Automated Reasoning in

Quantified Modal and Temporal Logics

Claudio Castellini

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2004

Abstract

This thesis is about automated reasoning in quantified modal and temporal logics, with an application to formal methods. Quantified modal and temporal logics are extensions of classical first-order logic in which the notion of truth is extended to take into account its necessity or equivalently, in the temporal setting, its persistence through time.

Due to their high complexity, these logics are less widely known and studied than their propositional counterparts. Moreover, little so far is known about their mechanisability and usefulness for formal methods.

The relevant contributions of this thesis are threefold: firstly, we devise a sound and complete set of sequent calculi for quantified modal logics; secondly, we extend the approach to the quantified temporal logic of linear, discrete time and develop a framework for doing automated reasoning via Proof Planning in it; thirdly, we show a set of experimental results obtained by applying the framework to the problem of Feature Interactions in telecommunication systems.

These results indicate that (a) the problem can be concisely and effectively modeled in the aforementioned logic, (b) proof planning actually captures common structures in the related proofs, and (c) the approach is viable also from the point of view of efficiency.

Acknowledgements

First and foremost, I wish to acknowledge my supervisor, Dr. Alan Smaill, who has been able, little by little, to teach me how to be a researcher on my own — which seems no small achievement. In all this time, he has also had the patience of bearing with all my moods, no easy task at all. It has been a wonderful experience to work with him.

My academic mentors: Alan Bundy (who was also my internal examiner), Muffy Calder, Anatoli Degtyarev, Clare Dixon, Michael Fisher, Paul Jackson, Alice Miller; Zohar Manna and the REACT group, plus Sandro Coglio; Roy Dyckhoff, my external examiner; Luca Viganò, Silvio Ghilardi and especially Rajeev Goré for teaching me a lot about modal logics ("it is gratifying to know that someone else apart from my mother has read one of my papers" — Goré 2002).

A final thought for all the friends I have made in Scotland, and that make me regret having left every day more. A full list would be impossible, so I'll mention just a few of them; all the others please accept my apologies. Thank you Ewen, Graham, Fiona, Alison, Julian, Annabel, Ben, Luke, Zac, the Edinburgh Pizza Connection (Paolo, Nicola, Massimo, Andrea, Manuela, Sara, Carlo, Luisa), Marcio, Manuel, Predrag, Somnuk, Yelena, Angel, Noel O'Regan and the Edinburgh Renaissance Singers, Jera, Branka, Tonya and Axel, Chaker for the coffee and God for keeping his place from the fire, Johnnie Saundie for the meat and Stevie for some useful herbal remedies, besides all the rest. Take it easy, mates.

Declaration

I hereby declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Portions of the work here described have been published previously in [CS00, CS01, CS02a, CS02b].

Dedicated to my parents, Marisa and Alfredo,

and to Marcella. Grazie di tutto, pisellini meravigliosi.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modal Logics are extensions of classical logic, in which one or more modal operators are introduced, and in which the notion of truth is much richer and more subtle. The subtleties of modal truth have made these logics, along the years, a fascinating subject, since their invention by Aristotle.

More recently, especially after Hintikka and Kripke's idea of a "possible worlds" semantics, put forward in the early 60's, modal logics have also become a subject of applied research, especially in the field of formal verification. In fact, it has quickly become clear that a family of particularly expressive modal logics, called Temporal Logics, could capture the behaviour of a number of complex, artificial systems (circuits, protocols, programs) the safety of which had to be verified.

However, while propositional modal and temporal logics have been widely studied, *quantified* modal and temporal logics have been quite neglected, mainly for two reasons: first, because the expressivity of propositional modal and temporal logics has so far been more or less enough to fulfil the requirements posed by practical applications; second, because quantification introduces a lot of complexity, actually far more complexity in the case of temporal logics. These two factors have limited the study of quantified modal and temporal logics, at least with respect to their propositional counterparts. This is clearly reflected in the common assumption that "modal logic" mainly

stands for “propositional modal logic” in the community lingo.

In this thesis we investigate quantified modal and temporal logics, both from the theoretician’s and the practitioner’s point of view; we try to give a hint on *why* they are sometimes needed, as opposed to propositional modal and temporal logics, and *how* they can be practically employed to solve problems for which propositional modal and temporal logics are not enough any longer.

In particular, we first give a systematic proof-theoretic presentation of a wide set of quantified modal logics, and, under some simplifying assumptions, show that this presentation retains some good properties from the point of view of automated reasoning.

We then move on to quantified temporal logics, focusing our attention upon one of the most used and well-known ones: First-Order Linear-Time Temporal Logic (**FOLTL** for short). We show that the theoretical framework devised for quantified modal logics can be, to some extent, extended to **FOLTL**, and that this new framework can be used as the starting point for building an automated reasoning system, based upon the Proof Planning paradigm.

Proof planning is an approach to automated theorem proving which reduces proof search by raising it to a meta-level. Whereas in classical theorem proving one explores step-by-step a search space of inference rules applied “backwards” to a goal formula, in proof planning the search is conducted with A.I.-style planning operators (*methods*) which describe common patterns of reasoning in the object logic via meta-logical pre- and post-conditions. Methods represent proof steps larger and “more intelligent” than a single inference, and they are applied to *meta-level goals*, which are meta-logical representations of (possibly multiple) goals in the object logic.

Lastly, we show that, on a selected case-study, automated reasoning in **FOLTL** via Proof Planning leads to significant new results; it especially overcomes the traditional limitations of finitary proof systems (e.g., model checking), allowing for infinite-state systems to be validated.

As a whole, this thesis represents an attempt at introducing quantified modal and

temporal logics into the toolbox of the automated reason community.

1.1 Original contributions

The main original contributions of this thesis are three:

1. *Sequent calculi for Quantified Modal Logics.* We devise a new framework, represented by a family of Gentzen-style sequent calculi for quantified modal logics which enjoy some theoretically and practically relevant properties. In particular, any quantified modal logic whose Kripke frame enjoys properties which can be finitarily axiomatised in first-order logic with equality are captured by the framework. The calculi are proved to be *modular*, *uniform*, *normalising*, *sound* and *complete* for each logic. Most of this material also appears in [CS02b].

2. *Proof Planning for* **FOLTL**. We devise a set of proof planning methods tailored for **FOLTL** and customised for the case-study presented, with a rather large degree of generality. These methods effectively capture the common structure encountered in **FOLTL** proofs, and embed intelligent macro-steps of inference, modelled upon those a human mathematician would perform.

3. *Formal Methods.* The case-study, *Feature Interactions* in telecommunications systems, is modelled in **FOLTL** via an intuitive and clear set of formulae, and it is shown that the proof planning approach solves a set of associated formal verification problems, without making any simplifying assumption of finiteness over the domain.

Item 1 can be seen as an extension of Basin, Matthews and Viganò's work on quantified modal logics ([BMV96, BMV97a, BMV98, Vig00]); Item 2 is, as far as we know, the first attempt at applying the proof planning paradigm to **FOLTL** and formal methods; while Item 3, although in our opinion not yet mature for the Formal Meth-

ods community, is an interesting, successful practical application of the framework devised. An initial result in that direction has been published in [CS02a].

Other relevant original contributions of this thesis are:

- A standard method of proving soundness and completeness of sequent calculi for modal and temporal logics, based upon the paradigm of labelled deduction and two-sorted first-order logic.

- The development of a practical, hands-on integration between the proof planner λCLAM and an object-level theorem prover — a step that is required if one needs to ensure that proof plans actually correspond to proofs and therefore represent sound derivations. Surprisingly, as far as we know it is the first time in λCLAM's long history that this is done. The integration scheme can be reused for any object logic and set of planning methods whatsoever.

- A rigorous although informal method of modelling complex systems in **FOLTL**, such as a network of telephone users.

1.2 Structure of the thesis

The thesis is organised as follows:

- Chapter 2 is a survey of relevant literature, including theoretical and practical work on various kinds of modal and temporal logics, proof planning and formal methods;

- Chapter 3 shows the theoretical framework which is used as the basis for automated reasoning in quantified modal and temporal logics;

- Chapters 4 and 5 describe how to build a proof planning environment for the logics tackled by the framework of the previous Chapter, by coupling the proof

planner λCLAM with an object-level theorem prover devised by ourselves according to some rigorous guidelines;

- Chapter 6 describes the way we have modelled our case-study in **FOLTL**, how we have defined the properties we were interested in checking, and the set of proof planning methods devised to solve the associated problems;
- Chapter 7 presents experimental results and discusses them in comparison with some highly relevant related work;
- and lastly Chapter 8 draws conclusions and outlines future work.

Appendices A and B contain some material which would have made the thesis slightly heavier to read. In particular an interactive session with FTL, our object-level theorem prover, is reported, and a proof of its correctness is shown, meaning that a precise correspondence between proofs (in the sequent calculi sense) and proofs (in the theorem proving sense) is established.

Chapter 2

Literature survey

This Chapter reviews a range of background material that is related to this thesis. Sections 2.1 and 2.2 survey modal and temporal logics from a rather theoretical point of view — in the latter Section, formal methods and the verification problem are also sketched, together with the characteristics of **FOLTL**; Section 2.3 provides an overview of automated reasoning techniques and systems developed for temporal logics; Section 2.4 reviews proof planning, and finally in Section 2.5 a brief history of the problem of Feature Interactions in telecommunication systems is given.

2.1 Modal logics and labelled deduction

It would be pretentious to give here a complete account of modal logics (the most comprehensive reference for modal logics today, in general, is probably [CH95], whereas, for a more first-order oriented reference, see [FM98]), so we first give a broad outline of its history, the problems which arose and the ways they were solved, and then focus on quantified modal logics with respect to the so-called *semantics of possible worlds*.

The history of modal logics traditionally begins with Aristotle and his work about the "way" (Latin *modus*, whence the name *modal*) in which a proposition P can be true or false (necessarily or possibly, with various shades in between). Modal logics have then been further developed in the second half of the twentieth century and their

axiomatisations studied widely, mostly for purely academical purposes.

Round the mid-50s, in particular, a proliferation of informal interpretations and attempts at axiomatising them had made the field quite obscure, both in meaning and notation. It was widely recognised that modal logics extend classical logic with two operators, $\Box$ and $\Diamond$; the informal reading of $\Box$, in particular, varied from "necessarily" to "it is believed" to "it is known" to "it is morally acceptable that". Under different informal reading, different sets of axioms could be accepted; for example, traditionally the (set of) axiom(s) $\Box p \supset p$ stands for "what is necessarily true, is true"; in a different context, e.g., when $\Box$ is interpreted as "mandatory", its meaning is less clear and is possibly not enforced by some reasonable models — it is difficult to believe that whatever is mandatory is true.

In this confusion of roles and lack of a unifying perspective, Hintikka and Kripke's idea of a "possible worlds" semantics expressed via a graph ([Hin62, Kri63]) represented a huge step ahead and boosted the relevance of modal logics to Computer Science.

The informal interpretation of the possible worlds semantics is that there exists a set of possible worlds, alternative to the one chosen as reference, in each of which the truth value of objects can be different; in this context, $\Box$ and $\Diamond$ stand for, in turn, "true in all possible worlds" and "true in at least one possible world". This interpretation has a clear intuitive reflection in the very concepts of necessity and possibility: what is necessarily true is true in all possible worlds, and what is possibly true is true in at least one possible world; and besides that, it also gives an account of what different set of modal axioms mean: for example, if $\Box p \supset p$ holds, then "whatever is true at all possible worlds is true at this very world", where our world is one of the possible worlds.

This remark serves as a basis for a further conceptual advancement, that of a generalisation of the notion of possible worlds into that of *accessible* worlds. It could be the case that each possible world has "access" to only a limited fraction of the total

possible worlds; in such a context, $\Box p \supset p$, asserted at an arbitrary world w_0, means that "whatever is true at all worlds accessible from w_0 is true at w_0 itself". Since the choice of w_0 is arbitrary, the axiom states that each world is accessible from itself. We now have a semantics based upon a set and a binary relation on it — that is, a *graph*. Previously obscure sets of axioms had now an immediate correspondence with the structure of the underlying graph, called *frame*. For instance, $\Box p \supset p$ corresponds to a reflexive relation and, with a slight abuse of language, a reflexive frame.

The first systematic presentation of these correspondences, due to van Benthem ([van84]), gave rise to the notion of characterisation of a class of frames by means of modal axioms and/or first-order properties. In other words, a class of frames (say, all reflexive frames) was shown to make all instances of a modal axiom ($\Box p \supset p$ in this example) true, and conversely, all instances of the axiom were shown to be true in reflexive frames. A class of frames corresponded then to a condition on the frame relation R (called *accessibility* relation), in this case: $\forall x.xRx$; and, at the same time, it could be characterised by a modal axiom.

But it was also discovered that the correspondence is not always this sharp. It could be the case, for instance, that it is impossible to characterise a class of frames via a modal axiom, but the properties of the class itself are expressible in first-order logic. Conversely, it may happen that a certain class of frames enjoys a property that *cannot* be described in first-order logic. The latter case is particularly interesting to us, mainly when it comes to classes of frames in which every "chain" of worlds is finite. By a compactness argument, such property cannot be expressed in first-order logic, but still, the class of frames may have a finite modal axiomatisation. See Section 4.1 for a more detailed discussion.

It is possible to reflect the semantics of possible worlds (or *Kripke semantics*) in the syntax of the modal language we adopt if we use the so-called Labelled Deduction. The basic idea in labelled deduction is that of using terms of a special language (*labels*) to add information to a formula. Although used for a long time in logics, the

systematisation of labels (and the birth of the term *labelled deduction*) is due to Gabbay ([Gab96]). The main conceptual advancement is that the main unit of information, rather than the formula, like in ordinary, non-labelled logics, is the *labelled formula*: a formula with some "additional" information attached.

In the most general setting, a label can denote any kind whatsoever of information one might want to attach to a formula in order to ease its management; in modal logics, it is straightforward to use labels to denote possible worlds. This idea, explored quite thoroughly by Basin, Matthews and Viganò (see [BMV97b, BMV96, BMV97a, BMV98] and, of course, the book [Vig00]), has the main advantage of generating proof systems (mostly, natural deduction systems and sequent calculi) which are really easy to read and modular with respect to a large family of modal and temporal logics. That is why we decided to adopt labelled deduction, together with the intuition that this presentation would ease mechanisation, which hopefully this very work witnesses.

Switching to quantified modal logics, it turns out that the situation is not much harder, if we make some simplifying assumptions. [FM98] is a good survey of quantified modal logics and the problems they pose, from a rather abstract and philosophical point of view; its main merit for the computer scientist is perhaps that of systematising the different possibilities offered by quantification in modal logics. In particular, free quantification in modal logics can give rise to references to objects that do not exist in some worlds, therefore making uncertain the very meaning of quantification. The concept of *denotation* of a term is deeply analysed, together with another important issue, that of *flexibility* of terms, that is, the possibility that the denotation of a term (this time, in the classical logic sense) changes through time. Lastly, a further relevant point is that of the variability of the domain of quantification, that is, whether we shall assume that the domain is fixed (constant domains) or that it changes when we move from one world to another (increasing or decreasing domains).

In [BMV98], sound and complete proof systems are given for a large class of modal logics with rigid (i.e., non flexible) designators; in those logics, the only truth values

which depend on worlds are those of predicates. Notwithstanding this restriction, the logics remain quite powerful. In [CS02b] (but see, of course, Chapter 3), that class of logics is further extended, although under the restriction of constant domains.

2.2 Temporal logics and formal methods

Temporal logics are extensions of classical (propositional or first-order) logics, incorporating a model of the flow of time, either as metric constraints or via a suitable semantics. In the latter case, the employed semantics is extended with respect to the classical one in order to take into account the way truth propagates through time, and as well the way time itself is modelled (e.g., discrete versus continuous, linear versus branching etc.).

As well, temporal logics can be seen as *particularly strong* modal logics[1]: the flow of time is a frame, in the sense of Kripke, whose properties make it isomorphic to structures one is more used to see as trees (branching time) or total orders (linear time). One more complication is that the frame can be required to be discrete, and in this case, by the Compactness Theorem, its properties are not even expressible in first-order logic — but see Chapter 4 for a fuller account.

The interested reader is referred to [Gor93]; perhaps the most interesting result cited there, from the point of view of the link between modal and temporal logics, is that, semantically speaking, the propositional modal logics **S4**, **S4**.**14**, **S4**.**3** and **S4**.**3**.**1** correspond in turn to the temporal logics of dense/discrete trees and of dense/discrete linear time. While this holds for the propositional fragments of the logics, we have no news about similar relationships for the quantified versions, although it is quite likely that something like this holds[2].

[1] by the term *strong* here we simply mean "big": insofar as a *logic* is a set of well-formed formulae true under some interpretation, a logic L is stronger than a logic L' if and only if $|L| > |L'|$. The more axioms are added to a logic (or, equivalently, the more structured its frame is), the bigger this logic is.

[2] personal communication with Rajeev P. Goré; see also [DS02].

2.2.1 The verification problem

The reason for the huge interest raised by temporal logics in computer science and artificial intelligence in the past thirty years probably lies in their expressivity and intuitiveness. Temporal logics allow us to model both the behaviour and the requirement of any dynamic system whatsoever, where *dynamic* here has the usual meaning of "changing through time"; and it is clear that such a wide definition incorporates such diverse and important notions as computer programs, mechanical and electrical devices, agents interacting in a wild environment, etc.

Therefore, by means of temporal logics, one can formally specify both a complex system and the properties it is required to enforce. The problem of verifying that a system behaves in the desired way, that is, that its specification meets the requirement, is nowadays prominent and is usually called *formal verification*; the development of temporal logics and their adaptation has contributed to the creation of a whole new community, that of *formal methods* — techniques which solve the verification problem, more or less automatically, and offer a substantial improvement over the usual testing techniques. Needless to say, this community bears strong links with industry, thanks to the financial and social importance of the verification problem.

2.2.2 History

Historically, it is probably wise to say that temporal logics as we know them today were initially conceived by Prior ([Pri67]) in order to give an account of tenses via modal logics; this naturally led to a vision of time embedded in a semantics for a particular (set of) modal logic(s).

After this pioneering era, temporal logics have been developed mainly in two ways: on one hand, they have been related to other formalisms such as finite and infinite automata, graph reachability and grammars. This side of the story, which we are not going to explore in depth, has led to the development of some very successful techniques

for the verification of systems, such as model checking ([Cla97]), which enforces systematic state space exploration via fix-point operators, and automata theory (see, e.g., [Var03]), maybe the best approach so far to the verification problem for hardware design.

On the other hand, researchers have developed tools and algorithms to tackle temporal logics directly. After an initial era of research about properties of terminating programs, ending up in Hoare's logic ([Hoa69]) and Lamport's TLA (Temporal Logic of Actions, [Lam94]), the attention has gradually shifted to *reactive*, "never terminating" systems, such as multi-agent systems, protocols, finite automata, fair transition systems, down to more practical instances such as operating systems, circuits, hybrid systems and bank transaction management systems.

Amir Pnueli and Zohar Manna have pioneered the use of temporal logics, both in their theoretical ([MP81]) and practical aspects ([AM85]); other relevant pieces of work include Gabbay's ([GHR94], a remarkable general survey about temporal logics) and Fisher's idea of executable temporal specification ([Fis97]). Although nowadays less active, temporal logic programming (i.e., temporal extensions of Prolog) has been practised for some time (see Section 2.3).

2.2.3 Propositional temporal logics

In practice, the choice of a temporal logic depends on the problem one is trying to solve; from this point of view, what substantially characterises a temporal logic is, in order of importance for this work:

1. whether it is point- or interval-based;
2. the connectives: future and/or past;
3. the structure of time (branching or linear, continuous/dense/discrete);
4. whether it is propositional or first-order.

(see [Eme90] for a thorough classification). It is not our intention to explore all the combinations and possibilities here, but let us just give some meaningful remarks. First of all, we restrict to future-time operators and point-based semantics. Although this may seem a strong assumption, to our knowledge the most successful approaches to formal methods via temporal logics so far are point-based and make no use of past-time operators. As well, for all our purposes the past horizon of time is limited, that is, there actually is a starting point in time, for example, when the system under examination is started; this means that past-time operators are somehow "easier" to handle, if required.

Over the years, Linear Temporal Logic (*LTL*) and Computational Tree Logic (*CTL* with its variant CTL^*) have a particular place: in both logics time is discrete, but in *LTL* time is isomorphic to $(\mathbf{N}, <)$; it is probably the most widely used linear-time logic. *CTL*, on the other hand, has a branching model of time. Besides the language, the essential difference is that in *CTL* one can model and investigate properties of single *paths* in the evolution of time; that is, one can ask that a certain properties holds on some paths (traces, executions of a program or of a protocol, etc.), whereas this is impossible in *LTL*. As a matter of fact, *LTL* is generally considered an approximation of *CTL*, and it is usually employed in its place, because it is much less complex. The problem of testing satisfiability of a formula (which is roughly equivalent to the verification problem) is $EXPTIME$-complete for *CTL*, $PSPACE$-complete for *LTL* and NP-complete for $LTL(\Diamond)$, that is, the fragment of *LTL* restricted to the operator $\Diamond$ ("eventually"). Notice that we are here referring to the propositional fragments of these logics, and, as one can see, the complexity is already high.

2.2.4 Quantified temporal logics

If we switch to the first-order variants of *LTL* and *CTL*, usually called **FOLTL** and $FOCTL$, the situation becomes hard. Quantified temporal logics add the possibility of quantifying over a first-order-like domain, whose objects in general depend on time itself. As one might expect, the main problems arise from the interaction between time

and quantification.

It is known that full **FOLTL** is not only undecidable, but non-recursively enumerable; as far as we know, one of the first attempts at studying it is [AM85], where a sound and complete proof system is given for a fragment of it, namely that of *timed* formulae; in [MMW94] a complete proof system is given for such a fragment.

After a long period of quiescence, **FOLTL** has recently received a great deal of attention, probably both because the time is ripe to explore it, and the limits of *LTL* have been touched. Therefore, a number of "well-behaved" fragments of it have been studied. The most remarkable and recent piece of work is probably [HWZ00] (along with the book [GKWZ03], Chapter 11), to which the reader is addressed. In particular, one of the biggest and probably the most interesting decidable fragment of **FOLTL** is the so-called *monodic* fragment, in which only one variable can appear under the scope of any temporal operator. The monodic fragment was proved to be decidable in the above mentioned paper by Hodkinson, Wolter and Zakharyaschev. It is also worth reminding the reader of [WZ00a], focused on first-order *modal* logics, showing once again that temporal logics are modal logics with a particularly strong Kripke semantics. Other interesting pieces of work come from Pliuškevičius ([Pli97, Pli00, Pli01]).

Still, the usefulness of such fragments is in question and is not clear at the time of writing, although some practical applications which fit into it have been found (see, e.g., [AVFY98, WZ00b]). It is anyway true that in the very last years, this branch of logics has flourished, also thanks to an EPSRC-funded project, led by Mikhail Zakharyaschev, officially named "Analysis and mechanisation of decidable first-order temporal logics". Started in late 2001, with the objective of analysing the monodic fragment of **FOLTL**, it can probably be summarised so far by the paper [HWZ01]. Personal communication with one of the team members indicates that the search of interesting case-studies for it is ongoing and is a relevant part of the project.

As far as $FOCTL$ is concerned, very recently some of its fragments have been studied (see [HWZ02] for a negative one and [BHWZ02] for a positive one). We are

not aware of any implementation using $FOCTL$, nor of any case-study requiring it.

2.3 Automated reasoning in temporal logics

Unsurprisingly, a number of tools have been developed in the years for reasoning with temporal logics. Here is a brief survey of them.

2.3.1 Temporal logic programming

The technique called *temporal logic programming* has flourished during the 80s and part of the 90s. The basic idea is to augment Prolog with temporal annotations in order to make logic programming with time feasible. [OM94] is a wide survey of the attempts at building a temporal logic programming paradigm. According to it, temporal logic programming systems are divided into interval-based and point-based:

1. **interval-based TLP systems.** *Tempura* ([Mos98]) works on discrete Interval Temporal Logic (ITL). Programs are specified in an imperative-programming style and the execution consists of reducing the intervals assigned to each operation until no further reduction is possible. Applications of Tempura to hardware design, motion representation and algorithm description are cited. *Tokio* ([AFMO85]) enforces a discrete, linear ITL in a Prolog fashion; its language is *de facto* a superset of Prolog.

2. **point-based TLP systems.** *Templog.* Proposed by Abadí and Manna in the late 80s, Templog ([AM89]) enriches classical Prolog with the temporal operators $\Box$, $\Diamond$ and $\bigcirc$. It implements a linear, discrete-time subset of $FOLTL$ and executes it using the TSLD resolution method. It has been shown that Templog is an instance of the CLP scheme, and that Templog programs can be translated to a two-sorted first-order language which can be successfully attacked by a variant of SLD. Gabbay's *Temporal Prolog* ([Gab87]) is seemingly the only TLP system

which tackles branching time temporal logic, even if it has some restrictions on the form of the clauses. Hrycej's *Temporal Prolog* ([Hry88]), on the other hand, integrates a temporal reasoner built on top of classical Prolog (and employing Allen's temporal constraint model) with a constraint solver written in C; interesting results are reported, thus encouraging a hybrid approach, even in the case of logic programming. *Starlog* ([CK91]) follows similar principles.

A citation apart is for MetateM ([FO92]), which has then evolved into a tool for reasoning about agent and concurrent processes, rather than pure temporal logics. Some successes are cited in this paper, such as an application to database queries. Fisher, it is worth remarking, also devised the Separated Normal Form for **FOLTL** formulae, together with Gabbay ([Fis92]), which is at the basis of some recent attempts at mechanisation of **FOLTL** (e.g., [DF01]).

The general picture of TLP languages seems discouraging. Groups which have been working with them have now turned to hybrid or imperative systems. Moreover, a relevant number of the revised TLP languages already employ forms of search-based techniques, thus slipping toward the CLP scheme.

2.3.2 Model checking

Model checking is probably the most successful approach to automated reasoning of the last fifteen years. Since the early times (see, e.g., [CG87]) it has attracted a lot of industrial interest, and has progressed in a considerable manner. Literature about it is nowadays so wide that it is impossible to list it; the interested reader could begin with [BCM$^+$92], in which the notion of *symbolic* model checking was introduced, along with [HV91] in which the contraposition between model checking and theorem proving is outlined and discussed, and then read the book [CGP99].

Model checking consists in exploring the state space generated by a dynamic system. Under a suitable formal model and semantics, a dynamic system can be viewed as an operator mapping the current state to the possible states at the next instant of

time; this way a cone of reachable states is generated. Usually one wants to ensure that a set of "bad" states will never be reached by the system (safety), or that a request will be eventually accomplished no matter when it is issued (liveness); in model checking, safety corresponds to (non-)inclusion of bad states in the reachability cone, while liveness corresponds to finding looping trajectories which include requests and fulfilments.

Usually the system and the property are modelled in different ways, and it has been argued (e.g., [HV91]) that this is exactly what makes model checking peculiar and better than theorem proving. Systems are usually modelled by a Kripke-based semantic structure, which takes into account the shape of the state and the transitions, whereas the properties are usually modelled by *LTL* or *CTL* formulae, in which safety and liveness can be succinctly and intuitively expressed.

Historically, the main problem affecting model checking is state space explosion, namely the impossibility of keeping track of the reachability cone because it becomes too big. The big improvement as far as this problem is concerned was the invention of symbolic model checking, in which data structures such as BDDs were employed to compactly represent sets of states (rather than explicitly enumerating them). Other approaches employ Boolean formulae and constraints (mostly expressed via linear arithmetic). Still, the question of how far model checking approaches can go is quite open, since every different approach has its drawbacks and outliers (for instance, the class of Boolean multipliers represent a particularly hard benchmark for BDD-based approaches).

One more characteristic issue of model checking is that, besides recent advancements, it tackles finite-state space problems. In the most efficient settings, this is simply due to the necessity of keeping explicit track, although symbolic, of the *whole* state space. The problem has been (recently) tackled mainly in specialised ways, targeted for the particular "sense" in which infinity creeps in a particular class of systems: a system can have unbounded data structures, an arbitrarily large number of components,

or maybe employ recursion; model checking is being adapted to each single case with some degree of success, by attaching some deductive or constraint-solving machinery to the generic algorithm (see, e.g., [Esp97, SUM99]) but, as far as we know, there is no general application of model checking to infinite state space systems.

Another approach widely used so far is that of assuming a finite number of elements in the domain of quantification of a system, thus approximating the infinite-state space problem to a finite one; in this case ordinary model checking works fine and has led to a number of results. But this idea has two drawbacks: first, it only gives negative results, that is, it can only *find bugs*, since we may safely assume in most cases that a bug in a finite approximation will be such in the infinite setting as well. But if the model checker cannot find any bugs, this tells us little about the real system. Second, since approximations need grounding, i.e., expansion of every term into all possible ground instances, this approach suffers heavily from combinatorial explosion, therefore becoming impracticable as soon as the domain grows.

Among the most recent, popular and well-established model checkers it is worth mentioning SPIN ([Hol97a]), SMV ([BCM$^+$92]), NuSMV ([CCGR99]) and UPPAAL ([BLL$^+$96]).

2.3.3 STeP

STeP is perhaps the most extensive attempt at formalising and systematising automated reasoning in quantified temporal logics so far. Led by Zohar Manna since the early 90s, the STeP team ([MBB$^+$95]) has an impressive series of publications and academic successes; so far, the system has been the basis for more than thirty PhD theses and a number of informal works[3].

Its core is represented by an ML routine effectively combining decision procedures for decidable theories intervening in automated verification of properties of programs, hybrid, real-time and reactive systems (see [BBC$^+$96, MS98]). Its use has also

[3] personal communication with Zohar Manna.

spawned some interesting theoretical tools for the field, such as *verification diagrams* ([MS99]) — a first-order annotated variant of labelled transition graphs, plus semantic information on transitions.

STeP was born as, and still is, a hybrid system, the main idea attached to it being that of exploiting diverse forms of reasoning, both deductive (*verification conditions*), model-based (model checking) and algorithmic (decision procedures). The input formalism is *SPL* (Simple Programming Language), a Turing-complete language in which one can imperatively specify the behaviour of a program or a set of concurrent processes; the relative speed of processes is abstracted away, so that they can be thought as effectively running in parallel. Sharing of variables and message passing is also allowed.

The most interesting idea in STeP probably lies in the verification conditions and the invariants strengthening machinery. STeP implements a small set of sequent-style deduction rules, tailored for the different kinds of properties required by the user (who can also direct the search thanks to an interactive interface). Based upon Manna and Pnueli's classification of temporal properties (see [MP91]), this set enforces a sound and relatively complete proof system for most safety and liveness properties, which are thus reduced to a set of first-order conditions obtained via the semantics of *SPL* (which is quantifier-free first-order logic). In order to aid the proof machinery, whenever induction is required (that is always, except for toy problems), an automatic mechanism for invariants strengthening is employed, which can generate inductive assertions implying the initial query.

STeP is, so far, one of the few systems which can effectively tackle **FOLTL**, although not in full; there are indeed classes of problems outside the scope of STeP, but (according to Manna himself, and it is not difficult to endorse this remark) at some point one has to specialise his framework, in order to get tangible results. STeP enjoys a remarkable degree of automation; for instance it has a mechanism for automatic generation of invariants during an inductive proof.

Manna and his group's achievements are summarised ([MP92, MP95]). A third book is in preparation.

2.3.4 Tableau-based systems

By the term *tableau-based systems*, we will refer throughout this Subsection to systems which enforce (semi-)automated reasoning in **FOLTL** in various application fields, such as planning, via syntactic-based methods, that is, methods which do not enforce the structure of the system being examined, but rather write it as a set of formulae in a temporal logic and then perform deduction on them.

One interesting line of research is that of encoding planning problems into **FOLTL**. Especially the works of Bacchus and Kabanza ([BK00]) and Cerrito and Cialdea Mayer ([ML02]) show that significant improvements can be obtained, with respect to well-known planning systems, if control information is encoded in **FOLTL** and then used to prune the search. The main advantage of using such a high-level language, apparently, is that it is expressive and compact, and it represents an easy way to encode domain-specific knowledge (a recurring problem in the planning community). Both works do not really try to automate **FOLTL** but use ground versions of it to specialise the approach, but the result is well worth remarking, especially if one thinks that literally a lot of specialised approaches for planning exist, including Boolean satisfiability and graph reachability.

Felty ([FT97]) proposed in 1997 a sound and complete mixed natural deduction / sequent calculus for propositional *LTL*, which is of little interest to our topic, but still the interactive fashion contained in her experimental results showed the linear temporal logic was a viable tool, at least for the formalisation of complex problems. In particular, she devised a method for translating in a reasonable manner *LTL* proofs into natural (English) language. Also in the Prosper project (see [HK99]) quite a lot of effort has been dedicated to this translation.

2.4 Proof planning

Proof planning is an approach to automated theorem proving originally designed to reduce proof search by raising it to a meta-level [Bun88, BvHS91]. Whereas in classical theorem proving one explores step-by-step a search space of inference rules applied "backwards" to a goal formula, in proof planning the search is conducted with *methods*, A.I.-style planning operators which describe common patterns of reasoning in the object logic via meta-logical pre- and post-conditions. Methods can and should represent proof steps larger and "more intelligent" than a single inference, and they are applied to *meta-level goals*, which are meta-logical representations of (possibly multiple) goals in the object logic.

Proof planning systems use methods to build an abstract proof tree, or *proof plan*, which can then be used to find an object level proof, e.g. by running tactics corresponding to methods. There need not be a guarantee that any corresponding object level proofs can be found or even exist.

Meta-level goals and the meta-logical formulae in method conditions can express both legal and heuristic statements about proof goals. Legal statements are about the form of the object goals, e.g. when a method *could* be applied. Heuristic statements help guide the proof search, e.g. saying when a method *should* be applied. Methods and meta-level goals are usually designed by system authors or users, and typically oriented toward a specific domain where a set of heuristics is known, e.g. summing series [Wal94]. In [Bun91] a methodology for good method design is described, proposing evaluation criteria such as generality and parsimony.

The intended advantage of proof planning is that the planning search space is significantly smaller than the original object level search space. Conversely, the plan space is likely to be incomplete. Both these things depend entirely on the particular method set. Another aim of proof planning is provide declarative, as opposed to procedural, specifications of methods which can be reasoned *about* mechanically, not just executed. This facilitates the automatic learning and adaptation of proof methods.

2.4.1 CLAM: Advance Planning

The first proof planning system was CLAM [BvHS91]. It built upon the *tactic* based approach to theorem proving, where common patterns of inference rules are captured in tactics, a small program which automates the search for a proof fragment by applying rules according to the given pattern. In CLAM, a method is considered to be specification of a tactic, providing conditions for its application and the effects it has on the goal. A given tactic may have multiple methods, corresponding to its use in different situations.

The CLAM system was designed to work in conjunction with a tactic-based theorem prover, specifically the *Oyster* system, an implementation of Martin-Löf's Type Theory. It constructs a proof plan which is used to guide *Oyster* to a proof, by replacing methods with their corresponding tactics [BvHHS90]. Hence planning is done in advance of proving.

CLAM method conditions are written in a declarative style, i.e. as meta-logical statements. However, in practice it is possible to write procedural style conditions in Prolog, and CLAM method designers often do this to e.g. improve their efficiency or implement complex strategies.

2.4.2 λCLAM: proof planning in a higher-order framework

λCLAM [RSG98] is the successor to the CLAM system. Like CLAM, λCLAM is an advance planning system, producing plans to be converted into tactics. Unlike CLAM, which was born with *Oyster* as the object-level theorem prover, there was no specific underlying tactic-based theorem prover for λCLAM, until FTL was built. Actually, the proof checking mechanism developed during the application of proof planning to the Feature Interaction problem represents the first attempt at getting proofs out of proof plans obtained via the proof planner λCLAM.

Method conditions are now written in λ-Prolog [NM86] a higher-order version of

Prolog. Having a higher-order meta-logic has allowed a much more concise, natural and declarative expression of methods.

Another significant aspect of λCLAM is the use of *methodicals* to 'join together' methods to specify larger ones, in much the same way that tactics are formed using tacticals. This is extremely useful when describing large and complex strategies — a common problem in CLAM. It also allows a more declarative specification of such strategies. A semantics for these method expressions, based on continuations, is being investigated at the time of writing (see, e.g., [Ric02]).

2.4.3 ΩMEGA

The ΩMEGA system is another proof planning implementation [Ker98], but differs from the CLAM family in a number of important aspects. Most importantly, it does not differentiate between methods, tactics and inference rules: everything is a method. When a method is applied, further planning is carried out to construct a proof that an object level proof exists. This process bottoms out with the application of methods corresponding to inference rules. Hence the proof plan is a hierarchy, both in the normal 'proof tree' sense, and in that some methods can be expanded to another proof plan. The architecture allows planning and proving to be interleaved, rather than planning being done in advance. This lets ΩMEGA to recover after forming faulty plans which have no corresponding proof.

Another important difference from CLAM is the system's division of preconditions into declarative and procedural aspects, as well method slots for posting constraints, and the use of constraint reasoning [Mel96].

2.5 Feature Interactions in telecommunication systems

As hopefully already conveyed by this very Chapter, **FOLTL** is not a very widely spread language for formalising and mechanically solving problems. On one hand, it

is far too complex to be tackled by the vast majority of existing tools (in many cases, it has to be remarked, tools are specialised on less expressive languages); on the other, it seems that the community is so far quite happy with the results obtained by working, at most, in propositional temporal logics.

The ideal case-study for using **FOLTL** should have at least the following characteristics: *(i)* it should require first-order temporal logic, rather than *LTL*, in order to be fully formalised and analysed; *(ii)* it should be relevant from the academic and/or the industrial point of view. Possibly, it also should have a clear and intuitive meaning, in order to maintain a fairly simple global view of the problem, and to ease the presentation.

The problem of Feature Interactions in Telecommunication System (FIs) fulfils these requirements. FIs is a prominent problem in early-stage formal methods. By this term we denote informally the application of mathematical validation techniques to the high-level design of a service of any kind. The advantage of early-stage formal methods principally lies in spotting and correcting bugs in the specification of a service *before* a single line of code is actually deployed.

Mainly studied, so far, in telecommunication systems, the problem arises when two or more *features*, services additional to a basic, standard service, have a conflict and misbehave. A paradigmatic example, deeply analysed in [Fel01] and mechanically verified in [CS02a] (see also Chapter 7), happens when a user of a large land phone service subscribes both to an anonymous calls rejection service and to a forwarding service. Obviously, whenever an anonymous call arrives and the callee is busy, the system has a conflict: should the anonymous call be rejected or forwarded?

The scenario is made quite complex by the fact that there are arbitrarily many users in the world, each one enjoying the so-called Basic Call Service, and that any of them may subscribe to any number of different features. Each feature alters the behaviour of a user, and complicates the interaction among users.

So far a number of approaches have tackled the FI problem with various degrees of

success, as documented in the Feature Interaction Workshop series. A remarkable survey is now available in [CKMRM02], in which the research so far is divided into three broad areas: service/software engineering, formal methods and on-line techniques.

