The Tableau Workbench: a framework for building automated tableau-based theorem provers

Pietro Abate

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

September 2007

© Pietro Abate

Typeset in Times by TEX and LATEX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Pietro Abate
11 September 2007

Abstract

The Tableau WorkBench

Theorem provers have now matured dramatically. On one hand, highly optimized and specific theorem provers like Fact [Horrocks and Patel-Schneider 1998] or MSPASS [Hustadt and Schmidt 2000], can test formulae with hundreds of symbols within a few seconds while provers like the LWB [Heuerding 1996] implement many different logics in a single framework. On the other hand, theorem provers like Isabelle [Paulson 1993] implement generic logical frameworks that are mainly tailored to interactive reasoning. Despite the great effort made to design intuitive user interfaces, researchers that are merely interested in mechanizing their favourite logic often find these tools too complex to modify or to program. For example, the LWB addresses these issues by implementing a large (but fixed) number of logics and by giving access to them through user interfaces that are easy to use and readily available from the web. However, this not viable when experimenting with new logics that have not already been implemented in the LWB by expert programmers.

The Tableaux Work Bench (TWB) is a generic framework for building automated theorem provers for arbitrary logics without learning complex programming languages. The TWB consists of a small core that defines its general architecture, some extra the machinery required to specify tableau-based theorem provers and an abstraction language for expressing new tableau rules. New logic modules are defined via the user interface and then translated and compiled with the proof engine to produce a specialized theorem prover for that logic.

The TWB targets a potentially broad audience: first the TWB can be used by those who are interested in experimenting with new logical calculi but who do not have the technical skills to modify existing theorem provers. Second, the TWB can be used as an aid to teach automated reasoning where the emphasis is on automated deduction methods rather than on programming techniques. Lastly, the TWB can be used by expert users to build custom theorem provers particularly tailored to specific applications or to compare optimisations techniques.

The main design criteria we adopted in the implementation of the TWB are generality, user friendliness and modularity. The TWB core defines only a very generic infrastructure which can be easily adapted to implement different deduction methods. It also offers a high level specification language to make it accessible to non-programmers who can re-use their pen-and-paper specifications with minor modifications to implement their favourite logics. Lastly, the TWB is built using a modular architecture and is composed of a set of loosely coupled libraries that can be individually replaced without compromising the soundness of the system.

Contributions of this Thesis

The main contribution of this thesis is to present a new tool, the `TWB`, designed to aid researchers to quickly experiment with new logics and for more experienced users to design complex decision procedures or to compare different optimisation techniques. Despite the existence of different specialized theorem provers, to the best of our knowledge, there is no other tool with similar characteristics to the `TWB`. We believe that the approach we have taken in designing the `TWB`, where users have the ability to quickly prototype an implementation for their calculus, can be of great help to the scientific community, particularly for researchers with great expertise in logic but little background in automated reasoning.

We also give a one-pass decision procedure for the unified branching time logic UB that extends the work in [Schwendimann 1998] for $PLTL$. The logic UB is the easiest temporal logic with a branching semantics. In the literature, to the best of our knowledge, there are no one-pass algorithms for this logic. Developing such an algorithm has many different advantages from an automated reasoning perspective. First, the algorithm is based on a straightforward depth-first search. This characteristic enables the reasoner to always consider one branch at a time, opening up the possibility of a parallel implementation. Second, by exploring one branch at a time, the amount of memory required, in the average case, will be less than other algorithms based on two passes where the entire result of the search must be kept in memory before starting the second pass. Most importantly, that this approach can easily be extended to other fix-point logics such as CTL, LCK, PDL and possibly, to CTL^* and the modal μ-calculus.

Overview of the dissertation

This dissertation is organized in three parts.

Part I consists of Chapters 1 and 2. In Chapter 1 we give a brief introduction to tableau methods for automated deduction, modal logic and functional and monadic programming. In Chapter 2 we review the major results in the literature about fix-point logics.

Part II consists of Chapters 4 to 8, in which we present the Tableaux Work Bench (`TWB`), a generic framework designed to implement tableau-based automated theorem provers. In Chapter 4 we give our motivations to develop the `TWB` and we outline a number of issues related to the design of generic theorem provers based on the tableau method. In Chapter 5 we give a detailed description of the `TWB` design. In Chapter 6 we present the `TWB` syntax to define tableau provers from a user perspective. In Chapter 7, we give examples of calculi implemented with the `TWB` and experimental results. Finally, in Chapter 8, we present a comparison with other well known theorem provers and we outline possible future developments of the `TWB`.

Part III consists of Chapter 3 in which we propose a single-pass tableau calculus for branching time logic UB. We give a proof sketch of soundness and completeness.

Contents

Abstract **iii**

I Preliminaries **1**

1 Background **3**
1.1 Tableau Methods for Automated Deduction . . . 3
1.2 Classical Propositional Logic . . . 4
1.3 Basic Normal Modal Logics . . . 4
1.4 Functional Programming . . . 6

2 Fix-Point Logics **11**
2.1 Temporal Logics . . . 11
2.1.1 Syntax and Semantics . . . 12
2.1.2 Decision Procedures . . . 15
2.2 Logic of Common Knowledge . . . 16
2.2.1 Syntax and Semantics . . . 17
2.2.2 Decision Procedures . . . 18
2.3 Propositional Dynamic Logic . . . 19
2.3.1 Syntax and Semantics . . . 19
2.3.2 Decision Procedures . . . 20
2.4 The Modal μ-calculus and Beyond . . . 20
2.5 Discussion . . . 21
2.6 Conclusion . . . 21

II Towards a General Decision Procedure for Fix-Point Logics **23**

3 A Single Pass Tableau Procedure for Unified Branching Time Logic **25**
3.1 Syntax and Semantics . . . 25
3.2 A One-pass Tableau Algorithm for UB . . . 30
3.2.1 The Rules . . . 31
3.2.2 The Search Strategy . . . 34
3.2.3 Example . . . 35
3.3 Soundness and Completeness . . . 36
3.4 Two pass decision procedure . . . 36
3.5 Extensions to Other Fix-point Logics . . . 37
3.5.1 Algorithmic Aspects . . . 38

