=6; f[4]=8; f[3]=2; f[2]=7; f[1]=3;
f[8]=6; f[7]=4; f[6]=1; f[5]=5; f[4]=8; f[3]=2; f[2]=7; f[1]=3;
f[8]=4; f[7]=2; f[6]=7; f[5]=5; f[4]=1; f[3]=8; f[2]=6; f[1]=3;
f[8]=2; f[7]=5; f[6]=7; f[5]=4; f[4]=1; f[3]=8; f[2]=6; f[1]=3;
f[8]=5; f[7]=7; f[6]=1; f[5]=4; f[4]=2; f[3]=8; f[2]=6; f[1]=3;
f[8]=2; f[7]=7; f[6]=5; f[5]=8; f[4]=1; f[3]=4; f[2]=6; f[1]=3;
f[8]=1; f[7]=7; f[6]=5; f[5]=8; f[4]=2; f[3]=4; f[2]=6; f[1]=3;
f[8]=5; f[7]=8; f[6]=4; f[5]=1; f[4]=7; f[3]=2; f[2]=6; f[1]=3;
f[8]=4; f[7]=8; f[6]=1; f[5]=5; f[4]=7; f[3]=2; f[2]=6; f[1]=3;
f[8]=4; f[7]=7; f[6]=1; f[5]=8; f[4]=5; f[3]=2; f[2]=6; f[1]=3;
f[8]=6; f[7]=2; f[6]=7; f[5]=1; f[4]=4; f[3]=8; f[2]=5; f[1]=3;
f[8]=6; f[7]=8; f[6]=2; f[5]=4; f[4]=1; f[3]=7; f[2]=5; f[1]=3;
f[8]=6; f[7]=4; f[6]=7; f[5]=1; f[4]=8; f[3]=2; f[2]=5; f[1]=3;
f[8]=1; f[7]=7; f[6]=4; f[5]=6; f[4]=8; f[3]=2; f[2]=5; f[1]=3;
f[8]=4; f[7]=7; f[6]=5; f[5]=3; f[4]=1; f[3]=6; f[2]=8; f[1]=2;
f[8]=3; f[7]=6; f[6]=4; f[5]=1; f[4]=8; f[3]=5; f[2]=7; f[1]=2;
f[8]=4; f[7]=1; f[6]=5; f[5]=8; f[4]=6; f[3]=3; f[2]=7; f[1]=2;
f[8]=5; f[7]=3; f[6]=8; f[5]=4; f[4]=7; f[3]=1; f[2]=6; f[1]=2;
f[8]=5; f[7]=7; f[6]=4; f[5]=1; f[4]=3; f[3]=8; f[2]=6; f[1]=2;
f[8]=4; f[7]=6; f[6]=8; f[5]=3; f[4]=1; f[3]=7; f[2]=5; f[1]=2;
f[8]=3; f[7]=6; f[6]=8; f[5]=1; f[4]=4; f[3]=7; f[2]=5; f[1]=2;
f[8]=5; f[7]=7; f[6]=1; f[5]=3; f[4]=8; f[3]=6; f[2]=4; f[1]=2;

Elapsed time: 37.31
```

8.6 Conclusion

We have now provided a platform on which to experiment on the concepts presented in Chapters 5, 6 and 7; this opens the way to cross-checking the contents of these chapters, and even as it is their culmination it serves as a background to their study. Implementation is the true test for theories; teaching a tableau prover to do simple arithmetic can be hard. Also, we found that the use of a theorem prover as a language engine demands a major increase in its capabilities to be worthwhile.

Next steps for *Dynamo* include adding equational reasoning capabilities, getting it to run faster, adding data types, the $\exists$ operator and the Kleene star, and implementing a tool to verify *Dynamo* programs using the calculus presented in Chapter 6.

Part III

Conclusions

Chapter 9

Conclusion

Computers are useless;
they can only give you answers.
–Pablo Picasso

We started out this journey with the intention of improving both the understanding and the landscape of automated reasoning tools for a variety of logics; in the course of this work, a translator from multi-modal logic into first order logic, a hybrid logic resolution theorem prover, a hybrid logic test set generator and a DFOL programming language were designed and implemented.