Service engineering deals with the creation of services, that is, their specification in a semi-formal way, for instance via natural language, their development and deployment; the paper focuses on software engineering, that is, on the early stages of service engineering, identifying two major approaches (focused techniques and process models) and concluding that the results available in literature, though promising, are still quite rough; in particular, it is stated that there are, rather surprisingly, still too few papers coming from the industrial world in order to assess the applicability of software engineering to avoid feature interactions.

On-line techniques are aimed at detecting and resolving feature interaction in complex, run-time environments, at the very time they happen. There is an obvious advantage in this, namely that of working in a real-life environment rather than on a (possibly abstract) model of the telecommunications system. Nevertheless, the authors' conclusion is that the difficulties arising in on-line feature detection are still hard; so hard, in fact, that most resolution methods boil down to the termination of the call originating the interaction — an indeed effective but user-annoying solution. More complex algorithms can become too expensive. Moreover, on-line techniques often critically depend on the network and its characteristics, so that change in the network architecture led to a forced redesign of the methods. Again, no relevant industrial-strength study is available at the time of writing.

Focusing on *formal methods* for Feature Interactions, which is also one of the objectives of this thesis, their application is restrained to the off-line detection of interactions via an abstract model of the system; this makes the technique independent of the actual implementation. Formal methods also force designers to think carefully about the characteristics of the service, and of the distance between what is conceived and what is actually meant by the specification; often the two things match poorly, since

abstract models imply the use of a formal (logical) language.

In general, in formal methods applied to FI, a model of the Basic Call Service and of the features is given, and then a requirement is expressed and checked against the model. More precisely, the authors split the approaches into three subcategories (we reproduce some of the references therein):

1. modelling of properties of the features and detection of interactions via unsatisfiability or inconsistency in terms of a logic. Such approaches make use of the so-called *Temporal Logic of Actions* ([Lam94]), *LTL* and first-order logic ([BJK94, JMN$^+$01, FN00]);

2. behavioural modelling via automata or transition systems and detection via specific properties such as deadlocks and nondeterminism (Finite State Machines and Automata, State Transition Diagrams and Constraint Programming) (e.g., [Blo97, AABdR00]);

3. modelling of both behaviour and properties via formal languages (*LTL*, *CTL*, TLA, Message Sequence Charts and even μ-calculus) (e.g., [CM01]).

The problem is generally stated as follows: let the notion of a feature F_1 *satisfying* a property ϕ_1, denoted $F_1 \models \phi_1$, be known, and also that $F_2 \models \phi_2$ for another feature/property pair; when the two features are somehow combined, denoted $F_1 \oplus F_2$, does $F_1 \oplus F_2 \models \phi_1 \wedge \phi_2$ hold? In most cases the question is translated in a suitable language and then a general-purpose automated reasoning tool (model checker, theorem prover, constraint solver) is used; but in nearly all cases, state-space explosion happens, and very few approaches can perform full state-space exploration, the other resorting to approximations (and leading thus to incomplete frameworks).

It is worth reminding that so far very few approaches have been given which tackle the problem in full, that is, making no approximation whatsoever on the number of entities in the world (in this case, users in the network); even in the most successful cases, potential sources of infinity are bounded or assumed small, and then finitary

techniques are applied. The notable exceptions are [FN00], where full first-order temporal logic is used for the formalisation, but then replaced by a grounded approximation[4], and [CM02a], where abstraction is used to show that, in a particular case, the approximated result holds for any number of users.

2.6 Chapter overview

In this Chapter we have surveyed the literature we consider relevant to this thesis. After a brief history of modal and temporal logics, we have introduced labelled deduction, and we have outlined how it has been successfully used in dealing with such logics. We have then introduced the verification problem and various methods employed to do automated reasoning in temporal logics; one can see the first point as the main drive for the second.

We have then outlined the paradigm of Proof Planning and how it can to improve the situation in automated reasoning in general, by defining an abstract search space in which macro-steps of intelligent (but potentially unsound) reasoning, called methods, are used to build proof plans, later on refined to proofs and then checked.

Lastly we have introduced the case-study we will be examining later on, Feature Interactions in telecommunication systems. With a few remarkable exceptions, the problem has so far been tackled by means of finitary approximations.

[4] Amy Felty was also going to study the problem without approximation, according to a personal communication received in 2001, but we have no relevant news yet.

Chapter 3

Sequent calculi for quantified modal logics

In this Chapter we aim to give a systematic presentation of Quantified Modal Logics (QMLs): uniform, intuitive, clear and complete for a class of QMLs as large as possible. For this we devise a family of labelled sequent calculi for QMLs (limited to constant domains and rigid designators) which captures all logics whose frame properties can be expressed as a finite set of first-order sentences, with no restriction whatsoever on their shape, and possibly employing equality.

Notwithstanding this generality, our sequent calculi retain some remarkable properties:

Modularity All calculi consist of a fixed base calculus for the weakest QML **QK**, plus one sequent rule for each first-order sentence expressing a property of the frame. This, together with the use of labels, makes the presentation clear and intuitive. In case the property of the frame requires equality, a few additional rules are added and modularity is retained;

Uniformity Each added rule is clearly related to the property it models, e.g., there is a rule for reflexivity, one for transitivity, etc. This avoids the need for rules

obscurely enforcing frame properties, as is usually the case in unlabelled presentations;

Normalisation All calculi are normalising, in that the rules which model frame properties can be used just at the top of the proof tree without loss of completeness, therefore simplifying the presentation and potentially aiding automated deduction;

Soundness / completeness All calculi are proved sound and complete with respect to their classes of frames; the proof of soundness and completeness is uniform, in that it is parametrised over the frame axioms.

With respect to this very Chapter, is it worth making some further remarks about related work. It is indeed not the first time that a labelled presentation of a wide spectrum of QMLs is given; the most remarkable piece of work so far is due to Viganò ([Vig00]), who has given labelled Natural Deduction systems and sequent calculi for a wide set of QMLs. His systems are sound, complete and normalising for all QMLs whose frame properties can be axiomatised by first-order Horn clauses without equality (the so-called *relational theories*).

Here we extend Viganò's work by giving sound, complete and normalising sequent calculi for all first-order axiomatise QMLs, with no restriction on the shape of frame axioms and possibly employing equality between worlds; moreover, we employ a different way of proving soundness and completeness of such systems. It is worth remarking that Viganò's choice of restricting to relational theories is dictated exactly by the necessity of keeping a normalisation property to his systems (see his Theorem 4.3.7 and subsequent discussion in [Vig00]); our systems retain soundness, completeness and normalisation for a much wider set of QMLs. For example, no normalising system for **QS4**.**3** is given in [Vig00], whereas our system $\mathcal{C}_{\mathbf{QS4.3}}$ is sound and complete exactly for **QS4**.**3**, and retains the normalisation property discussed in Subsection 3.2.4.

The choice of labelled deduction is motivated by at least three reasons: *(i)* the explicit use of labels makes the presentation much more intuitive, in that it generates uniform sequent systems, *(ii)* it helps to keep reasoning about the properties of the frame separate from reasoning about logical formulae, thus potentially aiding automated deduction, *(iii)* in the quantified case, in which we are interested, it gives rise to systems which can be inherently more powerful than unlabelled ones: see for instance [Ghi91], in which several unlabelled QMLs are proved incomplete with respect to their Kripke semantics.

From now on we will indicate *Kripke*-soundness and completeness just by the terms *soundness* and *completeness*.

The Chapter is structured as follows: in Section 3.1 some preliminaries are given about the language of our logics, proof theory and QMLs; in Section 3.2 our sequent calculi are defined, and their benefits, *in primis* their normalisation property, are discussed; and lastly, in Section 3.3 their soundness and completeness are stated and proved.

3.1 Preliminaries

In this Section we outline *(i)* the syntax of the language we will be using throughout the paper, *(ii)* the semantics of the logics generated by such language, *(iii)* a broad classification of QMLs, *(iv)* the basics of sequents and sequent calculi.

3.1.1 Syntax of the language

The syntax we present is standard in labelled deduction (see, e.g., [Gab96]). Let $\mathcal{V}$, $\mathcal{F}$, $\mathcal{P}$, $\mathcal{V}'$ and $\mathcal{F}'$ be nonempty pairwise disjoint sets; then

Definition 1 (Formulae) Logical terms *(***lt***)*, logical atoms *(***la***)*, logical formulae *(***lf***)*, labels *(***lab***)*, constraints *(***cst***) and* labelled formulae *(***labf***) are defined according to the following grammar:*

$$
\begin{array}{rcll}
\mathbf{lt} & ::= & x \mid f(\mathbf{lt}_1,\ldots,\mathbf{lt}_n) & \text{where } x \in \mathcal{V}\,, f \in \mathcal{F}\,, n \geq 0 \\
\mathbf{la} & ::= & p(\mathbf{lt}_1,\ldots,\mathbf{lt}_m) & \text{where } p \in \mathcal{P}\,, m \geq 0 \\
\mathbf{lf} & ::= & \mathbf{la} \mid \neg\mathbf{lf} \mid \mathbf{lf} \supset \mathbf{lf} \mid \forall x.\mathbf{lf} \mid \Box\mathbf{lf} & \text{where } x \in \mathcal{V} \\
\mathbf{lab} & ::= & 0 \mid t \mid g(\mathbf{lab}_1,\ldots,\mathbf{lab}_l) & \text{where } t \in \mathcal{V}',0, g \in \mathcal{F}',l \geq 0 \\
\mathbf{cst} & ::= & \mathbf{lab} \prec \mathbf{lab} \mid \mathbf{lab} \doteq \mathbf{lab} & \\
\mathbf{labf} & ::= & \mathbf{lf} \mathbin{@} \mathbf{lab} &
\end{array}
$$

Labelled formulae and constraints are collectively called formulae *and their set is denoted by* **forms**.

Other connectives such as $\wedge$, $\vee$, $\leftrightarrow$, $\exists$ and $\Diamond$ are defined from the above ones in the usual way, e.g., $\exists$ is $\neg\forall\neg$, $\Diamond$ is $\neg\Box\neg$ and so on. Also, a standard notion of *free variables* of a formula is assumed, and formulae with no free variables are called *sentences*. Lastly, we will employ a standard notion of sub-formulae of a formula and of a set of formulae.

Examples of logical formulae are: $\forall x.\Diamond\exists y.r(x,y)$, $\Box p(a)$ and $\Box(p_1 \wedge p_2) \leftrightarrow (\Box p_1 \wedge \Box p_2)$, where $p, r, p_1, p_2 \in \mathcal{P}$, $a \in \mathcal{F}$ and $x, y \in \mathcal{V}$; examples of constraints are $\tau_1 \prec \tau_2$ and $\tau \doteq \tau'$ where $\tau, \tau', \tau_1, \tau_2$ are labels; examples of labelled formulae are $p(a)$ @ 0, $p_1 \wedge p_2$ @ τ and $\forall x.p(x) \supset p(a)$ @ τ'. The @ operator is intended to bind less tightly than any other operator; the last example, for instance, means $\forall x.p(x) \supset p(a)$ holds at the world denoted by τ'.

3.1.2 Semantics and validity

We present a semantics which is largely based upon that given in [AM90]. See also, e.g., [CG92].

Definition 2 (Structure) *We call a* structure *a tuple* $\mathbf{M} = \langle \mathcal{W}, R, \mathcal{D}, I \rangle$ *where:*

- *the* set of possible worlds $\mathcal{W}$ *is a nonempty set, containing at least a distinguished element usually denoted by the symbol* 0*;*

- *the* accessibility relation $R \subseteq \mathcal{W} \times \mathcal{W}$ *is a binary relation over* $\mathcal{W}$ *;*
- *the* domain of quantification $\mathcal{D}$ *is a nonempty set disjoint from* $\mathcal{W}$ *;*
- *the* interpretation I *maps each world* $w \in \mathcal{W}$ *and predicate symbol* $p \in \mathcal{P}$ *to a predicate* $I(w,p)$ *over* $\mathcal{D}$*, and each function symbol* $f \in \mathcal{F}$ *to a* $\mathcal{D}$*-valued function* $I(f)$.

(notice that we include the interpretation I in the structure. Although slightly non standard, this makes the presentation easier to read). As is usual in modal logics, we will say that a structure has a property if and only if R in the structure has the property; for example, we will say that a structure is reflexive if and only if the associated R is reflexive, and so on. Note that, due to this semantics, the logics we consider have constant domains (i.e., the domain of quantification $\mathcal{D}$ is the same in all possible worlds) and rigid designators (i.e., the only "dynamic" objects are predicates).

Some more definitions: $\langle \mathcal{W}, R, \mathcal{D} \rangle$ is the *frame* on which the structure $\mathbf{M}$ is based. An *assignment* α is a function mapping variable symbols in $\mathcal{V}$ to values in $\mathcal{D}$. The assignment $\alpha^{[d/x]}$ assigns $d \in \mathcal{D}$ to x, leaving all the other symbols as in α. The *denotation* of a logical term s in the structure $\mathbf{M}$ w.r.t. α, written $s^{\mathbf{M},\alpha}$, is recursively defined as follows: if s is $v \in \mathcal{V}$, then $s^{\mathbf{M},\alpha} = \alpha(v)$; if s is $f(s_1,\ldots,s_n)$, then $s^{\mathbf{M},\alpha} = I(f)(s_1^{\mathbf{M},\alpha},\ldots,s_n^{\mathbf{M},\alpha})$.

To give a semantics to labels and constraints, we first introduce a further interpretation I_l mapping $\prec$ to R, $\doteq$ to the equality relation, 0 to the distinguished element in $\mathcal{W}$ and function symbols in $\mathcal{F}'$ to $\mathcal{W}$-valued functions; then we introduce a further assignment α_l mapping variable symbols in $\mathcal{V}'$ to elements of $\mathcal{W}$. The denotation of labels is analogous to that of logical terms (*de facto*, labels are terms of the labelling language): if τ is $t \in \mathcal{V}'$, then $\tau^{I_l,\alpha_l} = \alpha_l(t)$; if τ is $g(\tau_1,\ldots,\tau_n)$, then $\tau^{I_l,\alpha_l} = I_l(g)(\tau_1^{I_l,\alpha_l},\ldots,\tau_n^{I_l,\alpha_l})$. To ease the notation, we refer to elements of $\mathcal{W}$ with the letter w possibly decorated, and intend that $w, w_i, w', \ldots$ are the objects denoted by labels $\tau, \tau_i, \tau', \ldots$ That is, for example, $w = \tau^{I_l,\alpha_l}$.

Definition 3 (Truth in a structure) *A formula* φ *is* true in a structure $\mathbf{M}$ under the assignment α, *written* $\mathbf{M}, \alpha \models \varphi$, *if and only if:*

$$
\begin{array}{lll}
\mathbf{M}, \alpha \models \tau_1 \prec \tau_2 & \textit{iff} & (w_1, w_2) \in R \\
\mathbf{M}, \alpha \models \tau_1 \doteq \tau_2 & \textit{iff} & w_1 = w_2 \\
\mathbf{M}, \alpha \models p(s_1, \ldots, s_n) \mathbin{@} \tau & \textit{iff} & (s_1^{\mathbf{M},\alpha}, \ldots, s_n^{\mathbf{M},\alpha}) \in I(w, p) \\
\mathbf{M}, \alpha \models \neg\varphi \mathbin{@} \tau & \textit{iff} & \textit{not} \quad \mathbf{M}, \alpha \models \varphi \mathbin{@} \tau \\
\mathbf{M}, \alpha \models \varphi \supset \psi \mathbin{@} \tau & \textit{iff} & \textit{if} \quad \mathbf{M}, \alpha \models \varphi \mathbin{@} \tau \quad \textit{then} \quad \mathbf{M}, \alpha \models \psi \mathbin{@} \tau \\
\mathbf{M}, \alpha \models \forall x.\varphi \mathbin{@} \tau & \textit{iff} & \textit{for all } d \in \mathcal{D}\,, \quad \mathbf{M}, \alpha^{[d/x]} \models \varphi \mathbin{@} \tau \\
\mathbf{M}, \alpha \models \Box\varphi \mathbin{@} \tau & \textit{iff} & \textit{for all } w \in \mathcal{W} \quad \textit{and } t \in \mathcal{T}\,', \textit{ if } \alpha_l(t) = w \textit{ then} \\
 & & \quad \textit{if} \quad \mathbf{M}, \alpha \models \tau \prec t \quad \textit{then} \quad \mathbf{M}, \alpha \models \varphi \mathbin{@} t
\end{array}
$$

If a formula φ is true in $\mathbf{M}$ under any assignment α, we say that the structure $\mathbf{M}$ is a *model* for φ, and that φ is *true in the structure (model)* $\mathbf{M}$, written $\mathbf{M} \models \varphi$. Note that truth of a sentence is independent of the assignment.

If a formula φ is true in all structures based on a frame $\mathbf{F}$, we say it is *valid on the frame* $\mathbf{F}$, written $\mathbf{F} \models \varphi$. Lastly, if it is valid on all frames belonging to a class of frames C, we say it is *valid on the class of frames* C and write $\models^C \varphi$. In particular, when a modal logic **QL** is known to correspond to a class of frames, we write $\models^{\mathbf{QL}} \varphi$. So, for instance, $\models^{\mathbf{QS4}} \varphi$ means that φ is valid on the class of transitive, reflexive frames, and so on.

3.1.3 Quantified modal logics

We will refer to QMLs with constant domains and rigid designators simply as QMLs or "logics" and will denote them as **QK**, **QT** and so on. A thorough classification of their names, properties and characteristic axioms can be found, e.g., in [CH95]. In the same book one can see that QMLs are usually organised in a hierarchy, in which **QK** is the weakest one (Table 1 in [CH95]).

A relevant subset of them is characterised by classes of frames enjoying a set of properties which are expressible as a finite set of first-order sentences, possibly involv-

ing equivalence; Table 3.1 lists some of these properties, along with their corresponding names and first-order sentences[1]. Note that these sentences are naturally expressed in our language of labels.

We will call such logics *FO-axiomatisable* and indicate them generically as **QL**; we will say that the sentences which express their frame properties *axiomatise* them, and denote the set of those sentences as $\mathsf{FrmAx}(\mathbf{QL})$. If any of the sentences in $\mathsf{FrmAx}(\mathbf{QL})$ contains the symbol $\doteq$, we will say **QL** is a QML *with equality*, otherwise when necessary we will specify *without equality*.

3.1.4 Sequent calculi and provability

We give now some basic definitions, uniform with [TS96], Subsection 3.1. From now on, let Γ and Δ be finite multisets of formulae; when referring to the elements of Γ and Δ we will use the Greek letters $\{\gamma_1,\ldots,\gamma_l\}, l \geq 0$ and $\Delta = \{\delta_1,\ldots,\delta_m\}, m \geq 0$, with the assumption that they are *placeholders* for formula, rather than formulae.

Then a *sequent* is an expression $\Gamma \longrightarrow \Delta$. The γ_is are called *antecedents* and are intended conjunctively, while the δ_is are called *consequents* and are intended disjunctively; the sequent symbol can be read as a logical implication. Definition 3 and following are therefore straightforwardly extended to sequents: for any possible instantiation of the γs and δs, $\mathbf{M},\alpha \models \Gamma \longrightarrow \Delta$ if and only if $\mathbf{M},\alpha \models \gamma_1 \wedge \ldots \wedge \gamma_l \supset \delta_1 \vee \ldots \vee \delta_m$.

Let $n \geq 0$; then a *sequent rule* ρ is a pair (set of sequents, sequent), written

$$\frac{\Gamma_1 \longrightarrow \Delta_1 \quad \cdots \quad \Gamma_n \longrightarrow \Delta_n}{\Gamma \longrightarrow \Delta}\ \rho$$

where the $\Gamma_i \longrightarrow \Delta_i$'s are called *premises* and $\Gamma \longrightarrow \Delta$ is the *conclusion* of the rule. In displaying a sequent rule, generally, we highlight one formula in the conclusion (the *main* formula), and one or more formulae in each premise (the *active* formulae). The intuition is that the active formulae are introduced in the premises by manipulating the

[1] sentences and names of the properties are uniform with [Gol92], Chapter 1, except for the strong versions of weak density, directedness and connectedness, which are obtained by simply removing the antecedents of the implications, and atomicity, defined, e.g., in [van84].

Property (name)	Corresponding sentence
Seriality (D)	$\forall t \exists t'.t \prec t'$
Reflexivity (T)	$\forall t.t \prec t$
Irreflexivity	$\forall t.\neg\, t \prec t$
Symmetry (5)	$\forall t_0 t_1.t_0 \prec t_1 \supset t_1 \prec t_0$
Asymmetry	$\forall t_0 t_1.t_0 \prec t_1 \supset \neg\, t_1 \prec t_0$
Antisymmetry	$\forall t_0 t_1.(t_0 \prec t_1 \wedge t_1 \prec t_0) \supset t_0 \doteq t_1$
Transitivity (4)	$\forall t_0 t_1 t_2.(t_0 \prec t_1 \wedge t_1 \prec t_2) \supset t_0 \prec t_2$
Weak density	$\forall t_0 t_1.t_0 \prec t_1 \supset \exists t'.t_0 \prec t' \wedge t' \prec t_1$
Strong density	$\forall t_0 t_1 \exists t'.t_0 \prec t' \wedge t' \prec t_1$
Weak directedness (2)	$\forall t_0 t_1 t_2.(t_0 \prec t_1 \wedge t_0 \prec t_2) \supset \exists t'.t_1 \prec t' \wedge t_2 \prec t'$
Strong directedness	$\forall t_1 t_2 \exists t'.t_1 \prec t' \wedge t_2 \prec t'$
Weak connectedness (3)	$\forall t_0 t_1 t_2.(t_0 \prec t_1 \wedge t_0 \prec t_2) \supset (t_1 \prec t_2 \vee t_1 \doteq t_2 \vee t_2 \prec t_1)$
Strong connectedness	$\forall t_1 t_2.t_1 \prec t_2 \vee t_1 \doteq t_2 \vee t_2 \prec t_1$
Atomicity	$\forall t_1 \exists t'.t_1 \prec t' \wedge \forall t_2.t' \prec t_2 \supset t' \doteq t_2$

Table 3.1: properties of the accessibility relation as first-order sentences.

main formula via the sequent rule. We will use the term *frame rules* for rules whose active formulae are constraints, and *closing rules* for rules which have no premises. All other rules will be called *logical*.

A *sequent calculus* is a set of sequent rules. Recall that, since the γs and δs are placeholders for formulae, a sequent rule is really a *schema*, instantiated every time it appears in a derivation; an instance of a rule is then called an *inference*. For a more formal treatment of this concept, see, e.g., [Kan63] or the seminal [Gen35].

Assume a standard definition of *tree* (see, e.g., Subsection 2.2 of [Gal86]) and let $\mathcal{C}$ be a sequent calculus; then a *$\mathcal{C}$-derivation* of $\Gamma \longrightarrow \Delta$ is a tree in which every node

N_i is labelled with a pair $\langle \rho_i, \Gamma_i \longrightarrow \Delta_i \rangle$, where $\rho_i \in \mathcal{C}$, and has n children, where n is the number of premises of ρ_i. The root node is labelled by $\Gamma \longrightarrow \Delta$ and the leaves have no labelling rule. Slightly abusing the language, we will say that N_i is labelled by ρ_i, by $\Gamma_i \longrightarrow \Delta_i$, or by a formula ϕ_i, if ϕ_i is main in ρ_i.

A *branch* of a derivation is a tuple of nodes $(N_1, \ldots, N_k)$ such that *(i)* N_1 is the root of the derivation, *(ii)* N_{i+1} is a child of N_i for all $i = 1, \ldots, N_{k-1}$ and *(iii)* N_k is a leaf of the derivation. A *closed* branch is a branch in which N_k is labelled by a closing rule. A *closed* $\mathcal{C}$-derivation of a sequent $\Gamma \longrightarrow \Delta$ (also called a $\mathcal{C}$*-proof* of $\Gamma \longrightarrow \Delta$) is a $\mathcal{C}$-derivation of $\Gamma \longrightarrow \Delta$ and whose branches are all closed.

Definition 4 (Provability) *If* $\Gamma \longrightarrow \Delta$ *has a* $\mathcal{C}$*-proof, we write*

$$\vdash_{\mathcal{C}} \Gamma \longrightarrow \Delta$$

and say that $\Gamma \longrightarrow \Delta$ *is* provable *in* $\mathcal{C}$ *(it is* $\mathcal{C}$-provable*), or that* $\Gamma \longrightarrow \Delta$ *is a* theorem *of* $\mathcal{C}$ *(it is a* $\mathcal{C}$-theorem*).*

Two proofs will be called *similar* if and only if they prove the same sequent. Proof trees will be displayed, as is customary in Computer Science, with the root at the bottom, labelled by the sequent we want to prove — proof trees develop upward.

3.2 Sequent calculi for QMLs

In this Section we introduce and develop $\mathcal{C}_{\mathbf{QK}}$, a sequent calculus for **QK**; then a general procedure for strengthening $\mathcal{C}_{\mathbf{QK}}$ is outlined: first to QMLs without equality and then to all QMLs. A short discussion follows.

3.2.1 $\mathcal{C}_{\mathbf{QK}}$: a sequent calculus for QK

Assume from now on a standard definition of *substitution* of a variable in an expression E (formula, multi-set of formulae, sequent) as presented in [DV01], denoted $E[s/x]$

where s is a logical term or a label and x is, in turn, in ν or in ν'; then

Definition 5 ($c_{\mathbf{QK}}$) *Let $A \in$ **forms**, τ, τ' be labels, φ, ψ logical formulae and c a logical term; moreover, let $a \in \nu$ and $t' \in \nu'$; then $c_{\mathbf{QK}}$, a sequent calculus for* **QK**, *is visible in Table 3.2.*

Closing rule

$$\frac{}{\Gamma, A \longrightarrow A, \Delta}\ \mathsf{ax}$$

Logical rules

$$\frac{\Gamma \longrightarrow \varphi @ \tau, \Delta}{\Gamma, \neg\varphi @ \tau \longrightarrow \Delta}\ l\neg \qquad \frac{\Gamma, \varphi @ \tau \longrightarrow \Delta}{\Gamma \longrightarrow \neg\varphi @ \tau, \Delta}\ r\neg$$

$$\frac{\Gamma, \psi @ \tau \longrightarrow \Delta \quad \Gamma \longrightarrow \varphi @ \tau, \Delta}{\Gamma, \varphi \supset \psi @ \tau \longrightarrow \Delta}\ l\supset \qquad \frac{\Gamma, \varphi @ \tau \longrightarrow \psi @ \tau, \Delta}{\Gamma \longrightarrow \varphi \supset \psi @ \tau, \Delta}\ r\supset$$

$$\frac{\Gamma, \forall x.\varphi @ \tau, \varphi[c/x] @ \tau \longrightarrow \Delta}{\Gamma, \forall x.\varphi @ \tau \longrightarrow \Delta}\ l\forall \qquad \frac{\Gamma \longrightarrow \varphi[a/x] @ \tau, \Delta}{\Gamma \longrightarrow \forall x.\varphi @ \tau, \Delta}\ r\forall$$

$$\frac{\Gamma, \Box\varphi @ \tau, \varphi @ \tau' \longrightarrow \Delta \quad \Gamma, \Box\varphi @ \tau \longrightarrow \tau \prec \tau', \Delta}{\Gamma, \Box\varphi @ \tau \longrightarrow \Delta}\ l\Box \qquad \frac{\Gamma, \tau \prec t' \longrightarrow \varphi @ t', \Delta}{\Gamma \longrightarrow \Box\varphi @ \tau, \Delta}\ r\Box$$

Table 3.2: the calculus $c_{\mathbf{QK}}$ for **QK**. $A \in$ **forms**, τ, τ' are labels, φ, ψ logical formulae and c a logical term. $a \in \nu$ and $t' \in \nu'$ cannot appear free in the conclusion of $r\forall$ and $r\Box$.

$c_{\mathbf{QK}}$ is a variant of Gentzen's sequent calculus *LK* for classical logic ([Gen35]), except that

1. it is presented with no structural rules, but with a generalised closing rule and duplication of the main formula in $l\forall$ and $l\Box$ (by analogy, for instance, with system *G* in [Gal86], Definition 5.4.1);

2. it has two rules $r\Box$ and $l\Box$ for the $\Box$ operator, intuitively reflecting its semantics;

3. it is restricted to a minimal subset of operators ($\neg$, $\supset$, $\forall$ and $\Box$), with the assumption that rules for derived operators, such as $\exists$ and $\Diamond$, can be used here and there. They are obtained straightforwardly by composing rules in $\mathcal{C}_{\mathbf{QK}}$; for instance, $l\Diamond$ is obtained by considering the top and bottom sequents of the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \tau \prec t', \varphi \mathbin{@} t' \longrightarrow \Delta}{\Gamma, \tau \prec t' \longrightarrow \neg\varphi \mathbin{@} t', \Delta}\, r\neg}{\Gamma \longrightarrow \Box\neg\varphi \mathbin{@} \tau, \Delta}\, r\Box}{\Gamma, \neg\Box\neg\varphi \mathbin{@} \tau \longrightarrow \Delta}\, l\neg}{\Gamma, \Diamond\varphi \mathbin{@} \tau \longrightarrow \Delta}\ (\text{definition of } \Diamond)$$

It is possible to prove in $\mathcal{C}_{\mathbf{QK}}$ a number of characteristic axioms of **QK**; as an example, in Figure 3.1 we sketch the $\mathcal{C}_{\mathbf{QK}}$-proof of *Modal Modus Ponens*, and in Figure 3.2 the $\mathcal{C}_{\mathbf{QK}}$-proof of the *Converse Barcan Formula*, characteristic of quantified modal logics with constant domains[2]. Also, the *Rule of Necessitation* is naturally enforced: for any logical formula φ and label τ, if $\vdash_{\mathcal{C}_{\mathbf{QK}}} \varphi \mathbin{@} \tau$ then $\vdash_{\mathcal{C}_{\mathbf{QK}}} \Box\varphi \mathbin{@} \tau$. This can be easily shown by structural induction.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\psi \mathbin{@} t' \longrightarrow \psi \mathbin{@} t'}\,\mathsf{ax} \quad \dfrac{}{\varphi \mathbin{@} t' \longrightarrow \varphi \mathbin{@} t'}\,\mathsf{ax}}{\varphi \mathbin{@} t', \varphi \supset \psi \mathbin{@} t' \longrightarrow \psi \mathbin{@} t'}\, l\supset \quad \dfrac{}{0 \prec t' \longrightarrow 0 \prec t'}\,\mathsf{ax}}{0 \prec t', \varphi \mathbin{@} t', \Box(\varphi \supset \psi) \mathbin{@} 0 \longrightarrow \psi \mathbin{@} t'}\, l\Box \quad \dfrac{}{0 \prec t' \longrightarrow 0 \prec t'}\,\mathsf{ax}}{0 \prec t', \Box\varphi \mathbin{@} 0, \Box(\varphi \supset \psi) \mathbin{@} 0 \longrightarrow \psi \mathbin{@} t'}\, l\Box}{\Box\varphi \mathbin{@} 0, \Box(\varphi \supset \psi) \mathbin{@} 0 \longrightarrow \Box\psi \mathbin{@} 0}\, r\Box}{\Box\varphi \wedge \Box(\varphi \supset \psi) \mathbin{@} 0 \longrightarrow \Box\psi \mathbin{@} 0}\, l\wedge}{\longrightarrow \Box\varphi \wedge \Box(\varphi \supset \psi) \supset \Box\psi \mathbin{@} 0}\, r\supset$$

Figure 3.1: a $\mathcal{C}_{\mathbf{QK}}$-proof of Modal Modus Ponens.

[2] all proof sketches, although some unessential formulae may be omitted here and there for the sake of conciseness, are completely rigorous. Especially, we will leave out the copy of the main formula in the premises of rules $l\forall$ and $l\Box$, when they are not necessary.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(a)\ @\ t' \longrightarrow p(a)@t'}\ \mathsf{ax}}{\forall x.p(x)\ @\ t' \longrightarrow p(a)@t'}\ l\forall \quad \dfrac{}{0 \prec t' \longrightarrow 0 \prec t'}\ \mathsf{ax}}{0 \prec t', \Box\forall x.p(x)\ @\ 0 \longrightarrow p(a)@t'}\ l\Box}{\Box\forall x.p(x)\ @\ 0 \longrightarrow \Box p(a)@0}\ r\Box}{\Box\forall x.p(x)\ @\ 0 \longrightarrow \forall x.\Box p(x)@0}\ r\forall}{\longrightarrow \Box\forall x.p(x) \supset \forall x.\Box p(x)@0}\ r\supset$$

Figure 3.2: a $\mathcal{C}_{\mathbf{QK}}$-proof of the Converse Barcan Formula.

3.2.2 Sequent calculi for QMLs without equality

Assume standard notions of *prenex normal form* and *Skolemization* of a first-order formula (see, e.g., [Sho70]); then we introduce the following procedure which builds a sequent rule out of a first-order sentence:

Definition 6 (Strengthening) *Let ϕ be a first-order sentence in the language of labels not containing the equality symbol; then the* strengthening procedure*, yielding sequent rule $\mathsf{Str}(\phi)$, is defined as follows:*

1. *convert ϕ into prenex normal form and skolemize; call the new sentence ϕ^S;*
2. *add to $\mathcal{F}'$ the Skolem function(s) introduced at the previous step;*
3. *build a 2LK-derivation of $\Gamma, \phi^S \longrightarrow \Delta$ (see Table 3.5 in Subsection 3.3.1) in which every sequent labelling a leaf contains only constraints, Γ or Δ; when using rule $l\forall^*_\theta$, avoid copying the main formula into the premise;*
4. *finally, let $\Gamma \longrightarrow \Delta$ be the conclusion of $\mathsf{Str}(\phi)$, and let the leaves of the derivation obtained at the previous step be its premises.*

Note that rules obtained by the strengthening procedure are frame rules. This is appropriate, since they express properties of the accessibility relation and, in turn, of the frame. As already noted, when displayed in sequent calculi, rules are schemes; we will denote the placeholders by the letter τ, possibly decorated, meaning any label.

As an example of the strengthening procedure, consider a sentence not involving equality in Table 3.1, for instance 2; in order to obtain $\mathsf{Str}(2)$, first build its prenex normal form and skolemize:

$$2^S = \forall t_0 t_1 t_2.(t_0 \prec t_1 \wedge t_0 \prec t_2) \supset (t_1 \prec cv(t_0,t_1,t_2) \wedge t_2 \prec cv(t_0,t_1,t_2))$$

Let then $\mathcal{F}' = \{cv\}$; now "unfold" 2^S as shown in Figure 3.3; finally, build $\mathsf{Str}(2)$ by taking $\Gamma \longrightarrow \Delta$ as the conclusion, and the leaves of the proof tree in Figure 3.3 as the premises.

$$\dfrac{\dfrac{\dfrac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma \longrightarrow \tau_0 \prec \tau_2, \Delta}{\Gamma, \longrightarrow \tau_0 \prec \tau_1 \wedge \tau_0 \prec \tau_2, \Delta}\, r\wedge^* \quad \dfrac{\Gamma, \tau_1 \prec cv(\tau_0,\tau_1,\tau_2), \tau_2 \prec cv(\tau_0,\tau_1,\tau_2) \longrightarrow \Delta}{\Gamma, \tau_1 \prec cv(\tau_0,\tau_1,\tau_2) \wedge \tau_2 \prec cv(\tau_0,\tau_1,\tau_2) \longrightarrow \Delta}\, l\wedge^*}{\Gamma, (\tau_0 \prec \tau_1 \wedge \tau_0 \prec \tau_2) \supset \tau_1 \prec cv(\tau_0,\tau_1,\tau_2) \wedge \tau_2 \prec cv(\tau_0,\tau_1,\tau_2) \longrightarrow \Delta}\, l\supset^*}{\Gamma, \forall t_0 t_1 t_2.(t_0 \prec t_1 \wedge t_0 \prec t_2) \supset t_1 \prec cv(t_0,t_1,t_2) \wedge t_2 \prec cv(t_0,t_1,t_2) \longrightarrow \Delta}\, l\forall^*_\theta l\forall^*_\theta l\forall^*_\theta$$

Figure 3.3: application of Step 3 of the strengthening procedure to the sentence 2^S. The leaves of this derivation are the premises of rule $\mathsf{Str}(2)$, called wdir and visible in Table 3.4.

Note that there is no restriction on the shape of ϕ, except that it must not contain the equality symbol so far. Skolemization at Step 1 is carried on in a completely standard way.

A central issue is that

Proposition 7 (Termination of strengthening) *The strengthening procedure is terminating.*

Proof:

It suffices to show that every step of the procedure is terminating. Obviously, the only non-trivially terminating step is Step 3: for this, note that every application of a $2LK$ rule to ϕ^S reduces the number of connectives in it (provided that copying the main formula in $l\forall^*_\theta$ is forbidden), eventually leading to a set of sequents which contain only constraints.

•

The restriction on the use of duplicate formulae in rule $l\forall^*_\theta$ at Step 3 of the procedure is necessary in order to guarantee termination of the strengthening procedure.

For any QML without equality **QL**, let $\mathsf{Str}(\mathbf{QL})$ be the sequent calculus obtained by strengthening the sentences in $\mathsf{FrmAx}(\mathbf{QL})$, that is $\mathsf{Str}(\mathbf{QL}) = \{\rho \mid \rho = \mathsf{Str}(\phi), \phi \in \mathsf{FrmAx}(\mathbf{QL})\}$. Then a sequent calculus for **QL** can be built by taking $\mathcal{C}_{\mathbf{QK}} \cup \mathsf{Str}(\mathbf{QL})$.

This calculus is *modular*, in that it is obtained by adding to the (unchanged) basic calculus $\mathcal{C}_{\mathbf{QK}}$ a set of new rules, and *uniform*, in that (as Definition 6 suggests) each sequent rule in $\mathsf{Str}(\mathbf{QL})$ is clearly and intuitively related to a first-order sentence enforcing a property of the frame.

Moreover, by Proposition 7 and the fact that $\mathsf{FrmAx}(\mathbf{QL})$ is finite, we have that calculi obtained this way are *finitary*, in that they have a finite number of rules, and each rule has a finite number of premises.