III The Tableau WorkBench 39

4 A General Tableau Prover 41
4.1 Introduction . . . 41
4.2 Motivations and Target Audience . . . 42
4.3 Anatomy of a General Tableau Theorem Prover . . . 43
4.3.1 Tableau Methods . . . 43
4.3.2 Mechanizing Tableau Methods . . . 44
4.3.3 Removing Non-Determinism . . . 47
4.4 Design Criteria . . . 49
4.5 Development Techniques, Tools and Availability . . . 50

5 Overview of the TWB 53
5.1 The Core Algorithm . . . 53
5.2 Core Library Modules . . . 54
5.2.1 The Sequence Module . . . 55
5.2.2 The Node Module . . . 56
5.2.3 The Rule Module . . . 57
5.2.4 The Strategy Module . . . 58
5.2.5 The Visit Module . . . 60
5.3 User Data-Types Library . . . 61
5.4 Tableau Library Modules . . . 62
5.4.1 The Partition Module . . . 63
5.4.2 The UserRule Module . . . 63
5.4.3 The RuleContext Module . . . 63
5.5 User Syntax Library . . . 64
5.6 Summary . . . 64

6 Using the TWB 67
6.1 Defining the Calculus for a Logic . . . 67
6.1.1 Defining Connectives . . . 68
6.1.2 Defining Histories and Variables . . . 69
6.1.3 Defining Rules . . . 70
6.1.4 Defining the Strategy . . . 76
6.1.5 Formula Manipulation . . . 77
6.2 Building and Using a Theorem Prover for a Calculus . . . 79
6.2.1 Building the Prover . . . 80
6.2.2 Executing the Prover . . . 80

7 Case Studies and Experimental Results 83
7.1 Case Studies . . . 83
7.1.1 Basic modal logic S4 . . . 83
7.1.2 Propositional linear temporal logic . . . 87
7.1.3 Optimisations for Tableau-based Theorem Provers . . . 92
7.1.4 Benchmarks . . . 96

7.2 Experimental Results 97
7.2.1 Comparing Optimization Techniques 97
7.2.2 Comparison with the `LWB` 98

8 Related and Future Work **101**
8.1 Related Work 101
8.2 Future Work 103
8.2.1 Short Term Improvements 103
8.2.2 Long Term Improvements 104

Bibliography **119**

Part I

Preliminaries

Chapter 1

Background

In this chapter, in Section 1.1 we give an overview of theorem proving using tableau methods, in Section 1.2 we introduce calculi for classical propositional logic (CPL) and in Section 1.3 we present tableau calculi for the normal modal logics K, KT and $S4$. In Section 1.4 we give an overview of functional programming and monadic programming as it is relevant to understand the intricacies of the `TWB` presented in Chapter 5.

1.1 Tableau Methods for Automated Deduction

The tableau method is a convenient formalism for automated deduction that has many variants and exists for many logics [M D'Agostino and D Gabbay and R Hähnle and J Posegga et al. 1999]. It can be seen as a refutation procedure that decomposes a given set of formulae into a network of sets each representing a possible world in the Kripke semantics for the chosen logic [Kripke 1959]. The tableau method gives us a purely syntactic method of determining whether or not some given formula φ is valid in logic L or to determine whether φ is a (possibly global) logical consequence of a set of formulae Γ.

More specifically, a tableau calculus consists of a finite collection of rules with each rule telling us how to break down one logical connective into its constituent parts. The rules typically are expressed in terms of finite sets of formulae, although there are logics for which we must use more complicated data structures like multi-sets or lists. If there is such a rule for every logical connective then the procedure will eventually return a set which consists only of atomic formulae and their negations (or go into cycles), which cannot be broken down any further. To keep track of this process, the nodes of a tableau itself are set out in the form of a tree and the branches of this tree are created and assessed in a systematic way. Usually, such a systematic method for searching this tree gives rise to an algorithm for performing deduction. By running a tableau procedure for an initial set of formulae, the set is then recognized as satisfiable or unsatisfiable with respect to the semantics of the logic in question. This is not always the case for some temporal logics where checking satisfiability requires a second pass to model-check the initial pseudo model generated by the tableau procedure in the first pass.

A tableau rule with name (ρ) is composed of a numerator and a list of denominators. In the literature, tableau rules are often written as:

$$(\rho)\ \frac{n}{d_1 \dots d_m}$$

where n is the numerator, while $d_1 \dots d_m$ is a list of denominators. Each tableau rule contains one or more distinguished formulae called the *principal formula* and one or more formulae called *side formulae*. In the following we use the letters $A, B, C, P, Q, \cdots$ to denote principal formulae and $X, Y, Z \cdots$ for sets of side formulae. Each rule is usually named using the main connective of the principal formula (not shown in the example). A detailed analysis of tableau methods from an automated reasoning prospective will be carried out in Chapter 4.

1.2 Classical Propositional Logic

Definition 1.2.1. Let AP be a non-empty set of propositional variables and consider the connectives $\wedge$, $\vee$ and $\neg$ and the constants $\top$ (*true*) and $\bot$ (*false*). Then the language of classical propositional logic is defined using the BNF grammar below:

$$\begin{aligned} p &::= p_0 \mid p_1 \mid p_2 \mid \cdots \\ \varphi &::= p \mid \bot \mid \top \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \end{aligned}$$

Definition 1.2.2. A CPL-model M is associated with a valuation function $V : AP \rightarrow \{0,1\}$ that assigns to each atom $p \in AP$ a value in $\{0,1\}$.