9.1 On Empirical Evaluation and Modal-like Satisfiability Testing

In Part I of the thesis we focused on putting modal logic to work; in particular, we were interested in different ways of implementing solvers for the modal satisfiability problem. We saw how empirical evaluation is useful not only for comparison of competing reasoning tools, but also for guidance and evaluation in the development of said tools, as well as evaluation of the test sets themselves. We also saw the importance of having a proper test set in the case of HyLoRes development; had we developed hGen first, the urgency of improving paramodulation treatment would have been much more apparent. We also saw two different ways of putting a particular logic to work: having a tool to translate it into a logic that has tools already developed for it, in this case FOL, or writing a tool from scratch. Each method has its advantages and disadvantages: the translation method can be very easy to do in a naïve way, but improving it requires tweaking the translation with an eye on the workings of the tool we want to work with, which will require substantially more involvement and is always limited. The custom tool way also has its own compromises: on the one hand, one has complete control over the inner workings of the tool, but on the other hand it tends to be a much bigger effort.

9.2 On DFOL programming

In Part II, we concentrated on one thing we can do with one logic, and all the ways in which we can look at it. We took DPL, extended it until it was expressive enough for programming, and stated an executable program interpretation for it. We reviewed the first two versions of the *Dynamo* engine, and decided that getting closer to DFOL semantics would be simpler if we abandoned the state machine approach and used a tableau prover for the engine instead. In the meantime, we provided a Hoare calculus for verification of *Dynamo* programs, which being inspired in the semantics of the logic instead of the program state, is the same for any incarnation of the *Dynamo* engine. In the end, we implemented *Dynamo*, in the course of which we learned that while a theorem prover has no trouble with the concept of negation-as-failure, things like simple arithmetic and equality reasoning require the engine to be significantly enlarged. Another desired functionality, the Kleene star, had to be postponed; the study of how to do unrestricted looping and still produce meaningful results falls out of the scope of the present work. One thing that can be done is take advantage of the lazy processing engine of Haskell, and report models as they appear.

9.3 Threaded through: Haskell and Scientific Programming

The translators from modal to first order logic, HyLoRes, hGen, and all versions of *Dynamo*, all share a common property: they have been written in Haskell. The main benefit of programming in Haskell was that since we did not have to worry about all the little details of *how* we wanted our computation carried out, we had more time to consider optimizations to the bigger details of the algorithms and data structures; it is also less trouble to change them in order to experiment. Ultimately, if one wants a really fast program and can devote the time and resources to developing it, the imperative way will always work better, although it is always superseded by hardware-specific machine coding, which in turn is bested by task-specific hardware design. There is something else to be said for this ordering, which is that the insights gained for each approach are increasingly different: the tasks performed and therefore the knowledge required focus more and more on where the data goes and how cleverly it is stored, recalled and updated. But all these optimizations are vulnerable to improvements in algorithm *quality*; developing better heuristics and better data organization usually results in more dramatic results than fine-tuning your loops or using custom hardware, and more importantly, *gives a better insight on the nature of the problem.*

9.4 Equality Reasoning

Another thing that became a common theme between the two parts of this thesis is the need for equality reasoning treatment. Both in HyLoRes and in the *Dynamo* engine, we found some manifestation of equality being a stone in our shoe. Were these stones equal? Well... since tableaux and resolution are dual methods, the problems posed by equality are perforce different; also, in *Dynamo* we want the solution to the set of equations, while in HyLoRes we do not have interpreted function symbols and are looking for contradictory statements.

9.5 One Logic to Find them, one Logic to Bind them?

The two main threads in this work are not parallel, but come together in a place slightly outside this thesis. DFOL is, after all, a dynamic logic: it has tests, which either fail or succeed, and other operators bring us from states to (sets of) states. Furthermore, Hoare logic can be expressed in terms of First Order Dynamic Logic [Har79], so in a sense all the logics covered here belong to the family of 'modal-like' logics. In fact, there is a way to express the Hoare calculus we introduced entirely in DFOL, since the meaning of both existential and universal correctness triples can be encoded in it: we can write $\{A\}\phi\{B\}$ as $A^\bullet \rightarrow (\phi \rightarrow B^\bullet)$, and $(A)\phi(B)$ as $A^\bullet \rightarrow (\phi; B^\bullet)$, where $\cdot^\bullet$ is a translation from FOL to DFOL. So it is revealed; the formulas of DFOL can be seen as modalities, where the 'worlds' in a model are the assignments, and the transitions are of course regulated by the usual semantics of DFOL: tests represent transitions to either the failure state or the current state, an assignment to a variable v represents a transition to a v-variant of the current state, and so on.