As a simple example, in Figure 3.4 we sketch the proof of axiom $\Box p \supset \Box\Box p \ @ \ 0$, characteristic of transitive frames, in $\mathcal{C}_{\mathbf{QK}} \cup \{\mathsf{trans}\}$. Rule $\mathsf{trans} = \mathsf{Str}(4)$ is visible in Table 3.4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p@t_2 \longrightarrow p@t_2}\ \mathsf{ax} \quad \dfrac{\dfrac{}{0 \prec t_1 \longrightarrow 0 \prec t_1}\ \mathsf{ax} \quad \dfrac{}{t_1 \prec t_2 \longrightarrow t_1 \prec t_2}\ \mathsf{ax} \quad \dfrac{}{0 \prec t_2 \longrightarrow 0 \prec t_2}\ \mathsf{ax}}{0 \prec t_1, t_1 \prec t_2 \longrightarrow 0 \prec t_2}\ \mathsf{trans}}{0 \prec t_1, t_1 \prec t_2, \Box p@0 \longrightarrow p@t_2}\ l\Box}{0 \prec t_1, \Box p@0 \longrightarrow \Box p@t_1}\ r\Box}{\Box p@0 \longrightarrow \Box\Box p@0}\ r\Box}{\longrightarrow \Box p \supset \Box\Box p@0}\ r\supset$$

Figure 3.4: a proof of axiom $\Box p \supset \Box\Box p \ @ \ 0$, characterising transitive frames, in $\mathcal{C}_{\mathbf{QK}} \cup \{\mathsf{trans}\}$.

3.2.3 Sequent calculi for QMLs with equality

The equality symbol between labels is already present in our syntax (Definition 1) and it has a semantics (Definition 3). Let a *labelled logical atom* be a labelled formula in

which the logical formula is a logical atom; then

Definition 8 ($\mathcal{C}_{\mathbf{QK}}^{\doteq}$) *Let τ, τ' be labels and $t \in \mathcal{V}'$; then $\mathcal{C}_{\mathbf{QK}}^{\doteq}$, a sequent calculus for* **QK** *augmented for equality between labels, is the union of $\mathcal{C}_{\mathbf{QK}}$ (recall Table 3.2) and the rules visible in Table 3.3.*

Frame rules

$$\frac{}{\Gamma \longrightarrow \tau \doteq \tau, \Delta}\ \mathsf{refl}_{\doteq}$$

$$\frac{\Gamma[\tau'/t], \tau \doteq \tau' \longrightarrow \Delta[\tau'/t]}{\Gamma[\tau/t], \tau \doteq \tau' \longrightarrow \Delta[\tau/t]}\ \mathsf{sub}_{\doteq}$$

Table 3.3: rules for equality. $\mathcal{C}_{\mathbf{QK}}^{\doteq}$ is the union of these rules and $\mathcal{C}_{\mathbf{QK}}$. τ, τ' are labels and $t \in \mathcal{V}'$. In rule $\mathsf{sub}_{\doteq}$, the occurrences of τ replaced by τ' are in labelled logical atoms or constraints only.

Rules in Table 3.3 enforce basic properties of $\doteq$, for instance that assuming $\tau \doteq \tau'$, a label τ can be uniformly substituted with τ'[3]. Note that rule $\mathsf{sub}_{\doteq}$ is included in the set of frame rules although it can have active logical atoms; we choose to do this because both $\mathsf{refl}_{\doteq}$ and $\mathsf{sub}_{\doteq}$ deal with the symbol of equality, which is defined only between labels.

Definition 6 carries on straightforwardly for all QMLs (just remove the words "not containing the equality symbol" from it). The same is true for Proposition 7. For any QML **QL**, now possibly with equality, a sequent calculus for **QL** can be built by taking $\mathcal{C}_{\mathbf{QK}}^{\doteq} \cup \mathsf{Str}(\mathbf{QL})$. All properties defined and proved in the previous Subsection still hold: all calculi obtained as described above are modular, uniform and finitary.

As a non-trivial example, in Figure 3.5 we sketch the proof of axiom $\Diamond\Box p \supset \Box\Diamond p$ @ 0, characteristic of the logic of reflexive, weakly directed frames, in $\mathcal{C}_{\mathbf{QK}}^{\doteq} \cup$

[3] the restriction to labelled logical atoms and constraints is dictated by the completeness argument and will be clarified in Section 3.3.

$\{\mathsf{refl}, \mathsf{wconn}\}$. Rules $\mathsf{refl} = \mathsf{Str}(T)$ and $\mathsf{wconn} = \mathsf{Str}(3)$ are visible in Table 3.4. This proof is possible, as we expect, since the property of weak connectedness is strictly stronger than that of weak directedness. Note the use of $\doteq$.

3.2.4 The entailment rule: normalisation

Lastly, we introduce a rule which takes into account all frame rules seen so far. Let $\mathsf{FrmRl}(\mathbf{QL})$ be the union of $\mathsf{Str}(\mathbf{QL})$ and the rules in Table 3.3; then

Definition 9 ($c_{\mathbf{QL}}$ and the entailment rule) *For any logic* **QL***, let*

$$c_{\mathbf{QL}} = c^{\doteq}_{\mathbf{QK}} \cup \mathsf{ent}_{\mathbf{QL}}$$

where $\mathsf{ent}_{\mathbf{QL}}$ *(*entailment*) is the following rule:*

$$\frac{}{\Gamma \longrightarrow \Delta}\ \mathsf{ent}_{\mathbf{QL}} \qquad \textit{with } \vdash_{\mathsf{FrmRl}(\mathbf{QL})} \Gamma \longrightarrow \Delta.$$

According to the above Definition, in each $c_{\mathbf{QL}}$-proof, rule $\mathsf{ent}_{\mathbf{QL}}$ represents a subproof in which only rules in $\mathsf{FrmRl}(\mathbf{QL})$ are used. The calculi $c_{\mathbf{QL}}$, with respect to the calculi $c^{\doteq}_{\mathbf{QK}} \cup \mathsf{Str}(\mathbf{QL})$, have a restriction on the use of frame rules; since $\mathsf{ent}_{\mathbf{QL}}$ is a closing rule, it cannot be followed higher up in the tree by the application of any logical rule. In other words,

Proposition 10 (Normalisation) *Let* **QL** *be any FO-axiomatizable logic; then for every $c^{\doteq}_{\mathbf{QK}} \cup \mathsf{Str}(\mathbf{QL})$-proof there is a similar $c_{\mathbf{QL}}$-proof that is* normal*, in the sense that no logical rules are ever used above a frame rule.*

Proof: Trivial, from the facts that *(i)* no frame rules appear in any $c_{\mathbf{QL}}$, and that *(ii)* $\mathsf{ent}_{\mathbf{QL}}$ is a closing rule.

•

Again, all calculi $c_{\mathbf{QL}}$ retain the properties defined and proved in the previous Subsections: they are still modular, uniform and finitary.

$$\dfrac{\dfrac{\dfrac{}{p @ t_1 \longrightarrow p @ t_1}\,\mathsf{ax} \quad \dfrac{\dfrac{}{t_1 \prec t_1 \longrightarrow t_1 \prec t_1}\,\mathsf{ax}}{\longrightarrow t_1 \prec t_1}\,\mathsf{refl}}{\Box p @ t_1 \longrightarrow p @ t_1}\,l\Box \quad \dfrac{\dfrac{}{t_1 \prec t_1 \longrightarrow t_1 \prec t_1}\,\mathsf{ax}}{\longrightarrow t_1 \prec t_1}\,\mathsf{refl}}{\Box p @ t_1 \longrightarrow \Diamond p @ t_1}\,r\Diamond}{t_1 \doteq t_2, \Box p @ t_1 \longrightarrow \Diamond p @ t_2}\,\mathsf{sub}_{\doteq}$$

Branch $\boxed{3}$

$$\dfrac{\dfrac{\dfrac{}{p @ t_1 \longrightarrow p @ t_1}\,\mathsf{ax} \quad \dfrac{\dfrac{}{t_1 \prec t_1 \longrightarrow t_1 \prec t_1}\,\mathsf{ax}}{\longrightarrow t_1 \prec t_1}\,\mathsf{refl}}{t_2 \prec t_1, \Box p @ t_1 \longrightarrow p @ t_1}\,l\Box \quad \dfrac{}{t_2 \prec t_1 \longrightarrow t_2 \prec t_1}\,\mathsf{ax}}{t_2 \prec t_1, \Box p @ t_1 \longrightarrow \Diamond p @ t_2}\,r\Diamond$$

Branch $\boxed{2}$

$$\dfrac{\dfrac{\dfrac{}{p @ t_2 \longrightarrow p @ t_2}\,\mathsf{ax} \quad \dfrac{}{t_1 \prec t_2 \longrightarrow t_1 \prec t_2}\,\mathsf{ax}}{t_1 \prec t_2, \Box p @ t_1 \longrightarrow p @ t_2}\,l\Box \quad \dfrac{\dfrac{}{t_2 \prec t_2 \longrightarrow t_2 \prec t_2}\,\mathsf{ax}}{t_2 \prec t_2}\,\mathsf{refl}}{t_1 \prec t_2, \Box p @ t_1 \longrightarrow \Diamond p @ t_2}\,r\Diamond$$

Branch $\boxed{1}$

$$\dfrac{\dfrac{\dfrac{}{0 \prec t_1 \longrightarrow 0 \prec t_1}\,\mathsf{ax} \quad \dfrac{}{0 \prec t_2 \longrightarrow 0 \prec t_2}\,\mathsf{ax} \quad \boxed{1} \quad \boxed{2} \quad \boxed{3}}{0 \prec t_1, 0 \prec t_2, \Box p @ t_1 \longrightarrow \Diamond p @ t_2}\,\mathsf{wconn}}{0 \prec t_1, \Box p @ t_1 \longrightarrow \Box\Diamond p @ 0}\,r\Box}{\Diamond\Box p @ 0 \longrightarrow \Box\Diamond p @ 0}\,l\Diamond}{\longrightarrow \Diamond\Box p \supset \Box\Diamond p @ 0}\,r\supset$$

Bottom of the tree

Figure 3.5: a proof of axiom $\Diamond\Box p \supset \Box\Diamond p \ @\ 0$, characteristic of reflexive, weakly directed frames, in $\mathcal{C}^{\doteq}_{\mathbf{QK}} \cup \{\mathsf{refl}, \mathsf{wconn}\}$. The bottom subtree is at the root of the proof; the three subtrees above correspond to placeholders $\boxed{1}$, $\boxed{2}$ and $\boxed{3}$. Notice that this proof is *not* normal.

3.2.5 Discussion

The methodology outlined earlier on allows us to build sequent calculi for any FO-axiomatizable QML (with or without equality). As an extended example, Table 3.4 shows the rules obtained by application of the strengthening procedure to sentences in Table 3.1. We have given them mnemonic names, such as $\mathsf{refl} = \mathsf{Str}(T)$, and so on. As usual, labels in the rules of Table 3.4 are really placeholders.

Besides adding to the elegance of the presentation, modularity and uniformity are useful for the implementation of these logics. Such an implementation would indeed benefit from not having to be redone from scratch each time a new, stronger logic is needed; modularity could be reflected in modularity of the automated machinery.

Moreover, the property of normalisation reduces the search space during proof search in any $\mathcal{C}_{\mathbf{QL}}$. In principle, rule $\mathsf{ent}_{\mathbf{QL}}$ can be replaced by any reasoning method whatsoever for the first-order theory of $\mathsf{FrmAx}(\mathbf{QL})$, seen as a black box; in particular, any efficient machinery for equivalence reasoning can be employed. Normal proofs here can be seen as a generalised version of *regular* proofs in sequent calculi for logics with equality, an issue addressed, e.g., in [DV01].

Lastly, we show two more non-trivial examples. First, we recast the (non-normal) proof in Figure 3.5 in $\mathcal{C}_{\mathbf{QS4.3}}$; the result is visible in Figure 3.6, where, still, we indicate explicitly the use of frame rules, instead of using rule $\mathsf{ent}_{\mathbf{QS4.3}}$, for the sake of clarity; note however that, as expected, no logical rules are used above any frame rule, i.e., this proof *is* normal.

Second, in Figure 3.7 we show that McKinsey's axiom, $\Box\Diamond p \supset \Diamond\Box p$, characteristic of atomic frames, is provable in $\mathcal{C}_{\mathbf{QS4.1}}$, a calculus for logic **QS4.1** for which

$$\mathsf{FrmRl}(\mathbf{QS4.1}) = \{\mathsf{refl}_{\doteq}, \mathsf{sub}_{\doteq}, \mathsf{refl}, \mathsf{trans}, \mathsf{atom}\}$$

(see, e.g., [van84]). As we expect, this proof is also normal.

$$\frac{\Gamma, \tau \prec wit(\tau) \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{ser}$$

$$\frac{\Gamma, \tau \prec \tau \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{refl}$$

$$\frac{\Gamma \longrightarrow \tau \prec \tau, \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{irr}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma, \tau_1 \prec \tau_0 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{symm}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma \longrightarrow \tau_1 \prec \tau_0, \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{asymm}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma \longrightarrow \tau_1 \prec \tau_0, \Delta \quad \Gamma, \tau_0 \doteq \tau_1 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{antisymm}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma \longrightarrow \tau_1 \prec \tau_2, \Delta \quad \Gamma, \tau_0 \prec \tau_2 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{trans}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma, \tau_0 \prec hb(\tau_0, \tau_1), hb(\tau_0, \tau_1) \prec \tau_1 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{wdens}$$

$$\frac{\Gamma, \tau_0 \prec hb(\tau_0, \tau_1), hb(\tau_0, \tau_1) \prec \tau_1 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{sdens}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma \longrightarrow \tau_0 \prec \tau_2, \Delta \quad \Gamma, \tau_1 \prec cv(\tau_0, \tau_1, \tau_2), \tau_2 \prec cv(\tau_0, \tau_1, \tau_2) \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{wdir}$$

$$\frac{\Gamma, \tau_1 \prec cv(\tau_0, \tau_1, \tau_2), \tau_2 \prec cv(\tau_0, \tau_1, \tau_2) \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{sdir}$$

$$\frac{\Gamma \longrightarrow \tau_0 \prec \tau_1, \Delta \quad \Gamma \longrightarrow \tau_0 \prec \tau_2, \Delta \quad \Gamma, \tau_1 \prec \tau_2 \longrightarrow \Delta \quad \Gamma, \tau_1 \doteq \tau_2 \longrightarrow \Delta \quad \Gamma, \tau_2 \prec \tau_1 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{wconn}$$

$$\frac{\Gamma, \tau_1 \prec \tau_2 \longrightarrow \Delta \quad \Gamma, \tau_1 \doteq \tau_2 \longrightarrow \Delta \quad \Gamma, \tau_2 \prec \tau_1 \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{sconn}$$

$$\frac{\Gamma, \tau_1 \prec la(\tau_1), la(\tau_1) \doteq \tau_2 \longrightarrow \Delta \quad \Gamma, \tau_1 \prec la(\tau_1) \longrightarrow la(\tau_1) \prec \tau_2, \Delta}{\Gamma \longrightarrow \Delta}\ \mathsf{atom}$$

Table 3.4: frame rules obtained from sentences in Table 3.1 via the strengthening procedure. *wit* (the "witness" world), *hb* (the world "halfway between"), *cv* (the "convergent" world) and la (the "last" world) are Skolem functions, purposefully added to $\mathcal{F}'$ by the strengthening procedure.

3.3 Soundness and completeness

Recall Definitions 3 and 4, and let **QL** be any FO-axiomatizable QML (with or without equality); in this Section we prove that $\mathcal{C}_{\mathbf{QL}}$ is sound and complete for each **QL**, that

$$\dfrac{\dfrac{\dfrac{}{t_1 \prec t_1 \longrightarrow t_1 \prec t_1}\,\mathsf{ax}}{\longrightarrow t_1 \prec t_1}\,\mathsf{refl}}{t_1 \doteq t_2 \longrightarrow t_1 \prec t_2}\,\mathsf{sub}_{\doteq}$$

Branch $\boxed{3}$

$$\dfrac{\dfrac{}{0 \prec t_1 \longrightarrow 0 \prec t_1}\,\mathsf{ax} \quad \dfrac{}{0 \prec t_2 \longrightarrow 0 \prec t_2}\,\mathsf{ax} \quad \dfrac{}{t_1 \prec t_2 \longrightarrow t_1 \prec t_2}\,\mathsf{ax} \quad \boxed{3} \quad \dfrac{}{t_2 \prec t_1 \longrightarrow t_2 \prec t_1}\,\mathsf{ax}}{0 \prec t_1, 0 \prec t_2 \longrightarrow t_2 \prec t_1, t_1 \prec t_2}\,\mathsf{wconn}$$

Branch $\boxed{2}$

$$\dfrac{\dfrac{\dfrac{}{t_1 \prec t_1 \longrightarrow t_1 \prec t_1}\,\mathsf{ax}}{\longrightarrow t_1 \prec t_1}\,\mathsf{refl}}{}$$

Branch $\boxed{1}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p @ t_2 \longrightarrow p @ t_2}\,\mathsf{ax} \quad \dfrac{\dfrac{}{t_2 \prec t_2 \longrightarrow t_2 \prec t_2}\,\mathsf{ax}}{\longrightarrow t_2 \prec t_2}\,\mathsf{refl}}{p @ t_2 \longrightarrow \Diamond p @ t_2}\,r\Diamond \quad \dfrac{\dfrac{}{p @ t_1 \longrightarrow p @ t_1}\,\mathsf{ax} \quad \boxed{2}}{p @ t_1 \longrightarrow \Diamond p @ t_2, t_1 \prec t_2}\,r\Diamond}{0 \prec t_1, 0 \prec t_2, p @ t_1, \Box p @ t_1 \longrightarrow \Diamond p @ t_2}\,l\Box \quad \boxed{1}}{0 \prec t_1, 0 \prec t_2, \Box p @ t_1 \longrightarrow \Diamond p @ t_2}\,l\Box}{0 \prec t_1, \Box p @ t_1 \longrightarrow \Box \Diamond p @ 0}\,r\Box}{\Diamond \Box p @ 0 \longrightarrow \Box \Diamond p @ 0}\,l\Diamond}{\longrightarrow \Diamond \Box p \supset \Box \Diamond p @ 0}\,r\supset$$

Bottom of the tree

Figure 3.6: a proof of axiom $\Diamond\Box p \supset \Box\Diamond p \ @\ 0$, characteristic of reflexive, weakly directed frames, in $\mathcal{C}_{\mathbf{QS4.3}}$ — but frame rules are explicitly indicated. The bottom subtree is at the root of the proof; the subtrees above correspond to placeholders $\boxed{1}$ and $\boxed{2}$. Notice the difference with the proof in Figure 3.5, in which logical rules are used above rule wconn; this proof is actually normal.

is, whatever is $\mathcal{C}_{\mathbf{QL}}$-provable is **QL**-valid and vice-versa:

$$\dfrac{\dfrac{\dfrac{}{p \mathbin{@} t_2 \longrightarrow p \mathbin{@} t_2}\,\mathsf{ax}}{t_1 \doteq t_2, p \mathbin{@} t_1 \longrightarrow p \mathbin{@} t_2}\,\mathsf{sub}_{\doteq}}{la(0) \doteq t_1, la(0) \doteq t_2, p \mathbin{@} t_1 \longrightarrow p \mathbin{@} t_2}\,\mathsf{sub}_{\doteq}$$

Branch $\boxed{3}$

$$\dfrac{\dfrac{}{0 \prec la(0) \longrightarrow 0 \prec la(0)}\,\mathsf{ax} \quad \dfrac{}{0 \prec la(0) \longrightarrow 0 \prec la(0)}\,\mathsf{ax}}{\longrightarrow 0 \prec la(0)}\,\mathsf{atom}$$

Branch $\boxed{2}$

$$\dfrac{\dfrac{}{0 \prec la(0) \longrightarrow 0 \prec la(0)}\,\mathsf{ax} \quad \dfrac{}{0 \prec la(0) \longrightarrow 0 \prec la(0)}\,\mathsf{ax}}{\longrightarrow 0 \prec la(0)}\,\mathsf{atom}$$

Branch $\boxed{1}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\boxed{3} \quad \dfrac{}{la(0) \prec t_1 \longrightarrow la(0) \prec t_1}\,\mathsf{ax}}{la(0) \prec t_1, la(0) \doteq t_1, p \mathbin{@} t_2 \longrightarrow p \mathbin{@} t_1}\,\mathsf{atom} \quad \dfrac{}{la(0) \prec t_2 \longrightarrow la(0) \prec t_2}\,\mathsf{ax}}{la(0) \prec t_1, la(0) \prec t_2, p \mathbin{@} t_2 \longrightarrow p \mathbin{@} t_1}\,\mathsf{atom}}{la(0) \prec t_1, \Diamond p \mathbin{@} la(0) \longrightarrow p \mathbin{@} t_1}\,l\Diamond \quad \boxed{2}}{la(0) \prec t_1, \Box\Diamond p \mathbin{@} 0 \longrightarrow p \mathbin{@} t_1}\,l\Box}{\Box\Diamond p \mathbin{@} 0 \longrightarrow \Box p \mathbin{@} la(0)}\,r\Box \quad \boxed{1}}{\Box\Diamond p \mathbin{@} 0 \longrightarrow \Diamond\Box p \mathbin{@} 0}\,r\Diamond}{\longrightarrow \Box\Diamond p \supset \Diamond\Box p \mathbin{@} 0}\,r\supset$$

Bottom of the tree

Figure 3.7: a proof of McKinsey's axiom, $\Box\Diamond p \supset \Diamond\Box p \mathbin{@} 0$, characteristic of atomic frames, in $\mathcal{C}_{\mathbf{QS4.1}}$. Notice that this proof is normal, as expected. The bottom subtree is at the root of the proof; the subtrees above correspond to placeholders $\boxed{1}$, $\boxed{2}$ and $\boxed{3}$.

$$\vdash_{\mathcal{C}_{\mathbf{QL}}} \Gamma \longrightarrow \Delta \quad \text{iff} \quad \models^{\mathbf{QL}} \Gamma \longrightarrow \Delta.$$

3.3.1 Two-sorted first-order logic with equality

We now sketch two-sorted first-order logic with equality on one sort (which we call $2FOL$) and an associated sequent calculus. This machinery is needed for the proof of soundness and completeness. The following presentation is rather informal; the reader can check the details in [Gal86] and [DV01], which are the main sources of inspiration.

The vocabulary of $2FOL$ has three sets $\mathcal{V}'$, $\mathcal{F}'$ and $\mathcal{P}'$ of variable, function and predicate symbols, plus two symbols, ι and θ, called *sort symbols*; ι is the sort of individuals and θ is the sort of worlds. To each function and predicate symbol is associated an n-uple in $\{\iota,\theta\}^n$ (the *rank* of the symbol — see [Gal86], Subsection 5.2.1). Informally: the rank of a symbol associates a sort (or *type*) of each argument of the function or predicate associated with the symbol; for function symbols, it also states the type of the function itself. By default, $=_\theta \in \mathcal{P}'$ with rank (θ,θ). $=_\theta$ denotes equivalence among elements of sort θ.

The language of $2FOL$ is built out of terms and atoms into formulae by means of $\neg$, $\supset$ and $\forall$, analogously to what happens in first-order logic, but respecting the rank of each symbol.

A *structure* of $2FOL$ is a pair $\mathbf{M}' = \langle \mathcal{D}', I' \rangle$ in which $\mathcal{D}' = \mathcal{W}' \cup \mathcal{C}'$ where $\mathcal{W}'$ and $\mathcal{C}'$ are disjoint and are called *sorts*. Every term of $2FOL$ is associated via its rank to exactly one sort; we indicate this fact with the notation $t:\theta$ (if t denotes an element in $\mathcal{W}'$) or $t:\iota$ (if t denotes an element in $\mathcal{C}'$).

The *interpretation* I' maps function and predicate symbols to functions and predicates over $\mathcal{D}'$, respecting the rank of each symbol; in particular, it maps $=_\theta$ to the equality relation over $\mathcal{W}'$. An *assignment* in $2FOL$ is a function α' mapping variable symbols in $\mathcal{V}'$ to elements of either sort, depending on their rank. Given the standard notion of denotation of terms, truth of a $2FOL$ formula in $\mathbf{M}'$ under α' is the usual one for many-sorted first-order logics.

Definition 11 ($\mathcal{C}_{\mathbf{QK}}$) *Let A,B range over formulae, c_1,c_2,s,t terms and a_1,a_2 vari-*

ables of 2FOL; then 2LK, a sequent calculus for 2FOL, is visible in Table 3.5.

Closing rules

$$\frac{}{\Gamma, A \longrightarrow A, \Delta}\ ax^*$$

Rules for equality

$$\frac{}{\Gamma \longrightarrow \tau =_\theta \tau, \Delta}\ re^* \qquad \frac{\Gamma[t:\theta/x], s =_\theta t \longrightarrow \Delta[t:\theta/x]}{\Gamma[s:\theta/x], s =_\theta t \longrightarrow \Delta[s:\theta/x]}\ sub^*$$

Logical rules

$$\frac{\Gamma \longrightarrow A, \Delta}{\Gamma, \neg A \longrightarrow \Delta}\ l\neg^* \qquad \frac{\Gamma, A \longrightarrow \Delta}{\Gamma \longrightarrow \neg A, \Delta}\ r\neg^*$$

$$\frac{\Gamma, B \longrightarrow \Delta \quad \Gamma \longrightarrow A, \Delta}{\Gamma, A \supset B \longrightarrow \Delta}\ l\supset^* \qquad \frac{\Gamma, A \longrightarrow B, \Delta}{\Gamma \longrightarrow A \supset B, \Delta}\ r\supset^*$$

$$\frac{\Gamma, \forall x:\iota.A, A[c_1:\iota/x] \longrightarrow \Delta}{\Gamma, \forall x:\iota.A \longrightarrow \Delta}\ l\forall^*_\iota \qquad \frac{\Gamma \longrightarrow A[a_1:\iota/x], \Delta}{\Gamma \longrightarrow \forall x:\iota.A, \Delta}\ r\forall^*_\iota$$

$$\frac{\Gamma, \forall x:\theta.A, A[c_2:\theta/x] \longrightarrow \Delta}{\Gamma, \forall x:\theta.A \longrightarrow \Delta}\ l\forall^*_\theta \qquad \frac{\Gamma \longrightarrow A[a_2:\theta/x], \Delta}{\Gamma \longrightarrow \forall x:\theta.A, \Delta}\ r\forall^*_\theta$$

Table 3.5: the calculus $2LK$ for $2FOL$. A, B are formulae, c_1, c_2, s, t terms and a_1, a_2 variables of $2FOL$; a_1 and a_2 cannot appear free in the conclusion of $r\forall^*_\iota$ and $r\forall^*_\theta$. In rule sub^*, the occurrences of $s:\theta$ replaced by $t:\theta$ are in atomic formulae only, as in [DV01].

$2LK$ is a specialisation for two sorts of the calculus $G_=$ for many-sorted languages with equality presented, e.g., in [Gal86], Definition 10.5.1, where equality is admitted on one sort only, namely the sort θ; the presentation is also simplified with respect to Gallier's according to [DV01, Kan63]. $2LK$ consists of an axiomatic rule, rules for equality, rules for $\neg$ and $\supset$, and two pairs of rules for $\forall$, denoted $r\forall^*_\iota, r\forall^*_\theta, l\forall^*_\iota, l\forall^*_\theta$, in place of the usual ones for untyped quantifiers. We denote all $2LK$-rules with a

superscript $*$.

We have that

Theorem 12 *2LK is sound and complete for 2FOL (Lemma 10.5.1, Theorem 10.5.1 in [Gal86]; Section 1 of [DV01]).*

3.3.2 Embedding QMLs into $2FOL$

Now we define a translation which maps formulae and first-order sentences to terms and formulae of $2FOL$:

Definition 13 *Let the operator $[\![\cdot]\!]$ be defined as in Figure 3.8 (recall Definition 1).*

$[\![\cdot]\!]$ maps symbols of **QL** to primed symbols of $2FOL$, logical atoms to $2FOL$ atoms with one more argument (the label), and leaves other formulae as they stand, recursively, except that it respects the type of the quantifiers in $\forall$-formulae (which is always ι) and it unfolds $\Box$ operators in a way that is intuitively related to their semantics. As an example,

$$[\![\forall x.\Box p(x) \supset p(x) \mathbin{@} 0]\!] = \forall x' : \iota.\forall t' : \theta.0' \prec' t' \supset p'(x',t') \supset p'(x',0').$$

In the following, we drop the "prime" in order to ease the notation. The above example becomes

$$[\![\forall x.\Box p(x) \supset p(x) \mathbin{@} 0]\!] = \forall x : \iota.\forall t : \theta.\ 0 \prec t \supset p(x,t) \supset p(x,0).$$

$[\![\cdot]\!]$ is also straightforwardly extended to sequent rules:

$$\frac{\Gamma_1 \longrightarrow \Delta_1 \quad \cdots \quad \Gamma_n \longrightarrow \Delta_n}{\Gamma \longrightarrow \Delta}\ \rho \quad \overset{[\![\cdot]\!]}{\rightsquigarrow} \quad \frac{[\![\Gamma_1 \longrightarrow \Delta_1]\!] \quad \cdots \quad [\![\Gamma_n \longrightarrow \Delta_n]\!]}{[\![\Gamma \longrightarrow \Delta]\!]}\ [\![\rho]\!]$$

As it can be seen, $[\![\cdot]\!]$ preserves the number of premises of a sequent rule; therefore, it extends also to derivations: the $2FOL$-translation of a derivation is a derivation in which all sequent rules are $2FOL$-translations of sequent rules of $\mathcal{C}_{\mathbf{QL}}$. The same goes for proofs.

$$
\begin{array}{lcl}
[\![x]\!], \text{ with } x \in \mathcal{V} & = & x' : \iota \in \mathcal{V}\,' \\
[\![f(s_1,\ldots,s_n)]\!], \text{ with } f \in \mathcal{F} & = & f'([\![s_1]\!],\ldots,[\![s_n]\!]) \text{ with } f' \in \mathcal{F}\,' \\
[\![p]\!], \text{ with } p \in \mathcal{P} & = & p' \in \mathcal{P}\,' \\
[\![0]\!] & = & 0' \in \mathcal{F}\,' \\
[\![\prec]\!] & = & \prec' \in \mathcal{P}\,' \\
[\![\doteq]\!] & = & =_\theta \in \mathcal{P}\,' \\
[\![t]\!], \text{ with } t \in \mathcal{V}\,' & = & t' : \theta \in \mathcal{V}\,' \\
[\![g]\!], \text{ with } g \in \mathcal{F}\,' & = & g' \in \mathcal{F}\,' \\
[\![\tau_1 \prec \tau_2]\!] & = & [\![\tau_1]\!] \prec' [\![\tau_2]\!] \\
[\![p(s_1,\ldots,s_n) \mathbin{@} \tau]\!] & = & p'([\![s_1]\!],\ldots,[\![s_n]\!],[\![\tau]\!]) \\
[\![\neg\varphi \mathbin{@} \tau]\!] & = & \neg[\![\varphi \mathbin{@} \tau]\!] \\
[\![\varphi \supset \psi \mathbin{@} \tau]\!] & = & [\![\varphi \mathbin{@} \tau]\!] \supset [\![\psi \mathbin{@} \tau]\!] \\
[\![\forall x.\varphi \mathbin{@} \tau]\!] & = & \forall x : \iota.\ [\![\varphi \mathbin{@} \tau]\!] \\
[\![\Box\varphi \mathbin{@} \tau]\!] & = & \forall t : \theta.\ [\![\tau \prec t]\!] \supset [\![\varphi \mathbin{@} t]\!] \\
[\![\Gamma]\!] & = & \{[\![\gamma]\!] \mid \gamma \in \Gamma\} \\
[\![\Gamma \longrightarrow \Delta]\!] & = & [\![\Gamma]\!] \longrightarrow [\![\Delta]\!] \\
[\![\phi]\!], \phi \text{ a first-order sentence} & = & \phi
\end{array}
$$

Figure 3.8: the definition of $[\![\cdot]\!]$, a $2FOL$-translation mapping formulae, sequents and first-order sentences to formulae and sequents of $2FOL$. Translations of first-order sentences ϕ include the rank of bound variables, which is invariably θ.

3.3.3 Soundness and completeness

For any **QL**, let $\mathsf{FrmAx}^S(\mathbf{QL})$ be the set of first-order sentences obtained by skolemizing and converting in prenex normal form the sentences in $\mathsf{FrmAx}(\mathbf{QL})$. Let also, as usual, Γ and Δ be finite multisets of **forms**, with the restriction that the labels appear-

ing in $\Gamma \cup \Delta$ must not contain any Skolem functions. In order to show soundness and completeness of $c_{\mathbf{QL}}$ for **QL**, we prove that the following statements are equivalent:

1. $\Gamma \longrightarrow \Delta$ is a theorem of $c_{\mathbf{QL}}$,
2. $[\![\Gamma \cup \mathsf{FrmAx}^S(\mathbf{QL}) \longrightarrow \Delta]\!]$ is a theorem of $2LK$,
3. $[\![\Gamma \cup \mathsf{FrmAx}(\mathbf{QL}) \longrightarrow \Delta]\!]$ is valid in $2FOL$,
4. $\Gamma \longrightarrow \Delta$ is valid in **QL**.

Figure 3.9 graphically depicts the situation.

$$\vdash_{c_{\mathbf{QL}}} \Gamma \longrightarrow \Delta \ \boxed{1} \quad \Longleftrightarrow \quad \boxed{4} \ \models^{\mathbf{QL}} \Gamma \longrightarrow \Delta$$

$$\Updownarrow \qquad\qquad \Updownarrow$$

$$\vdash_{2LK} [\![\Gamma \cup \mathsf{FrmAx}^S(\mathbf{QL}) \longrightarrow \Delta]\!] \ \boxed{2} \quad \Longleftrightarrow \quad \boxed{3} \ \models^{2FOL} [\![\Gamma \cup \mathsf{FrmAx}(\mathbf{QL}) \longrightarrow \Delta]\!]$$

Figure 3.9: a schematic representation of the proof of correctness. Instead of proving that 1 implies 4 (soundness) and that 4 implies 1 (completeness), we prove that 1, 2, 3 and 4 are equivalent.

We proceed by first proving equivalence 1-2 (Proposition 14), then equivalence 2-3 (Proposition 25), and lastly equivalence 3-4 (Proposition 26).

Proposition 14 (Equivalence 1-2) *Items 1 and 2 are equivalent, that is*

$$\vdash_{c_{\mathbf{QL}}} \Gamma \longrightarrow \Delta \quad \textit{iff} \quad \vdash_{2LK} [\![\Gamma \cup \mathsf{FrmAx}^S(\mathbf{QL}) \longrightarrow \Delta]\!].$$

Proof: We show that every $c_{\mathbf{QL}}$-proof can be $2FOL$-translated, and that every $2LK$-proof of a $2FOL$-translated sequent is similar to a $2LK$-proof that actually *is* the $2FOL$-translation of a $c_{\mathbf{QL}}$-proof.

1 implies 2: in order to show that every $c_{\mathbf{QL}}$-proof can be 2*FOL*-translated, we show that every rule in $c_{\mathbf{QL}}$ can be 2*FOL*-translated. Recall Tables 3.2, 3.3 and 3.5; by case analysis,

- *closing rule:* straightforwardly, $[\![\mathsf{ax}]\!] = ax^*$.
- *logical rules:* as well, $[\![r\neg]\!] = r\neg^*$, $[\![l\neg]\!] = l\neg^*$, $[\![r\supset]\!] = r\supset^*$ and $[\![l\supset]\!] = l\supset^*$. For example:

$$\frac{\Gamma, \varphi \mathbin{@} \tau \longrightarrow \psi \mathbin{@} \tau, \Delta}{\Gamma \longrightarrow \varphi \supset \psi \mathbin{@} \tau, \Delta}\, r\supset \quad \overset{[\![\cdot]\!]}{\rightsquigarrow} \quad \frac{[\![\Gamma]\!], [\![\varphi \mathbin{@} \tau]\!] \longrightarrow [\![\psi \mathbin{@} \tau]\!], [\![\Delta]\!]}{[\![\Gamma]\!] \longrightarrow [\![\varphi \mathbin{@} \tau]\!] \supset [\![\psi \mathbin{@} \tau]\!], [\![\Delta]\!]}\, r\supset^*$$

  Moreover, $[\![r\forall]\!] = r\forall^*_\iota$ and $[\![l\forall]\!] = l\forall^*_\iota$. For example:

$$\frac{\Gamma \longrightarrow \varphi[a/x] \mathbin{@} \tau, \Delta}{\Gamma \longrightarrow \forall x.\varphi \mathbin{@} \tau, \Delta}\, r\forall \quad \overset{[\![\cdot]\!]}{\rightsquigarrow} \quad \frac{[\![\Gamma]\!] \longrightarrow [\![\varphi[a/x] \mathbin{@} \tau]\!], [\![\Delta]\!]}{[\![\Gamma]\!] \longrightarrow \forall x{:}\iota.[\![\varphi \mathbin{@} \tau]\!], [\![\Delta]\!]}\, r\forall_\iota$$

  Lastly, $[\![r\Box]\!]$ is the composition of $r\forall^*_\theta$ and $r\supset^*$, whereas $[\![l\Box]\!]$ is the composition of $l\forall^*_\theta$ and $l\supset^*$. For example:

$$\frac{\Gamma, \tau \prec t' \longrightarrow \varphi \mathbin{@} t', \Delta}{\Gamma \longrightarrow \Box\varphi \mathbin{@} \tau, \Delta}\, r\Box \quad \overset{[\![\cdot]\!]}{\rightsquigarrow} \quad \frac{\dfrac{[\![\Gamma]\!], [\![\tau \prec t']\!] \longrightarrow [\![\varphi \mathbin{@} t']\!], [\![\Delta]\!]}{[\![\Gamma]\!] \longrightarrow [\![\tau \prec t']\!] \supset [\![\varphi \mathbin{@} t']\!], [\![\Delta]\!]}\, r\supset^*}{[\![\Gamma]\!] \longrightarrow \forall t{:}\theta.[\![\tau \prec t]\!] \supset [\![\varphi \mathbin{@} t]\!], [\![\Delta]\!]}\, r\forall^*_\theta$$

- *frame rules:* again, $[\![\mathsf{refl}_{\doteq}]\!] = re^*$, $[\![\mathsf{sub}_{\doteq}]\!] = sub^*$ and, by Definition 6, frame rules obtained by the strengthening procedure are finite compositions of 2*LK* rules.

This completes the proof of implication 1-2.

2 implies 1: this case is more complicated. From now on, let $[\![\mathbf{forms}]\!]$ denote the image of **forms** under $[\![\cdot]\!]$, that is $[\![\mathbf{forms}]\!] = \{\psi \mid \psi = [\![\varphi]\!], \varphi \in \mathbf{forms}\}$; moreover, let any 2*FOL*-formula which is the translation of a formula $\varphi \in \mathbf{forms}$ be denoted as $[\![\varphi]\!]$; lastly, let us assume that Π is a 2*LK*-proof of $[\![\Gamma \cup \mathsf{FrmAx}^S(\mathbf{QL}) \longrightarrow \Delta]\!]$, for some logic **QL** and Γ, Δ multisets of **forms**.

We want to show that there is a $2LK$-proof Π', similar to Π, which is the translation of a $c_{\mathbf{QL}}$-proof of $\Gamma \longrightarrow \Delta$. In order to do that, we first establish a sufficient condition for a $2LK$-proof to be the translation of a $c_{\mathbf{QL}}$-proof, and then we show that, for every Π, there is a similar Π' which enjoys the condition.