Definition 1.2.3. Let M be CPL-model and V its valuation function. Then the satisfaction relation $M \vDash \varphi$ of a formula φ in M is defined as follows:

$M \vDash p$ iff $p \in AP$ and $V(p) = 1$

$M \vDash \neg\varphi$ iff $M \nvDash \varphi$

$M \vDash \varphi \vee \psi$ iff $M \vDash \varphi$ or $M \vDash \psi$

$M \vDash \varphi \wedge \psi$ iff $M \vDash \varphi$ and $M \vDash \psi$

Definition 1.2.4. We say that a formula φ is *satisfiable* if there is a CPL-model M for φ where $M \vDash \varphi$. We say that a formula φ is *valid* (a tautology) if $\neg\varphi$ is not satisfiable.

For example, the formula $p \rightarrow q$ is satisfiable in CPL, while the formula $\varphi \rightarrow \varphi$ is valid in CPL. Given a CPL-formula φ, the negation normal form (nnf) of φ is an a formula φ' where implications are replaced by their definitions, negations are pushed down to atoms using De Morgan's laws and double negations are eliminated. It is well known that every CPL-formula has a logically equivalent formula in nnf.

Figure 1.1 shows the tableau rules for CPL. Given a formula φ in nnf, the tableau procedure for CPL gives us a method of determining whether φ is valid. The tableau procedure tests if the set $\{\neg\varphi\}$ is satisfiable in CPL. If this is **not** the case, then φ must be valid. In the following we assume familiarity with classical propositional logic.

1.3 Basic Normal Modal Logics

Basic normal modal logics are extensions of classical propositional logic. As well as having the usual apparatus of classical propositional logic, modal logics are also equipped with two

$$(\bot)\ \frac{A;\neg A;X}{\bot} \quad (\wedge)\ \frac{A\wedge B;X}{A;B;X} \quad (\vee)\ \frac{A\vee B;X}{A;X \mid B;X}$$

Figure 1.1: Tableau Rules for Propositional Classical Logic

additional operators called the necessity and possibility operators with syntax $\Box$ (box) and $\Diamond$ (diamond), respectively. We define the language of propositional modal logic by extending the definition for propositional classical logic as follows.

Definition 1.3.1. Let AP be a non-empty set of propositional variables and consider the connectives $\wedge$, $\vee$, $\neg$, $\Box$ and $\Diamond$ and the constant $\top$ and $\bot$. Then the language of the basic normal modal logic is defined by the BNF grammar below:

$$\begin{array}{lcl} p & ::= & p_0 \mid p_1 \mid p_2 \mid \cdots \\ \varphi & ::= & p \mid \bot \mid \top \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \end{array}$$

Modal logic semantics are usually given in terms of Kripke models [Hughes and Cresswell 1996]. Intuitively, the idea behind a Kripke model is that in addition to the current "state of affairs" or "world", there are a number of other related possible worlds. Therefore if something is necessarily true in a world w then it must be true in all possible worlds related to w.

Definition 1.3.2. A model is a triple $M = (W,R,V)$ where W is a non-empty set of possible worlds, R is a binary relation over W and $V : AP \rightarrow 2^W$ is an evaluation function that associates to each atomic proposition $p \in AP$ a set of worlds in W where p is true.

If $M = (W,R,V)$ is a model, we say that a world $v \in W$ is reachable (or accessible) from a world $w \in W$ iff wRv. We now extend the notion of satisfaction for propositional logic to define the semantics of the basic normal modal logics.

Definition 1.3.3. Let $M = (W,R,V)$ be a model. Then the satisfaction relation $M,w \vDash \varphi$ for a formula φ and a world $w \in W$ is recursively defined as follows:

$M,w \vDash p$ iff $p \in AP$ and $w \in V(p)$

$M,w \vDash \neg\varphi$ iff $M,w \nvDash \varphi$

$M,w \vDash \varphi \vee \psi$ iff $M,w \vDash \varphi$ or $M,w \vDash \psi$

$M,w \vDash \varphi \wedge \psi$ iff $M,w \vDash \varphi$ and $M,w \vDash \psi$

$M,w \vDash \Box\varphi$ iff $\forall v \in W$ if wRv then $v \vDash \varphi$

$M,w \vDash \Diamond\varphi$ iff $\exists v \in W$ such that wRv and $v \vDash \varphi$.

Definition 1.3.4. A formula φ is *satisfiable* if there exists a model $M = (W,R,V)$ and a world $w \in W$ such that $M,w \vDash \varphi$. A formula φ is *valid* if $\neg\varphi$ is not satisfiable.

$$(K)\ \frac{\Diamond P; \Box X; Z}{P; X} \quad (T)\ \frac{\Box P; X}{P; \Box P; X} \quad (S4)\ \frac{\Diamond P; \Box X; Z}{P; \Box X}$$

Figure 1.2: Tableau Rules for Basic Modal Logics K, T and $S4$

The family of basic modal logics are characterized by certain kinds of models (frames) corresponding to particular restrictions on the reachability relation R. Many of these restrictions are definable in terms of first-order logic (FO) when the binary predicate $R(x,y)$ represents the reachability relation. For example, the normal modal logic K assumes no restriction on the binary relation R. Extensions of K are characterized by restriction on R. For example, the logic KT is characterized by reflexive models while the logic $S4$ is the logic of transitive and reflexive Kripke frames. Reflexive and transitive restrictions on the reachability relations are respectively expressed by the following FO conditions:

Reflexivity : $\forall w : R(w,w)$

Transitivity : $\forall s,t,u : (R(s,t) \wedge R(t,u)) \rightarrow R(s,u)$.

Tableau calculi for basic modal logics K, KT and $S4$ can be obtained by using the rules from the tableau calculus for CPL in Figure 1.1 and the rules in Figure 1.2. Then we have:

tableau calculus for $K = CPL$-rules + (K)-rule

tableau calculus for $KT = CPL$-rules + (K)-rule + (T)-rule

tableau calculus for $S4 = CPL$-rules + $(S4)$-rule + (T)-rule.