9.6 Final Remarks

A theory is useful only when it is used; it is our hope that the tools developed in the course of preparing this work make the involved logics more useful than they already are, by providing a testing lab to try out ideas and see how they work. And how well they do.

On the course of this study, then, some tools apt for studying and experimenting with nonclassical logics have been developed:

- HyLoRes: `http://www.science.uva.nl/~juanh/hylores`

- *Dynamo*: `http://www.science.uva.nl/~juanh/dynamo`
- hGen: `http://www.science.uva.nl/~juanh/hGen`

Bibliography

[AB98] K. Apt and M. Bezem. Formulas as programs. In *PNA-R9809*. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-3711, October 1998.

[ABC$^+$02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Integrating boolean and mathematical solving: Foundations, basic algorithms and requirements. In *Proceedings of Joint AISC 2002 and Calculemus 2002*. Springer, July 2002.

[ABPS98] K. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014–1066, 1998.

[AdNdR01] C. Areces, H. de Nivelle, and M. de Rijke. Resolution in modal, description and hybrid logic. *Journal of Logic and Computation*, 11(5):717–736, 2001.

[AE92] Y. Auffray and P. Enjalbert. Modal theorem proving: An equational viewpoint. *Journal of Logic and Computation*, 2(3):247–297, 1992.

[AG03] C. Areces and D. Gorín. Ordered resolution for hybrid logics. Submitted, 2003.

[AGHdR00] C. Areces, R. Gennari, J. Heguiabehere, and M. de Rijke. Tree-based heuristics in modal theorem proving. In W. Horn, editor, *Proc. ECAI'2000, Berlin, Germany*. IOS Press, 2000.

[AH02a] C. Areces and J. Heguiabehere. Direct resolution for modal-like logics. In *Proceedings of WIL 2002*. Kurt Gödel Society, 2002.

[AH02b] C. Areces and J. Heguiabehere. Hylores: A hybrid logic prover based on direct resolution. In *Proceedings of Advances in Modal Logic 2002*, 2002. refereed.

[AH03] C. Areces and J. Heguiabehere. hGen: A random CNF formula generator for Hybrid Languages. In *Proceedings of Methods for Modalities 3*, 2003.

[Apt00] K. Apt. A denotational semantics for first-order logic. In *Proc. of the Computational Logic Conference*, pages 53–69. Springer, 2000.

[Are00] C. Areces. *Logic Engineering. The Case of Description and Hybrid Logics.* PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, Amsterdam, The Netherlands, October 2000.

[AV02] K. Apt and C. Vermeulen. First-order logic viewed as a constraint programming language. In A. Voronkov and M. Baaz, editors, *Proceedings of LPAR02*, pages 19–35. Springer, 2002.

[BBKdN98] P. Blackburn, J. Bos, M. Kohlhase, and H. de Nivelle. Automated theorem proving for natural language understanding, 1998.

[BdRV01] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge University Press, 2001.

[BFH$^+$92] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. In *Proceedings KR-92*, 1992.

[BG98] L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In *Automated deduction—a basis for applications, Vol. I*, pages 353–397. Kluwer Acad. Publ., Dordrecht, 1998.

[BG01] L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01], chapter 2, pages 19–99.

[BGdR03] S. Brand, R. Gennari, and M. de Rijke. Constraint programming for modelling and solving modal satisfiability. In *Proceedings CP 2003*, 2003.

[BHS00] P. Balsiger, A. Heuerding, and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, 2000.

[Bli] Bliksem Version 1.10B. URL: `http://www.mpi-sb.mpg.de/~bliksem/`. Accessed January 16, 2000.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM Journal of Computing*, 4(4):412–430, 1975.

[CAS] The CADE ATP System Competition. `http://www.cs.miami.edu/~tptp/CASC/`. Site accessed on July 17, 2003.

[CGM$^+$97] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Model checking safety critical software with spin: An application to a railway interlocking system. In *Proceedings of the Third SPIN Workshop*, 1997. Twente, the Netherlands.

[CGV02] D. Calvanese, G. De Giacomo, and M. Vardi. Reasoning about actions and planning in ltl action theories. In *Proc. of the 8th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2002)*, pages 593–602, 2002.

[DHK98] Gilles Dowek, Therese Hardin, and Claude Kirchner. Theorem proving modulo. Technical Report RR-3400, 1998.

[DP60] S. Davis and M. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.