A *subset* of Π is a subset of the nodes of Π; let $N \in \Pi$ be labelled by $[\![\varphi]\!]$; then

Definition 15 *A* trail *of* N, $\mathsf{Tr}(N)$, *is a subset of* Π *for which the following properties hold:*

1. $\mathsf{Tr}(N)$ *is a tree and* N *is the root node;*
2. *let* $N_i \in \mathsf{Tr}(N)$*; let* N_j, $j = 1\ldots,n$ *be its children, each one labelled by* $[\![\varphi_j]\!]$*; then every* $[\![\varphi_j]\!]$ *is active in* N_i*;*
3. *no node of* $\mathsf{Tr}(N)$ *is labelled by a duplicate* $\forall$*-formula introduced in the premises by a* $l\forall^*_\theta$ *rule.*

$\mathsf{Tr}(N)$ *is said to* belong *to* Π*, which is said to be its* parent.

Informally speaking, the trail of N is the subset of Π by which $[\![\varphi]\!]$ is "completely unfolded".

Let $(N_1,\ldots,N_k)$ be a branch of Π; then a *path* in Π is a tuple of nodes $(N_n,\ldots,N_m)$ such that $1 \leq n \leq m \leq k$, and its *length*, $len(N_n,\ldots,N_m)$, is the number of nodes between N_n and N_m. The *sparsity* of a trail $\mathsf{Tr}(N)$ in Π is defined as $\sum len(N',\ldots,N'')$ for all $N',N'' \in \Pi$ such that N'' is a child of N' in $\mathsf{Tr}(N)$. Intuitively, the sparsity of a trail indicates how "far away" from each other the nodes of $\mathsf{Tr}(N)$ are in its parent. If the sparsity is 0, the trail is called *compact*. Informally speaking, a compact trail is also a proper subtree of its parent.

Definition 16 (Compactness of a proof) *A* $2LK$*-proof* Π *will be called* compact *if and only if:*

1. *for every node* $N \in \Pi$ *labelled by* $[\![\varphi]\!]$, $\mathsf{Tr}(N)$ *belongs to* Π *and is compact;*

2. *the union of all such trails is* Π.

Informally, a compact proof is the union of a finite set of compact trails, each one labelled by a $2FOL$-formula $[\![\varphi]\!]$. We are now ready to prove that the property of compactness is sufficient for a $2FOL$-proof to be the translation of a $c_{\mathbf{QL}}$-proof:

Lemma 17 *If* Π *is compact, then there is a* $c_{\mathbf{QL}}$*-proof* Θ *such that* $[\![\Theta]\!] = \Pi$.

Proof: Immediate from Definition 15 and the proof of implication 1-2: every translation of a $c_{\mathbf{QL}}$-rule ρ is the compact trail of a node N in a $2LK$-proof, labelled by $[\![\rho]\!]$.

•

Thanks to this Lemma, in order to prove implication 2-1, it suffices to show that for every Π there is a similar, compact Π'. To carry on, we first need two useful results from Proof Theory:

Lemma 18 (Inversion Lemma for $2LK$**)** *For all* $\rho \in 2LK$ *except* ax^* *and* re^**, if the conclusion of* ρ *is* $2LK$*-provable, so are all the premises.*

Proof: By induction on the depth of a proof, that is, on the length of the longest branch in the proof. See Proposition 3.5.4 in [TS96] for the details. The Proposition also trivially extends to rule sub^*.

•

Given the notions of *adjacency* and *permutability* of sequent rules in $2LK$, adapted from Definition 5.3.1 in [TS96],

Lemma 19 (Permutation Lemma for $2LK$**)** *Let* $\rho, \rho' \in 2LK$. *Then* ρ *is always permutable below* ρ'*, except when* $\rho = l\forall_\iota^*$ *and* $\rho' = r\forall_\iota^*$*, or when* $\rho = l\forall_\theta^*$ *and* $\rho' = r\forall_\theta^*$. *The new proof is similar to the original one.*

Proof: As in Lemma 5.3.10 in [TS96], specialised for two sorts and no structural rules. The definition of permutability obviously takes into account the fact that no rule is permutable where it is not applicable, i.e., that rule α can be permuted below rule β only if the main formula in α is not active in β and vice-versa.

•

Now we proceed by case analysis on the shape of $[\![\Gamma \cup \mathsf{FrmAx}^S(\mathbf{QL}) \longrightarrow \Delta]\!]$, considering in turn three sub-cases, and showing that in each (more and more complex) sub-case, there is Π' which is similar to Π and compact.

Sub-case (I) Let the set of sub-formulae of $\Gamma \cup \Delta$ contain no $\Box$-formulae and let $\mathsf{FrmAx}^S(\mathbf{QL})$ be empty. By structural induction on the shape of the sub-formulae of $[\![\Gamma]\!]$ and $[\![\Delta]\!]$, it is clear that every node $N \in \Pi$ is labelled by a $2LK$-rule displayed in Table 3.5, *except* $r\forall^*_\theta$ and $l\forall^*_\theta$.

But, each of these rules is the translation of a single $c_{\mathbf{QL}}$-rule (recall the proof of implication 1-2); therefore, by Definition 15, every node in Π is a single, compact trail. Then Π is compact by Definition 16, and obviously similar to itself.

Sub-case (II) Suppose now that there is at least a node $N \in \Pi$ labelled by a $\Box$-formula. We first state a corollary of Lemma 19:

Corollary 20 *An application of rule $r\supset^*$ or $l\supset^*$ can be permuted below or above any other rule, preserving similarity.*

Let us call a $\Box$-*trail* the trail of a node N labelled by the translation of a $\Box$-formula; then

Theorem 21 (Existence and compactness of $\Box$-trails) *Let $N \in \Pi$ be labelled by the translation of a $\Box$-formula; then there is a $2LK$-proof Π' similar to Π such that:*

1. $\mathsf{Tr}(N)$ *belongs to* Π',
2. $\mathsf{Tr}(N)$ *is compact.*

Proof: *(1)*: by contradiction. Consider node N: by the conclusion of $r\forall^*_\theta$ we know that

$$\vdash_{2LK} \Gamma \longrightarrow \forall t:\theta.[\![\tau \prec t]\!] \supset [\![\varphi \mathbin{@} t]\!], \Delta.$$

Now if *(1)* is false, then by Definition 15, there can be no proof Π' in which a node N' above N is labelled by $r\supset^*$, and its main formula is active in N. This means that

$$\nvdash_{2LK} \Gamma \longrightarrow [\![\tau \prec a]\!] \supset [\![\varphi \mathbin{@} a]\!], \Delta$$

where a does not appear free in the former sequent. But this contradicts Lemma 18, when $\rho = r\forall^*_\theta$. An analogous argument holds on the left. *(2)*: by Corollary 20, the child of N in $\mathsf{Tr}(N)$ can be permuted in Π so that the sparsity of $\mathsf{Tr}(N)$ eventually becomes 0, that is, $\mathsf{Tr}(N)$ is compact.

•

Let then Π' be such a proof: by this very Theorem, all $\Box$-trails in Π belong to Π', and they are all compact. Moreover, as it can be easily checked, Π' does not contain any new nodes labelled by $\Box$-formulae; and, since by the same inductive argument of Sub-case I, the only nodes in Π' not falling in the previous Sub-case are exactly those in all $\Box$-trails, all nodes in Π' belong to a compact trail. By Definition 16 then, Π' is compact, and it is similar to Π by this Theorem again.

As an example, let Π be the following $2LK$-proof of theorem $[\![\Box(p \vee \neg p) \mathbin{@} 0]\!]$ (all bound variables have sort θ — we omit it for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma']\!], 0 \prec t', p(t') \longrightarrow p(t'), [\![\Delta']\!]}\, ax^*}{[\![\Gamma']\!], 0 \prec t' \longrightarrow p(t'), \neg p(t'), [\![\Delta']\!]}\, r\neg^*}{[\![\Gamma']\!], 0 \prec t' \longrightarrow p(t') \vee \neg p(t'), [\![\Delta']\!]}\, r\vee^*}{[\![\Gamma']\!] \longrightarrow 0 \prec t' \supset p(t') \vee \neg p(t'), [\![\Delta']\!]}\, r\supset^*}{\begin{array}{c}\vdots \text{ subproof \#1} \\ [\![\Gamma]\!] \longrightarrow 0 \prec t' \supset p(t') \vee \neg p(t'), [\![\Delta]\!]\end{array}}$$
$$\dfrac{}{[\![\Gamma]\!] \longrightarrow \forall t.0 \prec t \supset p(t) \vee \neg p(t), [\![\Delta]\!]}\, r\forall^*_\theta$$

Assume, without loss of generality, that subproof #1 is compact, and let Π' be the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma']\!], 0 \prec t', p'(t') \longrightarrow p'(t'), [\![\Delta']\!]}\, ax^*}{[\![\Gamma']\!], 0 \prec t' \longrightarrow p'(t'), \neg p'(t'), [\![\Delta']\!]}\, r\neg^*}{[\![\Gamma']\!], 0 \prec t' \longrightarrow p'(t') \vee \neg p'(t'), [\![\Delta']\!]}\, r\vee^*}{\vdots \text{ subproof \#1}}}{[\![\Gamma]\!], 0 \prec t' \longrightarrow p'(t') \vee \neg p'(t'), [\![\Delta]\!]}}{[\![\Gamma]\!] \longrightarrow 0 \prec t' \supset p'(t') \vee \neg p'(t'), [\![\Delta]\!]}\, r\supset^*}{[\![\Gamma]\!] \longrightarrow \forall t.0 \prec t \supset p'(t) \vee \neg p'(t), [\![\Delta]\!]}\, r\forall^*_\theta$$

It turns out that $\Pi' = [\![\Theta]\!]$, where Θ is the following $c_{\mathbf{QL}}$-proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma', 0 \prec t', p \mathbin{@} t' \longrightarrow p \mathbin{@} t', \Delta'}\, \mathsf{ax}}{\Gamma', 0 \prec t' \longrightarrow p \mathbin{@} t', \neg p \mathbin{@} t', \Delta'}\, r\neg}{\Gamma', 0 \prec t' \longrightarrow p \vee \neg p \mathbin{@} t', \Delta'}\, r\vee}{\vdots \text{ subproof \#1}}}{\Gamma, 0 \prec t' \longrightarrow p \vee \neg p \mathbin{@} t', \Delta}}{\Gamma \longrightarrow \Box(p \vee \neg p) \mathbin{@} 0, \Delta}\, r\Box$$

and that Π' is similar to Π' and compact, as we expect from Theorem 21.

Sub-case (III) Suppose, lastly, that there is $N \in \Pi$ labelled by $\phi^S \in \mathsf{FrmAx}^S(\mathbf{QL})$. Another immediate corollary of Lemma 19 is that

Corollary 22 *An application of rule $l\forall^*_\theta$, $l\neg^*$, $r\neg^*$, $l\supset^*$, $r\supset^*$ can be permuted* above *any other rule, preserving similarity.*

Let us call a *frame trail* $\mathsf{Tr}(N)$, where N is labelled by the translation of ϕ^S; then

Theorem 23 (Existence and compactness of frame trails) *Let $N \in \Pi$ be labelled by the translation of a frame axiom ϕ^S; then there is a 2LK-proof Π' similar to Π such that:*

1. $\mathsf{Tr}(N)$ *belongs to* Π',
2. $\mathsf{Tr}(N)$ *is compact.*

Proof: *(1)*: by the same argument of Theorem 21 and repeated application of Lemma 18. *(2)*: by the same argument of Theorem 21: by Corollary 22, and by the fact that by Definition 6, ϕ^S has the shape $\forall\overline{x}.P(\overline{x})$, where $P(\overline{x})$ is quantifier-free and appears on the left of a sequent.

•

Let then Π' be such a proof: by this very Corollary, all frame trails in Π belong to Π', and they are all compact. Moreover, again, Π' does not contain any new nodes labelled by frame axioms.

However here, differently from the previous Sub-case, the use of the $l\forall^*_\theta$ rule *can* spawn nodes which do not belong to any trail; in fact, by the same inductive argument of Sub-cases I and II, this is the only case of nodes in Π' not falling in the previous Sub-cases. So it remains to prove that there is a further $2LK$-proof, call it Π'', similar to Π and Π', in which such nodes belong to a compact trail.

As an example of "bad" behaviour, consider Figure 3.10, illustrating a proof involving the axiom of symmetry (indicated as 5 to ease the notation — recall Table 3.1). The problem arises from the very shape of frame axioms, which can have, in general, two or more outer universal quantifiers.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{[\![\Gamma]\!],5,t_1 \prec t_0 \longrightarrow [\![\Delta]\!]} \qquad \dfrac{\dfrac{\dfrac{\vdots}{[\![\Gamma]\!],5 \longrightarrow t_0 \prec t_1', t_0 \prec t_1, [\![\Delta]\!]} \quad \dfrac{\vdots}{[\![\Gamma]\!],5, t_1' \prec t_0 \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}\, l\supset^*}{[\![\Gamma]\!],5,\boxed{t_0 \prec t_1' \supset t_1' \prec t_0} \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}}{[\![\Gamma]\!],5,\boxed{\forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0} \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}\, \boxed{l\forall^*_\theta}}{[\![\Gamma]\!],5,\forall t_1.t_1 \prec t_0, t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta]\!]}\, l\supset^*}{[\![\Gamma]\!],5,\forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta]\!]}\, l\forall^*_\theta}{[\![\Gamma]\!],5 \longrightarrow [\![\Delta]\!]}\, l\forall^*_\theta$$

Figure 3.10: an example of "bad" frame trail: an application of rule $l\forall^*_\theta$, boxed in the Figure, generates a duplicate $\forall$-formula which is not the translation of any formula in **forms** and spawns nodes not belonging to any trail. Bad nodes are boxed, as well as their main formulae.

Let $N' \in \Pi$ be labelled by a duplicate formula ψ. It must be the case that ψ was generated by a $l\forall^*_\theta$ labelling a node in a frame trail; call the frame axiom at the root of the trail ϕ^S. Now since ϕ^S is in prenex normal form, it must be the case that $\phi^S =$

$\forall x_1 \ldots x_n.\psi$, and that there is a copy of ϕ^S in the sequent labelling N'. This is evident in Figure 3.10, at the node labelled by the boxed $l\forall^*_\theta$.

Let then $N_1, \ldots, N_n$ be n new nodes inserted just below N', such that *(a)* N_1 is labelled by ϕ^S and $l\forall^*_\theta$, *(b)* for all $N_i, i = 1, \ldots, n$, N_i is labelled by $\forall x_i \ldots x_n.\psi$ and $l\forall^*_\theta$. Let, lastly, N' be labelled by the active formula in N_n. This way we obtain a new proof Π'' similar to Π' which contains a trail $\mathsf{Tr}(N_1)$ labelled by ϕ^S including the old bad nodes.

Figure 3.11 shows the effect of this operation on the example of Figure 3.10.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{[\![\Gamma]\!], 5, t_1 \prec t_0 \longrightarrow [\![\Delta]\!]} \quad \dfrac{\vdots}{[\![\Gamma]\!], 5 \longrightarrow t_0 \prec t_1', t_0 \prec t_1, [\![\Delta]\!]}}{[\![\Gamma]\!], 5, \forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0, t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta]\!]}\, l\forall^*_\theta}{[\![\Gamma]\!], 5, \forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta]\!]}\, l\forall^*_\theta}{[\![\Gamma]\!], 5 \longrightarrow [\![\Delta]\!]}\, l\supset^*}{[\![\Gamma]\!], \boxed{5} \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}}{[\![\Gamma]\!], 5, \boxed{\forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0} \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}\, \boxed{l\forall^*_\theta}}{[\![\Gamma]\!], 5, \boxed{t_0 \prec t_1' \supset t_1' \prec t_0} \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}\, \boxed{l\forall^*_\theta}}{[\![\Gamma]\!], 5 \longrightarrow t_0 \prec t_1', t_0 \prec t_1, [\![\Delta]\!] \quad [\![\Gamma]\!], 5, t_1' \prec t_0 \longrightarrow t_0 \prec t_1, [\![\Delta]\!]}\, \boxed{l\supset^*}$$

Figure 3.11: the example of Figure 3.10, "cured": a new node has been inserted in the proof, making the old bad nodes part of a new frame trail.

By repeated application of this method, all nodes not falling in the above cases can be revamped into nodes belonging to frame trails; more formally, there is Π'' similar to Π which meets Definition 16 and is therefore compact.

In order to carry the proof of implication 2-1 to the end, one last simple result is needed:

Lemma 24 *A rule* ρ *whose active formulae are atomic can be permuted above until it is at the top of the proof tree.*

Proof: By the definition of permutability (Definition 5.3.1 in [TS96]), a rule ρ is permutable above a rule ρ' only if none of the active formulae of ρ is main in ρ'.

But the only rule in which the main formulae are atomic is ax^* which is closing, and therefore is at the top of the proof tree.

•

By applying this result repeatedly to all compact frame trails in Π', we get a final 2*LK*-proof in which all frame trails either appear at the top of the proof tree or have frame trails above them. By Definition 9, such a proof is the translation of a $c_{\mathbf{QL}}$-proof.

This also holds for the frame rule $\mathsf{sub}_{\doteq}$, and it is precisely the reason for the restriction on its application (recall Table 3.5).

This completes the proof of implication 2-1 and therefore of Proposition 14.

As a final example, let Π be the following 2*LK*-proof in which the axiom of symmetry 5 (recall Table 3.1) has been employed:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma'''],t_1 \prec t_0 \longrightarrow [\![\Delta''']\!]}\,ax^* \quad \dfrac{}{[\![\Gamma'''] \longrightarrow t_0 \prec t_1,[\![\Delta''']\!]}\,ax^*}{\begin{array}{c}\vdots \text{ subproof \#4}\\ [\![\Gamma'']\!],t_1 \prec t_0 \longrightarrow [\![\Delta'']\!]\end{array} \quad \begin{array}{c}\vdots \text{ subproof \#5}\\ [\![\Gamma'']\!] \longrightarrow t_0 \prec t_1,[\![\Delta'']\!]\end{array}}\,l{\supset}^*}{[\![\Gamma'']\!],t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta'']\!]}}{\vdots \text{ subproof \#3}}}{[\![\Gamma']\!],t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta']\!]}\,l\forall^*_\theta}{\vdots \text{ subproof \#2}}}{[\![\Gamma]\!],\forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta]\!]}\,l\forall^*_\theta}{\vdots \text{ subproof \#1}}$$

Let Π' be the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma'''],t_1 \prec t_0 \longrightarrow [\![\Delta''']\!]}\,ax^* \quad \dfrac{}{[\![\Gamma'''] \longrightarrow t_0 \prec t_1,[\![\Delta''']\!]}\,ax^*}{[\![\Gamma''']\!],t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta''']\!]}\,l{\supset}^*}{[\![\Gamma''']\!],\forall t_1.t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta''']\!]}\,l\forall^*_\theta}{[\![\Gamma''']\!],\forall t_0 t_1.t_0 \prec t_1 \supset t_1 \prec t_0 \longrightarrow [\![\Delta''']\!]}\,l\forall^*_\theta}{\begin{array}{c}\vdots \text{ subproof \#4/\#5}\\ \vdots \text{ subproof \#3}\\ \vdots \text{ subproof \#2}\\ \vdots \text{ subproof \#1}\end{array}}$$

It turns out that $\Pi' = [\![\Theta]\!]$, where Θ is the following $c_{\mathbf{QK}}$-proof:

$$\dfrac{\dfrac{\dfrac{}{\Gamma''', t_1 \prec t_0 \longrightarrow \Delta'''}\ \mathsf{ax} \quad \dfrac{}{\Gamma''' \longrightarrow t_0 \prec t_1, \Delta'''}\ \mathsf{ax}}{\Gamma''' \longrightarrow \Delta'''}\ \mathsf{symm}}{\begin{array}{l}\vdots\ \text{subproof \#4/\#5}\\ \vdots\ \text{subproof \#3}\\ \vdots\ \text{subproof \#2}\\ \vdots\ \text{subproof \#1}\end{array}}$$

and that Π' is similar to Π, it is compact, and the only displayed frame rule appears at the top of the tree, as we expect from Theorem 23 and Corollary 24.

•

Proposition 25 (Equivalence 2-3) *Items 2 and 3 are equivalent, that is*

$$\vdash_{2LK} [\![\Gamma \cup \mathsf{FrmAx}^S(\mathbf{QL}) \longrightarrow \Delta]\!] \quad \textit{iff} \quad \models^{2FOL} [\![\Gamma \cup \mathsf{FrmAx}(\mathbf{QL}) \longrightarrow \Delta]\!].$$

Proof: Since $[\![\mathbf{forms}]\!] \cup \mathsf{FrmAx}^S(\mathbf{QL})$ is a strict subset of the formulae of $2FOL$, this equivalence follows from Theorem 12, with the remark that the $2FOL$ theory of $\mathsf{FrmAx}^S(\mathbf{QL})$ is a conservative extension of that of $\mathsf{FrmAx}(\mathbf{QL})$ (see, e.g., [Sho70], p. 55).

•

Proposition 26 (Equivalence 3-4) *Items 3 and 4 are equivalent, that is*

$$\models^{2FOL} [\![\Gamma \cup \mathsf{FrmAx}(\mathbf{QL}) \longrightarrow \Delta]\!] \quad \textit{iff} \quad \models^{\mathbf{QL}} \Gamma \longrightarrow \Delta.$$

Proof: Since $[\![\cdot]\!]$ extends to sequents straightforwardly, it suffices to prove the Proposition for single formulae. The Proposition is proved by showing that, given a model in **QL** for $\varphi \in \mathbf{forms}$, there is a corresponding model for $[\![\varphi]\!]$ in $2FOL$, and vice-versa.

Let $\mathbf{M} = \langle \mathcal{W}, R, \mathcal{D}, I \rangle$ and α be a structure and an assignment of **QL**, and let $\mathbf{M}' = \langle \mathcal{D}', I' \rangle$ and α' be a structure and an assignment of $2FOL$ such that:

1. $\mathcal{D}' = \mathcal{W} \cup \mathcal{D}$,

2. I' interprets $\prec'$ as R and $\doteq$ as $=_\theta$,

3. for any predicate symbol $p \in \mathcal{P}$,

$$(s_1^{\mathbf{M},\alpha},\ldots,s_n^{\mathbf{M},\alpha}) \in I(w,p) \text{ iff } (s_1^{\mathbf{M}',\alpha'},\ldots,s_n^{\mathbf{M}',\alpha'},\tau^{\mathbf{M}',\alpha'}) \in I'(p')$$

4. $\alpha'(v:\iota) = d' \in \mathcal{D}'$ iff $\alpha(v) = d \in \mathcal{D}$,

5. $\alpha'(t:\theta) = w' \in \mathcal{D}'$ iff $\alpha(t) = w \in \mathcal{W}$.

It turns out that $\mathbf{M},\alpha \models \varphi$ iff $\mathbf{M}',\alpha' \models [\![\varphi]\!]$. This is proved by structural induction. Base cases:

- logical atoms: $\mathbf{M},\alpha \models p(s_1,\ldots,s_n)\ @\ \tau$ if and only if $(s_1^{\mathbf{M},\alpha},\ldots,s_n^{\mathbf{M},\alpha}) \in I(w,p)$ if and only if $(s_1^{\mathbf{M}',\alpha'},\ldots,s_n^{\mathbf{M}',\alpha'},\tau^{\mathbf{M}',\alpha'}) \in I'(p')$ if and only if $\mathbf{M}',\alpha' \models p'([\![s_1]\!],\ldots,[\![s_n]\!],[\![\tau]\!])$ that is $\mathbf{M}',\alpha' \models [\![p(s_1,\ldots,s_n)\ @\ \tau]\!]$.

- $\prec$-constraints: $\mathbf{M},\alpha \models \tau_1 \prec \tau_2$ if and only if $(w_1,w_2) \in R$ if and only if $\mathbf{M}',\alpha' \models [\![\tau_1]\!] \prec' [\![\tau_2]\!]$ that is $\mathbf{M}',\alpha' \models [\![\tau_1 \prec \tau_2]\!]$.

- $\doteq$-constraints: $\mathbf{M},\alpha \models \tau_1 \doteq \tau_2$ if and only if $w_1 = w_2$ if and only if $\mathbf{M}',\alpha' \models [\![\tau_1]\!] =_\theta [\![\tau_2]\!]$ that is $\mathbf{M}',\alpha' \models [\![\tau_1 \doteq \tau_2]\!]$.

Step cases: assume that $\mathbf{M},\alpha \models \varphi@\tau$ if and only if $\mathbf{M}',\alpha' \models [\![\varphi@\tau]\!]$, and $\mathbf{M},\alpha \models \psi@\tau$ if and only if $\mathbf{M}',\alpha' \models [\![\psi@\tau]\!]$. Then

- negation: $\mathbf{M},\alpha \models \neg\varphi\ @\ \tau$ if and only if not $\mathbf{M},\alpha \models \varphi\ @\ \tau$ if and only if not $\mathbf{M}',\alpha' \models [\![\varphi\ @\ \tau]\!]$ that is $\mathbf{M}',\alpha' \models [\![\neg\varphi\ @\ \tau]\!]$.

- implication: $\mathbf{M},\alpha \models \varphi \supset \psi\ @\ \tau$ if and only if not $\mathbf{M},\alpha \models \varphi\ @\ \tau$ or $\mathbf{M},\alpha \models \psi\ @\ \tau$ if and only if not $\mathbf{M}',\alpha' \models [\![\varphi\ @\ \tau]\!]$ or $\mathbf{M}',\alpha' \models [\![\psi\ @\ \tau]\!]$ that is $\mathbf{M}',\alpha' \models [\![\varphi \supset \psi\ @\ \tau]\!]$.

- quantification: $\mathbf{M},\alpha \models \forall x.\varphi@\tau$ if and only if for all $d \in \mathcal{D}$ it is the case that $\mathbf{M},\alpha^{[d/x]} \models \varphi@\tau$ if and only if $\mathbf{M}',(\alpha^{[d/x]})' \models [\![\varphi@\tau]\!]$ if and only if $\mathbf{M}',\alpha'^{[d/x]} \models [\![\varphi@\tau]\!]$ if and only if $\mathbf{M}',\alpha' \models \forall x.[\![\varphi@\tau]\!]$ if and only if $\mathbf{M}',\alpha' \models [\![\forall x.\varphi@\tau]\!]$.

- necessitation ($\Box$): it reduces to the previous cases for quantification and implication, since the domain of quantification of $2FOL$ includes w .

As far as frame properties are concerned, sentences in $\mathsf{FrmAx}(\mathbf{QL})$ enforce exactly those properties of $\prec$ which are needed by the accessibility relation R in order to make the model $\mathbf{M}$ a model of **QL** (recall Definition 2 and subsequent discussion). Since $\prec$ is interpreted as R, this completes the proof of Proposition 26.

•

Propositions 14, 25 and 26 together lead to

Theorem 27 (Soundness and completeness) $c_{\mathbf{QL}}$ *is sound and complete for any FO-axiomatizable logic* **QL**.

3.3.4 Discussion

As it stands, the strengthening procedure (see Definition 6) might seem to hinder completeness, because of the forbidden duplication of $\forall$-formulae when using rule $l\forall^*_\theta$. In general, a *LK*-like sequent calculus with no weakening and contraction rules, as is $2LK$, will be incomplete if duplication is disallowed. The simplest example of a first-order theorem which cannot be proved if duplication is restricted is $\exists x.\forall y.p(x) \supset p(y)$, for any unary predicate p in the signature. Intuitively incompleteness arises from the impossibility of matching terms introduced by generative and non-generative rules.

But this doubt is actually void. *In primis*, notice that the strengthening procedure does not aim to *prove* a formula, but rather to "unfold" it, in the sense given by the procedure itself. By Proposition 7, the procedure terminates, but also notice that it is deterministic, up to placeholders renaming and the order of the premises of the sequent rule obtained.

In secundis, the procedure acts on *Skolemised sentences* placed on the left-hand side of a sequent, rather than on generic formulae; therefore, no complex interplay between terms introduced by generative and non-generative rules can happen at all,

since *no generative rules* are ever used. The only quantifier rule that can be ever used in the procedure is exactly $l\forall^*_\theta$.

In tertiis, as is shown in the example of Figures 3.10 and 3.11, every time the completeness argument requires a multiple use of rule $l\forall^*_\theta$ in the proof of a $2FOL$-theorem, there is a similar $2LK$-proof in which multiple uses of the appropriate sequent rule, obtained via strengthening, appears. Intuitively, whenever a universal quantifier needs be used more than once in a $2LK$-proof, the appropriate *sequent rule* can be used multiple times in the corresponding $\mathcal{C}_{\mathbf{QL}}$-proof.

3.4 Chapter overview

In this Chapter we have devised a family of labelled sequent calculi for QMLs with constant domains and rigid designators, whose frame properties can be axiomatised in first-order logic with equality. We have proved that these calculi are sound, complete, modular, uniform and normalising.

We could say that the first two properties are important to the theoretician, establishing what can and cannot be proved in the calculi; the third and fourth matter to the modal logician, giving a way to build new sequent calculi thanks to some simple guidelines; and the fifth is crucial to the practitioners, that is, to those who want to do automated reasoning in QMLs. Normalisability means that logical and frame reasoning will never be intertwined in any proof, or, better, that for any proof in which this happens, there exists a similar one in which it does not. Therefore, in principle, any external machinery can be used as a black box to perform the task of checking entailment.

The work exposed in this Chapter extends and generalises Basin, Matthews and Viganò's work of the late 90s.

Chapter 4

A framework for automated reasoning in QMLs and FOLTL

In this Chapter we build upon the theoretical work expounded in Chapter 3 and develop a formal framework for automated reasoning in QMLs and First-Order Linear-Time Temporal Logic (from now on, **FOLTL**).

Our framework consists of:

1. a labelled sequent calculus for **FOLTL**, called $\mathcal{C}_{\mathbf{FOLTL}}$, obtained by extending our language with some new symbols and $\mathcal{C}_{\mathbf{QS4.3}}$ with a set of new sequent rules;

2. an interactive, tactic-based theorem prover for QMLs and **FOLTL**, called FTL, in which $\mathcal{C}_{\mathbf{FOLTL}}$ is implemented;

3. a λProlog module which acts as a "bridge" between the proof planner λCLAM and FTL, in the spirit of Proof Planning.

The next three sections expand the above items, in turn, with the exception that Section 4.2 does not describe FTL in detail — that is left to Chapter 5 — but, rather, how the paradigm of tactic-based theorem proving has been adapted to $\mathcal{C}_{\mathbf{QL}}$ and $\mathcal{C}_{\mathbf{FOLTL}}$.

4.1 Extending $\mathcal{C}_{\mathbf{QL}}$ to FOLTL

In Chapter 3 we have defined a family of labelled sequent calculi for FO-axiomatizable QMLs; we have also proved a number of properties of the calculi. Here we extend the approach to **FOLTL**.

FOLTL is a very strong quantified modal logic, in the sense that it is a quantified modal logic whose frame is isomorphic to the natural numbers. Its propositional fragment, usually called *LTL*, is obtained by adding to the propositional modal logic of linear, discrete frames, called **S4**.**3**.**1**, two modal operators, called "next" and "until" (see [DS02] — $\Box$ and $\Diamond$ can then be defined in terms of "next" and "until"); so, a sensible way to extend $\mathcal{C}_{\mathbf{QL}}$ to **FOLTL** could be to axiomatize in first-order logic the properties of such a frame and then to employ the strengthening procedure to get another member of the $\mathcal{C}_{\mathbf{QL}}$ family. Adding two modal operators would be no problem, since it would suffice to mimic their semantic definitions, exactly as it has been done for $\Box$ and $\Diamond$.

Unfortunately, this is impossible. The class of frames characterising the logic **S4**.**3**.**1** is exactly the set of frames isomorphic to the natural numbers; using the Compactness Theorem, it can be shown that no finite set of first-order sentences can axiomatize such a class of frames. This holds *a fortiori* for *LTL* and for their quantified counterparts, **QS4**.**3**.**1** and **FOLTL**.

In fact, it is possible to characterise **S4**.**3**.**1** *modally*, by adding the so called *Dummett axiom* to the axioms for **S4**.**3**, that is, $T, 4$ and 3; the Dummett axiom,

$$\Box(\Box(p \supset \Box p) \supset p) \supset (\Diamond\Box p \supset \Box p),$$

forces reflexive, transitive and weakly-connected frames, characteristic of **S4**.**3**, to assume the shape of a set of *balloons*; a balloon is a finite chain of single, reflexive worlds at whose end lies a cluster of worlds, all accessible to one another — in graph theory words, a clique. Also, it can be shown that whatever is valid in **S4**.**3**.**1** is valid on the

frame $\langle \mathbb{N}, \leq \rangle$ where $\mathbb{N}$ is the set of natural numbers and $\leq$ is the standard less-than-or-equal relation over the naturals (see [Gor93]).

But, similarly, the Dummett axiom is not expressible in first-order logic at all; so there is no way of applying here the general procedure outlined in Chapter 3, even in a hypothetical propositional case. In a sense, this is reassuring, since it has been proved that **FOLTL** is not only undecidable (it is stronger than first-order logic), but also that it is non recursively enumerable, therefore there can exist no finitary sequent calculus for it ([HWZ00]). If the strengthening procedure were actually applicable to **FOLTL**, we would be contradicting that result.

So, the problem arises here of how to build a labelled sequent calculus for a quantified modal logic characterised by a linear, discrete frame, isomorphic to the set $\mathbb{N}$ — which cannot be done by means of the strengthening procedure defined in Chapter 3.

Substantially, two types of strengthening are required: *(i)* we have to strengthen the *syntax* and *semantics* of our language, to take into account the higher complexity of the frame and the new modal operators for "next" and "until"; and *(ii)* we also have to build *new sequent rules* to be added to a suitable member of the $\mathcal{C}_{\mathbf{QL}}$ family, to give an account of how the new symbols (predicates, functions, modal operators) behave. We tackle these issues in the following Subsections.

4.1.1 Strengthening the syntax and semantics

One first, very basic observation is that, since we work in Labelled Deduction, we can work out Item *(i)* above just by enriching our *language*, since in this framework semantical properties of the frame can be expressed in the language itself — this is precisely one of the pillars of Labelled Deduction.

In our setting, properties of the frame are expressed by *labels*, *constraints* and *frame rules*, and our starting point is therefore that of enriching the labelling language. One reasonable way appears that of somehow "building" the natural numbers into the language by means of a Peano-style successor function and by viewing the accessibility

relation as the standard $\leq$ relation over $\mathbb{N}$. This can be easily accomplished by adding to $\mathcal{F}'$ a new function symbol, σ, interpreted as the unary function *successor-of*, and by interpreting the accessibility relation $\prec$ exactly as $\leq$.

Moreover, we introduce some new modal operators: the unary operator $\bigcirc$ ("next") and the binary operators $\Box^*$ ("bounded always"), $\mathcal{U}$ ("until") and $\mathcal{W}$ ("weak until"). The intuitive meaning of these operators, according to the standard interpretation of *LTL*, is:

- $\bigcirc p$ @ τ holds if and only if p @ $\sigma(\tau)$ holds, that is, if and only if p holds at the instant immediately after the instant denoted by τ; and
- $\Box^{\tau'} p$ @ τ holds if and only if there is an instant τ' in the future of τ such that p holds from τ to τ'.
- $p \mathcal{U} q$ @ τ holds if and only if there is an instant in the future of τ at which q holds, and p holds in the meantime;
- $p \mathcal{W} q$ @ τ holds if and only if $\Box p$ @ τ holds, or $p \mathcal{U} q$ @ τ holds. This operator extends $\mathcal{U}$ allowing for the "persisting" condition p to possibly hold forever, with q never happening.

Recall Section 3.1.1; to take these additions into account, it suffices to replace Definition 1 with the following:

Definition 28 (FOLTL formulae) *Like Definition 1, except:*

$$\begin{aligned} \textbf{lab} &::= 0 \mid t \mid \sigma(\textbf{lab}) && \text{where } t \in \mathcal{V}', 0, \sigma \in \mathcal{F}' \\ \textbf{lf} &::= \textbf{la} \mid \neg\textbf{lf} \mid \textbf{lf} \supset \textbf{lf} \mid \forall x.\textbf{lf} \mid \Box\textbf{lf} \\ &\quad \mid \bigcirc\textbf{lf} \mid \Box^{\textbf{lab}}\textbf{lf} \mid \textbf{lf}\ \mathcal{U}\ \textbf{lf} \mid \textbf{lf}\ \mathcal{W}\ \textbf{lf} && \text{where } x \in \mathcal{V} \end{aligned}$$

Now recall Subsection 3.1.2; the semantics is reshaped as follows:

Definition 29 (FOLTL structure) *We call a* **FOLTL** structure *a structure*

$$\mathbf{M} = \langle \mathbb{N}, \leq, \mathcal{D}, I \rangle$$

where $\mathbb{N}$ *and* $\leq$ *denote the set of natural numbers and the standard less-than-or-equal binary relation; moreover,* I_l *maps* $\doteq$ *to the standard equality relation over* $\mathbb{N}$ *and* $\sigma \in \mathcal{F}'$ *to the standard Peano successor function. All the rest is unchanged w.r.t. Definition 2 and subsequent ones in Subsection 3.1.2.*

Note that this new definition remarkably reflects the notion of a *standard model* in [AM90]. A standard model is precisely what is needed in that paper to take into account the semantics of **FOLTL** (therein called *First-Order Temporal Logic*).

Lastly, recall Subsection 3.1.2 again; the new notion of *truth in a structure* is obtained like this:

Definition 30 (Truth in a FOLTL structure) *A formula* φ *is* true in a **FOLTL** structure $\mathbf{M}$ under the assignment α, *written* $\mathbf{M}, \alpha \models \varphi$, *if and only if:*

$$\begin{array}{lll}
\mathbf{M}, \alpha \models \tau_n \doteq \tau_m & \textit{iff} & n = m \\
\mathbf{M}, \alpha \models \tau_n \leq \tau_m & \textit{iff} & n \leq m \\
\mathbf{M}, \alpha \models \tau_n < \tau_m & \textit{iff} & n < m \\
\mathbf{M}, \alpha \models \bigcirc \varphi \ @ \ \tau_i & \textit{iff} & \mathbf{M}, \alpha \models \varphi \ @ \ \sigma(\tau_i) \\
\mathbf{M}, \alpha \models \Box^{\tau_n} \varphi \ @ \ \tau_i & \textit{iff} & \textit{for all } m \in \mathbb{N}, \\
 & & (\mathbf{M}, \alpha \models \tau_i \leq \tau_m \textit{ and } \mathbf{M}, \alpha \models \tau_m \leq \tau_n) \textit{ implies} \\
 & & \mathbf{M}, \alpha \models \varphi \ @ \ \tau_m \\
\mathbf{M}, \alpha \models \varphi \mathcal{U} \psi \ @ \ \tau_i & \textit{iff} & \textit{there is } n \in \mathbb{N} \textit{ such that} \\
 & & \mathbf{M}, \alpha \models \tau_i \leq \tau_n \textit{ and } \mathbf{M}, \alpha \models \psi \ @ \ \tau_n \textit{ and} \\
 & & \mathbf{M}, \alpha \models \Box^{\tau_n} \varphi \ @ \ \tau_i \\
\mathbf{M}, \alpha \models \varphi \mathcal{W} \psi \ @ \ \tau_i & \textit{iff} & \mathbf{M}, \alpha \models \Box \varphi \ @ \ \tau_i \textit{ or} \\
 & & \mathbf{M}, \alpha \models \varphi \mathcal{U} \psi \ @ \ \tau_i
\end{array}$$

All the rest is unchanged w.r.t. Definition 3 and subsequent ones in Subsection 3.1.2.