In this this work, we assume familiarity with basic modal logics. For an introduction to these notions see the introductory textbooks by Hughes and Cresswell [Hughes and Cresswell 1996] and by Fitting [Fitting 1983]. Of course, the function nnf defined for classical propositional logic can be extended for classical normal modal logics by considering the duality $\Box = \not\Diamond \not$ and the De Morgan's laws.

1.4 Functional Programming

> *Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.*
>
> *comp.lang.functional (faq)*

Hughes, the father of the Haskell programming language, argues that since modularity is the key to successful programming, the functional programming style is vitally important to solve real world problems [Hughes 1989; Wood 2001]. Functional programs are easier to

understand and to maintain, are often shorter than programs written in imperative style [Hudak and Jones 1994] and more easily verifiable using formal methods [Filliatre and Letouzey 2003]. Even if the methodological benefits of functional programming are well known, the vast majority of programs are still written in imperative languages such as C. This contradiction can be explained on the one hand by the historical lack of performance of functional programs and on the other, by the inherent conceptual difficulty of writing functional programs.

In the last 20 years, advances in compilers technology and the development of new programming techniques have considerably narrowed the gap. Even if functional languages like OCaml [Chailloux et al. 2000] are still slower then C/C++ in particular domains, the difference in performance has been considerably reduced in the last decade[1] [Verna 2006; Garret 2000]. Moreover, research devoted to develop efficient data structures for functional languages [Okasaki 1996] and books focusing on functional programming [Paulson 1991] have now taken this style into the mainstream and made it palatable to develop industrial-grade software [Wadler 2006; Hudak and Jones 1994].

The functional programming style achieves modularity via the ability to compose functions in an abstract way, and to have control of the data flow that is made explicit (sometimes, in pure functional languages, painfully explicit). Functional programming languages are divided in two categories: *pure* languages, such as Haskell [Jones et al. 1999] and *impure* languages such as OCaml [Chailloux et al. 2000]. In impure languages, programmers have access to a number of constructs with *side effects* such as exception or assignments. Pure languages on the other hand, are easier to reason about because of the absence of computation with side effects.

In the rest of this work, we assume familiarity with basic concepts from functional programming. For an introduction to functional programming refer to [Paulson 1991].

Monadic Programming

An important programming technique, developed to make pure functional languages more accessible without introducing unwanted features, is the monadic programming style. In fact, "Shall be pure or impure ?" is the motivating question for Wadler's work in monadic programming. The concept of monads arises from category theory and was first applied by Moggi to structure the denotational semantics of programming languages [Moggi 1989]. Wadler took monads a step closer to the real world by giving a number of examples to simulate impure features in a pure functional language [Wadler 1992]. The Haskell compiler, for example, is structured using monads [Jones et al. 1999].

We now give an introduction to monads, following Wadler [Wadler 1993], to highlight how monads can be used to capture "computational effects" such as maintaining a global state of the computation, or to express non-deterministic computations. In general, a monad is a construct to abstract a computation ("a program is a function from values to computations"), to increase modularity and to regain some flexibility peculiar to impure functional languages.

The basic monad is a triple $\mathscr{M} = (M, return, bind)$ consisting of a type constructor M and a pair of polymorphic functions[2], *return* and *bind*. For example, if $\mathscr{M}$ is a monad and we

[1] See `http://shootout.alioth.debian.org/` for a detailed empirical comparison.

[2] Since monads are usually presented using Haskell syntax, the function *bind* is often written as >>= or used as an infix operator as '*bind*'. We write the function bind as a binary function. Moreover we use *return* instead of the

wish to convert a function $f : a \to b$ into its monadic form, we will write $f : a \to b\ M$, thus encapsulating the result of the function into an object with a well defined behaviour.

To describe the basic monads used in the `TWB`, we adopt the OCaml syntax: for an overview see [Chailloux et al. 2000]. The type[3] signature of the basic Monad $M = (M, return, bind)$ is shown below where `return` accepts a value and returns a monad (computation) while `bind` accepts a monad and a function and returns a new monad.

```
type 'a m
val return : 'a -> 'a m
val bind   : 'a m -> ('a -> 'b m) -> 'b m
```

Monads can define a number of computations. The *identity* monad, the simplest monad, is defined as follows:

```
type 'a m    = 'a
let return a = a
let bind m f = f m
```

The identity monad is a monad that does not embody any computational strategy. It simply applies the bound function to its input without any modification, therefore returning a computation that is no different from its input. The identity monad is very seldom used in applications. It only plays a fundamental role in the theory of monad transformers [Liang et al. 1995], where any monad transformer applied to the identity monad yields a non-transformer version of that monad. As a trivial example, we give two functions (below), one in monadic style, the other one by applying a function directly to its arguments.

The first function `fib` returns an integer, the second function `aux_fib` is an auxiliary (monadic version) of the function `fib` where we "open" the recursion. The third function `mfib` is the monadic version of `fib` that accepts an integer and returns a value of type *int m* which in the specific case of the identity monad (*M* in the example) is the same as *int*.

```
let rec fib = function
     |0 -> 1
     |1 -> 1
     |n -> fib (n-1) + fib (n-2)

let aux_fib f = function
     |0 -> M.return 1
     |1 -> M.return 1
     |n -> M.bind (f (n-1)) (fun y ->
              M.bind (f (n-2)) (fun x -> M.return (x + y)))

let rec mfib = aux_fib mfib
```

The *state monad* (below) represents a computation that accepts an initial state and returns a value paired with the final state:

more traditional *unit* because *unit* is a keyword in OCaml.

[3] *'a m* is a type where *'a* is a type variable, and *m* is a type constructor. For example, *'a list* is a generic list type and *int list* is the type for a list of integers. In Haskell, type constructors are of the form *Ma*, where *M* is the type constructor and *a* is a type variable.

```
type state
type 'a m     = state -> ('a * state)
let return a = fun s -> (a,s)
let bind m f = fun s -> let (a,y) = m s in f a y
```

The state monad is often used to maintain the state of the computation in a transparent and modular way and it allows the implementation of "side effects" typical of impure programming languages, such as I/O, without compromising the overall soundness of the program. It is also used to simulate in-place updates in non-imperative data structures without violating referential transparency[4]. For example, in the `TWB`, the state monad is used to maintain the state of the computation of the strategy evaluator (cf. Section 5.2.4) without requiring any global variables.