[DS94] E Dijkstra and C. Scholten. *redicate Calculus and Program Semantics.* MIT Press, Cambridge, Massachusetts, 1994.

[EdC89] P. Enjalbert and L. Fariñas del Cerro. Modal resolution in clausal form. *Theoretical Computer Science*, 65(1):1–33, 1989.

[FDGM$^+$98] E. Franconi, G. De Giacomo, R. MacGregor, W. Nutt, and C. Welty, editors. *Proceedings of the 1998 International Workshop on Description Logics (DL'98)*, 1998. Available at `http://sunsite.informatik.rwth-aachen.de/publications/ceur-ws`.

[FHMV95] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge.* The MIT Press, 1995.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag New York, 2nd edition, 1996.

[GGST98] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 626–635. Morgan Kaufmann, San Francisco, California, 1998.

[GHC] the glasgow haskell compiler homepage. `http://www.haskell.org/ghc/`. Site accessed on June 19, 2003.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[GNU] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`.

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* Springer, 1982.

[Gol92] R. Goldblatt. *Logics of Time and Computation.* CSLI Lecture Notes. CSLI, Stanford, second edition, revised and expanded edition, 1992.

[GP92] V. Goranko and S. Passy. Using the universal modality: gains and questions. *Journal of Logic and Computation*, 2:5–30, 1992.

[Grä01] E. Grädel. Why are modal logics so robustly decidable? In G. Paun, G. Rozenberg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science. Entering the 21st Century*, pages 393–408. World Scientific, 2001.

[GS90] J. Groenendijk and M. Stokhof. Dynamic montague grammar. In L. Kalman and L. Polos, editors, *Papers from the Second Symposium on Logic and Language.* Akademiai Kiadoo, 1990.

[GS91] J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14(1):39–100, 1991.

[GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures—the case study of modal K. In *Proceedings CADE-96*, 1996.

[GvMW00] I. Gent, H. van Maaren, and T. Walsh., editors. *SAT 2000.* IOS Press, 2000.

[Häh01] R. Hähnle. Tableaux and related methods. In Robinson and Voronkov [RV01], chapter 3, pages 100–178.

[Har79] D. Harel. *First Order Dynamic Logic.* Springer, 1979.

[HdR01] J. Heguiabehere and M. de Rijke. The random modal qbf test set. In *Proceedings IJCAR Workshop on Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics*, pages 58–67, 2001.

[Her89] A. Herzig. *Raisonnement automatique en logique modale et algorithmes d'unification.* PhD thesis, Université Paul-Sabatier, Toulouse, 1989.

[Hil03] T. Hillenbrand. Citius altius fortius: Lessons learned from the theorem prover waldmeister. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[Hin88] J. Hintikka. Model minimization - an alternative to circumscription. *Journal of Automated Reasoning*, 4(1):1–13, March 1988.

[HKT84] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic Volume II — Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984.

[Hoa69] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):567–580, 583, 1969.

[HPSS00] I. Horrocks, P. Patel-Schneider, and R. Sebastiani. An analysis of empirical testing for modal decision procedures. *Logic Journal of the IGPL*, 8:293–323, 2000.

[HS96] A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, and S4. Technical report IAM-96-015, University of Bern, Switzerland, 1996.

[HS97] U. Hustadt and R. Schmidt. On evaluating decision procedures for modal logic. In *Proceedings IJCAI-97*, pages 202–207, 1997.

[HyL] The hybrid logics homepage. `http://www.hylo.net`. Site accessed on July 17, 2003.

[Kam81] H. Kamp. A theory of truth and semantic representation. In J. Groenendijk, T. Janssen, and M. Stokhof, editors, *Formal Methods in the Study of Language: Part 1*, pages 277 – 322. Mathematisch Centrum, 1981.

[Koh00] M. Kohlhase. Model generation for Discoure Representation Theory. In *ECAI Proceedings*, 2000.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP 4*, pages 569–574, 1974.

[Lad77] R. Ladner. The computational complexity of provability in systems of modal logic. *SIAM Journal on Computing*, 6:467–480, 1977.

[LFdC88] A. Herzig L. Fariñas del Cerro. Linear modal deductions. In E. Lusk and R. Overbeek, editors, *Proceedings of* 9^{th} *International Conference on Automated Deduction, CADE-88*, volume 310 of *Lecture Notes in Computer Science*, pages 487–499. Springer, 1988.