Note that we have replaced the symbol $\prec$, so far used to indicate the accessibility relation, with the usual symbol $\leq$. Note also that we now use *m* and *n* to denote worlds, as they can be safely identified with natural numbers — τ_n and τ_m indicate the *labels* which denote *n* and *m*.

One further note is necessary. Our definition of $\mathcal{U}$ is slightly stronger than the usual one. We require that, if $p \mathcal{U} q$ holds, *p* must hold *also* at the future instant in which *q* holds, whereas this is usually not required. This choice is motivated by the shape of the system we will be trying to model as a case-study, and will be clarified in Chapter 6, Section 6.3.1.

4.1.2 Building a $\mathcal{C}_{\mathbf{QL}}$ for FOLTL

Now that our language is rich enough to express **FOLTL** formulae, we have to find a suitable $\mathcal{C}_{\mathbf{QL}}$ for strengthening. We first make two simple observations:

1. the propositional modal logic of linear, discrete frames, **S4**.**3**.**1**, is obtained by adding the Dummett axiom to **S4**.**3**; as well,

2. it seems reasonable to believe that the relationship between **S4**.**3**.**1** and *LTL* somehow carries on between **QS4**.**3**.**1** and **FOLTL**, that is, that **QS4**.**3**.**1** is the restriction of **FOLTL** to the operator $\Box^1$;

In view of this we choose $\mathcal{C}_{\mathbf{QS4.3}}$, our sound and complete calculus for **QS4**.**3**, as the basis for a labelled sequent calculus for **FOLTL**, that we will indicate as $\mathcal{C}_{\mathbf{FOLTL}}$. Of course, it is not incidental that $\prec$, in **QS4**.**3**, is reflexive, transitive and weakly connected, which is something any partial order such as $\leq$ must enjoy.

We therefore extend $\mathcal{C}_{\mathbf{QS4.3}}$ with two kinds of new rules: *(i)* rules which model the behaviour of the modal operators $\bigcirc$, $\Box^*$ and $\mathcal{U}$, both on the left and on the right of sequents, and *(ii)* rules which model the behaviour of $\leq$ and σ.

[1] this belief is corroborated by a personal communication with Rajeev Goré.

Rules of type *(i)* are shown in Table 4.1, while rules of type *(ii)* (that is, frame rules) appear in Table 4.2.

Logical rules

$$\frac{\Gamma, \varphi \mathbin{@} \sigma(\tau) \longrightarrow \Delta}{\Gamma, \bigcirc\varphi \mathbin{@} \tau \longrightarrow \Delta}\, l\bigcirc \qquad \frac{\Gamma \longrightarrow \varphi \mathbin{@} \sigma(\tau), \Delta}{\Gamma \longrightarrow \bigcirc\varphi \mathbin{@} \tau, \Delta}\, r\bigcirc$$

$$\frac{\Gamma, \Box^{\tau_n}\varphi \mathbin{@} \tau, \varphi \mathbin{@} \tau_c \longrightarrow \Delta \quad \Gamma, \Box^{\tau_n}\varphi \mathbin{@} \tau \longrightarrow \tau \leq \tau_c, \Delta \quad \Gamma, \Box^{\tau_n}\varphi \mathbin{@} \tau \longrightarrow \tau_c < \tau_n, \Delta}{\Gamma, \Box^{\tau_n}\varphi \mathbin{@} \tau \longrightarrow \Delta}\, l\Box^*$$

$$\frac{\Gamma, \tau \leq \tau_d, \tau_d < \tau_n \longrightarrow \varphi \mathbin{@} \tau_d, \Delta}{\Gamma \longrightarrow \Box^{\tau_n}\varphi \mathbin{@} \tau, \Delta}\, r\Box^*$$

$$\frac{\Gamma, \tau \leq \tau_a, \psi \mathbin{@} \tau_a, \Box^{\tau_a}\varphi \mathbin{@} \tau \longrightarrow \Delta}{\Gamma, \varphi\, \mathcal{U}\, \psi \mathbin{@} \tau \longrightarrow \Delta}\, l\mathcal{U}$$

$$\frac{\Gamma \longrightarrow \tau \leq \tau_b, \varphi\, \mathcal{U}\, \psi \mathbin{@} \tau, \Delta \quad \Gamma \longrightarrow \psi \mathbin{@} \tau_b, \varphi\, \mathcal{U}\, \psi \mathbin{@} \tau, \Delta \quad \Gamma \longrightarrow \Box^{\tau_b}\varphi \mathbin{@} \tau, \varphi\, \mathcal{U}\, \psi \mathbin{@} \tau, \Delta}{\Gamma \longrightarrow \varphi\, \mathcal{U}\, \psi \mathbin{@} \tau, \Delta}\, r\mathcal{U}$$

Table 4.1: rules for modal operators introduced in **FOLTL**. $\tau_a, \tau_d \in \mathcal{V}'$ cannot appear free in the conclusion of $l\mathcal{U}$ and $r\Box^*$.

Here it seems reasonable to provide, in rules $\Box^*$ and $r\mathcal{U}$, a duplicate main formula in the premises for completeness reasons, as it happens, for instance, for rules $l\forall$ and $r\exists$. In general, it is likely that any classical logic based sequent calculus needs duplication of formulae in order to retain completeness, either in the form of structural rules (weakening and contraction) or as duplicate premises in non-generative rules involving existential quantifiers on the right or universal quantifiers on the left. This is the case for $\mathcal{C}_{\mathbf{QL}}$ and, consequently, for $\mathcal{C}_{\mathbf{FOLTL}}$.

Once again, it is worth recalling that $\mathcal{C}_{\mathbf{FOLTL}}$ cannot be complete for **FOLTL**; but nothing prevents us from trying to retain completeness for the largest possible fragment of **FOLTL** we are interested in, that is, for the problem we are trying to solve.

Completeness also is the main reason why we have introduced the operator $\Box^*$. Recall that, according to its semantics, $\mathcal{U}$ consists of an outer existential quantifier and an inner universal quantifier over time instants; by using $\Box^*$, we somehow separate the

existential part of $\mathcal{U}$ from the universal one. This is very useful, as we have noticed during some earlier work (see [CS02a]), when $\mathcal{U}$-formulae appear as assumptions on the left of a sequent. In that case, when rule $l\mathcal{U}$ is employed, it generates a $\Box^*$-formula on the left which, thanks to the duplicated formula in rule $l\Box^*$, can be reused many times. In that paper we actually face the problem: if $\Box^*$-formula duplication were not allowed, it would be impossible to carry on some of the proofs presented therein.

Frame rules

$$\frac{}{\Gamma \longrightarrow \sigma(\tau) \doteq 0, \Delta}\ \mathsf{not}_0$$

$$\frac{\Gamma, \sigma(\tau') \leq \tau'' \longrightarrow \Delta}{\Gamma, \tau' < \tau'' \longrightarrow \Delta}\ \mathsf{l}< \qquad \frac{\Gamma \longrightarrow \sigma(\tau') \leq \tau'', \Delta}{\Gamma \longrightarrow \tau' < \tau'', \Delta}\ \mathsf{r}<$$

$$\frac{\Gamma \longrightarrow \varphi\ @\ 0, \Delta \quad \Gamma, \varphi\ @\ t \longrightarrow \varphi\ @\ \sigma(t), \Delta}{\Gamma \longrightarrow \Box\varphi\ @\ \tau, \Delta}\ \mathsf{ind}$$

Table 4.2: frame rules for **FOLTL**. $t \in \mathcal{V}'$ cannot appear free in the conclusion of ind.

On the other hand, Table 4.2 gives rules for some basic properties of 0, σ and $\leq$, plus a simple time-induction rule.

Soundness and completeness

Rules added to $\mathcal{C}_{\mathbf{QS4.3}}$ to get to $\mathcal{C}_{\mathbf{FOLTL}}$, that is, rules in Tables 4.1 and 4.2, can be proved sound via a simple semantic argument — in fact, they simply reflect the semantics of the associated operators, predicates and functions.

As far as completeness is concerned, we are not in the position, so far, to make any formal claim about it; what we can say is that:

1. it is possible to prove the Dummett axiom in $\mathcal{C}_{\mathbf{FOLTL}}$ (see [CS01], where the proof is carried out in a close relative of this calculus, therein called $\mathcal{TL}_{\mathsf{ind}}$). This suggests that $\mathcal{C}_{\mathbf{FOLTL}}$ could be complete for (propositional!) **S4.3.1**; since

the proof makes use of the induction rule, the hypothesis is much weaker in the case of **QS4.3.1**;

2. an interesting open question: is $\mathcal{C}_{\mathbf{FOLTL}}$ complete and/or terminating for propositional *LTL*?

3. another interesting question: is $\mathcal{C}_{\mathbf{FOLTL}}$ complete for the monodic fragment of **FOLTL**?

4. in [CS02a], a simplified version of this machinery, namely without the σ function and the $\bigcirc$ operator, was used to manage the first successful experiment in Feature Interactions. Thanks to the introduction of $\Box^*$, it is probably possible to work out a completeness proof for $\mathcal{C}_{\mathbf{FOLTL}}$ with respect to the $\bigcirc$-free fragment of **FOLTL** — the newly introduced rules do not seem to invalidate any of the assumptions made for the proof of Section 3.3. Anyway, this is future work.

4.2 Tactic-based theorem proving in $\mathcal{C}_{\mathbf{QL}}$ and $\mathcal{C}_{\mathbf{FOLTL}}$

Having set up a theoretical framework for reasoning about QMLs and **FOLTL**, and aiming to do automated reasoning via Proof Planning in $\mathcal{C}_{\mathbf{QL}}$ and $\mathcal{C}_{\mathbf{FOLTL}}$, the first step has been to build a theorem prover which implements the calculus — the *object-level* theorem prover. We describe here how tactic-based theorem proving has been adapted to our case.

Let us call *goal* a pair (proof, sequent); then a *tactic* is a predicate over goals. Operationally, tactics are "steps" from a goal to another: we have a goal, we want to reduce it to a simpler one (that could be a set of simpler subgoals), and a tactic does exactly that. The hope is that, eventually, all subgoals will be trivially true; assuming the soundness of tactics, that means the link between the initial goal and its proof has been established.

FTL uses *basic tactics* to enforce sequent rules (one rule, one tactic) and *compound*

tactics to enforce repeated, conditional and exhaustive application of tactics (both basic and compound). The following Subsections explain how the mechanism works. The reader interested in other applications of tactic-based theorem proving might want to have a look at [Fel93].

4.2.1 Basic tactics

FTL uses *basic tactics* to enforce sequent rules: one rule, one basic tactic. The standard embedding of a rule in a basic tactic happens through a straightforward definition:

```
<tactic>
  <position>
  <input-goal>
  <output-goal> :-
  <preconditions>,
  <effects>.
```

The idea is that a basic tactic, applied to the *input-goal*, produces the *output-goal*, provided that the *preconditions* about the input-goal are met; the *effects* are applied to compute the output-goal. The integer parameter *position* specifies the main formula, when more than a candidate is found.

Usually, the preconditions specify what the shape of the input-goal has to be in order for the basic tactic (rule) to be applicable, that is, whether the sequent in the output-goal contains a candidate main formula for this rule; the effects remove the main formula from the premises, if it is the case, and print some information about the operations performed by the basic tactic. But this is just the simplest case.

Without going into detail, here is the definition[2] of the basic tactic embedding rule $l\neg$:

```
tlnot
```

[2] the representation given here is slightly simplified with respect to the actual implementation.

```
Pos
((lnot Pos P) proves (Gamma --> Delta))
(P proves (Gamma' --> ((Phi at Tau)::Delta))) :-
member Pos (not Phi at Tau) Gamma,
delete Pos Gamma Gamma'.
```

(As is customary in logic programming, sets are implemented via lists; in the example above, and in all subsequent ones, we denote the sequent symbol by `-->` and set union by `::`.) As it can be seen, basic tactic `tlnot` links the two goals ($l\neg(Pos,P)$, $\Gamma \longrightarrow \Delta$) and ($P$, $\Gamma' \longrightarrow \{\varphi \ @ \ \tau\} \cup \Delta$) provided that formula $\neg\varphi \ @ \ \tau$ is actually member number *Pos* of Γ (precondition), and, if so, builds the antecedent of the premise, Γ', by removing the formula from Γ (effect).

Note that rule $l\neg$ is here used as a *proof constructor*: it takes a position *Pos* and a proof P as input and outputs a new proof $l\neg(Pos,P)$. The idea is that if there is a proof P of the sequent in the input-goal (e.g., the premise of the rule), and the tactic assumes so, a proof $l\neg(Pos,P)$ of the sequent in the output-goal (the conclusion of the rule) can be built. Therefore, the tactic `tlnot` really acts as a wrapper for the rule: it defines the rule and states the side conditions under which it can be applied.

A more interesting case occurs when the rule wrapped by a basic tactic has two or more premises. Here is the definition of the basic tactic embedding rule $r\wedge$:

```
trand
  Pos
  ((rand Pos P1 P2) proves (Gamma --> Delta))
  (and_goal (P1 proves (Gamma --> ((Phi at Tau)::Delta')))
            (P2 proves (Gamma --> ((Psi at Tau)::Delta')))) :-
  F = (Phi and Psi at Tau),
  member Pos (Phi and Psi at Tau) Delta,
  delete Pos Delta Delta'.
```

Here we use the *goal constructor* `and_goal` to build a multiple goal: to solve it means to solve both argument goals. Rule $r\wedge$ acts here as a constructor taking as inputs an integer and *two* proofs: it builds a proof $r \wedge (Pos, P_1, P_2)$ of the sequent in the output-goal if there are two proofs P_1 and P_2 which prove the sequents in the premises in turn.

With the aid of some simple recursion machinery, tactics can be employed to build a full proof of a sequent; the mechanism sketched above shows that a proof is actually a higher-order λProlog term, representing a tree — the proof-tree of the conclusion of a sequent rule. Tactics embedding closing rules (e.g., `ax`) provide the bottom of the recursion: their output-goal is just an ad-hoc constant, `true_goal`.

λProlog is a declarative programming language and therefore this approach works either way, but usually *backward reasoning* is employed, that is:

1. start with a goal $(P, \longrightarrow \varphi \ @ \ 0)$ as input-goal, where φ is the logical formula to be proved and P is an uninstantiated metavariable;

2. if the input-goal is actually `true_goal`, stop; otherwise,

3. apply a suitable tactic to the input-goal and, for each output-goal generated, go back to Item 2.

This is exactly what happens in FTL: in the interactive mode, the user supplies the appropriate tactic at each step, until all subgoals are `true_goal`s. The result of the

computation, if it terminates successfully, is a higher-order λProlog term representing the proof of the sequent in the original input-goal. This term is a faithful representation of the proof tree; as an example, here is the term output by FTL applied to the triviality $\forall x.p(x) \supset (p(a) \wedge p(b))$:

```
(rimp 1
  (rand 1
    (lall 1 (ax 1 1))
    (lall 1 (ax 1 1))
  )
)
```

Compare it with the corresponding proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(a) \longrightarrow p(a)}\,\mathsf{ax}}{\forall x.p(x) \longrightarrow p(a)}\,l\forall \quad \dfrac{\dfrac{\dfrac{}{p(b) \longrightarrow p(b)}\,\mathsf{ax}}{\forall x.p(x) \longrightarrow p(b)}\,l\forall}{\forall x.p(x) \longrightarrow p(a) \wedge p(b)}\,r\wedge}{\longrightarrow \forall x.p(x) \supset p(a) \wedge p(b)}\,r\supset$$

Of course, in general there is no guarantee that such a term exists; but it can be shown that a tactic-based theorem prover implemented respecting the guidelines given in [Fel93] — and FTL is heavily based on Felty's work — is *correct*, in the sense that for every theorem φ and proof P proving it, there are higher-order λProlog terms `P` and `Phi` such that the λProlog goal `P proves Phi` is derivable from the λProlog program implementing the prover, and vice versa. See Appendix B for such a proof specialised for FTL, and refer once again to [Fel93] for a thorough explanation.

4.2.2 Compound tactics

As well as basic tactics, compound tactics link two goals, but they also bear some operational content. They are independent from the object logic as they enforce, among other things, repeated, conditional and exhaustive application of tactics.

The main differences to basic tactics are that *(i)* besides the input- and output-goal, they usually have one or more additional arguments (the tactics to be manipulated); *(ii)* the goals are usually specified in a completely generic way, since their shape is in many cases not known *a priori*; and *(iii)* there are no preconditions and effects, but rather an operational specification of how they behave. The operational content is the body of the clause defining the tactic:

```
<tactic>
  <tactic1> ... <tacticN>
  <input-goal>
  <output-goal> :-
  <operational content>.
```

Thanks to standard recursion techniques, a compound tactic can enforce any operation on tactics (proofs), preserving soundness. As an example, compound tactics `then_tac` and `orelse_tac` enforce sequential and conditional application of two tactics:

```
then_tac
  Tac1 Tac2
  InGoal
  OutGoal :-
  Tac1 InGoal MidGoal, Tac2 MidGoal OutGoal.

orelse_tac
  Tac1 Tac2
  InGoal
  OutGoal :-
  Tac1 InGoal OutGoal; Tac2 InGoal OutGoal.
```

`then_tac` simply applies the first tactic `Tac1` to the input-goal `InGoal` and gets a middle-goal `MidGoal`, then applies the second tactic `Tac2` to the middle-goal and

obtains the output-goal `OutGoal`. Both tactics must succeed in order for `then_tac` to succeed. Note that, using the standard λProlog unification mechanism, the actual shape of the goals is neglected — they appear as generic metavariables `InGoal` and `OutGoal`.

Similarly, `orelse_tac` applies `Tac1` and `Tac2` in a disjunctive fashion: if `Tac1` fails, `Tac2` is attempted.

The argument tactics of a compound tactic do not have to be basic tactics — they can be compound as well. This allows the construction of "higher-level" compound tactics such as `repeat_tac`:

```
repeat_tac
  Tac
  InGoal
  OutGoal :-
  orelse_tac
     (then_tac Tac (repeat_tac Tac))
     fail_tac
     InGoal OutGoal.
```

`repeat_tac` acts as follows: it applies `Tac` to the input-goal (`then_tac Tac ...`) and then recursively calls itself (`repeat_tac Tac`). This has the effect of keeping on applying `Tac` to the new goal obtained at each step, until `true_goal` is obtained. In case the repeated application of `Tac` does not yield `true_goal`, the first argument of the outer `orelse_tac` tactic fails, giving way to its second argument `fail_tac`, which always fails. In other words, this tactic eagerly applies a tactic `Tac` until the input-goal is *solved*.

A number of other compound tactics can be defined; again, refer to [Fel93] for a thorough list.

A final remark is worthwhile, about the use of λProlog, and in general about a higher-order programming language, rather than simple Prolog. Besides other reasons

which do not concern the tactics mechanism directly (and which will be outlined in Chapter 5), we can observe here two advantages we get:

1. firstly, quantification over predicates lets a metavariable be used in place of a predicate name, followed by its arguments:

   ```
   Tac1 InGoal MidGoal, ...
   ```

   This allows a freer use of the clauses in the program: metavariable `Tac1` in the example may range over all tactics, which are defined by program clauses;

2. secondarily, λProlog is a *strongly typed* language, i.e., all terms must have been assigned a type in the signature of the program or, alternatively, it must be possible for the compiler to dynamically deduce their type; therefore, it is possible to use different types for a term by giving a partial specification of their arguments. For instance, tactic `then_tac` is declared like this:

   ```
   type then_tac      (goal -> goal -> o) ->
                      (goal -> goal -> o) ->
                      goal -> goal -> o.
   ```

   (`o` is the predefined type for Boolean values, i.e., *true* or *false*, and is the standard target type for predicates). It takes two tactics, each one of type (`goal -> goal -> o`), as arguments and outputs a tactic (its target type is `goal -> goal -> o`). Now, the type of `then_tac` is *not* that of a tactic, which would be `goal -> goal -> o` alone, but the type of term

   ```
   then_tac Tac1 Tac2
   ```

is — it is the λ-application of a term of the type indicated above to two terms of type `goal -> goal -> o`. Therefore we can freely employ it inside another tactic as a tactic argument, as is the case of `repeat_tac`, visible above.

4.3 Proof Planning and FTL

Notwithstanding its modular construction, $\mathcal{C}_{\mathbf{FOLTL}}$ has too many rules, and probably in any standard proof the branching factor would be high. This makes the situation quite hard to manage for any automated theorem prover, even taking into account backward, goal-directed reasoning, the use of metavariables and so on. On the other hand, recent results about the complexity of **FOLTL** make this fact unsurprising, and convey the idea that some highly abstract form of reasoning is necessary.

This is why we turn our attention to Proof Planning. The idea in Proof Planning is that an object-level theorem prover is guided by a high level specification of a proof (a *proof plan*) generated by the proof planner.

But we face a non trivial problem here: there must be some soundness-preserving form of *translation* between the proof planner, which outputs proof plans as trees of methods and sequents, and the object-level theorem prover, which must build a proof out of it — and a proof is usually much more complicated than a proof plan.

Although sometimes neglected in the Proof Planning literature, this is a crucial step: to finally have a *proof* of the original theorem is the only argument that can be used to support soundness of the methods employed by the planner, and of the instantiations of the methods that actually appear in the proof plan. On the other hand, a correct translation of the proof plan is the only chance for the object-level theorem prover to actually obtain a proof, assuming, as we have said above, that the problem is too complex for it to be solved automatically.

Such a "bridge" is actually a pair of λProlog modules. One of them implements **FOLTL** as an object logic in λCLAM; the other takes care of the actual translation to

and fro FTL's object logic. Here we give a sketch of the interaction between FTL and λCLAM, which happens through the bridge.

Normally, the bridge would be used to translate a proof plan and a sequent in λCLAM's internal syntax into a proof and a sequent in FTL's internal syntax; but, since λCLAM's internal syntax is quite hard to read and write, we have opted for a more flexible bridge, capable of operating both ways. We write the sequent to be proved in FTL, translate the sequent to λCLAM, and, once and if λCLAM returns a proof plan for it, we translate the λCLAM sequent and the proof plan back into an FTL sequent and a tactic.

This tactic, which is usually quite complicated but not as detailed as the proof we aim at obtaining, is then applied to the FTL sequent, and, if everything goes well, a proof of the sequent is obtained. Once this is done, FTL is invoked to check that the proof actually proves the sequent.

Figure 4.1 graphically sketches the situation.

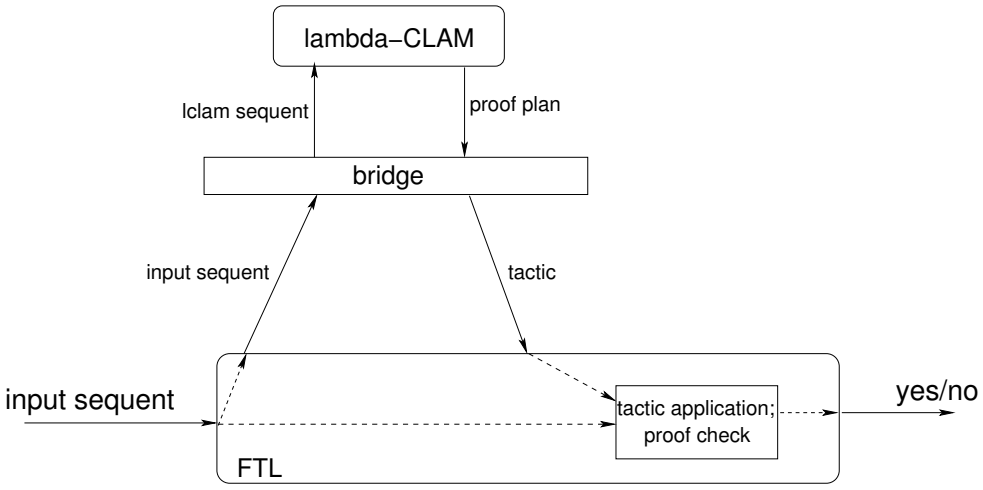

Figure 4.1: a broad representation of the interaction between λCLAM and FTL.

The sequence of operations goes as follows:

1. an input sequent, i.e., a theorem to be proved, is written in the input language of FTL, which is a quite plain λProlog coding of **FOLTL** — in the original spirit

of the project, we want to be able to retain the expressiveness of such a logic, therefore it must be possible to express our problems in a straightforward way;

2. the input sequent is translated by the bridge module into λCLAM's internal syntax. First every term and atomic formula in the language of FTL is wrapped by a metapredicate which outputs a term in the syntax of λCLAM; then the logical and modal operators are translated into their equivalents in λCLAM;

3. λCLAM's proof plan engine is called, and hopefully a proof plan for the translation of the input sequent is returned;

4. the proof plan is translated into a tactic and passed to FTL, which finally

5. applies the tactic to the input sequent and checks that the resulting proof does prove it.

There is only one item which deserves more explanations, and it is Item 4. In Proof Planning, every method has an associated (basic) tactic; in λCLAM, methods are declared like this:

```
atomic <theory-name> <method-name>
  <input-goal>
  <preconditions>
  <effects>
  <output-goal>
  <tactic>.
```

The declaration of a method bears some resemblance with that of a tactic — and this is not surprising, since they both encapsulate an operator that can be applied in a tree-like structure: tactics build proof trees and methods build proof plans. An uninteresting difference is that a method is here declared in an abstract way, that is, there is a predicate `atomic` which states that *method-name* is the name of a method belonging

to the theory *theory-name*. This is, of course, an implementation choice of λCLAM and is not characteristic of Proof Planning in general.

The added value of a method with respect to a tactic is that a method enforces an informal, incomplete and even in some cases unsound step of human-like reasoning; therefore it must have an associated (set of) tactic(s), representing its translation in the object logic, which is completely formal.

Thanks to this precise association, in `bridge.mod` the translation of a proof plan into a tactic may happen using a few λProlog clauses. The structure of a proof plan is recursively examined and, at each node, the information on which method was applied is extracted; in parallel, a tree of tactics is built, in which each node is in turn labelled with the tactic associated with the method found in the proof plan.

Both methods and tactics, in our implementation, bear information about which is the main formula at each node, so that the information may be simply passed from methods to tactics.

Although tricky here and there, the actual implementation of the proof checking mechanism, roughly corresponding to the "proofcheck" block in Figure 4.1, is defined by a simple λProlog clause:

```
proofcheck ProofPlan Query :-
  translate_plan ProofPlan Tactic,
  translate_formula Query Phi,
  checkFTL Tactic Proof Phi',
  print "Plan proofchecked!".
```

The predicate `checkFTL` takes a tactic, a formula and an uninstantiated proof object as input, and tries to execute the tactic on the formula, step by step building the corresponding proof, which is finally output.

One last point is worth being remarked: in λProlog tactics acting as tactic constructors can be easily built using the abstraction mechanism. In an earlier example

(see [CS02a]) we have devised a case-split method for a class of **FOLTL** formulae representing an example of Feature Interactions (see Chapter 7). In that case, the case-split method opened three branches. The method definition looked like this:

```
atomic fi casesplit
  <input-goal>
  true
  true
  <output-goal>
  (T1\ T2\ T3\ (function_of T1 T2 T3)).
```

Here `function_of` indicates the complex tactic which realised the method in full $\mathcal{C}_{\mathbf{FOLTL}}$ detail; at three particular locations in the proof tree, three tactics had to be attached, resulting from the further development of the proof plan, and therefore coming out of the translation of further methods. Thanks to λ-abstraction, the `<tactic>` slot can be filled by what really is a *function* of three tactics, or better, a *tactic constructor* `(T1\T2\ T3\(function_of T1 T2 T3))`, whose purpose is that of building the proof tree in the correct way.

4.4 Chapter overview

In this Chapter we have given a recipe for a sequent calculus for **FOLTL** which is, hopefully, ready to be used in our framework. At the price of giving up completeness, we have extended $\mathcal{C}_{\mathbf{QS4.3.1}}$ keeping in mind its good characteristics. In particular, we hope, in most situations, to be able to use the entailment rule in $\mathcal{C}_{\mathbf{FOLTL}}$ as well, although this will not be always true.

Then we have outlined how tactic-based theorem proving works, and we have shown that the approach can be easily adapted for all QMLs described in Chapter 3, and for **FOLTL** as well.

Lastly we have outlined how we have coupled the proof planner λCLAM and the object-level theorem prover FTL, in order to build a proof planning system for **FOLTL**.

Chapter 5

A tactic-based theorem prover for $\mathcal{C}$QL

In this Chapter we describe FTL, the quantified modal / temporal logic theorem prover we have developed. Although it has been conceived since the beginning as an object-level theorem prover to be coupled with λCLAM, it turns out that it stands on its own as an interactive prover for the logics which are the subject of the previous Chapters, as well. Under this respect, it can be seen as a similar machinery to that implemented in Isabelle in [Vig00].

The choice of reimplementing such a machinery from scratch, rather that using Isabelle or some other well-established framework, was originally motivated at least by the following issues:

1. we wanted to have a prover which would have been easy to integrate with our proof planner λCLAM;

2. as well, we wanted to have full control over the machinery, down to the finest possible degree;

3. lastly, we wanted to use a higher-order programming language, in order to reuse all the knowledge acquired during the years in which λCLAM had been developed by the Mathematical Reasoning Group at the University of Edinburgh.

Although results in automated TP obtained using FTL were shown in some early

publications ([CS00, CS01]), FTL is *not* conceived as an automated theorem prover — it contains no machinery for proof search (besides a simple exhaustive tactic applicator) and is completely unusable for automated theorem proving when we want to prove any slightly-more-than-trivial theorems.

One wants to employ proof planning mostly on complex, undecidable (or worse) domains and logics, in order to take advantage of its abstraction capabilities; therefore it would have anyway been no point trying to make FTL more and more automatic. Most of the effort has been concentrated since the beginning on the proof planning strategies, that is to say, on the upper level, rather than on the capabilities of FTL. All we needed FTL to provide was: soundness of λCLAM's methods, that is, proof checking. Automation was not required.

Having said that, FTL is written in λProlog and works fine as a stand-alone interactive prover for **FOLTL**, as well as for all other calculi in $\mathcal{C}_{\mathbf{QL}}$. The properties of $\mathcal{C}_{\mathbf{QL}}$, which extend to $\mathcal{C}_{\mathbf{FOLTL}}$ to a certain extent, have been heavily exploited while designing FTL — in particular, the property of modularity has enabled us to build a separate λProlog module to take care of frame reasoning. Again, this is an outcome of Labelled Deduction and fits well with its spirit.

This Chapter describes FTL. Tactic-based theorem proving and its application to our problem was already described in Chapter 4; in what follows, after a short introduction to λProlog and its peculiarities with respect to ordinary logic programming languages (Section 5.1), we give a high-level account of FTL's design, showing how it fits with $\mathcal{C}_{\mathbf{QL}}$ (Section 5.2), then moving on to some details about the actual implementation (Section 5.3).

Moreover, Appendix A shows an interactive session in FTL, in order to give an idea of how it works in practice; and Appendix B gives a proof of its correctness. The term *correctness* here has the meaning explained in the previous Chapter — it obviously does not mean FTL aims at being complete and/or terminating for any logic whatsoever.

5.1 A quick introduction to λProlog

In the subsequent Sections, we will refer to some basic concepts of the higher-order logic programming language λProlog, without going into detail (the interested reader should refer, e.g., to [Mil93, NM98] and [Mil98]). Nevertheless, a quick introduction is needed.

As a start, λProlog can be thought of as ordinary Prolog, but:

1. all terms are *typed*;
2. the language of clauses is extended, admitting λ-abstraction and λ-application;
3. the language of goals is extended, allowing for *universal* and *implicational* goals;
4. programs can be split among *modules*, each module consisting of a set of type declarations (the *signature* of the program) and a set of clauses (its *body*).

The typing system

The λProlog typing system is *strong* and *polymorphic*, that is, *(i)* all constants must have a unique type, and for all variables, a unique type must be inferable at compile-time; and *(ii)* both new types and type constructors can be declared, and type declarations can contain logical variables. An immediate example is that of lists:

```
kind person      type.
kind list        type -> type.

type average_age (list person) -> integer.
type head        (list A) -> A.
```

Here we have declared a type (`person`), a unary type constructor (`list`) and two functions, one operating on lists of persons and returning an integer number, and one

operating on lists of anything and returning an object of the same type of those contained in the list. Notice that the λProlog keyword `type` is overloaded, being used both in the definition of types and type constructors, and in the declaration of the types of predicates.

Predicates in λProlog have `o` as target type (the last type of a type specification, that is, the type of the function itself). `o` can be thought as the type of Boolean truth or falsehood at the object level.

Abstraction and application

λProlog allows metavariables to range over functions and predicates and, in the spirit of λ-calculus, allows for λ-abstraction and application. The notation is `(x\ f)` for $\lambda x.f$ and `(f x)` for $f(x)$.

Operationally, it uses a higher-order unification algorithm to perform unification of higher-order terms. Although higher-order unification is well-known to be non-terminating (see, e.g., [Hue75]), undecidable problems arise quite seldom.

A typical example (modelled upon [Mil98], and actually used in FTL) is that of applying a predicate to each element of a list:

```
type map  (A -> B -> o) -> (list A) -> (list B) -> o.


map _ nil nil.
map P [H|Tail] [H'|Tail'] :- P H H', map P Tail Tail'.
```

Consider `map`'s type specification. It is a predicate (its target type is `o`) linking a predicate of type `(A -> B -> o)` (a binary predicate, whose arguments have type `A` and `B`) and two lists, typed accordingly with the types of the argument predicate (`(list A)` and `(list B)`). Notice the use of the logical variable `P`, ranging over predicates.

The effect of `map` is that, given a predicate and a list of arguments for it, it generates a new list in which every element is obtained via the argument predicate. For example, let `double` a predicate linking an integer and its double:

```
type double   int -> int -> o.


double N M :- M is N * 2.
```

Then the goal `?- map double [1,2,3] L.` produces the answer

```
L = [2,4,6]
```

This predicate — just an example of smart higher-order logic programming — is very useful to extend an algorithm to a whole list, without rewriting any new code.

Goals

Universal goals are solved by adding a new constant to the signature of the program, and then trying to solve a new goal in which the universally quantified variable has been replaced by this constant — a system which resembles universal rules and their side conditions in classical logic. The notation is `?- pi x\ (P x)` for the goal represented by $\forall x.P(x)$. The newly-introduced constant cannot be unified with any variable already appearing in the signature.

Implicational goals are solved by adding to the program clauses the antecedents of the implication, and then trying to solve the consequent. The notation is `?- p => q` for the goal represented by $p \supset q$.

Notice that universal goals augment the signature of a program, whereas implicational goals augment its body. Recalling the assert/retract mechanism in Prolog, the use of implicational goals can be seen as a better logically founded mechanism for asserting clauses in the program's database[1].

Modules

Lastly, perhaps the most interesting concept in working with λProlog is that of modules, and of *accumulation* versus *importing* of λProlog modules.

[1] there is no clause retraction mechanism in λProlog, which can sometimes cause trouble.

An accumulated module is just textually included into another one, much like what happens in the C language with the preprocessor directive `#include` or in ordinary Prolog. The signatures and clauses of the two modules are merged (modulo multiple inconsistent definitions) and all goals can be indifferently solved using one module or the other's clauses.

On the other hand, the relation between a module (let us call it "father") which imports another module (the "child") is a little more complicated. The two signatures are merged as it happens during accumulation, but the clauses of the child are made dynamically available to the father, meaning that they act as antecedents in an implicational goal. As an example, consider these two module declarations (the example comes, once again, from [Mil98]):

```
module modA.
type p o.
p.

module modB
import modA.
type q o.
p :- q.
```

In this case, the goal `?- p.` is trivially derivable from within module modA (the child), but not from within module modB (the father). When the query for `p` is attempted, the clause `p :- q` is found and augmented with the only clause of the child, yielding clause `p :- (p => q)`. The subgoal `p => q` is generated, the fact `p.` is added to the father's clauses and `?- q.` is attempted with no result. Notice that, had modA been accumulated into modB, the query would have been immediately solved thanks to the fact `p.` in modA.

The big advantage of using λProlog modules is that the resulting code can take advantage of them and be remarkably modular. By modularity here, we mean that

changes affecting a module should not affect the rest of the code, with beneficial effects on bug detection.

In our case, this characteristic actually is what we need. Most quantified modal and temporal logics, in our framework, share the reasoning machinery and the syntax (see Chapters 3 and 4); therefore, it seemed an interesting (if viable) idea to keep reasoning on the frame properties in a separate module, or at least, to exploit λProlog's modules mechanism to keep it separate from "logical" reasoning. The goal was to minimise the burden of changes in the system, as different logics were tackled — which was tantamount to changing the frame properties.

5.1.1 Search, metavariables and Skolem functions

As is customary in Automated Reasoning, FTL implements some standard techniques to speed up the search and/or to reduce the search space. We give a brief outline of two of these techniques, since they are not mentioned in the previous Chapters but the proof of correctness we give below (Appendix B) requires them. This also lets us describe the λProlog search mechanism a little more in detail. All information sketched below can be found mainly in [Fel93], and is anyway quite standard nowadays in logic programming.

Search in λProlog

Let $\langle \Sigma, \mathcal{P} \rangle$ be a λProlog signature and set of clauses (program); search is then performed via six primitives:

1. **AND** A conjunction of goals $G_1 \wedge G_2$ is derivable from $\langle \Sigma, \mathcal{P} \rangle$ if and only if both G_1 and G_2 are derivable from it;

2. **OR** A disjunction of goals $G_1 \vee G_2$ is derivable from $\langle \Sigma, \mathcal{P} \rangle$ if and only if either G_1 or G_2 is derivable from it;

3. **INSTANCE** An existential goal $\exists x.G$ (i.e., a standard Prolog-like goal) is derivable from $\langle \Sigma, \mathcal{P} \rangle$ if and only if there is some Σ-term t of the same type as x such that $G[t/x]$ is derivable from it;

4. **GENERIC** A universal goal $\forall x.G$ is derivable from $\langle \Sigma, \mathcal{P} \rangle$ if and only if $G[k/x]$ is derivable from $\langle \Sigma \cup \{k\}, \mathcal{P} \rangle$ where k has the same type as x and is *not* in Σ;

5. **AUGMENT** An implicational goal $P \supset G$ is derivable from $\langle \Sigma, \mathcal{P} \rangle$ if and only if G is derivable from $\langle \Sigma, \mathcal{P} \cup \{P\} \rangle$;

6. **BACKCHAIN** An atomic goal A is derivable from $\langle \Sigma, \mathcal{P} \rangle$ if and only if either $A \in |\mathcal{P}|_\Sigma$ or $P \supset A \in |\mathcal{P}|_\Sigma$ and P is derivable from $\langle \Sigma, \mathcal{P} \rangle$. The set $|\mathcal{P}|_\Sigma$ is defined as the smallest set of clauses such that *(i)* $\mathcal{P} \in |\mathcal{P}|_\Sigma$ and *(ii)* if $\forall x.D \in |\mathcal{P}|_\Sigma$ and t is a Σ-term of the same type as x, then $D[t/x] \in |\mathcal{P}|_\Sigma$.