One of the benefits of using the monadic programming style is that the behaviour of a function can be modified compositionally. For example, we can modify the function `mfib` to compute also the number of recursive calls when evaluated. In the next example, we assume *M* implements the state monad. We first define a function `tick` that executes a function and then increments a counter. The function `tick` accepts a function `f`, a value `n` and value `c` representing the current state; it returns a tuple composed of the result of the evaluation of the function `f` with argument `n` in the current state `c`, and the new state `k+1`. Then, the function `cfib` is the new function that behaves as the function `mfib` but now also counts the number of recursive calls done by the function when evaluated:

```
let tick f n c = let (r,k) = f n c in (r,k+1)
let rec cfib n c = tick (aux_fib cfib) n c
```

Monads can also be used to implement non-deterministic computations, where the evaluation of an expression returns a list of possible answers. Before defining such a monad, we need first to extend the structure of the basic monad with a distinguished zero value *mzero* and a binary operation *mplus* to group monads. The type signature of this new monad (usually called MonadPlus in the Haskell nomenclature) is as follows:

```
type 'a m
val return : 'a -> 'a m
val bind   : 'a m -> ('a -> 'b m) -> 'b m
val mplus  : 'a m -> 'a m -> 'a m
val mzero  : 'a m
```

The simplest monad that encapsulates non-deterministic computation is the *list monad*. Note that since OCaml implements an eager evaluation semantics, using the native OCaml list module will force the computation of all possible answers at once, wasting computational resources[5]. Instead of the default (eager) OCaml list module, we use a lazy sequence *Seq*[6] where values are computed only if needed. Operators of the list monad are definable in terms of basic list operations. The list monad is defined as follows:

```
type 'a m     = 'a list
let return a = Seq.push a Seq.empty
```

[4]Referential transparency is a property of computer programming languages, essentially stating that any expression may be replaced with its result without changing the behaviour of the program.

[5]Haskell, on the other hand is a lazy language by design and this complication is not needed.

[6]See Section 5.2.1 for details about the *Seq* module.

```
let bind m f = Seq.join (Seq.map f m )
let mplus    = Seq.append
let mzero    = Seq.empty
```

List monads are often used to implement backtracking algorithms. However, using the list monad to implement such algorithms has limitations related to efficiency and advanced monad composition techniques. A more efficient backtracking monad has been devised in [Hinze 2000] and in [Kiselyov et al. 2005]. In Section 5.2.4 we describe in detail a new monad that is the result of a non-standard composition of a state monad and a backtracking monad.

Another interesting effect that can be easily coded into the monadic framework is the continuation-passing style technique [Friedman 1988], that has been used, for example, to model sophisticated programming language features. Continuations represent the future of a computation as a function from an intermediate result to the final result. In a nutshell, if we consider the evaluation of an expression $f(e)$, a continuation k of $f(e)$ represents the future of the partial computation of $f(e)$ after the expression e has been evaluated. A continuation k is a function of one argument. In order to evaluate e wrt. k, the continuation k is applied to the result e' of the computation of e. In continuation passing style, computations are built from sequences of nested computations terminated by a final continuation which produces the final result. The basic idea is to add to each function an extra “continuation” argument, and to further transform the body of the function. Therefore instead of returning its value, it passes this value on to its extra continuation argument. The *continuation monad* encapsulates continuations as a monadic value. A continuation monad can be defined as follows:

```
module ContMonad : Monad = struct
    type answer
    type 'a m     = ('a -> answer) -> answer
    let return a = fun k -> k a
    let bind m f = fun k -> m (fun x -> f x k)
end
```

Continuation monads often also provide the “Call/cc” function (Call with current continuation) found in scheme and Standard ML. The operator “Call/cc” can be seen as a `GOTO` in a functional setting, where given a function f and a continuation k, it discards the current continuation of the function f and replaces it with a new continuation k, effectively “jumping” to k. The function that implements such an operation is:

```
let callCC f k = f (fun x _ -> k) k
```

Fix-Point Logics

Modal logics are widely used in many areas in computer science, particularly for the specification and verification of software and hardware, for knowledge representation in databases, and in artificial intelligence. The most important reason for this wide application is the balance modal logics offer between computational complexity and expressivity. For example, the validity problem for basic modal logics like K and $S4$ is PSPACE-complete, while for $S5$, it is NP-complete. However, modern applications of modal languages often require more expressive power to reason about properties of a system such as termination or fairness. Extending basic modal logics by adding features like transitive closure, path quantifiers, and least and greatest fix points substantially increases its expressive power without losing decidability. Temporal logics like $PLTL$ are indeed PSPACE-complete, while other logics such as computational tree logic (CTL), propositional dynamic logic (PDL) and the logic of common knowledge (LCK) are decidable in EXPTIME [Fagin et al. 1995; Emerson and Jutla 2000]. In this chapter we introduce several representative logics with fix-point operators and rooted tree models: namely CTL^*, CTL, $PLTL$, PDL and LCK.

2.1 Temporal Logics

Temporal logic was first introduced by Arthur Prior [Prior 1957] under the name of Tense Logic to provide a formal logical notation for the formulation and resolution of philosophical issues concerning time. In the last five decades, tense logic has been studied in a range of fields, from computer science to linguistics.

In computer science, temporal logic has been mainly used as a means to reason about correctness proprieties of *concurrent* and *reactive* systems[1] such as safety, liveness, total correctness and fairness of deterministic and non-deterministic computations [Emerson 1996; Manna and Pnueli 1983]. In artificial intelligence [Shanahan 1997], temporal logic has been used to address the problems of reasoning about the mutable and immutable properties of the environment over time, and in linguistics, in relation to the analysis and interpretation of tenses in natural languages [Richards et al. 1989]. In computer science, the most popular temporal logics are the propositional linear temporal logic $PLTL$, the computational tree logic CTL and

[1] A reactive system is a program that maintains an ongoing interaction with the environment. Examples include operating systems, network communication protocols, air traffic control systems. Reactive systems can either be implemented on sequential or parallel architectures. Such systems are also called "open" systems.

the full computational tree logic CTL^*.