[LS02] M. Lange and C. Stirling. Model checking fixed point logic with chop. In M. Nielsen and U. H. Engberg, editors, *Proc. 5th Conf. on Foundations of Software Science and Computation Structures, FOSSACS'02*, volume 2303 of *LNCS*, pages 250–263, Grenoble, France, April 2002. Springer.

[Mas99] F. Massacci. Design and results of the Tableaux-99 non-classical (modal) system competition. In *Proceedings Tableaux'99*, 1999.

[Min89] G. Mints. Resolution calculi for modal logics. *amst*, 143:1–14, 1989.

[MSP] MSPASS V 1.0.0t.1.2.a. URL: `http://www.cs.man.ac.uk/~schmidt/mspass`. Accessed June 11, 2003.

[NN92] H. Nielson and F. Nielson. *Semantics with Applications.* John Wiley and Sons, 1992.

[Ohl88] H.J. Ohlbach. A resolution calculus for modal logics. In E. Lusk and R. Overbeek, editors, *Proceedings of* 9^{th} *International Conference on Automated Deduction, CADE-88*, volume 310 of *Lecture Notes in Computer Science*, pages 500–516. Springer, 1988.

[Oka01] C. Okasaki. An overview of edison. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier, 2001.

[ONdRG00] H. Ohlbach, A. Nonnengart, M. de Rijke, and D. Gabbay. Encoding two-valued non-classical logics in classical logic. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier, 2000.

[Par78] R. Parikh. The completeness of propositional dynamic logic. In *Proceedings of the* 7^{th} *Symposium on Mathematical Foundations of Computer Science*, volume 64, pages 403–415. Springer, 1978.

[Pel85] D. Peleg. Concurrent dynamic logic. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 232–239. ACM Press, New York, 1985.

[Pra76] V. Pratt. Semantical considerations on Floyd–Hoare logic. In *Proceedings 17th IEEE Symposium on Foundations of Computer Science*, pages 109 – 121, 1976.

[PS98] P. Patel-Schneider. DLP system description. In Franconi et al. [FDGM+98], pages 87–89. DLP is available at `http://www.bell-labs.com/user/pfps`.

[PSHvH02] P. Patel-Schneider, I. Horrocks, and F. van Harmelen. Reviewing the design of daml+oil: An ontology language for the semantic web. In R. Dechter, M. Kearns, and R. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 792–797, July 2002.

[PSS03] P. Patel-Schneider and R. Sebastiani. A new general method to generate random modal formulae for testing decision procedures. *Journal of Artificial Intelligence Research*, 18:351–389, May 2003.

[PSV02] G. Pan, U. Sattler, and M. Vardi. Bdd-based decision procedures for k. In *Proceedings of the Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2002.

[PT85] S. Passy and T. Tinchev. PDL with data constants. *Information Processing Letters*, 20:35–41, 1985.

[PT91] S. Passy and T. Tinchev. An essay in combinatory dynamic logic. *Information and Computation*, 93(2):263–332, 1991.

[RAC] The racer system homepage. `http://www.fh-wedel.de/~mo/racer/`.

[Ram87] A. Ramsay. *Formal Methods in Artificial Intelligence*. Cambridge University Press, 1987.

[Rei95] W. Reif. The KIV Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009. Springer Verlag, 1995.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[RSV01] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science, 2001.

[RV01] A. Robinson and A. Voronkov, editors. volume I. Elsevier Science, 2001.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In *Machine Intelligence, 4*, pages 135–150. American Elsevier, New York, 1969.

[*SA] *SAT Homepage. URL: :`http://www.mrg.dist.unige.it/~tac/StarSAT.html`.

[Sch97] R. Schmidt. *Optimised Modal Translation and Resolution.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[SE88] C. Sedogbo and M. Eytan. A tableau calculus for DRT. *Logique et Analyse*, pages 379–402, 1988.

[Seg82] K. Segerberg. *A completeness theorem in the modal logic of programs*, pages 36–46. Polish Science Publications, 1982.

[Smu68] R. Smullyan. *First-Order Logic.* Springer, 1968.

[SPA] SPASS Version 1.0.3. URL: `http://spass.mpi-sb.mpg.de/`. Accessed June 11, 2003.

[Tac99] A. Tacchella. *SAT system description. In *Proceedings DL'99*, 1999.