Metavariables

A well-known problem in automated reasoning is that of appropriately instantiating bound variables in non-generative quantifiers, that is, existential quantifiers appearing with positive polarity or vice-versa (for a precise definition of polarity, especially in temporal logics, refer, e.g., to [AM90]). In our framework, for instance, every time a $r\exists$ rule is employed we have to guess a term of the language, having sort ι, with which to substitute the bound variable in the active formula of the rule. This case happens as well with non-generative rules for modal operators such as $r\Diamond$ and $l\Box^*$.

Besides sort information, there is no clue on what term to use; since we usually have Skolem functions in the language of $\mathcal{C}_{\mathbf{QL}}$, and work with the natural numbers in the case of $\mathcal{C}_{\mathbf{FOLTL}}$, the Herbrand universe is infinite and non-determinism is extreme. Therefore, some smart, soundness-preserving instantiation mechanism must be enforced.

The choice we adopt, and which is adopted as well in λCLAM, is that of using *metavariables* in non-generative rule applications. In place of the guessed term, a

λProlog variable is substituted, which is carried on to the very end. (By "the very end" here we mean a point at which it is actually possible to find an instantiation for that variable via the higher-order unification mechanism, or something smarter.) This well-known idea dates back to the late 50s at least, if not to Herbrand, and has been recently revisited and thoroughly described in [DV01], where proof systems such as ours are called *free variable systems*, since the idea of metavariables is closely related to the (first-order) concept of free variable in a formula.

The use of metavariables is particularly appealing and useful in our case, where frame reasoning is highly separated from logical reasoning (see Section 5.2): two *different* unification algorithms can in principle be used to perform the final instantiation of the metavariables appearing in a proof tree — one for the sort θ of possible worlds (time instants) and one for the sort of everything else, ι — without affecting each other.

As an example, Figure 5.1 shows the $\mathcal{C}_{\mathbf{QK}}$-proof of the Converse Barcan Formula (recall Figure 3.2), in which two metavariables, X of sort ι and T of sort θ, are employed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(X)\ @\ T \longrightarrow p(a)@t}\,\mathsf{ax}}{\forall x.p(x)\ @\ T \longrightarrow p(a)@t}\,l\forall \quad \dfrac{}{0 \prec t \longrightarrow 0 \prec T}\,\mathsf{ax}}{0 \prec t, \Box\forall x.p(x)\ @\ 0 \longrightarrow p(a)@t}\,l\Box}{\Box\forall x.p(x)\ @\ 0 \longrightarrow \Box p(a)@0}\,r\Box}{\Box\forall x.p(x)\ @\ 0 \longrightarrow \forall x.\Box p(x)@0}\,r\forall}{\longrightarrow \Box\forall x.p(x) \supset \forall x.\Box p(x)@0}\,r\supset$$

Figure 5.1: the $\mathcal{C}_{\mathbf{QK}}$-proof of the Converse Barcan Formula (recall Figure 3.2), with the use of metavariables. We follow the Prolog convention here: metavariables names begin with a capital letter or with an underscore. The proof tree is closed by the unification $\{X \leftarrow a, T \leftarrow t\}$.

Skolem functions

Another big issue is that of, sometimes, *restricting* the scope of unification, once metavariables have been introduced in the language. It is the case, dual to the previous one, of sequent rules involving generative quantifiers such as $r\forall$ and $l\forall$: in this case, the term which substitutes the bound variable in the active formula of the rule must be *fresh*, that is, it must not appear in the conclusion of the rule itself. This side condition enforces soundness of the rule itself.

Usually, this proviso is enforced thanks to *Skolem functions* restricting unification. In λProlog there is a straightforward and logically well motivated way of enforcing such a proviso: whenever a fresh constant is required, the GENERIC directive is used at the metalevel to introduce a term which will not unify with any metavariable *already* present in the signature of the program, that is, in the proof tree.

5.2 High-level design

FTL consists of about one thousand lines of λProlog code, distributed among six modules; modules are hierarchically conceived, i.e., the basic syntax and operation of the object language are defined in simple "bottom" modules, which are then accumulated or imported into "higher" ones which deal with more and more complex functions. The top module contains the main predicate, that is, the interactive theorem prover itself. The hierarchical structure of FTL is depicted in Figure 5.2.

In the Figure, each box represents a module; a solid arrow from box *A* to box *B* indicates that module *A* is included into module *B*, while a dotted arrow denotes importing in the same way. In λProlog, accumulation and importing propagate through the hierarchy, meaning that, for instance, module syntax is accumulated by all modules Also, as one can see, only one dotted line appears in Figure 5.2, meaning that the only case of a module being imported into another is that of frame being imported into basic_tacs.

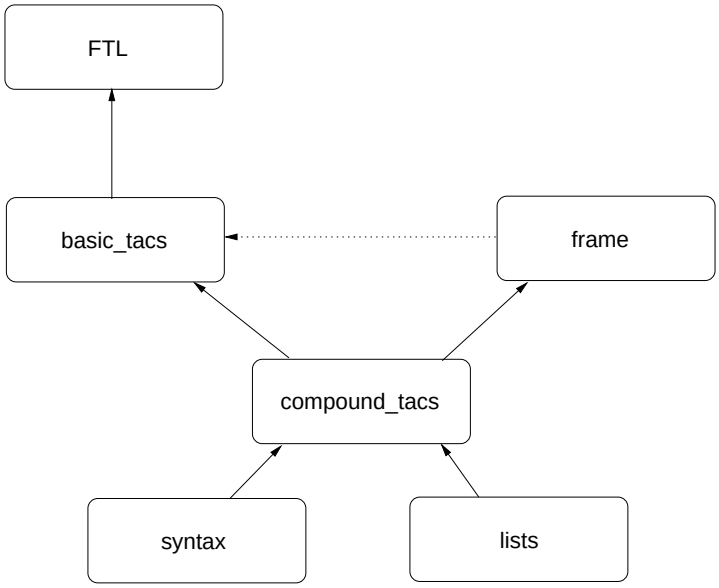

Figure 5.2: the hierarchy of modules that constitute FTL.

FTL can also (and perhaps more proficiently) be viewed as a set of interplaying sets of modules — aggregates of λProlog modules defining specific functionalities or dealing with well-delimited parts of the object logic; since every accumulated module is textually included into its accumulator, in the end such a view is equivalent to counting the top modules, that is, modules which are not accumulated into any other module, but possibly are imported into some other ones.

It is in this respect that we have confined all frame reasoning in a module called frame. Figure 5.3 offers an architectural view of FTL (syntax and lists are not represented for conciseness).

FTL is the overall top module and offers one main functionality: the interactive theorem prover. frame, on the other hand, accumulates all syntax and basic operations (compound tactics in module compound_tacs, among others) and then defines frame rules and associated tactics, in the end grouped in the entailment rule.

In this respect the modularity of λProlog reflects the modularity of our sequent calculi $\mathcal{C}_{\mathbf{QL}}$: FTL knows all the bits of syntax needed for frame reasoning (for instance,

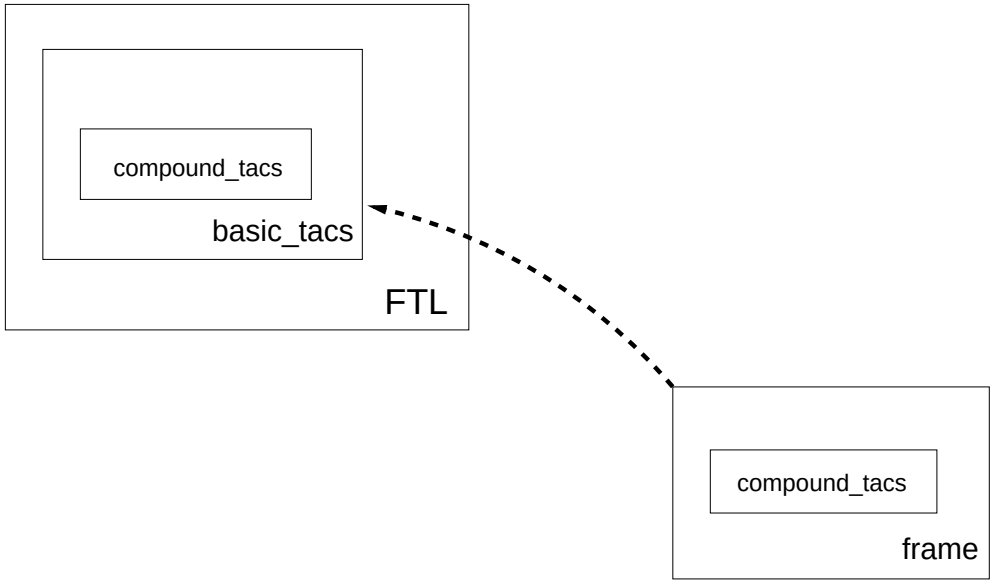

Figure 5.3: the architecture of FTL. Modules syntax and lists are omitted for the sake of conciseness.

the Skolem functions introduced by the strengthening procedure) since they are in the signature, but has no knowledge of frame rules and tactics. If a tactic is modified or deleted, or a new tactic is added, we only need to change module frame.

Rather than giving the details of the code, it is better here to point the interested reader to some appropriate references. The TP machinery and interaction with the user closely resemble the code found in λProlog literature, for instance in [Fel88, Fel89, Fel93], and really present nothing new; on the other hand, the implementation of tactics in λProlog, and how proofs are built out of tactics, has already been described in Section 4.2.

5.3 Implementation

A description of what each module does follows.

lists

This module contains primitive operations on lists such as *member*, *nth_member* and *delete*, plus a very basic higher-order predicate *map*, which, given a predicate P and a list l, generates a list l' in which every element is the result of applying P to the corresponding element of l. Also, a couple of printing facilities appear in this module.

syntax

This module defines the sort ι and *logical formulae* as `kind`s, plus all the basic (not frame-related) syntactic elements, more or less exactly as they can be read off Section 3.1.1. This includes Boolean operators, first-order quantifiers and modal/temporal operators. These objects are viewed as logical formulae constructors, i.e., unary or binary functions from the set of logical formulae to itself. *Labelled formulae* are defined as another `kind`, having one constructor only — the @ operator. As well, the sort θ and the kinds of sequents, goals and proofs are defined. Lastly, this module defines the sequent constructor `-->`, joining two lists of formulae, and the goal constructor `proves`, building a goal out of a proof and a sequent.

compound_tacs

This module contains compound tactics, most of which have already been described in Subsection 4.2.2, so we will not go into detail. Note that, in the FTL module hierarchy (recall Figure 5.2), this module appears *below* module basic_tacs, notwithstanding the name. Although somewhat surprising at the beginning, it is clear that compound tactics, from the point of view of the metalanguage, are *more general* than basic tactics, since they can be applied to any object logics whatsoever. Operationally, basic tactics, which are special to the logic **QL** implemented by the prover, must rely on compound tactics to be applied.

frame

This module contains all that is related to frame reasoning. The signature defines the symbols 0, $\prec$ and $\doteq$ as constructors for the sort θ; the clauses define frame rules and the associated tactics, as they are defined in Table 3.4, plus the substitution algorithm for the application of rule $\mathsf{sub}_{\doteq}$. This algorithm is really a simple recursive term substitution predicate operating on single formulae. The `map` higher-order predicate takes care, then, of extending it to lists of formulae, when rule $\mathsf{sub}_{\doteq}$ is applied. Lastly, in this module the tactic `tent` is defined, corresponding to the entailment rule ent. So far we have not been highly concerned with it, therefore it is implemented as a simple depth-bounded iterative deepening exhaustive application of all rules in $\mathsf{FrmRl}(\mathbf{QL})$, for each **QL**; but, of course, there is no restriction on how this procedure is designed, as long as it returns a valid unification, if it is the case, when it finds a positive answer to the entailment problem.

basic_tacs

Again, basic tactics and their use have been described somewhere else (see Subsection 4.2.1); besides that, this module defines the only tactic intended for automated reasoning in FTL: it enforces an exhaustive, eager application of all tactics taken from a given list. The list also includes tactic `tent`, whose inner working is hidden thanks to the fact that this module *imports* module frame. Any change to that module has no effect to the code written here.

FTL

This is the top module: it actually defines *(i)* the predicate `top`, which enforces interactive theorem proving, presenting the user with a sequent and asking for a tactic, and *(ii)* the predicate `check`, which simply checks that a given proof is actually the proof of a given sequent. This latter one heavily relies on `top`: it just uses the proof as a guide to choose a tactic which is then applied to the sequent. The answer is positive if the

proof tree is closed. `check` is, actually, the proof-checking machine which is "seen" by λCLAM during the proof checking phase.

5.4 Chapter overview

This short Chapter, after a quick introduction to the beauty of the higher-order logic programming language λProlog, contains the overview of FTL, our interactive, tactic-based theorem prover for **FOLTL**, which is meant to be the object-level theorem prover coupled with the proof planner λCLAM. First a high-level design view has been given, and then some details about the implementation have been described.

Chapter 6

Proof planning for FOLTL and Feature Interactions

6.1 Introduction

In Section 4.3 we have outlined the basic ideas which realise the interaction between the proof planner λCLAM and the object-level theorem prover FTL. In this Chapter we describe how we have specialised Proof Planning for **FOLTL** and, in particular, for the case-study of Feature Interactions in telecommunication systems (FIs). All is said in this Chapter relies on the architecture outlined in the above cited Section.

The Chapter is organised as follows: first, Section 6.2 describes a preliminary experiment which has been published in [CS02a]. The experiment is an interesting initial attempt at applying Proof Planning to a very restricted subset of **FOLTL**, modelling FIs in a rather naïve way. The experiment is based upon Amy Felty's work ([Fel01]).

The experiment relied on one, very specialised, method; its generalisability was quite low. Willing then to generalise and extend our result, we have moved to a more complex framework, consisting of a new, more general **FOLTL** model of FIs (Section 6.3) plus a wide set of methods with a high range of applicability (Section 6.4).

Lastly (Section 6.5), we outline the experimental methodology we have followed

in Chapter 7, which is far from trivial. *How* to carry experiments on, *what* data to report on and *why* has been a major concern while setting up the test set and finding the solutions. The evaluation methodology follows Francisco Cantu's papers and Ph.D. thesis ([CBSB96a, CBSB96b, CO97]).

6.2 Proof Planning for Feature Interactions: a preliminary result

According to its most general definition, a *feature* is a service marketed to the customer of a company, usually in addition to a basic service. In the past decade at least, this problem, as experimented in large telephone networks, has received great attention (see, e.g., [GBGO00]), both from the academical and the industrial world. In this particular setting, the basic service is represented by the plain telephone switch network connecting users to one another; features are additional services such as call-waiting and call-forwarding. Features are specified and implemented without any knowledge of what other features may be concurrently required by other users in the network. This facilitates modular design but also introduces potential undesired / unwanted behaviours when more than one feature is activated.

A well-known example is the interaction arising between Anonymous Call Rejection (ACR) and Call Forwarding Busy Line (CFBL). Informally, ACR prescribes that anonymous calls (i.e., calls from a user hiding her number) should be rejected, while CFBL prescribes that all calls to the subscriber should be forwarded to a third party if the subscriber is busy. Assume user x subscribes to both features: what happens if anonymous user y calls x while he is engaged? Should y's call be rejected according to ACR or forwarded to z according to CFBL? The situation is usually repaired by establishing a priority relation among features, and in this case, ACR would have priority over CFBL.

The way we have tackled the problem in this preliminary setting has been to closely

follow the methodology outlined in [FN00]; in particular, we have modelled the system in a very similar way, and then we have mimicked the hand-made proof of the interaction arising between ACR and CFBL as a proof plan. The plan has then been automatically verified using FTL.

We now describe the model and the methodology to obtain the proof plan. In the following, we will omit labels attached to a formula whenever the label is 0.

- The global behaviour of the telephone system is expressed as a set of invariant (i.e., wrapped by a $\Box$ operator) universally quantified first-order sentences;
- a feature, denoted by the subscript i, is specified via a formula like this:

$$\Box\forall\overline{x}\, e_i(\overline{x}) \supset\ [p_i(\overline{x})\ \mathcal{U}\ (r_i(\overline{x}) \vee d_i(\overline{x}))] \tag{6.1}$$

  informally meaning "after the feature is **e**nabled, a **p**ersisting condition holds until the feature is **r**esolved or **d**ischarged".

- a feature interaction (that is, an undesired behaviour) is found between two features 1 and 2 if the previous formulae together imply $\Box\forall\overline{x}\neg G(\overline{x})$, where $G(\overline{x})$ is:

$$e_1(\overline{x}) \wedge e_2(\overline{x}) \wedge [(p_1(\overline{x}) \wedge p_2(\overline{x}))\ \mathcal{U}\ (\neg p_1(\overline{x}) \wedge \neg p_2(\overline{x}) \wedge \neg d_1(\overline{x}) \wedge \neg d_2(\overline{x}))] \tag{6.2}$$

  informally meaning "enable the features at the same time, let them persist, then force them to resolve". The idea is that $G(\overline{x})$ represents the required behaviour, and $\Box\forall\overline{x}\neg G(\overline{x})$ is the *denial* of it. If the above implication is valid, then the required behaviour will never be possible, for any combination of users.

Here is how we have formalised the problem:

(i) ACR is defined by instantiating the schema 6.1 as follows:

$$\begin{aligned} e_1(x,y) &= has\_acr(x) \wedge \neg display(y) \wedge call\_req(y,x) \\ p_1(x,y) &= call\_req(y,x) \\ r_1(x,y) &= acr\_announce(x,y) \\ d_1(y) &= onhook(y) \end{aligned}$$

informally meaning: if user x has activated ACR and user y is anonymous, y will be trying to call x until either x sends a rejection message or y hangs up.

(ii) CFBL is defined by instantiating the same schema as follows:

$$\begin{aligned} e_2(x,y,z) &= has\_cfbl(x) \wedge \neg idle(x) \wedge \\ &\quad \neg \exists t.\, forwarding(t,x,z) \wedge call\_req(y,x) \\ p_2(x,y) &= call\_req(y,x) \\ r_2(x,y,z) &= forwarding(y,x,z) \\ d_2(y) &= onhook(y) \end{aligned}$$

informally meaning: if user x has activated CFBL, is not idle and there are no calls to him being currently forwarded, y will be trying to call x until either the call is forwarded to z or y hangs up.

(iii) System axioms enforce simple properties of the predicates involved in the definition of the features, e.g.,

$$\Box \forall xy\, \neg(onhook(x) \wedge call\_req(x,y))$$

informally meaning: it is impossible to hang up and be trying to call someone at the same time.

(iv) Finally, the requirement is obtained by instantiating scheme 6.2 with the above definitions of p_i, e_i and d_i, $i = 1,2$ (from now on, we omit the explicit reference to the variables $\overline{x}$ for conciseness).

In the end we are trying to prove validity of the formula:

$$\begin{aligned}
\Box\forall\overline{x}\,[e_1 \supset\ p_1\ \mathcal{U}\ (r_1 \vee d_1)]\ \ &\wedge \\
\Box\forall\overline{x}\,[e_2 \supset\ p_2\ \mathcal{U}\ (r_2 \vee d_2)]\ \ &\wedge \\
\Box\forall\overline{x}[SA]\ \ &\supset \\
\Box\forall\overline{x}\neg G\ \ &
\end{aligned} \tag{6.3}$$

where *SA* denotes system axioms. Note that 6.3 involves a quite free mix of unary, binary and ternary predicates, temporal operators and first-order quantifiers, in such a way that it does not fall into any known well-behaved fragment of **FOLTL**, as far as we know.

According to the hand-made proof, let us suppose G holds and try to derive a contradiction. By the definitions of $\Box$ and $\mathcal{U}$,

1. if G holds (as an invariant), then we can fix an arbitrary time $t_0 \geq 0$ at which both e_1 and e_2 hold; also, there is a time $t_G \geq t_0$ such that p_1 and p_2 hold until $\neg p_1 \wedge \neg p_2 \wedge \neg d_1 \wedge \neg d_2$ holds at t_G;

2. since both ACR and CFBL hold, they must be enabled at t_0; also, there are times $t_{ACR}, t_{CFBL} \geq t_0$ such that p_1 and p_2 hold until the features are either resolved or discharged respectively at t_{ACR} and t_{CFBL}.

The key to the proof is the relative positions of t_G, t_{ACR} and t_{CFBL}; Figure 6.1 is an example case in which $t_{ACR} < t_G$ and $t_{CFBL} > t_G$.

There are three sub-cases to be considered for t_G and t_{ACR} (i.e., $t_G < t_{ACR}$, $t_G > t_{ACR}$ and $t_G = t_{ACR}$) and three for t_G and t_{CFBL}, but it turns out that the situation is simpler:

- consider G and ACR: if $t_G < t_{ACR}$ then both $\neg p_1$ and p_1 must hold at t_G, which leads to a contradiction. Analogously, consider G and CFBL: if $t_G < t_{CFBL}$ then both $\neg p_2$ and p_2 must hold at t_G;

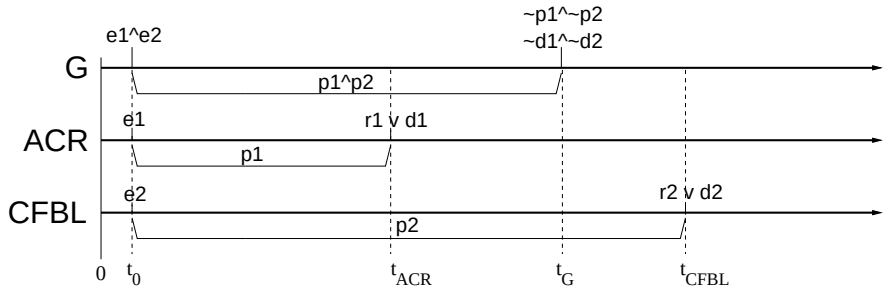

Figure 6.1: a graphical representation of the interaction between ACR and CFBL. In this case, $t_{ACR} < t_G$ and $t_{CFBL} > t_G$.

- consider G and ACR again: if $t_{ACR} < t_G$ then both p_1 and $r_1 \vee d_1$ must hold at t_{ACR}, which leads to a contradiction if the system axioms are taken into account. Analogously for G and CFBL, in which case a contradiction is derived from p_2 and $r_2 \vee d_2$ at t_{CFBL} w.r.t. the system axioms;

- lastly, consider the remaining case in which $t_G = t_{ACR} = t_{CFBL}$: by propositional reasoning, r_1 and r_2 must hold together with the system axioms, which once again leads to a contradiction.

As already noted in [Fel01], system axioms are not involved in the first two cases, ruled out by simple propositional considerations. The remaining three cases are solved by first-order reasoning because no temporal operators are involved in the system axioms. In order to mimic this neat, intuitive and rigorous (although not formal) way of reasoning, we set up a λCLAM method called *fi_case_split* which simply splits the goal of proving formula 6.3 into three first-order subgoals (see Figure 6.2).

λCLAM finds the proof plan in about one minute on an Ultra 10 Sun machine without any backtracking, as we expect. The proof plan is then translated into a (big) tactic which is fed to FTL, which applies it to the formula and generates the actual proof of the formula itself.

It is interesting to have a closer look at the process of translation of the proof plan into an FTL tactic. In particular, the first-order reasoning which happens in λCLAM dur-

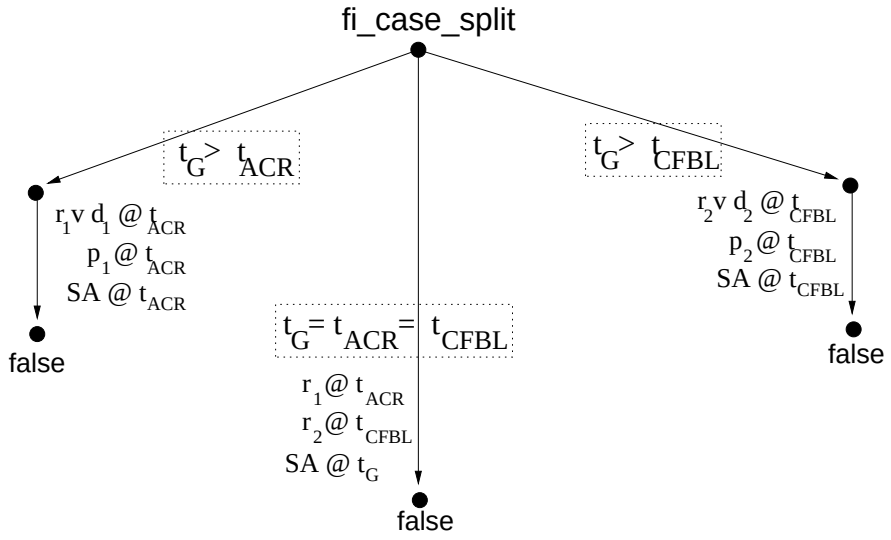

Figure 6.2: the method *fi_case_split* applied to the interaction between ACR and CFBL. The three generated subgoals are closed by first-order reasoning.

ing the exploration of the three subgoals opened by *fi_case_split* only involves atomic methods, which embed inference rules of $\mathcal{C}_{\mathbf{FOLTL}}$ and are translated directly into basic tactics.

The case is quite different with the translation of the *fi_case_split* method itself, corresponding to a quite complicated sub-proof tree, visible in Figure 6.3. In particular, rules $l\mathcal{U}$ are employed once each for G and ACR, introducing time points t_G and t_{ACR}; then, strong connectedness (rule sconn) generates three sub-cases in which $t_G < t_{ACR}$, $t_G > t_{ACR}$ and $t_G = t_{ACR}$. The first two sub-cases are ruled out respectively by immediate contradiction and using the first sub-case of the *fi_case_split* method; the third is brought forward, with the assumption that $t_G = t_{ACR}$.

Then, in perfect analogy, rule $l\mathcal{U}$ introduces time t_{CFBL} for CFBL and strong connectedness opens three more sub-cases, the first two of which are respectively closed by immediate contradiction, and by the second sub-case of *fi_case_split*. We are left with the assumptions $t_G = t_{ACR}$ and $t_G = t_{CFBL}$, and this branch is closed by the third sub-case of *fi_case_split*.

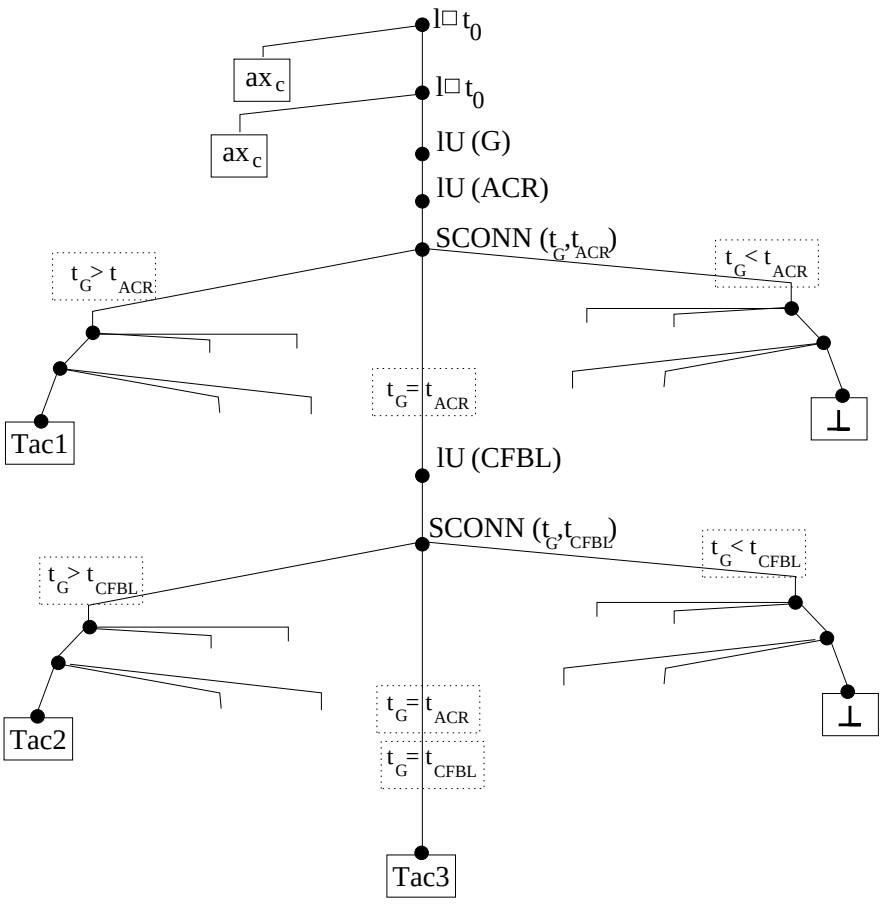

Figure 6.3: the tactic tree obtained translating the *fi_case_split* method. $Tac1$, $Tac2$ and $Tac3$ are the tactics corresponding to the three sub-cases of the method. Branches which look open in the Figure are closed by rules $l\Box^*$, not shown.

6.2.1 Discussion

As a preliminary result, the experiment just described is encouraging. The spirit behind proof planning is that of capturing the common structure in proofs dealing with a particular problem, by means of proof plans — exactly as it happens in this example. Several of the proofs devised for these problems actually share the common structure seen above; this indicates that FIs are definitely a good benchmark for proof planning.

In fact, on one hand, naïve as it may appear, the framework presented in this Section will work for any two features whose shapes resemble the schema 6.1 and system axioms not containing temporal operators. Although this represents a small fragment

of **FOLTL** (for one thing, it does not contain the $\bigcirc$ operator), a relevant part of the FI community is adopting a similar formalism; see, e.g., [JMN+01, FN00, CM01]. In short, there is some degree of generality in this technique.

Moreover, the subtree in Figure 6.3 remarkably reflects the structure of the hand-made proof and formally justifies it; its execution as a tactic proves the original formula in FTL and ensures soundness of the proof plan. But the more remarkable property is that it clearly shows a sort of "pattern" in the way the u -formulae are exploited and searched for contradiction: first use the u in *G* "against" that in *ACR*; once one branch only is left, use the u in *G* once again "against" that in *CFBL*; if in the end one branch only remains, try to close it by first order reasoning.

On the other hand, one of the main drawbacks of this experiment is that, although both the planning and the checking phases are automatic, the time spent by the user in order to devise the plan and the tactics related to the methods employed has been very long. In particular, the tactic associated with method *fi_case_split* contains something like 150 basic tactics, some of them applied to a precise formula in the antecedents or consequent of a sequent. For example, the uppermost node labelled by rule lu in the subtree of Figure 6.3 is enforced via tactic `(l_until_tac 3 0 tC)` — that is, the user had to specify not only that lu was to be used, but also on which antecedent formula (number 3) and with which label (in the example, t_C).

Moreover, the *order* in which basic tactics appear in the tactic associated with *fi_case_split* is absolutely crucial. One wrong position and the execution would not go through any more, preventing the system from proving soundness of the proof plan.

In the end, also for this quite simple example, the user had to manually build the tactic associated with *fi_case_split* step by step, ensuring at each step that labels had been instantiated correctly and so on. *Once the tactic was built*, planning and proof checking ran smoothly and quickly. Seen from this perspective, the advantage in using proof planning as opposed to good old interactive theorem proving is quite unclear — we must resort to the degree of generality the method has, as explained earlier on. The

weak points are that (a) *fi_case_split* is too specialised to solve this particular problem; and (b) the method really consists of smaller, simpler macro-steps, such as "expand an $\mathcal{U}$ operator on the right" or "use strong connectedness", each of which should be realised by a separate method. Enforcing point (b) would make most methods *reusable*.

In general, we cannot expect the whole process to be totally automatic; but still, in order to gain a clear edge over interactive theorem proving, we should be able to show that

1. the method is *really* general to some extent, that is: methods enforce macro-steps of reasoning which are common, if not ubiquitous, and can be used in many different places, maybe with small modifications;

2. as a consequence of this, the time the user spends for setting up each single proof plan *diminishes* as more and more plans are devised. An analysis of the human time required to solve the problems, rough as it may be, is necessary.

6.3 Modelling Feature Interactions in FOLTL

In this Section we give an outline of how we have modelled FIs in a more general and modular way. This time the starting point has been Calder and Miller's recent and, in our opinion, most comprehensive work [CM02b]. Their work seems to us to be the most successful application of formal methods to the problem so far.

In their framework, the phone network is seen as a *finite* set of *users*, each one gifted with at least a minimal set of abilities — being able to answer the phone, to dial a number, to hang up and so on. This basic set is called *Basic Call Service* (BCS). The environment also takes care of establishing connections among users. Features are added incrementally to each user. Notice that, in the end, there is little difference between the properties enforced by the BCS and those enforced by each single feature; one can see the BCS as a set of very basic features given to the user by default.

In the cited work, each user is modelled as a finite automaton, whose structure depends on the feature(s) he/she has subscribed to. Each feature, including the BCS, corresponds to a (set of) *LTL* formula(e) describing the properties the automaton associated with the feature should ensure; automata are then described by ProMeLa models and their adherence to the required properties is checked via language inclusion thanks to the model checker SPIN. SPIN and ProMeLa are well-known to the formal methods community (see, e.g., [Hol93, Hol97b]).

For instance, each user with the BCS should be able to hear the busy tone if she dials herself; this is expressed in *LTL* as $\Box(p \supset (\neg r \mathcal{W} q))$ where p states that the user has dialled herself, r that she is back to the idle state, and q that she hears the busy tone. Note the precision with which this sentence describes the required property: "if you dial yourself, then either you will never get back to idle, or, sooner or later, you will hear the busy tone, *meanwhile not being idle*".

As is pointed out in the very same paper, the expressivity of the $\mathcal{W}$ operator can and must be exploited in full here; it would be tempting, for example, to express the above property as $\Box(p \supset \Diamond q)$, something like "it is always the case that the busy tone will eventually follow the action of dialling yourself", employing the "eventually" operator $\Diamond$. Indeed the formula would look simpler and easier to handle, but it would also be too weak, since (a) it would be true in a scenario in which the user hears the busy tone *later on*, not necessarily as a result of this very call; (b) it would be false in a scenario in which the user failed to progress infinitely often, that is, for some reason the network took an infinite time to process her call. In fact, using the $\mathcal{W}$ operator, we can actually specify *what must hold* while we are waiting for an event to happen, and we can also be satisfied if the event *never happens*.

6.3.1 The Basic Call Service

The example use of $\mathcal{W}$ seen just above has actually suggested to us the idea of modelling the system via a collection of $\mathcal{W}$-invariants, that is, $\mathcal{W}$-formulae wrapped by a

□. The basic intuition is that if they can be used to express a requirement so precisely, it must be the case that they can express the *behaviour* of our system as precisely as that.

So we model the BCS as follows: each user stays in a *state*, for instance, his phone is idle, waiting for a call or waiting to dial a number. Along time, he can stay forever in this state, or move to another one, provided that, eventually, a *transition* happens, that is, either he performs an action (for instance, he off hooks, that is, he lifts the receiver), or the environment changes (say, someone tries to call him). A graphical representation of what a user can do is visible in Figure 6.4.

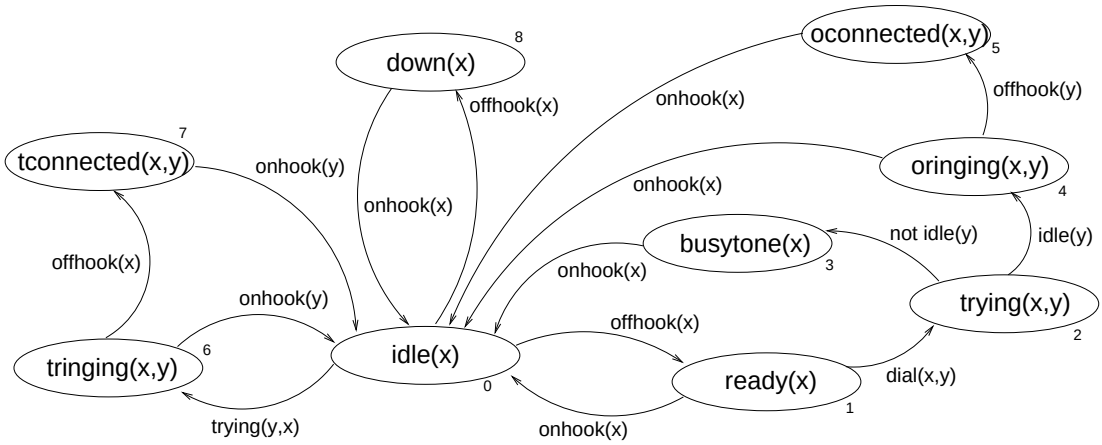

Figure 6.4: A graphical representation of what a user can do.

In the Figure each node represents the fact that a user, say *x*, is in a state, say *idle*; outgoing edges denote transitions. For instance, an *idle* user can $offhook$ and be either *down* (if the network is temporarily out of order) or *ready* to dial someone; alternatively, if someone is *trying* to call him, he will get to the $tringing$ state, meaning he hears the ringing tone as the terminating party. States are numbered for conciseness reasons — see below.

States and actions are actually modelled as first-order predicates, as is visible in Table 6.1.