The nature of the underlying structure of time on which the formalism is based has long been debated [Emerson and Halpern 1983; Emerson and Halpern 1986; Lamport 1981; Vardi 2001; Stirling 1987]. This academic discussion, apart from the philosophical implications concerning determinacy versus free will, has led to the formulation of a uniform framework which allows for the comparison of different formalisms [Emerson and Halpern 1986]. Where linear time is the correct model to use in order to characterize the set of all execution sequences which a program generates, branching time models are used to reason about the set of all execution trees generated by a program. Consequently, the former is useful to reason about sequential deterministic systems, while the latter is adequate to reason about non-deterministic or concurrent systems which are designed to explore all possible choices.

In his "final showdown", Vardi argues in favour of linear time logic for model checking [Vardi 2001]. His position is that, in spite of the success of CTL, it suffers from many fundamental limitations as a specification language for model checking problems.

Lamport noticed that since linear and branching time logics correspond to two distinct views of time, it is not surprising that $PLTL$ and CTL are expressively incomparable [Lamport 1981]. For example, the $PLTL$ formula FGp expressing that "Sometime in the future, p is always true" is not expressible in CTL. Conversely, the CTL formulae $AFAGp$, which express "in all paths, eventually there exists a state s where in all paths from s, p is always true" is not expressible in $PLTL$.

Concerning expressivity of branching time logics, CTL is essentially as good as CTL^* to express invariance proprieties (i.e. partial or total correctness). However CTL is not suitable for expressing fairness proprieties [Goranko 2000]. For example, fairness of a resource allocation in a distributed system can be expressed in CTL^* as:

$$A\ (G\,F(\text{resource_requested}) \rightarrow F\ (\text{resource_granted}))$$

The closest translation of this fairness property in CTL is:

$$AGAF(\text{resource_requested}) \rightarrow AF\ (\text{resource_granted})$$

The logic $ECTL$ [Emerson 1990] is an extension of CTL which incorporates simple fairness constraints. It has also been shown that both CTL and $ECTL$ can be extended to CTL^+ and $ECTL^+$, where boolean combinations of temporal modalities are allowed. CTL^+ has the same expressive power as CTL [Emerson and Halpern 1985], while $ECTL^+$ is strictly more expressive then $ECTL$.

2.1.1 Syntax and Semantics

In this section, we provide the syntax and semantics of $PLTL$, CTL and CTL^*. We first consider the language CTL^* which extends and subsumes CTL and $PLTL$, as well as a number of systems [Emerson and Halpern 1986]. In Chapter 3, we give a single-pass tableau calculus for the logic UB which could potentially serve as a starting point to develop a single-pass tableaux based algorithm for these logics.

2.1.1.1 Syntax

CTL^* is an extension of the language for propositional logic with temporal connectives. In particular we consider countably many propositional variables AP and the connectives $\neg, \wedge, E, \bigcirc, U$ where E is an existential path selector and U is the *until* operator.

With regard to the CTL^* syntax, it is often useful to classify formulae as state formulae or path formulae. The class of state formulae is composed of those that are true or false when evaluated in a state while the class of path formulae are those true or false of paths. State and path formulae are defined as follows:

State Formulae:

S1 each atomic proposition $p \in AP$ is a state formula

S2 if φ, ψ are state formulae then so are $\varphi \wedge \psi$ and $\neg\psi$

S3 if ψ is a path formula then $E\psi$ and $A\psi$ are state formulae

Path Formulae:

P1 each state formula is also a path formula

P2 if φ, ψ are path formulae then so are $\varphi \wedge \psi$ and $\neg\psi$

P3 if φ, ψ are path formulae then so are $\bigcirc\psi$ and $\varphi\ U\ \psi$

Other modalities are expressed by duality. The "before" modality B is defined as the dual of until via $(\varphi\ B\ \psi) := \neg(\neg\varphi\ U\ \psi)$. The "generally" modality and "sometimes" modality are defined, respectively, as $G\varphi := (\perp B\ \varphi)$ and $F\varphi := (\top\ U\ \varphi)$. The universal path selector A is defined as $\neg E \neg$.

The minimal set satisfying the rules S1-3 and P1-3 forms the language of CTL^*. The syntax of the logic CTL is obtained by restricting the syntax to disallow boolean combinations and nesting of linear time operators. For example, $AGF\varphi$ or $E(F\varphi \wedge (\psi\ U\ \phi))$ are CTL^* formulae, but not CTL formulae. Formally, the CTL syntax is obtained by replacing P1-3 with

P0 if ψ, φ are path formulae then so are $\bigcirc\psi$ and $\varphi\ U\ \psi$

Finally, the set of path formulae generated by rules S1 and P1-3 defines the syntax of $PLTL$.

2.1.1.2 Semantics

Different semantics have been proposed to interpret temporal logics: see [Reynolds 2001] for an overview. We adopt the semantics given in [Emerson and Halpern 1986] where temporal logics are viewed as a special kind of modal logic. In this framework, the truth value of a formula can change in different worlds in a Kripke frame and temporal operators allow us to reason about different points in time. It is then natural to evaluate formulae with respect to a state and a path in the transition system, rather than just a world as in traditional modal logics.

We now define the semantics of CTL^* over a Kripke structure. CTL and $PLTL$ formulae are interpreted over the same structure. We first define the notion of a transition frame.

Definition 2.1.1. A transition frame is a pair (S,R) where:

1. S is a non-empty set of states
2. R is a total binary relation over S.

Note. "total" in Definition 2.1.1 means that for every $s \in S$, there is some $t \in S$ such that $s\,R\,t$. In modal logic, the word "seriality" is often used to describe such relations. A transition frame is also called a serial Kripke frame. Note also that the totality condition does not restrict the modelling power of the language. In CTL^* a "terminated execution" of a program is usually seen as an endless loop of its last state.