[TAN] TANCS: Tableaux Non-Classical Systems Comparison. `http://www.dis.uniroma1.it/~tancs`. Site accessed on January 17, 2000.

[tCvEH01] B. ten Cate, J. van Eijck, and J. Heguiabehere. Expressivity of extensions of dynamic predicate logic. pages 61–66, 2001.

[Var97] M. Vardi. Why is modal logic so robustly decidable? In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 31, pages 149–184. AMS, 1997.

[vB83] J. van Benthem. *Modal Logic and Classical Logic.* Bibliopolis, 1983.

[vB86] J. van Benthem. Partiality and nonmonotonicity in classical logic. *Logique et Analyse*, 1986.

[vB96] J. van Benthem. *Exploring Logical Dynamics.* CSLI Publications, 1996.

[vBV92] J. van Benthem and W. Meyer Viol. Logical semantics of programming. Manuscript, 1992.

[vE98a] J. van Eijck. Dynamo — a language for dynamic logic programming. 1998.

[vE98b] J. van Eijck. Programming with dynamic predicate logic. Technical Report CT-1998-06, ILLC, 1998. Available from `www.cwi.nl/~jve/dynamo`.

[vE99a] J. van Eijck. Axiomatising dynamic logics for anaphora. *Journal of Language and Computation*, 1:103–126, 1999.

[vE99b] J. van Eijck. Powering decision machines with dynamo. In J. Gerbrandy, M. Marx, M. de Rijke, , and Y. Venema, editors, *Essays dedicated to Johan van Benthem on the Occasion of his 50th Birthday.* ILLC, Amsterdam, 1999.

[vEdV92] J. van Eijck and F.-J. de Vries. Dynamic interpretation and hoare deduction. *Journal of Logic, Language and Information*, 1(1):1–44, 1992.

[vEHN01] J. van Eijck, J. Heguiabehere, and B. Ó Nualláin. Tableau reasoning and programming with dynamic first order logic. *Logic Journal of the IGPL*, 9(3):411–445, 2001.

[Ver03] C. Vermeulen. Decidability and axiomatisation of a denotational semantics for first order logic. Manuscript, CWI, 2003.

[Vis98] A. Visser. Contexts in dynamic predicate logic. *Journal of Logic, Language and Information*, 7(1):21–52, 1998.

[Vor95] A. Voronkov. The anatomy of Vampire. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[Vor01] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In R.Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. 1st. International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 13–28, Siena, Italy, June 2001.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in LNCS. Springer Verlag, 1995.

[Wal94] C. Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation.* Research Notes in Artificial Intelligence. Morgan Kaufmann, 1994.

[X S] Spin – formal verification. `http://www.spinroot.com/spin/whatispin.html`.

[Zam89] N. Zamov. Modal resolutions. *Soviet Mathematics*, 33(9):22–29, 1989.

Samenvatting

Formele logica is de studie van noodzakelijke waarheden en systematische methoden met als doel deze waarheden helder uit te drukken en rigoreus te demonstreren. Dit proefschrift gaat over de automatisering van de conclusies die mogelijk zijn gemaakt door zekere logica's, over de evaluatie van deze automatiserings methoden en mogelijke toepassingen hiervoor.

De afgelopen jaren is de efficientie van het automatische bewijzen van stellingen voor modale logica enorm toegenomen, tegelijkertijd is het gebied van evaluatie van deze stelling bewijzers gerijpt. We zullen een aantal van de strategieën zien die gebruikt worden om middelen voor automatisch redeneren voor deze logicas te ontwikkelen en zien wat de rol is van empirische beoordeling in dit proces. We zullen ook zien hoe Dynamic Predicate Logic (DPL) geïnterpreteerd kan worden als programmeer taal, en hoe programma's die geschreven zijn in die taal gemakkelijk formeel gecontroleerd kunnen worden. Uiteindelijk zullen we zien hoe automatisch redeneren gebruikt kan worden als een motor voor berekeningen. Dit werk gaat dan over *middelen*: hun ontwikkeling, beoordeling en mogelijke toepassingen.