The temporal behaviour of the generic user with BCS is then enforced via a set of

$idle(x)$	user x is idle
$ready(x)$	user x is ready to dial (i.e., he has off-hooked)
$trying(x,y)$	user x has dialled user y and is trying to connect to him
$busytone(x)$	user x hears the busy tone (tried to connect to a busy user)
$oringing(x,y)$	user x is ringing user y
$oconnected(x,y)$	user x is connected to user y, as originator
$tringing(x,y)$	user x is being rung by user y
$tconnected(x,y)$	user x is connected to user y, as terminator
$down(x)$	user x is down (out of order)
$offhook(x)$	user x off-hooks
$onhook(x)$	user x on-hooks
$dial(x,y)$	user x dials user y

Table 6.1: States and actions of a user having BCS.

universally-quantified formulae:

1. *State correspondence.* For each state, a natural number is associated to it:

$$\forall x.\Box\ idle(x) \leftrightarrow at\_state(0,x)$$

$$\forall x.\Box\ ready(x) \leftrightarrow at\_state(1,x)$$

   and so on;

2. *Mutual exclusion.* No user can be in two states simultaneously:

$$\forall n,m,x.\Box\ at\_state(n,x) \wedge at\_state(m,x) \supset n = m$$

3. *Progress.* For each state, either the user remains in the state forever, or a transition (either action or environmental change) happens:

$$\forall x.\Box\ idle(x) \supset idle(x)\,\mathcal{W}\ (offhook(x) \vee \exists t.trying(t,x))$$

$$\forall x.\Box\ ready(x) \supset ready(x)\,\mathcal{W}\ (onhook(x) \vee \exists t.dial(x,t))$$

and so on;

4. *Trigger.* For each state and transition, if they happen simultaneously then the user will be in a new state at the next instant:

$$\forall x.\Box\ idle(x) \wedge offhook(x) \supset (\bigcirc ready(x) \vee \bigcirc down(x))$$

$$\forall x.\Box\ idle(x) \wedge trying(y,x) \supset \bigcirc tringing(y,x)$$

and so on;

5. *Initial state.* Every user is initially idle:

$$\forall x.idle(x)$$

6. *System axioms.* These are invariants, relating some states to some other ones; in the BCS only two of them are needed:

$$\forall x.\Box\ (oconnected(x,y) \leftrightarrow tconnected(y,x))$$

$$\forall x.\Box\ (oringing(x,y) \leftrightarrow tringing(y,x))$$

We have also found useful to define an invariant, to be used in some inductive proofs. The invariant states that every user is always in at least one state:

$$\forall x.\Box\ \ idle(x) \vee ready(x) \vee \exists t.trying(x,t) \vee$$
$$\exists t.oringing(x,t) \vee busytone(x) \vee down(x) \vee$$
$$\exists t.oconnected(x,t) \vee \exists t.tringing(t,x) \vee \exists t.tconnected(t,x)$$

It is easy to see that this actually *is* an invariant of the system, by induction:

- *Base case.* Trivially, by the initial state formula, every user is $idle$ at time 0;

- *Step case.* Assume user x is in a state $STATE$ at time τ; then by the related progress formula (the one having $STATE$ as the antecedent of the implication) either x will stay in $STATE$ forever, or eventually a transition will happen. If it is the first case, x will be in $STATE$ at $\tau+1$, QED; if it is the second, then either the transition happens right now, or it happens at some time in the future. If it is the first case, by the related trigger formula (the one having $STATE$ and the required transition in the antecedent of the implication), x will be in a new state at $\tau+1$, QED; lastly, if it is the second case, x will still be in $STATE$ at $\tau+1$.

A few remarks on this model must be done. The model starts out as an attempt at modelling a finite state automaton in **FOLTL**; but it then goes beyond that in a number of ways. Firstly, notice how mutual exclusion between states is gracefully handled in **FOLTL** via state correspondence and mutual exclusion formulae. There is no limit on the number of states a model can have; the price to pay is linear in their number (i.e., one state correspondence formula per state).

Secondly, the model intuitively enforces some subtle properties of a real phone network. For example, a user that has been called (the *terminator*) cannot terminate a call, whereas the user who has called (the *originator*) can; this is reflected in Figure 6.4: if $tconnected(x,y)$ holds, that is, x has been called by y, then she cannot get back to $idle$, unless y decides to hang up.

Consider Figure 6.4 again. This system enjoys a high degree of non-determinism, represented in the diagram by the facts that:

(a) a state can have more than one outgoing arrow, meaning that more transitions are allowed from each state; for instance, user x in state $oringing(x,y)$ can decide to hang up, getting back to state $idle(x)$, or can go "spontaneously" to state $oconnected(x,y)$ when user y off-hooks.

(b) from a state, given a transition, a user can move to more than one single next state;

for example, an *idle* user can off-hook and be either *down* or *ready*;

(c) any user can permanently stay in a state, without taking any transitions[1];

(d) the diagram is parametric over the users, that is, there is no restriction on what the *x*s and *y*s can be.

All these issues are dealt with, we believe, in an elegant an intuitive way by the progress and trigger formulae; each progress formula states that a user can either remain in a state, or a transition can happen at any time; this is handled via the $\mathcal{W}$ operator. Each trigger formula, on the other hand, tells us what is going to happen next, if a user is in a state and a transition happens. This is actually why we chose in the beginning (see Chapter 4), to adopt a form of $\mathcal{U}$ operator which is slightly stronger that the standard one, requiring that, given $p \mathcal{U} q$, it is the case that, when q happens, p will *still* be true. Consider for example the transition from *idle* to *down* of user *x*. Let us assume that, a certain point in time, the user actually is idle, that is, $idle(x)$ holds. Thanks to the progress formula

$$\forall x. \Box\ idle(x) \supset idle(x) \mathcal{W}\ (offhook(x) \vee \exists t. trying(t,x))$$

it is the case that either (1) *x* will stay *idle* forever (which is a perfectly sensible option, all in all), or he will stay *idle* for a finite amount of time, until, eventually, (2) either he will $offhook$ or (3) someone will be *trying* to connect to him.

Assume option (2) becomes true; thanks to the stronger form of $\mathcal{U}$, the above formula guarantees that there is an instant in the future at which *both* $idle(x)$ and $offhook(x)$. If this is true, trigger formula

$$\forall x. \Box\ idle(x) \wedge offhook(x) \supset (\bigcirc ready(x) \vee \bigcirc down(x))$$

guarantees that, at least in one case, at the next state *x* will be *down*. Notice that the use of the stronger form of $\mathcal{U}$ just makes the presentation easier to read: we could as

[1] this should really be represented by self-loops on each node of the diagram, but we omit them for the sake of readability.

well have used the usual $\mathcal{U}$, in which case the trigger formulae should have employed one more $\bigcirc$ operator, for instance

$$\forall x.\Box\ idle(x) \wedge \bigcirc offhook(x) \supset (\bigcirc ready(x) \vee \bigcirc down(x))$$

Notice also that this model somehow "links" users with one another, as one would expect, via predicate sharing. For instance, assume x is *idle* and y tries to connect to him; in this case we want him to reach the state $tconnected(x,y)$. This will happen if there exists a user t such that $trying(t,x)$ can be proved. This will be provided if another user, call her u, is actually trying to connect to x: in that case, $trying(u,x)$ will hold, making the transition true and allowing the right instantiation of variables.

Lastly, notice that in principle there is no restriction, in this model, on the complexity formulae can have, besides the structure highlighted in the above list. For instance, we expect a trigger formula to look like $\forall\overline{x}.\Box\ (STATE(\overline{x}) \wedge TRANS(\overline{x}) \supset \bigcirc NEXT\_STATE(\overline{x}))$; but what $STATE(\overline{x})$, $TRANS(\overline{x})$ and $NEXT\_STATE(\overline{x})$ are is left to the necessity of the modeller. Really, we have tried to limit the complexity of the formulae in order for them to be able to capture the behaviour we were interested in.

6.3.2 Features: introducing OCS

In Calder and Miller's work, the fact that a user subscribes to one or more features actually modifies the graph representing its automaton. For instance, subscription to a "ring back when free" feature is shown to make the associated automaton very different from the BCS one. We believe in our setting features can be made user-dependent via first-order predicates defining that a user subscribes to one or more features; new axioms will take care of the invariant properties of those predicates; and augmenting the graph transitions with the new predicates, or introducing new states defined by the feature will enforce the new behaviours required by the features themselves.

When adding the capability of a new feature to the automaton, we have tried to

maintain it as simple as possible. As an example, we have introduced a simple feature, called *Originating Call Screening* (OCS). According to [CM02b], a user subscribing to OCS has a predefined list of users, calling whom is prohibited.

A new predicate, $ocs(x,y)$ declares that user x has user y on his screening list. It is reasonable to state, as a system axiom, that nobody can be on his own's screening list ([CM02b]):

$$\forall x.\Box\ \neg ocs(x,x)$$

In order to prevent calling a screened user, the trigger formula determining the transition from state *ready* to state *trying* is modified from $\forall x,y.\Box\ ready(x) \wedge dial(x,y) \supset \bigcirc trying(x,y)$ to

$$\forall x,y.\Box\ ready(x) \wedge dial(x,y) \wedge \neg ocs(x,y) \supset \bigcirc trying(x,y)$$

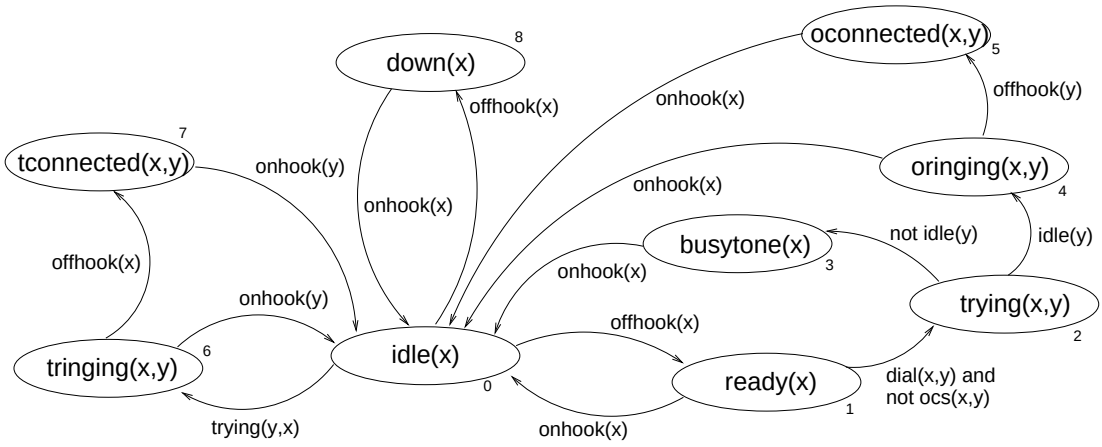

Figure 6.5: A graphical representation of what a user can do, when OCS is enabled.

Figure 6.5 pictures the new situation — notice the change in the transition from *ready* to *trying*. Lastly, the invariant defined for the BCS needs be extended this way:

$$\forall x.\Box \;\; idle(x) \vee ready(x) \vee busytone(x) \vee down(x) \vee$$
$$\exists t.(trying(x,t) \wedge \neg ocs(x,t)) \vee \exists t.(oringing(x,t) \wedge \neg ocs(x,t)) \vee$$
$$\exists t.(oconnected(x,t) \wedge \neg ocs(x,t)) \vee \exists t.(tringing(t,x) \wedge \neg ocs(t,x)) \vee$$
$$\exists t.(tconnected(t,x) \wedge \neg ocs(t,x))$$

that is, all states reachable "after a dial action has been performed" must be tagged with a side condition stating that user *x* can be there provided the user he is calling / ringing and so on is *not* on his screening list. Again, it is easy to prove that this invariant holds for the automaton pictured in Figure 6.5, that is, under the new set of hypotheses.

6.4 Designing proof plans for Feature Interactions

Much in the spirit of proof planning, in order to devise proof plans for FIs, we have been driven by:

1. the goal we were trying to reach, that is, the *shape* of the property we were trying to prove;

2. the experience already gathered in the field, that is, the preexisting techniques devised to solve an analogous problem; this primarily includes model checking, used in [CM02b] and probably one of the the most successful formal methods so far.

In the rest of this Section we describe a list of properties we have tried to verify for the BCS and BCS plus OCS (results are visible in Chapter 7), and the methods we have devised for them. The first Subsection reports on some general-purpose methods, not devised specifically for a property; the remaining Subsections analyse properties of the BCS, which we have tried to prove in the setting of the BCS automaton (Figure 6.4) and properties of the BCS augmented with OCS.

Each property has an associated number, as in [CM02b], which we have possibly decorated with *a*, *b* and so on, to denote slight variations of the same property.

Subsections are ordered by increasing difficulty. Methods are not reported in full, but described synthetically, sometimes by means of pseudo-code, enclosed in `< ... >`.

6.4.1 General-purpose methods

Mutual exclusion

One relevant problem which is potentially encountered when modelling a system in a state-based way in a logic is mutual exclusion between states[2]. The proof system must somehow be aware that no user can be in two states simultaneously — this fact, by the way, comes in many crucial points of the proof we have explored.

Since we use first-order predicates to denote that a user is in a state, it would be tempting to add a quadratic number of system axioms to the BCS specification, such as, for instance, $\forall x.\Box\,\neg(idle(x) \wedge ready(x))$, $\forall x.\Box\,\neg(ready(x) \wedge down(x))$, and so on. This would soon make the specification unmanageably large, clumsy and hard to maintain, in the sense that adding a new state would require adding more axioms. That is why we have chosen, in the first place, to number each state, and then to have one single mutual exclusion axiom.

There arises the necessity, then, of having a method which will detect mutual exclusion when present, and close the related search branch. Such a method is called `detect_mutex`:

```
atomic fi detect_mutex
  (seqGoal (Hyps >>> _ ))
  true
  true
  trueGoal
  < ... tactic ... > :-
  < 1. find State1 @ Tau1 in Hyps
```

[2] besides intuition, this is confirmed by a personal communication by Muffy Calder.

```
    2. find State2 @ Tau1 in Hyps
    3. find correspondence axioms in Hyps,
    4. find mutex axiom in Hyps
  >.
```

As already explained (recall Chapter 4, Section 4.3), a method is declared along with a theory name (`fi`), a method name (`detect_mutex`), pre- and post-conditions (both empty, that is `true` in the example), an input and output goal and an associated tactic. Operational content can be given to the method using the classical Prolog-style `:-` operator: the application of the method will succeed only if the content can be executed.

Given a sequent in which the antecedents are called `Hyps`, the above method returns the goal `trueGoal` (that is, it closes the branch) if (a) two different state formulae, (b) the related correspondence axioms, and (c) the mutual exclusion axiom can be found among the hypotheses.

The basic schema to build a tactic associated to a method, which has been successfully employed in the rest of the work, actually consists of (1) identifying a small set of hypotheses we are interested in in this branch, (2) "isolating" them, simplifying the sequent to be proved, then (3) trying to prove it by propositional logic, (4) possibly restoring all eliminated hypotheses, if this is needed. Propositional reasoning is devoted to FTL rather than being taken care of by λCLAM. This increases modularity of the system and makes it more open, in the sense that in principle any propositional theorem prover could be used in place of FTL.

This highly refined interplay between λCLAM and FTL can be seen as a sophisticated way of gaining control over the object-level theorem prover, from the point of view of the proof planner. λCLAM tells FTL what can be *neglected* to prove this particular branch.

In general, the tactic associated to a method can be somehow seen as a translation of the operational content of the method into a set of $c_{\mathbf{FOLTL}}$ rules. This logically

reflects the need of formally justifying the behaviour of the planner.

As an example, here is how the tactic associated to `detect_mutex` works:

```
(then_tac (tselect L R [StateName1 at Tau1, StateName2 at Tau1,
                        Corr1 at zero, Corr2 at zero, Mutex at zero] [])
(then_tac (tisol L R _ _)
(then_tac (repeat_go_tac (pre tlall))
(then_tac (pre tlbox) (pair_tac
  (then_tac (pre tlbox) (pair_tac
    (then_tac (pre tlbox) (pair_tac
      tauto_pl
      tent))
    tent))
  tent))
)))
```

Tactics are linked in a sequence thanks to `then_tac`. First `tselect` and `tisol` eliminate from the sequent all that does *not* look like items (a), (b) and (c) above. What remains must be enough to close this branch; the rest of the tactic therefore strips away $\forall$s and $\Box$s from the hypotheses (tactics `tlall`, `tlbox`) and then calls upon tactics `tauto_pl` and `tent` on each remaining branch of the proof, in order to close them respectively by propositional reasoning and entailment.

Using invariants

It has been sometimes necessary, especially in inductive proofs, to use the invariants described in Section 6.3. A method called `use_invariant` takes care of using the invariant to strengthen the goal, as is usually done in inductive reasoning. The well-known associated rule states that:

$$\frac{\longrightarrow I \quad \longrightarrow \Box(I \supset \bigcirc I) \quad \longrightarrow \Box(I \supset \varphi)}{\longrightarrow \Box\varphi}$$

if I can be proved to actually be an invariant (it holds forever from now on), and if it always *implies* the property φ we are interested in, then one can prove $\Box(I \supset \varphi)$ rather than the weaker $\Box\varphi$. I can be assumed in the hypotheses, provided we know that the invariant is an invariant.

The method employs a related FTL tactic. As it was the case earlier on, this method too can be seen as a rather smart wrapper for a tactic which, if left uncontrolled, would cause instability in the system (that is, it could potentially cut in a new formula at each inference step).

6.4.2 CTL-like E-path properties

The system must be able to eventually connect two users, at least under suitable conditions: it must be possible for *any two users* to be connected to each other (this will no longer be true once we add OCS, as one can figure out). Generalising the idea, every single state must be reachable, under suitable conditions; were it not so, the state could be neglected, and the related node could be deleted from the graph.

For instance, an $idle$ user who $offhooks$ must be able to get to the *ready* state at least in one case — that means that no user will always be out of service.

Properties such as these require *path quantifiers* such as those found in CTL / $FOCTL$ to be expressed; for example, the branching time model typical of CTL would allow one to use the E existential path quantifier and state that there is at least one time path such that any user will be *ready*. Intuitively speaking, "if everything goes right", any $idle$ user who $offhook$s will eventually be *ready*.

In our setting there is no way to directly express these concepts (actually, this is why linear-time logics are sometimes seen as approximations of branching-time logics); still, it seems fundamental to be able to prove them, especially for a telephone network: it is necessary to guarantee that the system *will* work fine, if it is not out of service!

So we adopt a "trick" similar to that of [CM02b]: we look for a *path* in the graph of Figure 6.4 leading to the required state.

- **Property 1a (BCS):** *it is possible for a user to eventually get ready.*

$$\forall x.\Diamond\ ready(x)$$

- **Property 1b (BCS):** *it is possible for a user to eventually connect to someone.*

$$\forall x.\Diamond\ \exists t.oconnected(x,t)$$

- **Property 1c (BCS):** *it is possible for a user to eventually be connected to someone.*

$$\forall x.\Diamond\ \exists t.tconnected(t,x)$$

The intuitive idea is to mimic *backward-reachability* as it happens in model checking. This has been done in a compound method (or methodical) called `exists_path`, which operates like this: first recognise what state appears in the $\Diamond$-formula in the goal; then

1. check whether we are already in the initial state, that is, check that $idle(x)$ can be proved; if so, stop. Otherwise,
2. find all trigger formulae which trigger the state we are currently in; for each of them,
3. find a related progress formula; set the state in the antecedent of the progress formula as the current state, and go back to the beginning.

Property **1a**, for example, should be proved like this: since $ready(x)$ is not the initial state, find what trigger formulae lead to *ready* (the reader can more easily check the situation on the graph in Figure 6.4). The only candidate is the one stating that if

we are $idle$ and we $offhook$ we will get to *ready*. The related progress formula has $idle(x)$ in the antecedent, so we are done.

Notice that the goal includes a $\Diamond$ operator: "eventually we will be ready". This implies proving that there exists a time in the future of 0 at which $ready(x)$ holds; but this will be easily proved using the transitivity property *n* times, where *n* is the number of states we have "back-traversed" before finding the initial state. The intuitive content of this method is: the property is true if it is true *along one path*:

```
compound fi exists_path
  (complete_meth
    (then_meth (repeat_meth mrall)
    (then_meth mrdia (pair_meth
      (repeat_meth
        (orelse_meth prove_init
        (orelse_meth find_trigger find_prog)))
      ent_meth
    )))
  )
  _
  true.
```

First, all $\forall$ (`repeat_meth mrall`) and the $\Diamond$ (`then_meth mrdia`) operators are stripped away. Notice that `mrdia` opens two branches, managed by the pairing method `pair_meth`. On one hand, the *second* branch is supposedly closed via a method called `ent_meth`, wrapping the entailment rule, since it represents the constraint that the time at which the $\Diamond$-formula in the goal was proved is in the future.

On the other hand, the first branch is taken care of by a repeated application of `prove_init`, `find_trigger` and `find_prog`, which enforce exactly the three steps outlined above.

Methods `find_trigger` and `find_prog` neglect all transitions not leading toward the state we want to reach. This is done via a tactic called `close_tac` which arbitrarily closes a proof branch without any explanation. One can view this tactic as a very carefully controlled application of the cut rule; for each progress formula, the resulting proof employs the tactic $n-1$ times, where n is the number of possible transitions that can be taken from the current state (including the self-loop transition implicit in the $\mathcal{W}$ operator); and, for each trigger formula, it is used $m-1$ times, where m is the number of states a user can get to by taking that transition. In this case, proof planning literally "directs" the search.

6.4.3 First-order invariants

By a first-order invariant we will denote, at this stage, a first-order formula (that is, not containing temporal operators) which must be proved to hold at all times. An example is Property **4** of the BCS in [CM02b], stating that "the dialled number is the same as the number of the connection attempt". In our model, this is enforced exactly by a transition formula, and could therefore be proved by a single application of the axiomatic tactic.

Rather, in order to test reasoning by invariants, we prefer to generalise a little this property, and try and prove it via the BCS invariant seen in Section 6.3.1:

- **Property 4a (BCS):** *the user we are trying to dial is the same as the user we have just dialled.*

$$\forall x,y.\Box\ (ready(x) \wedge dial(x,y)) \supset \bigcirc trying(x,y)$$

- **Property 4b (BCS):** *if I am ringing y and she offhooks, I'll be next connected to her.*

$$\forall x,y.\Box\ (oringing(x,y) \wedge offhook(y)) \supset \bigcirc oconnected(x,y)$$

- **Property 4c (BCS):** *if I am ringing y and she offhooks, she'll be next connected to me.*

$$\forall x,y.\Box\,(oringing(x,y) \wedge offhook(y)) \supset \bigcirc tconnected(y,x)$$

(In fact, Property **4c** needs the use of a system axiom.) Reasoning by using an invariant will here work like this: (1) rather than the goal directly, try to prove that the goal is implied by the invariant; (2) assuming the invariant is a *n*-ary disjunction and it appears among the hypotheses, open up *n* branches; (3) in $n-1$ branches, the proof will go through thanks to the detection of mutual exclusion (i.e., two state formulae will be asserted in the hypotheses at the same time), while the remaining branch will hopefully be closed by propositional reasoning.

The resulting method, called `all_paths`, looks like this:

```
compound fi all_paths
  (complete_meth
    (then_meth use_invariant
    (then_meth (repeat_meth mrall)
    (then_meth mrbox
    (then_meth mrimp
    (then_meth (try_meth (repeat_meth mlor))
    (orelse_meth detect_mutex foreach_state)
  ))))))
  _
  true.
```

Method `use_invariant` (see Section 6.4.1) employs the invariant; immediately after, the $\forall$ and $\Box$ operators are stripped away from the goal, and the implication is eliminated. Then, method `mlor`, wrapping the tactic for $l\vee$, is exhaustively applied to

open as many branches as there are disjuncts in the invariant. Lastly (`(orelse_meth detect_mutex foreach_state)`), each branch is either closed by mutual exclusion detection or via a method which isolates what are considered to be the right hypotheses, and then uses propositional reasoning. The use of system axioms is actually taken care of by this last method, which gives the compound method a rather high degree of generality.

6.4.4 Weak-until invariants

The situation becomes much more difficult, as expected, when the invariant to be proved has a $\mathcal{W}$ operator in it (recall Section 6.3). Property 2 of BCS in [CM02b] belongs to this category:

- **Property 2 (BCS):** *if I dial myself, I'll hear the busy tone before getting back to idle.*

$$\forall x.\Box\ (ready(x) \wedge dial(x,x)) \supset (\neg idle(x)\,\mathcal{W}\ busytone(x))$$

Once again, the proving strategy is inspired by model checking, this time by *forward reachability*: start from the state specified in the antecedent of the goal (in this case, $ready(x)$) and find a trigger telling us what happens if we take the transition specified in the same place (in this case, $dial(x,x)$). Open one branch for each transition found.

Then for each branch, that is, following each possible path forward, check whether we have reached the state on the right hand side of the $\mathcal{W}$ in the goal (in this case, $busytone(x)$). If it is the case, stop. Otherwise, find a progress formula and identify what transitions can be taken from this state; again, open a branch for each possible transition and, for each one, close the branch by mutual exclusion detection. Then go back to the beginning.

```
compound fi forward_search
  (then_meth (repeat_meth mrall)
  (then_meth mrbox
  (then_meth mrimp
  (then_meth mland
  (repeat_meth
    (then_meth trigger_to_trans
    (then_meth (try_meth (repeat_meth mlor))
    (then_meth (orelse_meth fulfilled_evt state_to_prog)
    (then_meth (try_meth (repeat_meth mlor))
    (orelse_meth detect_mutex id_meth
  ))))))))))
  _
  true.
```

The idea is that of keeping the left hand side of the $\mathcal{W}$ in the goal satisfied by mutual exclusion detection, until the right hand side is satisfied because the required state is reached.

6.4.5 Weak-until invariants (cont'd)

A slightly different kind of invariants, involving a $\mathcal{W}$ too, is Property 3:

- **Property 3a (BCS):** *if I am trying to connect to y, I'll keep on trying until I'll hear the busy tone or I'll be ringing her.*

$$\forall x,y.\Box\, trying(x,y) \supset trying(x,y)\,\mathcal{W}\ (busytone(x) \vee oringing(x,y))$$

- **Property 3b (BCS):** *if I am ready, I'll stay ready until I'll get back to idle or I'll be trying to connect to someone.*

$$\forall x,y.\Box\ ready(x) \supset ready(x)\ \mathcal{W}\ (idle(x) \vee \exists t.trying(x,t))$$

Slightly simpler than Property **2**, this is proved in a similar way, but trying to identify a trigger formula corresponding to the required goal. If this is not the case, the same method seen above (`state_to_prog`) is employed to let the system progress.

```
compound fi forward_progress
  (then_meth (repeat_meth mrall)
  (then_meth mrbox
  (then_meth mrimp
  (repeat_meth
    (then_meth state_to_prog
    (then_meth (try_meth (repeat_meth mlor))
    find_trigger
  ))))))
  _
  true.
```

6.4.6 Invariants for OCS

Once we add OCS to the system in the way outlined in Section 6.3.2, we expect in the first place that the characteristic property of OCS must be provable, stating that the model of OCS is appropriate. In particular:

- **Property 9 (OCS):** *assuming user x has a user t on his screening list, x may not be connected to t as originator.*

$$\forall x.\Box\ \exists t.(ocs(x,t) \supset \neg oconnected(x,t))$$

In order to prove Property **9**, we employ the invariant defined for OCS in Section 6.3.2 and open a search branch for each possible state user x is in; all of them but one are closed by detection of mutual exclusion, except for one, which goes through by propositional reasoning — this is reasonable, since the use of the OCS axiom $\forall x.\Box\ \neg ocs(x,x)$ should match with the condition $\neg ocs(x,x)$ when dealing with the transition from *ready* to *trying*. The associated method, called `inductive_box`, is a slight variant of `use_invariant` (see Section 6.4.1).

Moreover, with OCS on, we expect some of the BCS properties to be still provable, while some others are not. In particular, we are interested in proving that (a) Property **2** of BCS (see Subsection 6.4.4) is still valid, whereas (b) Property **3b** (see Subsection 6.4.5) is not. (a) is justified by the fact that no user may have himself on his screening list (this is a property of OCS reflected in a system axiom), and there is no reason why he should not hear the busytone if he self-dials; (b) cannot obviously work any longer, since a *ready* user trying to dial a screened user will never be *trying* to connect to her.

Let us note, by the way, that Property **9** in our model somehow represents the *negation* of Property **1** ("on some path a connection between any two users is possible"); therefore, the fact that is can be proved can be also seen as a proof of the interaction arising between it and Property **1**.

As far as Property **2** is concerned, we expect the very same methods employed to prove its validity with BCS to carry the proof on in this case.

On the other hand, Property **3b** represents a slightly more complex case. Planning the property does not go through as expected; in order to detect the interaction then, we set up a user, *alice*, and add a system axiom stating that she is *not* on anyone's screening list:

$$\forall x.\Box\ \neg ocs(x, alice)$$

Then we try to prove that, under this condition, Property **3b** specialised for Alice *does* hold:

$$\forall x,y.\Box\ ready(x) \supset ready(x)\,\mathcal{W}\ (idle(x) \vee trying(x,alice))$$

This should be proved via the very same method `forward_progress` seen for Property **3b**. Notice that here the open question of *how to systematically detect an interaction* arises — a discussion about this issue is left to Chapter 8.

6.5 Experimental methodology

When reporting on experiments, or evaluating how successful a technique has been with respect to its competitors, one should take into account not only how long was for the tool to (dis)prove correctness of the system, but also how long it took the user to devise the method.

This is all the more important in our framework, since proof planning tries to identify and mechanise common shapes in proofs. One would then expect that, once the idea on how to prove a statement has been formulated and implemented in a set of methods, subsequent, similar proofs would require less time, if not by the tool, at least by the user. In other words, there should be reusal of human time, as well as of methods.

We adopt Francisco Cantu's ([CO97]) three-fold classification of the tasks the user has to perform when trying to automate the proof of correctness of a system. Coherently we will refer to the user as to the *proof engineer*, that is, the developer of a formal proof for systems design and verification. Tasks (and the required human time) is divided at three level: *user tasks*, *proof tasks* and *tool tasks*.

User tasks have to do with formalising a problem and using λCLAM/FTL to solve it:

1. providing formal definitions of the specification of the system, of the required properties, of the language used;

2. building a set of methods to solve the problem and running λCLAM to obtain a proof plan;

3. executing the plan in order to get a proof of correctness; if the proof does not go through, revise the formalisation and try again.

In case this process does not work, proof tasks should come into play. These tasks deal with tuning proof techniques without modifying the tool:

1. alter the methods, its side conditions, its operational content, or its input/output goals;
2. provide a missing invariant (lemma) or modify an existing one;
3. alter the tactics associated with a method which should have been applied but failed to, or that was applied in the wrong place;
4. possibly, extend the theory with new operators or rules (this should hardly be the case).

Finally, the possibility of finding bugs in the code must be considered. If all the above fails, the proof engineer will probably have to resort to

1. provide new or different tacticals / methodicals, in order to circumvent a deficiency or inefficiency of the planner or prover;
2. find and correct wrong declarative content (i.e., predicates which work in the theory of λProlog but fail so to do because of the order in which clauses or predicates are written);
3. change the imported / accumulated modules structure (again, hopefully this will not happen);
4. circumvent bugs in the λProlog simulator.

In Chapter 7 we will report the human time required by the proof engineer, besides the timing of λCLAM/FTL, divided by task level. Although obviously not precise as the tool timings, these numbers will give an indication on how the method, in its entirety, works.

6.6 Chapter overview

This dense Chapter has dealt with the application of proof planning to **FOLTL**, and in particular to the case-study of Feature Interactions (FIs) in telecommunication systems. We have first described a preliminary experiment which was carried out in the early days of this research.

Aiming then to get a more general, flexible and extensible framework, we have shown how to concisely and precisely model FIs in **FOLTL** as a set of formulae defining the status of a user along time. The Basic Call Service and one feature, Originating Call Screening, have been introduced.

The Section after contains details of the methods we have devised for the properties of the system we are interested in. Our methods mimic simple steps of human reasoning such as “reaching a goal state from the initial state” or “explore all possible paths from now on, until a goal state is found”.

Lastly, we have outlined the experimental methodology we will be following in the next Chapter. Its main peculiarity is that, in evaluating the outcome of the experiments, we will be taking into account the human time required by the proof engineer, as well as the CPU time required by the machine in order to solve the problem. This will give a more precise idea of what it is like to tackle FIs via proof planning.

Chapter 7

Experimental results

In this Chapter we show data about the experiments carried out, and comment on the results obtained, also in comparison with some highly relevant related work. In Section 7.1 we present the results, first synthetically, then analysing them experiment by experiment; and in Section 7.2 we compare them with those obtained in [CM02b].

7.1 Test results

Table 7.1 reports the experimental results obtained by λCLAM/FTL in verifying the properties of models for BCS and BCS+OCS outlined in Chapter 6. Columns report, for each model and property proved (e.g. BCS and property **1a**), data about the proof plan and the proof (depth *d*, number of nodes #N, CPU Time in seconds), total CPU Time needed by planning and proof checking in seconds, and human time required to devise the solution (User, Proof, Tool tasks time and total human time, in man-hours).

Each row represents an experiment; all experiments define that a model (BCS or BCS+OCS) validates the required property, except for the third-last row, labelled OCS-3b, which denotes that property **3b** of the BCS interacts with OCS. The last two rows show average values and total timings.

All experiments were run on a PC equipped with an AMD K6 200MHz processor, 256 MB on board memory and Linux 2.4.7. We employed a special version of the

λProlog environment Teyjus v1.0-b33, especially patched for the MRG group at the University of Edinburgh, and λCLAM v4.0.0 (2002). The heap space of the λProlog compiler / simulator was raised to 512 MB in order to avoid heap overflow.

	Proof plan			Proof				Human time			
Property	*d*	#N	Time	*d*	#N	Time		U	P	T	
BCS+1a	13	15	11	23	31	2	**13**	2	100	200	**302**
BCS+1b	19	21	24	66	92	7	**31**	1	10	1	**12**
BCS+1c	15	17	15	38	52	3	**18**	1	1	1	**3**
BCS+4a	28	44	49	39	322	17	**66**	4	10	20	**34**
BCS+4b	28	44	58	39	321	20	**78**	1	1	2	**4**
BCS+4c	28	44	58	39	327	20	**78**	1	1	5	**7**
BCS+2	17	19	20	48	97	10	**30**	10	70	100	**180**
BCS+3a	14	16	11	41	112	14	**25**	4	10	10	**24**
BCS+3b	14	16	11	43	111	14	**25**	1	1	1	**3**
OCS+2	17	19	21	57	112	13	**34**	1	1	1	**3**
OCS+9	32	80	76	41	341	96	**172**	8	5	20	**33**
OCS-3b	14	16	11	47	110	20	**31**	20	10	10	**40**
Averages	20	29.6	30.4	43.4	169	19.7		3.5	18.3	30.9	
Totals			365			236	**601**	54	220	371	**645**

Table 7.1: Experimental results. Columns report, for each model and property proved, data about the proof plan and the proof (depth d, number of Nodes #N, CPU Time in seconds), total CPU Time needed by planning and proof checking in seconds, and human time required to devise the solution (User, Proof, Tool tasks time and total human time, in man-hours).

We now comment on each single experiment, analysing the structure of the plans

and proofs obtained and the CPU and human time needed.

7.1.1 Properties 1a, 1b, 1c

Let us first consider the three proofs of CTL-like E-path properties. Both the structure of the plans and proofs are quite similar. In fact, the proof plans resort to finding a path from *idle* to the required state, and its depth is perfectly correlated with the distance on the graph. Consider Figure 6.4: to prove Property **1a** ("eventually on some path every user gets *ready*") the proof planner only needs discover that any user can get to *ready* from *idle* in one step. Analogous considerations hold for Property **1b** ("eventually on some path every user gets *oconnected*", the required state is four steps away) and **1c** ("eventually on some path every user gets *tconnected*", the required state is two steps away). Timings, depths and numbers of nodes roughly reflect this proportion.

This is a clear quantitative indication that *the proof plan has captured the common structure in the three proofs*. The structure of the plans and proofs (not displayed) also corroborate this claim.

By inspection of the proofs, one can also see that they contain one application of `close_tac` per each transition neglected, as expected (see Subsection 6.4.2). These lemmas are necessary to let the proof carry on to the end and "close" the unwanted search branches.

As far as timings are concerned, proof planning dominates over proof checking, as expected, and Property **1b** is the hardest. Interestingly, the ratio between the depth and number of nodes of both the plans and the proofs are quite low, meaning that the plan / proof trees are, in all cases, quite narrow. This means proof planning is actually guiding the search in an efficient way. In fact, the proof of Property **1b** has a depth of 66 nodes, which is, if considered from the point of view of "simple" theorem proving, remarkable — it would probably be very hard for a general-purpose automatic theorem prover to find such a deep proof in such a complex logic.

As far as human time is concerned, consider Property **1a**: it was no great problem

to invent the proof plans (User time 2 man-hours) but it was quite hard to build the correct machinery, both in terms of methods (Proof time 100 man-hours) and in terms of adjusting the system (Tool time 200 man-hours). In fact this was the very first attempt, and, as expected, took a long time to set up.

The times scale down radically, however, if we proceed on to the other Properties, as expected, especially because *the very same set of methods work fine for all three of them*.

7.1.2 Properties 4a, 4b, 4c

Proof plans and proofs of these Properties employ a 9-fold invariant, that is, a disjunction with 9 disjuncts, which opens up 9 search branches. This "shapes" them so that they present remarkable similarities in structure. Since neither λCLAM nor FTL manage directly *n*-ary disjunctions, both the proof plans and the proofs look a little deeper than they actually are (8 binary nodes, each one employing rule $l\vee$, must be used). The remarkable number of nodes of the proofs really comes by 8 similar search branches. Notice also that these proofs do *not* include the proof of the invariant itself.

On a smaller scale, there is a pattern in human times which is similar to that one can see for the previous set of Properties. Tool time appears a little larger (5 man-hours) for Property **4c** since it was necessary to code and use a system axiom in that case, in order to have the proof go through.

7.1.3 Property 2

This problem required a big effort in human terms, as shown in the Table, since it was necessary to devise a way of proving an invariant with a $\mathcal{W}$ operator in it. In particular, a number of different methods were required, and it was not at all clear how to translate the intuitive ideas behind them into tactics. The final numbers about this plan / proof are slightly deceiving, since two lemmas were required.

One lemma states that if $p \mathcal{U} q$ holds, then eventually $p \wedge q$ will be true. This is given by the semantics of the operator itself. This lemma could be proved automatically.

The second lemma is more interesting, and reflects a weakness in the calculus $c_{\mathbf{FOLTL}}$. This lemma states an instance of the following: if p @ t and $\Box p$ @ $t+1$ hold, then it also holds that $\Box p$ @ t. The statement could be proved by the well-known inductive definition of $\Box p$:

$$\Box p \stackrel{\text{def}}{=} p \wedge \bigcirc \Box p$$

To treat this case, and similar ones, with an acceptable degree of generality, some form of modal μ-calculus (that is, fix-point definitions) would be required. We believe this is an interesting line of future work.

7.1.4 Properties 3a, 3b

Another pair of very similar plans and proofs, proved by the same set of methods. The first one required some effort on the human side, while the second was proved quite easily.

7.1.5 OCS and 9, 2, 3b

The invariance of Property **2** with OCS on could be proved with little or no modification to the methods explained above, when no feature was enabled. As one can see, the human time required was small. If compared with the figures above, for BCS+**2**, the proof is somehow deeper and larger because of the added complexity of OCS.

Validating Property **9** with OCS requires the largest effort of the whole benchmark set. This is due to the use of the OCS invariant, which introduces complexity in each branch of both the proof plan and the proof. The remarkably high human time was required in order to find a suitable way of expressing the property itself, and to devise a suitable invariant.

Finally, the proof of the only interaction in the benchmark, that is, OCS and Property **3b**, proved quite complex to be devised, as is witnessed by the human time reported under User time. Actually, there is still no systematic way of determining how to detect an interaction when there is one, and this single problem needed some 20 man-hours to find out how to discover it.