A (full)path represents the flow of time. Fullpaths are also commonly referred to as histories[2], and states as moments [Zanardo 1992].

Definition 2.1.2. A *fullpath* in (S,R) is an infinite sequence $b = s_0, s_1, s_2 \ldots$ of states such that for each $i \geq 0$, $s_i\, R\, s_{i+1}$. We write b^i to denote the suffix path $s_i, s_{i+1}, s_{i+2} \ldots$.

Definition 2.1.3. A model $M = (S,R,L)$ is a transition system (S,R) and a labelling function $L : S \to 2^{AP}$ which associates with each state s a set $L(s)$ of atomic propositions true at state s.

Formulae in CTL^*, CTL and $PLTL$, are interpreted using the semantics in Definition 2.1.4. When formula φ is true in a fullpath b and a model M, we write $M, b \Vdash \varphi$.

Definition 2.1.4. Let $M = (S,R,L)$ be a model. The satisfaction relation $M, b \Vdash \varphi$ of a path formula φ in a fullpath b (P1-3) and satisfaction relation $M, s \Vdash \varphi$ of a state formula φ in a state $s \in S$ (S1-3) are recursively defined as follows:

S1: $M, s \Vdash p$ with $p \in AP$ iff $p \in L(s)$
S2: $M, s \Vdash \neg\psi$ iff $M, s \nVdash \psi$
$M, s \Vdash \varphi \vee \psi$ iff $M, s \Vdash \varphi$ or $M, s \Vdash \psi$
$M, s \Vdash \varphi \wedge \psi$ iff $M, s \Vdash \varphi$ and $M, s \Vdash \psi$
S3: $M, s \Vdash A\varphi$ iff $\forall b = s, s_1, s_2 \ldots M, b \Vdash \varphi$
$M, s \Vdash E\varphi$ iff $\exists b = s, s_1, s_2 \ldots M, b \Vdash \varphi$
P1: $M, b \Vdash \varphi$ iff $M, s_0 \Vdash \varphi$ where $b = s_0, s_1, s_2 \ldots$
P2: $M, b \Vdash \neg\varphi$ iff $M, b \nVdash \varphi$
$M, b \Vdash \varphi \vee \psi$ iff $M, b \Vdash \varphi$ or $M, b \Vdash \psi$
$M, b \Vdash \varphi \wedge \psi$ iff $M, b \Vdash \varphi$ and $M, b \Vdash \psi$
P3: $M, b \Vdash (\varphi\ U\ \psi)$ iff $\exists i\,.\,[M, b^i \Vdash \psi$ and $\forall j\,.\,(j < i$ implies $M, b^j \Vdash \varphi)]$
$M, b \Vdash \bigcirc\varphi$ iff $M, b^1 \Vdash \varphi$

Definition 2.1.5. A state formula φ (resp. a path formula) is *valid* iff for every model $M = (S,R,L)$ and every state s (resp. path b) in M, we have $M, s \Vdash \varphi$ (resp. $M, b \Vdash \varphi$).

Definition 2.1.6. A state formula φ (resp. a path formula) is *satisfiable* iff there is a model $M = (S,R,L)$ and some state s (resp. path b) in M, such that $M, s \Vdash \varphi$ (resp. $M, b \Vdash \varphi$).

Note. $PLTL$ formulae in this setting should be regarded as "pure path formulae" and therefore interpreted only according to rules P1-3 in Definition 2.1.4. $PLTL$ models are often presented just as Kripke models over a *linear* flow of time commonly represented by $(\mathbb{N}, \succ, L)$ where $\mathbb{N}$ is the set of natural numbers.

[2]Not to be confused with the "histories" introduced in Section 4.3.1.

2.1.2 Decision Procedures

In this section, we present a brief overview of the principal results concerning decision procedures for the satisfiability problem in temporal logics. The development of proof methods for temporal logics, in particular for *PLTL* and *CTL*, have followed three main approaches: tableaux, automata and resolution. Tableaux-based approaches attempt to systematically construct a structure from which a model can be extracted. Translations to Büchi automata reduce the problem of determining whether the language recognized by the automaton obtained by translating a formula is non-empty. Resolution based approaches involve the translation of a temporal formula into a normal form and the application of inference rules of the *temporal resolution calculus* [Fisher 1991; Dixon 1996].

2.1.2.1 *PLTL*

In the literature there are different tableau calculi for *PLTL*. Wolper gives a two-pass algorithm that first builds a (cyclic) graph using a tableau algorithm and then prunes "PLTL-inconsistent" nodes in a second pass via a connected component analysis [Wolper 1985].

The first pass applies the tableau rules to construct a finite rooted cyclic graph. The second pass prunes nodes that are unsatisfiable because they contain contradictions like $\{p, \neg p\}$, and also remove nodes which give rise to "bad loops". Bad loops are defined as loops in the model where one or more eventualities are not satisfied. The second phase usually leads to an analysis of the strongly connected component of the graph. Typical description of this second phase can be found in [Wolper 1985]. The main practical disadvantage of such two-pass methods is that the cyclic graph built in the first pass has a size which is *always* exponential in the size of the initial formula. So the very act of building this graph immediately causes EXPTIME behaviour even in the average case (the validity problem for *PLTL* is in PSPACE). For a large formula, this may pose a substantial space cost because, in the worst case scenario, the number of nodes in the graph will be 2^{4l}, where l is the length of the initial formula [Wolper 1985]. The requirement also removes the possibility of parallelisation, an optimisation that has been sometimes used for automated tableau provers. Janssen gives an efficient version of Wolper's algorithm based on binary decision diagrams (BDDs) [Janssen 1989]. Another more recent and refined algorithm to check *PLTL* satisfiability using a two phase procedure is in [Lichtenstein and Pnueli 2000]. A tableau system for *PLTL* based on the idea of using linear constraints over integers is in [Schmitt and Goubault-Larrecq 1997].