Dit proefschrift is ingedeeld in twee hoofd gedeeltes. Deel I, Evaluation in Modal and Hybrid Theorem Proving, gaat over de huidige en bestaande pogingen op het gebied van stelling bewijsvoering in modale en hybride logica en het belang van beoordeling in het ontwerp en vergelijking van stelling bewijzers evenals in de beoordeling van de standaarden zelf. In Hoofdstuk 2 zullen we de evolutie van de standaardisering in modale logica stelling bewijsvoering bespreken en zullen we een hybride logica standaard introduceren. In Hoofdstuk 3 spreken we over de verschillende methoden voor het vertalen van modale logica naar First Order Logic (FOL), over het voordeel te gebruiken van de jaren van ontwikkeling die zijn gegaan in FOL stelling bewijsvoering en over hoe verschillende methoden te vergelijken. In Hoofdstuk 4 beschrijven we andere kijk op stelling bewijsvoering in niet klassieke logica: ontwikkeling van jouw eigen gespecialiseerde stelling bewijzer. We beschrijven de theorie en implementatie van HyLoRes, een oplossing gebaseerde stelling bewijzer voor hybride logica; we beschrijven ook hoe het testen een onaangetast deel van de ontwikkeling was.

In Deel II, Programming with Dynamic First Order Logic, onderzoeken wij het gebruik van Dynamic First Order Logic (DFOL) als een programmeer taal. In Hoofdstuk 5 geven we enige achtergrond van het concept model 'formules als programma's'; we introduceren het concept van een uitvoerbare interpretatie van DFOL($\cup$), en beschrijven twee steeds betrouwbaar wordende benaderingen van de interpratie. In Hoofdstuk 6 leggen we uit waarom DFOL($\cup, \sigma$) een goede kandidaat is voor een programmeer taal en beschrijven we een Hoare calculus daarvoor. In Hoofdstuk 7 beschrijven we een reken tabel voor DFOL($\cup, \sigma$) die zelfs een betere benadering geeft voor de uitvoerbare interpretatie van DFOL($\cup, \sigma$) en kan gebruikt worden als een programmeer taal moter en in Hoofdstuk 8 beschrijven we de implementatie van zo'n moter en laten we een aantal voorbeeld runs.

In Deel III, Conclusie, kijken we terug op wat er geleerd is in de delen I en II, wat ze gemeenschappelijk hebben en waar ze elkaar ontmoeten.

Abstract

Formal logic is the study of necessary truths and of systematic methods for clearly expressing and rigorously demonstrating such truths. This thesis is about the automation of the inferences made possible by certain logics, about the evaluation of these automation methods, and some possible uses for them.

The last few years have seen a huge increase in the efficiency of theorem provers for modal and modal-like logics, and together with it the field of evaluation of these theorem provers has matured considerably. We will see some of the strategies used to develop automatic reasoning tools for these logics, and the role of empirical evaluation in this process. We will also see how Dynamic Predicate Logic (DPL) can be interpreted as a programming language, and how programs written in that language can be easily subjected to formal verification. Finally, we will see how automated reasoning can actually be used as a computation engine. This work is then about *tools*: their development, evaluation, and possible uses.

This thesis is organized in two main parts. Part I, Evaluation in Modal and Hybrid Theorem Proving, deals with current and existing efforts in the field of modal and hybrid logic theorem proving, and the importance of evaluation in the design and comparison of theorem provers as well as in the evaluation of the benchmarks themselves. In Chapter 2 we'll review the evolution of benchmarking in modal logic theorem proving, and introduce a hybrid logic benchmark. In Chapter 3 we talk about the different methods for translating Modal Logic to First Order Logic (FOL), to take advantage of the years of development that went into FOL theorem proving, and how different methods compare. In Chapter 4 we describe another approach to theorem proving in non-classical logics: developing your own specialized theorem prover. We describe the theory and implementation of HyLoRes, a resolution-based theorem prover for hybrid logics; we also describe how testing was an integral part of development.

In Part II, Programming with Dynamic First Order Logic, we explore the use of Dynamic First Order Logic (DFOL) as a programming language. In Chapter 5 we give some background to the 'formulas as programs' paradigm; we in-

troduce the concept of an executable interpretation of DFOL($\sqcup$), and describe two increasingly faithful approximations to the interpretation. In Chapter 6 we explain why DFOL($\sqcup, \sigma$) is a good candidate for a programming language and describe a Hoare calculus for it. In Chapter 7 we describe a tableau calculus for DFOL($\sqcup, \sigma$) which gives an even better approximation to the executable interpretation of DFOL($\sqcup, \sigma$) and can be used as a programming language engine, and in Chapter 8 we describe the implementation of such an engine and show some example runs.

In Part III, Conclusion, we reflect on what was learned from Parts I and II, what they had in common, and where they meet.