7.1.6 Averages and totals

Some final considerations about the figures reported in Table 7.1. From the "Averages" row, one can see that:

- the average proof plan is 20 nodes deep and contains about 30 nodes in total (ratio: 0.66). This suggests that the shape of a proof plan is quite narrow and deep;
- the average proof is about 43 nodes deep and has 169 nodes in total (ratio: 0.25), which seems to suggest that there is a lot more "decoration" in a proof than in a proof plan. This somehow agrees with the idea that the proof plan abstracts away much more than is allowed in a proof;
- proof planning time dominates over proof checking time by a factor of 3 to 2. This is sensible as well, since most of the "intelligence" of the system lies in the plan rather than in the proof, although the tactics in the methods can be rather involved, if not require some degree of automation themselves;
- the whole set of benchmarks can be solved on a rather slow machine in something more than 10 minutes of CPU time, but the total human time required to set the machinery up was some 4 man-months full-time, assuming one man-month full-time is 160 man-hours;
- there is definite dominance of Tool time over Proof time, and of Proof time over User time, meaning that it is always simpler to invent planning strategies than to

realise them, and that it is simpler to realise them than to have your tool actually execute them!

7.2 Comparison with related work

Calder and Miller's work (see e.g., [CM00, CM01, CM02b]) is the main source of inspiration to the experimental test-set presented in this Chapter. It is anyway hard to quantitatively compare the results obtained by Calder and Miller and ours, since (1) the machines used are rather different, and (2) there is no indication on the human time required by Calder and Miller's approach in their papers.

From a qualitative point of view our approach has, in general, a precise advantage over Calder and Miller's (and any other model-checking-based approach), since our proofs use no finitary approximation whatsoever. But it must as well be remarked that, in [CM02a], the authors extend their approach to an unbounded number of users, thanks to an abstraction-based technique. Moreover, their model is much more detailed and realistic than ours, also thanks to the use of a well-established modelling language such as ProMeLa; lastly, our approach works by *proving* formulae via sequent calculi, meaning that if a formula is not valid, there is so far little chance of finding a counterexample to it (a counter-model of the formula), which, in the case of formal methods, is usually desirable.

As a final remark, notice that in [CM02b] the authors solve the problem for a wider set of properties than ours. The restricted set of properties we have considered is mainly due to the fact that, rather than introducing a fully-fledged, new technique to the Formal Methods community, we wanted to demonstrate how Proof Planning could be effectively adapted to a real case-study. From this point of view, more work is needed for this approach to be deployed in the community.

7.3 Chapter overview

This last Chapter presents the experimental results obtained by λCLAM/FTL applied to FIs. The whole benchmark is solved in less than 10 minutes CPU time, but it has required some 4 man-months of human time.

Comparison with related work is encouraging, modulo the smaller level of detail of our model, but the greater generality of the results we obtain (i.e., no finitary approximation is used).

Some simple but interesting considerations about the shape of the plans and proofs show that proof planning serves here exactly as a guidance for the object-level theorem prover, as is in its spirit, and that it effectively captures recurring patterns in proofs.

Chapter 8

Conclusions

The work behind this thesis was originally funded by EPSRC in a project called MFOTL (*Mechanising First-Order Temporal Logics*). In the project proposal, dating back to 1998, one can read:

> The aim of this project is to apply novel techniques from artificial intelligence to the development of a *practical* theorem-proving system for First-Order Temporal Logics (FOTL).
>
> [...]
>
> Thus, this project will combine work on temporal and inductive theorem-proving in order to investigate how a proof-planning approach may be used to improve first-order temporal theorem-proving.

We claim that the original aim of the project is fulfilled by the results achieved in this thesis:

1. the sequent calculi developed and exposed in Chapter 3 lay the theoretical basis to an AR-oriented approach to quantified modal logics (see also [CS02b]);

2. Chapters 4 and 5 extend the approach to **FOLTL** and describe how a sequent calculus for this logic can be mechanised in an interactive, tactic-based theorem prover based upon the higher-order logic programming language λProlog; the proof planner λCLAM is then coupled with the theorem prover in order to fully realise the proof planning approach to **FOLTL** (see also [CS00, CS01]);

3. Chapters 6 and 7 show that the approach as a whole is viable, at least for a significant case-study which had been tackled mostly in a finitary way so far. An initial result in this direction was published also in [CS02a].

With respect to the original aim of the project, it must be remarked that this thesis is slightly biased toward theoretical results. In particular, there is no mention of quantified *modal* logics in the proposal; but we have felt that such a general approach as that presented in Chapter 3, which almost came out as a by-product of our study on temporal logics, deserved a full treatment.

Also, there is less focus upon *inductive* theorem proving in **FOLTL** in this thesis, than indicated in the project proposal. Although proofs by induction using invariants are generated during the examination of the case-study, one cannot say that the system is generating new induction schemes or applying old ones in a smart way.

Nevertheless, we believe that the original contributions of this thesis have advanced the state-of-the-art in automated reasoning for modal and temporal logics, at least in the following ways:

- a new, systematic presentation of quantified modal logics has been given, which generalises and enlarges previous results in the field;
- proof planning has been applied to a logic which had never been tackled by the paradigm; in doing this, a set of new proof planning methods has been devised;
- the proof planner λCLAM has actually been coupled with an object-level theorem prover, so that its proof plans have been validated in the object logic and proved to represent sound ways of reasoning; as far as we know, this is the first time it has been done;
- a well-known problem in formal methods, namely that of Feature Interactions in telecommunication systems, has been explored and solved to some extent without any finitary approximation.

Lastly, it is interesting to remark that the proof obtained by λCLAM/FTL when applied to the case-study (see Chapter 7) are remarkably deep and large, the deepest being 66 nodes deep, and the largest consisting of 341 nodes, not taking into account the proofs of the invariants required, and a few lemmas which were necessary. The proofs were obtained in a reasonable amount of CPU time; and the object logic is not only undecidable, but non recursively enumerable. These results are remarkable from the point of view of standard automated theorem proving; still, from the standpoint of proof planning, they look like a starting point — much more can be achieved.

8.1 Future work

Here are some of the possible lines of future work, or open questions that, in our opinion, deserve more investigation:

- *On quantified modal and temporal logics.* The theoretical framework of Chapter 3 could be further extended to non-constant domains, or to more temporal logics (say, **FOCTL**); more soundness and completeness proofs could be studied (for instance: is $c_{\mathbf{FOLTL}}$ complete for monodic **FOLTL**? and for propositional *LTL*?); lifting the limitation that the set of frame axioms must be finite could lead us to a calculus complete for **FOLTL** (for instance, adding the ω-rule as an infinite set of frame axioms?);

- *On* λCLAM *and* FTL. The system as a whole could be re-engineered in a number of ways, willing to strengthen it or make it more robust. Among the possible options: re-implementing it in a more stable framework than λProlog (say Isabelle, or some imperative language); making it more open to the interaction with other object-level theorem provers; improving the presentation of results and the user interface, in order to make it more user-friendly;

- *On Feature Interactions/1.* Does the system scale up, e.g., to a more complex

model, or if we add more features, or if we look for more complex properties (liveness, response, absence of starvation)?

- *On Feature Interactions/2.* How to systematically detect an interaction? That is: can we devise a proof planning schema able to detect interactions in a more general way that that outlined in Chapter 6? This is, maybe, the most important open question regarding the experiments carried on in this thesis, and the practical applicability of our method at all.

- *On proof planning applied to* **FOLTL**. Can the approach used in Chapter 6 to model users in the problem of FI be applied to more problems in formal methods? And if so, can proof planning be adapted to work in that case?

Appendix A

An interactive session in FTL

Recall Figure 3.4; what follows is a recorded session in which precisely that proof tree is generated. Symbols such as `<lc-0-1>` and `_2766` denote, respectively, fresh constants as introduced by generative rules, and metavariables as introduced by non-generative rules.

Tactics are denoted by a `t` prepended to an acronym for the associated rule's name; for instance, `tlbox` is the tactic wrapping rule $l\Box$. Two modes are possible to apply a tactic: either we specify one number (two in the case of rule `ax`) indicating which is the main formula (e.g., `tax 1 1.` applies rule `ax` to the first antecedent and the first consequent), or we prepend the keyword `pre` to the tactic, meaning that the first formula left-to-right which has the required shape (e.g., $\Box$-formulae for $r\Box$ and $l\Box$ and so on) must be used.

```
Welcome to Teyjus

Copyright (C) 1999 Gopalan Nadathur
Teyjus comes with ABSOLUTELY NO WARRANTY
This is free software, and you are welcome to redistribute it
under certain conditions.  Please view the accompanying file
"COPYING" for more information.
```

```
[ftl] ?- name "ax4" Phi, top (Phi at zero).

----- Premises:
----- Conclusions:
(1) box phi imp box (box phi) at zero

***** Tactic? pre trimp.
----- Premises:
(1) box phi at zero
----- Conclusions:
(1) box (box phi) at zero

***** Tactic? pre trbox.
----- Premises:
(1) zero bef <lc-0-1>
(2) box phi at zero
----- Conclusions:
(1) box phi at <lc-0-1>

***** Tactic? pre trbox.
----- Premises:
(1) <lc-0-1> bef <lc-0-2>
(2) zero bef <lc-0-1>
(3) box phi at zero
----- Conclusions:
(1) phi at <lc-0-2>

***** Tactic? pre tlbox.
```

```
----- Premises:
(1) phi at _1740
(2) <lc-0-1> bef <lc-0-2>
(3) zero bef <lc-0-1>
(4) box phi at zero
----- Conclusions:
(1) phi at <lc-0-2>

***** Tactic? tax 1 1.
----- Premises:
(1) <lc-0-1> bef <lc-0-2>
(2) zero bef <lc-0-1>
(3) box phi at zero
----- Conclusions:
(1) zero bef <lc-0-2>
(2) phi at <lc-0-2>

***** Tactic? ttrans.
----- Premises:
(1) <lc-0-1> bef <lc-0-2>
(2) zero bef <lc-0-1>
(3) box phi at zero
----- Conclusions:
(1) _2706 bef _2707
(2) zero bef <lc-0-2>
(3) phi at <lc-0-2>

***** Tactic? tax 2 1.
```

```
----- Premises:
(1) <lc-0-1> bef <lc-0-2>
(2) zero bef <lc-0-1>
(3) box phi at zero
----- Conclusions:
(1) <lc-0-1> bef _2728
(2) zero bef <lc-0-2>
(3) phi at <lc-0-2>

***** Tactic? tax 1 1.
----- Premises:
(1) zero bef <lc-0-2>
(2) <lc-0-1> bef <lc-0-2>
(3) zero bef <lc-0-1>
(4) box phi at zero
----- Conclusions:
(1) zero bef <lc-0-2>
(2) phi at <lc-0-2>

***** Tactic? tax 1 1.

----> Proof found. Formula box phi imp box (box phi) at zero
----> is proved by proof
rimp 1 (
         rbox 1 (W1\
          rbox 1 (W2\
           lbox 3
            (ax 1 1)
```

```
            (trans (ax 2 1) (ax 1 1) (ax 1 1))
        )))
```

```
yes
```

Appendix B

Correctness of the implementation

In this Appendix we give a proof of the correctness of the implementation of $\mathcal{C}_{\mathbf{QL}}$ in FTL. The word *correctness* here means that, for every object logic **QL** or **FOLTL**, a formula has a proof in $\mathcal{C}_{\mathbf{QL}}$ if and only if a convenient meta-term can be derived in FTL via the standard λProlog search operations. Needless to say, this fact is not related with any decidability or completeness argument whatsoever; what it tells us is that there is a precise correspondence between provable formulae (theorems) and λProlog terms in FTL, that is, that objects of type `proof` obtained by means of FTL faithfully represent proofs of theorems in **QL**.

The proof follows closely the one found in [Fel93], to which the interested reader is referred to, once again. FTL is carefully written in order to adhere to the design guidelines found in that paper, so that, modulo bugs in the code, that proof is easily adapted to our case.

Without loss of generality and for the sake of simplicity, from now on, we carry the proof on only for a subset of the object logic, namely that required for $\mathcal{C}_{\mathbf{QK}}$ (it is clear how to extend it). What we want to do here is to establish a correspondence between objects of the object-language (goals) and objects of the metalanguage (higher-order terms). The first step is, then, to establish a mapping between the basic symbols of the object-language and objects in the metalanguage. First of all, the two sorts of our

language, ι and θ, are defined as types:

```
kind iota     type.
kind theta    type.
```

Then, let us assume the existence of a bijective mapping Φ between symbols in the sets defined in Subsection 3.1.1 and constants of the metalanguage. Let $\mathsf{dom}(\Phi)$ denote the domain of Φ. Φ maps elements of $\mathscr{P}$, $\mathscr{F}$ and so on to constants of suitable type `iota`, `theta`, `lformula` and the like. For instance, assuming the existence of a ternary Skolem function $cv \in \mathscr{F}'$, as is done, e.g., in Subsection 3.2.2, $\Phi(cv) = \mathtt{cv}$ where, using the λProlog notation to represent types, `cv` has type `theta -> theta -> theta -> theta`. Therefore, metalevel *typing* of the constants provides the *rank* of the terms of the object-language[1]. Φ is actually realised in the signature of the modules of FTL, for instance

```
type   cv  theta -> theta -> theta -> theta.
```

Let us also assume the existence of a mapping ρ from first-order variables to metavariables of type `iota` or `theta`, their type being derivable at compile-time by the λProlog interpreter; then, by means of Φ and ρ object-level terms can be encoded in the meta-logic like this:

$$
\begin{array}{rcll}
\langle\langle x\rangle\rangle & := & \rho(x) & \\
\langle\langle p(s_1,\ldots,s_m)\rangle\rangle & := & \Phi(p)\ \langle\langle s_1\rangle\rangle,\ldots,\langle\langle s_m\rangle\rangle & \text{where } p \in \mathsf{dom}(\Phi) \\
\langle\langle A\ @\ \tau\rangle\rangle & := & \langle\langle A\rangle\rangle\ \mathtt{@}\ \langle\langle\tau\rangle\rangle & \\
\langle\langle \neg A\rangle\rangle & := & \mathtt{neg}\ \langle\langle A\rangle\rangle & \\
\langle\langle A \supset B\rangle\rangle & := & \langle\langle A\rangle\rangle\ \mathtt{imp}\ \langle\langle B\rangle\rangle & \\
\langle\langle \forall x.A\rangle\rangle & := & \mathtt{all\ x\backslash}\ \langle\langle A\rangle\rangle & \text{where } \rho(x) = \mathtt{x} \\
\langle\langle \Box A\rangle\rangle & := & \mathtt{box}\ \langle\langle A\rangle\rangle & \\
\langle\langle \Gamma \longrightarrow \Delta\rangle\rangle & := & \langle\langle\Gamma\rangle\rangle\ \mathtt{-->}\ \langle\langle\Delta\rangle\rangle &
\end{array}
$$

[1] this is a slight abuse of language: the rank of a symbol, as defined in Subsection 3.3.1, only concerns sorts ι and θ, whereas here we consider *every* object of the syntax as having a rank, e.g., nullary elements of $\mathscr{P}$ have the rank of propositions, which is neither ι nor θ, etc.

(we use the convention that the encoding of a set of formulae is represented as the list of the encodings). This encoding satisfies the condition that, for any first-order terms or formulae M and N and variable x, $\langle\langle [N/x]M \rangle\rangle =_\beta [\langle\langle N \rangle\rangle / \mathtt{x}] \langle\langle M \rangle\rangle$.

It is also the case that a meta-term of type `iota`, `theta`, `lformula` or `formula` can be decoded by inverting the above encoding; the decoding of term `M` is indicated by $||\mathtt{M}||$ and is defined analogously as the encoding above.

Let now $\langle \Sigma, \mathcal{P} \rangle$ represent the λProlog program which implements FTL, restricted to $\mathcal{C}_{\mathbf{QK}}$; then

Theorem 31 (Correctness of the implementation)

1. *Let* Π *be the proof in* $\mathcal{C}_{\mathbf{QK}}$ *of a sequent* $\Gamma \longrightarrow \Delta$*; then there is an object* `P` *of type* `proof` *such that* `P proves` $\langle\langle \Gamma \longrightarrow \Delta \rangle\rangle$ *is derivable from* $\langle \Sigma, \mathcal{P} \rangle$*;*

2. *Let the clause*

   ```
   P proves Gamma --> Delta
   ```

   be derivable from $\langle \Sigma, \mathcal{P} \rangle$*; then the sequent* $||$`Gamma --> Delta`$||$ *is derivable in* $\mathcal{C}_{\mathbf{QK}}$.

Proof: item 1 follows from this argument: suppose $S_1, \ldots, S_n, n \geq 0$ are the premises of rule $\rho \in \mathcal{C}_{\mathbf{QK}}$, and S is its conclusion; let `P1`, …, `Pn` be variables of type `proof`; finally, let Σ' and $\mathcal{P}'$ be the signature and program

$$\begin{aligned} \Sigma' &:= \Sigma \cup \{\texttt{type P1 proof.}, \ldots, \texttt{type Pn proof.}\} \\ \mathcal{P}' &:= \mathcal{P} \cup \{\texttt{P1 proves } \langle\langle S_1 \rangle\rangle., \ldots, \texttt{Pn proves } \langle\langle S_n \rangle\rangle.\} \end{aligned}$$

Then there is a term `P` of type `proof` such that the goal `P proves` $\langle\langle S \rangle\rangle$ is derivable from $\langle \Sigma', \mathcal{P}' \rangle$. This is shown by induction on the height of Π, examining all rules in $\mathcal{C}_{\mathbf{QK}}$ and the associated tactics.

For the base cases (rule ax and a few more), `P proves` $\langle\langle S\rangle\rangle$ must be exactly one of the `P1 proves` $\langle\langle S_i\rangle\rangle$., modulo the λProlog higher-order unification algorithm[2], therefore it is immediately derived by BACKCHAIN.

For the step cases, we just show some significant cases in detail, namely: *(i)* for tactic `trimp`, wrapping rule $r\supset$, which shows how the proof carries on for rules dealing with Boolean operators; *(ii)* for tactic `trall`, wrapping rule $r\forall$, which shows how the proof carries on for rules dealing with generative quantifiers, and *(iii)* for tactic `tlall`, wrapping rule $l\forall$, which shows how the proof carries on for rules dealing with non-generative quantifiers, introducing metavariables.

- If the rule is $r\supset$, then S has the shape $\Gamma \longrightarrow \varphi \supset \psi@\tau, \Delta$. By the induction hypothesis, the clause `P1 proves` $\langle\langle \Gamma, \varphi@\tau \longrightarrow \psi@\tau, \Delta\rangle\rangle$ is derivable from $\langle \Sigma', \mathcal{P}'\rangle$. Then by BACKCHAIN on the clause for tactic `trimp`, enforcing rule $r\supset$,

  ```
  trimp Pos
    ((rimp Pos P) proves (Gamma --> Delta))
    (P proves [Phi at Tau|Gamma] --> [Psi at Tau|Delta']) :-
    delete Pos (Phi imp Psi at Tau) Delta Delta'.
  ```

  `P proves` $\langle\langle \Gamma \longrightarrow \varphi \supset \psi@\tau, \Delta\rangle\rangle$ is derivable from $\langle \Sigma', \mathcal{P}'\rangle$;

- If the rule is $r\forall$, then S has the shape $\Gamma \longrightarrow \forall x.\varphi@\tau, \Delta$. By the induction hypothesis, the clause `P1 proves` $\langle\langle \Gamma \longrightarrow \varphi[a/x]@\tau, \Delta\rangle\rangle$ is derivable from $\langle \Sigma' \cup \{\texttt{type a i.}\}, \mathcal{P}'\rangle$. As already stated, the encoding $\langle\langle\rangle\rangle$ is such that

  $$\langle\langle \varphi[a/x]\rangle\rangle =_\beta \texttt{x\(}(\langle\langle \varphi\rangle\rangle \texttt{a})$$

  Therefore, the above goal is equivalent to `(P a) proves (x\(`$\langle\langle\varphi\rangle\rangle$ `a))`; by the GENERIC directive, goal `pi a\((P a) proves (x\(`$\langle\langle\varphi\rangle\rangle$ `a)))` is derivable

[2] since metavariables introduced in sequents only stand for first-order variables, this is standard first-order unification.

from $\langle \Sigma', \mathcal{P}' \rangle$; now by BACKCHAIN on the clause for tactic `trall`, enforcing rule $r\forall$,

```
trall Pos
  ((rall Pos P) proves (Gamma --> Delta))
  (forall_goal a\ ((P a) proves Gamma --> [(Phi a) at Tau|Delta'])) :-
  delete Pos (all Phi at Tau) Delta Delta'.
```

the goal `P proves (all x\⟨⟨φ⟩⟩)` is derivable from $\langle \Sigma', \mathcal{P}' \rangle$; notice the use of the `forall_goal` goal constructor (in the sense of a proof/sequent pair), which wraps the `pi` directive necessary to introduce a fresh term in the derivation. Notice also that the proof term `P` is required here to have the fresh term `a` as an argument;

- If the rule is $l\forall$, then S has the shape $\Gamma, \forall x.\varphi@\tau \longrightarrow \Delta$. By the induction hypothesis, the clause `P1 proves` $\langle\langle \Gamma, \varphi[t/x]@\tau \longrightarrow \Delta \rangle\rangle$ is derivable from $\langle \Sigma, \mathcal{P}' \rangle$; here the signature Σ must either contain an explicit statement that t has the same type of x, or be such that this information is derivable at compile-time. By the same argument of the Item above, the above goal is equivalent to

  `P proves (x\(⟨⟨φ⟩⟩ t))`

  .

  By BACKCHAIN on the clause for tactic `tlall`, enforcing rule $l\forall$,

  ```
  tlall Pos
    ((lall Pos P) proves (Gamma --> Delta))
    (P proves [(Phi C) at Tau|Gamma] --> Delta) :-
    member Pos (all Phi at Tau) Gamma.
  ```

  the goal `P proves (all x\⟨⟨φ⟩⟩)` is derivable from $\langle \Sigma', \mathcal{P}' \rangle$ (notice that here no `delete` operation is performed, since we want a copy of the main formula to be

in the premise). The metavariable `C` in place of the term *t* is actually carried to the end of the proof, until either the tree is not closed, and this clause is backtracked over, or a unification is performed by a closing rule (see the base case of this proof).

Item 1 follows from a similar argument, relying on the existence of λProlog derivations obtained (only) via the six primitives described in Section 5.1.1. Altogether, they constitute a sound and complete (with respect to intuitionistic logic) non-deterministic search procedure for the language of λProlog higher-order hereditary Harrop formulae (see Theorem 2.1 in [Fel93]).

Briefly, let `P,P1,...,Pn` be $n+1$ distinct variables of type `proof`; moreover, let `S,S1,...,Sn` be $n+1$ distinct variables of type `sequent`. Let Σ' be the signature Σ, plus the signatures for `P, P1,...,Pn`, plus a declaration of type `i` for every variable appearing free in `P, S, S1,...,Sn`. If the clause `(P proves S)` is provable from $\langle \Sigma', \mathcal{P} \cup \{\texttt{(P1 proves S1)},\ldots,\texttt{(Pn proves Sn)}\}\rangle$ then there is a proof of $||S||$ in $\mathcal{C}_{\mathbf{QK}}$.

The proof is analogous to (1) by induction on the structure of `P`.

•

Bibliography

[AABdR00] Rafael Accorsi, Carlos Areces, Wiet Bouma, and Maarten de Rijke. Feature as constraints. In Muffy Calder and Evan Magill, editors, *Proceedings of the 6th International Workshop on Feature Interactions in Telecommunications and Software Systems*, Glasgow, Scotland, May, 17–19 2000. IOS Press.

[AFMO85] T. Aoyagi, M. Fujita, and T. Mota-Oka. Temporal logic programming language tokio: programming in tokio. *Logic Programming '85*, 221:128–137, 1985.

[AM85] M. Abadí and Z. Manna. Non-clausal temporal deduction. *Logic of programs*, 85:1–15, 1985.

[AM89] Martin Abadí and Zohar Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–296, September 1989.

[AM90] Martin Abadí and Zohar Manna. Nonclausal deduction in first-order temporal logic. *Journal of the ACM*, 37(2):279–317, April 1990.

[AVFY98] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In ACM, editor, *PODS '98. Proceedings of the ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, June 1–3, 1998, Seattle, Washington*, pages 179–187, New York, NY 10036, USA, 1998. ACM Press.

[BBC+96] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, New Brunswick, NJ, USA, July 1996. Springer Verlag.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BHWZ02] S. Bauer, I. Hodkinson, F. Wolter, and M. Zakharyaschev. On non-local propositional and one-variable quantified CTL^*. In *Proceedings of TIME 2002*, pages 2–9. IEEE Computer Science Press, 2002.

[BJK94] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services, 1994.

[BK00] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

[BLL+96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1996.

[Blo97] Johan Blom. Formalisation of requirements with emphasis on feature interaction detection. In *Proceedings of the Fourth International Workshop on Feature Interactions in Telecommunications Systems*, 1997.

[BMV96] David Basin, Sean Matthews, and Luca Vigano. A topography of labelled modal logics. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 75–92. Kluwer Academic Publishers, March 1996.

[BMV97a] D. Basin, S. Matthews, and L. Vigano. A new method for bounding the complexity of modal logics. *Lecture Notes in Computer Science*, 1289:89–102, 1997.

[BMV97b] David Basin, Sean Matthews, and Luca Vigano. Labelled propositional modal logics: Theory and practice. *Journal of Logic and Computation*, 7(6):685–717, 1997.

[BMV98] David Basin, Seán Matthews, and Luca Viganò. Labelled modal logics: Quantifiers. *Journal of Logic, Language, and Information*, 7(3):237–263, 1998.

[Bun88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bun91] Alan Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture

Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[BvHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[CBSB96a] Francisco Cantu, Alan Bundy, Alan Smaill, and David Basin. Experiments in automating hardware verification using inductive proof planning. Research Paper 828, Dept. of Artificial Intelligence, University of Edinburgh, 1996. Also in Proceedings of the Formal Methods for Computer-Aided Design Conference, 1996.

[CBSB96b] Francisco Cantu, Alan Bundy, Alan Smaill, and David Basin. Experiments in automating hardware verification using inductive proof planning. In M. Srivas and A. Camilleri, editors, *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, pages 94–108. Springer-Verlag, 1996.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. 11th International Computer Aided Verification Conference*, pages 495–499, 1999.

[CG87] E. M. Clarke and O. Grumberg. Research on Automatic Verification of Finite-State Concurrent Systems. In *Annual Review of Computer Science*, pages 269–290, Carnegie Mellon University, Pittsburgh, 1987.

[CG92] Giovanna Corsi and Silvio Ghilardi. Semantical aspects of quantified modal logic. In Cristina Bicchieri and Maria Luisa Dalla Chiara, editors, *Knowledge, belief, and Strategic Interaction*, pages 167–196. Cambridge University Press, 1992.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CH95] Max J. Cresswell and G. E. Hughes. *A New Introduction to Modal Logic*. Routledge, London, 1995.

[CK91] John G. Cleary and Vinit N. Kaushik. Updates in a temporal logic programming language. Technical Report 91/427/11, University of Calgary, April 1991.

[CKMRM02] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 2002.

[Cla97] Edmund M. Clarke. Model checking. *Lecture Notes in Computer Science*, 1346:54–56, 1997.

[CM00] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, 2000.

[CM01] Muffy Calder and Alice Miller. Using SPIN for feature interaction analysis — A case study. In M.B. Dwyer, editor, *Model checking software: 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001: proceedings*, pages 143–162. Springer, 2001. Lecture Notes in Computer Science No. 2057.

[CM02a] M. Calder and A. Miller. Automated verification of any number of concurrent, communicating processes. In *Proceedings of the 17th IEEE Automated Software Engineering (ASE 2002)*, pages 227–230, 2002.

[CM02b] Muffy Calder and Alice Miller. Feature interaction detection by pairwise analysis of LTL properties. Submitted to FMSD - Formal Methods

in Software Development; still unpublished at the time of writing (January 2005), April 2002.

[CO97] F. Cantu-Ortiz. *Proof Planning for Automating Hardware Verification*. PhD thesis, Dept of Artificial Intelligence, 1997.

[CS00] C. Castellini and A. Smaill. A modular, tactic-based approach to first-order temporal theorem proving. In *Proceedings of ICTL 2000*, Dresden, Germany, 2000.

[CS01] C. Castellini and A. Smaill. Tactic-based theorem proving in first-order modal and temporal logics. In E. Giunchiglia and F. Massacci, editors, *Proceedings of IJCAR WS10: Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics*, TR DII 14/01. University of Siena, June 2001.

[CS02a] C. Castellini and A. Smaill. Proof planning for feature interactions: A preliminary report. In M. Baaz and A. Voronkov, editors, *Proceedings of LPAR 2002*, LNAI 2514, pages 102–114. Springer, October 2002.

[CS02b] C. Castellini and A. Smaill. A systematic presentation of quantified modal logics. *Logic Journal of the IGPL*, 10(6):571–599, 11 2002. Also available as Informatics Research Report EDI-INF-RR-0150, University of Edinburgh.

[DF01] Anatoli Degtyarev and Michael Fisher. Towards first-order temporal resolution. *Lecture Notes in Computer Science*, 2174:18–32, 2001.

[DS02] S. Demri and Ph. Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Information and Computation*, 174(1):84–103, 2002. Full version of the STACS'98 paper.

[DV01] A. Degtyarev and A. Voronkov. Equality reasoning in sequent-based calculi. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 10, pages 611–706. Elsevier Science, 2001.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[Esp97] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[Fel88] A. Felty. Specifying theorem provers in a higher-order programming language. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 61–80. Springer-Verlag, 1988.

[Fel89] A. Felty. *Specifying and implementing theorem provers in a Higher Order Programming Language*. PhD thesis, University of Pennsylvania, 1989.

[Fel93] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, 1993.

[Fel01] Amy Felty. Temporal logic theorem proving and its application to the feature interaction problem. In E. Giunchiglia and F. Massacci, editors, *Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics*, TR DII 14/01. University of Siena, June 2001.

[Fis92] Michael Fisher. A normal form for first-order temporal formulae. *Lecture Notes in Computer Science*, 607:370–384, 1992.

[Fis97] Michael Fisher. A normal form for temporal logics and its applications in theorem-proving and execution. *Journal of Logic and Computation*, 7(4):429–456, August 1997.

[FM98] Melvin Fitting and Richard L. Mendelsohn. *First Order Modal Logic*. Kluwer Academic Publishers, Dordrecht, 1998.

[FN00] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. In *Feature Interactions Workshop*. IOS Press, 2000.

[FO92] M. Fisher and R. Owens. From the past to the future: Executing temporal logic programs. In *Proceedings of the Conference on Logic Programming and Automated Reasoning (LPAR)*, 624, St. Petersberg, Russia, July 1992. Springer-Verlag: Heidelberg, Germany.

[FT97] Amy Felty and Laurent Thery. Interactive theorem proving with temporal logic. *Journal of Symbolic Computation*, 23(4):367–397, 1997.

[Gab87] D. Gabbay. Modal and temporal logic programming. In *Temporal Logics and Their Applications*, pages 197–237. Academic Press, London, 1987.

[Gab96] Dov M. Gabbay. *Labelled Deductive Systems, Volume 1*. Oxford University Press, Oxford, 1996.

[Gal86] J. Gallier. *Logic for Computer Science*. Harper & Row, New York, 1986.

[GBGO00] Nancy Griffeth, Ralph Blumenthal, Jean-Charles Gregoire, and Tadashi Ohta. A feature interaction benchmark for the first feature interaction detection contest. *Computer Networks (Amsterdam, Netherlands: 1999)*, 32(4):389–418, April 2000.

[Gen35] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen (1969)*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969 edition, 1935.

[Ghi91] Silvio Ghilardi. Incompleteness results in Kripke semantics. *The Journal of Symbolic Logic*, 56(2):517–538, June 1991.

[GHR94] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic*. Clarendon Press, Oxford, 1994.

[GKWZ03] Dov Gabbay, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. *Many-dimensional modal logics: theory and applications*. Studies in Logic, 148. Elsevier Science, 2003.

[Gol92] Robert Goldblatt. *Logics of Time and Computation*. Center for the Study of Language and Information, Stanford, California, second edition, 1992.

[Gor93] R. Goré. Cut-free sequent and tableau systems for propositional diodorean modal logics. Technical Report UMCS-93-8-3, University of Manchester, Computer Science Department, August 1993.

[Hin62] Jaakko Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, New York, 1962.

[HK99] Alexander Holt and Ewan Klein. A semantically-derived subset of English for hardware verification. In *Proc. 37th Annual Meeting of the Association for Computational Linguistics: Maryland, USA*, pages 451–456. Association for Computational Linguistics, 1999.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hol93] G. J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, April 1993. also in: Proc. 11th PSTV91, INWG/IFIP, Stockholm, Sweden.

[Hol97a] Gerard J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–95, May 1997.

[Hol97b] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[Hry88] T. Hrycej. Temporal prolog. In Yves Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 296–301, Munich, FRG, August 1988. Pitman Publishers.

[Hue75] G. P. Huet. Unification in typed lambda calculus. In G. Goos and J. Hartmanis, editors, λ-*Calculus and Computer Science Theory*, pages 192–212. Springer-Verlag, Berlin, DE, 1975. Lecture Notes in Computer Science 37.

[HV91] Joseph Y. Halpern and Moshe Y. Vardi. Model checking vs. theorem proving: A manifesto. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 325–334, San Mateo, CA, USA, April 1991. Morgan Kaufmann Publishers.

[HWZ00] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.

[HWZ01] Ian Hodkinson, Frank Wolter, and Michael Zakharyaschev. Monodic fragments of first-order temporal logics: 2000–2001 A.D. *Lecture Notes in Computer Science*, 2250:1–23, 2001.

[HWZ02] Ian Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *Logic in Computer Science*, pages 393–402, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[JMN+01] Bengt Jonsson, Tiziana Margaria, Gustaf Naeser, Jan Nyström, and Bernhard Steffen. Incremental requirement specification for evolving systems. *Nordic Journal of Computing*, 8(1):65–87, Spring 2001.

[Kan63] S. Kanger. A simplified proof method for elementary logic. In *Computer Programming And Formal Systems, Studies in Logic*, pages 87–93. North-Holland Publ. Co., Amsterdam, 1963.

[Ker98] Manfred Kerber. Proof planning: A practical approach to mechanized reasoning in mathematics. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction, a Basis for Application – Handbook of the German Focus Programme on Automated Deduction*, chapter III.4, pages 77–95. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[Kri63] Saul A. Kripke. Semantical analysis of modal logic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[MBB+95] Zohar Manna, Nikolaj Bjørner, Anca Browne, Edward Chang, Michael Colón, Luca de Alfaro, Harish Devarajan, Arjun Kapur, Jaejin Lee, Henny Sipma, and Tomás E. Uribe. STeP: The stanford temporal prover. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Soft-*

ware Development, volume 915 of *Lecture Notes in Computer Science*, pages 793–794. Springer-Verlag, 1995.

[Mel96] E. Melis. Progress in proof planning: Planning limit theorems automatically. Technical Report SR-97-08, University of the Saarland, 1996.

[Mil93] Dale Miller. A proposal for modules in lambda-prolog. In *Extensions of Logic Programming*, pages 206–221, 1993.

[Mil98] Dale Miller. λProlog: An introduction to the language and its logic. Unpublished as yet., 1998.

[ML02] Marta Cialdea Mayer and Carla Limongelli. Linear time logic, conditioned models, and planning with incomplete knowledge. *Lecture Notes in Computer Science*, 2381:70–84, 2002.

[MMW94] H. McGuire, Z. Manna, and B. Waldinger. Annotation-based deduction in temporal logic. In Dov M. Gabbay and Hans Jürgen Ohlbach, editors, *Proceedings of the 1st International Conference on Temporal Logic*, volume 827 of *LNAI*, pages 430–444, Berlin, July 1994. Springer.

[Mos98] B. C. Moszkowski. Compositional reasoning using interval temporal logic and tempura. *Lecture Notes in Computer Science*, 1536:439–464, 1998.

[MP81] Z. Manna and A. Pnueli. Verification of temporal programs: the temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, New York, 1981.

[MP91] Z. Manna and A. Pnueli. Completing the temporal picture (logic). *Theoretical Computer Science*, 83(1):97–130, June 1991.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[MS98] Z. Manna and H. B. Sipma. Deductive verification of hybrid systems using STeP. *Lecture Notes in Computer Science*, 1386:305–318, 1998.

[MS99] Z. Manna and H. B. Sipma. Verification of parameterized systems by dynamic induction on diagrams. *Lecture Notes in Computer Science*, 1633:25–43, 1999.

[NM86] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*. Springer Verlag, Oxford, 1986.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logics for Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Clarendon Press, Oxford, England, 1998.

[OM94] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. *Lecture Notes in Computer Science*, 827:445–479, 1994.

[Pli97] R. Pliuškevičius. On the completeness and decidability of a restricted first order linear temporal logic. *Lecture Notes in Computer Science*, 1289:241–254, 1997.

[Pli00] Regimantas Pliuškevičius. On an ω-decidable deductive procedure for non-Horn sequents of a restricted FTL. *Lecture Notes in Computer Science*, 1861:523–537, 2000.

[Pli01] Regimantas Pliuškevičius. Deduction-based decision procedure for a

clausal miniscoped fragment of FTL. *Lecture Notes in Computer Science*, 2083:107–120, 2001.

[Pri67] Arthur N Prior. *Past, Present and Future*. Oxford University Press, Oxford, 1967.

[Ric02] Julian Richardson. A semantics for proof plans with applications to interactive proof planning. *Lecture Notes in Computer Science*, 2514:337–351, 2002.

[RSG98] J. D. C. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with Lambda-Clam. In Claude Kirchner and Hélène Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133, Lindau, Germany, July 1998.

[Sho70] J.R. Shoenfield. *Mathematical logic*. Addison-Wesley, 1970.

[SUM99] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. *Formal Methods in System Design: An International Journal*, 15(1):49–74, July 1999.

[TS96] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1996.

[van84] Johan van Benthem. Correspondence theory. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, volume 165 of *Synthese Library*, chapter II.4, pages 167–247. D. Reidel Publishing Co., Dordrecht, 1984.

[Var03] Moshe Y. Vardi. Automated verification: Graphs, logic, and automata. In *IJCAI-03, Proceedings of the Eighteenth International Joint Confer-*

ence on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003, pages 1603–1606. Morgan Kaufmann, 2003.

[Vig00] Luca Viganò. *Labelled Non-Classical Logics*. Kluwer Academic Publishers, Dordrecht, 2000.

[Wal94] Toby Walsh. Analogical proof planning. In T. Dartnall, editor, *Artificial Intelligence and Creativity*. Kluwer, 1994.

[WZ00a] F. Wolter and M. Zakharyaschev. Decidable fragments of first-order modal logics. *Journal of Symbolic Logic*, 2000.

[WZ00b] Frank Wolter and Michael Zakharyaschev. Spatio-temporal representation and reasoning based on RCC-8. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 3–14, San Francisco, 2000. Morgan Kaufmann.