Schwendimann gives a single pass tableau calculus for *PLTL* using histories and "synthesized" histories which are variables that are instantiated at the leaves and used to pass information toward the root [Schwendimann 1998]. This calculus explores the search space in a depth first fashion, and therefore, it reclaims space upon backtracking. However, it is usually slower than the two-pass method since different branches can repeat the same (redundant) work. As far as we know, this is the only direct single pass algorithm for *PLTL*.

Fisher introduced resolution rule inferences for *PLTL* in [Fisher 1991]. The first step to use temporal resolution is to transform the *PLTL* formula into *separated normal form* (SNF) by replacing non-atomic sub-formulae with new propositions and recursively removing all occurrences of the until operator. The second step uses standard resolution techniques using special inference rules to reason about temporal connectives. See [Fisher et al. 2001] for an overview

of resolution methods for temporal logics. Recently Hustadt and Konev implemented Fisher's algorithm [Hustadt and Konev 2003]. An interesting benchmark comparison of various theorem provers based on different decision procedure for *PLTL* is given in [Hustadt and Schmidt 2002].

2.1.2.2 *CTL*

A tableau-based decision procedure for satisfiability-checking in *CTL* is well known [Emerson and Halpern 1985; Ben-Ari et al. 1981]. This method is also based on a two pass algorithm, where a graph is built by a tableau procedure and then a second pass prunes "*CTL*-inconsistent" nodes by an iterative marking algorithm. Reynolds gives a tableau-based decision procedure for *ECTL* [Reynolds 2005].

Wolper, Vardi and Sistla proposed satisfiability checking algorithms based on automata [Wolper et al. 1983]. The basic idea is that, to show that a *CTL* formula φ is valid, we first build a Büchi automata for $\neg\varphi$. Then, if the language accepted by this automaton is empty then the formula φ is valid. Otherwise, since the algorithms checking emptiness of automata are constructive, if the language is not empty, then the algorithm produces a model for $\neg\varphi$. In the literature there are also approaches based on games where a formula is satisfiable if there exists a winning strategy for one player [Lange 2002].

An extension of the resolution method for *PLTL* to *CTL* is in [Bolotov and Fisher 1999].

2.1.2.3 CTL^*

For temporal logics with richer modalities such as CTL^*, the tableau construction is not directly applicable. Methods based on Tree Automata and Büchi Automata can be found respectively in [Vardi and Wolper 1988] and [Sistla et al. 1987].

2.2 Logic of Common Knowledge

The logic of knowledge was first investigated by early Greek philosophers and in the early 1960's was first formalized by Hintikka [Hintikka 1962]. The 1960's also saw a flourishing interest in the area of research interested in capturing the inherent proprieties of knowledge. More recently, researchers from different fields such as distributed systems, artificial intelligence, linguistic and economics have become interested in reasoning about knowledge, in particular considering the common knowledge operator (see [Fagin et al. 1995] for references).

The logic of common knowledge (*LCK*) is a multi-modal logic which can be used to talk about certain epistemic situations among a group of agents. In particular, the notion of common knowledge - where everyone knows, everyone knows that everyone knows, etc. - provides means to make universal statements about the knowledge of a group ("any fool knows"). See [Fagin et al. 1995] for an extensive overview.

LCK is based on multi-modal extensions of basic logics like $S5$ or $KD4$, depending on different philosophical assumptions, and a common knowledge operator, often written as C. The logic has been axiomatized in [Halpern and Moses 1985] and in [van der Hoek and Meyer 1997]. Let A be a group of n agents and $[a]$ be the knowledge operator for "agent a knows",

of each agent $a \in A$. We define the operator E ("everybody knows") as the conjunction of the individual knowledge of each agent. Then $E\varphi$ is an abbreviation for $[a_1]\varphi \wedge [a_2]\varphi \wedge \cdots \wedge [a_n]\varphi$, with $\{a_1, \cdots, a_n\} = A$.

There are two equivalent interpretations of the common knowledge operator. In the *iterative interpretation*, $C\varphi$ is seen as the infinite conjunction $E\varphi \wedge EE\varphi \wedge \cdots$. Note that if the number of agents is one then $E\varphi = \Box\varphi$. Alternatively, in the *fix-point interpretation*, $C\phi$ is defined as a solution of the fix-point equation $X = E\varphi \wedge EX$. Intuitively, this says that common knowledge of φ holds in a situation X where everyone in the group knows that φ holds and that they are in situation X. In fact, $C\varphi$ is the *greatest* solution of this fix-point equation and these two interpretations are equivalent [Fagin et al. 1995].

As we shall see in Section 2.2.2, there are different calculi for *LCK* that are cut-free, but that contain an infinitary rule. To the best of our knowledge, there are no cut-free calculi for *LCK* that are sound, complete and free of an infinitary rule. We will focus on the fix-point characterization of the common knowledge operator.

2.2.1 Syntax and Semantics

2.2.1.1 Syntax

We consider countably many propositional variables AP, a finite non-empty set of agents A and the connectives $\neg, \wedge, [a]$ with $a \in A$ with $AP \cap A = \emptyset$ and the constants $\bot$ and $\top$. The formulae of *LCK* are then defined using the BNF grammar below:

$$\begin{array}{rcl} p & ::= & p_0 \mid p_1 \mid p_2 \mid \cdots \\ \varphi & ::= & p \mid \top \mid \bot \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \\ & & \mid [a]\varphi \mid \langle a\rangle\varphi \\ & & \mid [C]\varphi \mid \langle C\rangle\varphi \\ & & \mid [E]\varphi \mid \langle E\rangle\varphi \end{array}$$

The abbreviations for $\langle a\rangle$ and $\vee$ are as usual. We define $[E]$ and $\langle E\rangle$ as:

$$[E]\varphi = \bigwedge_{a\in A} [a]\varphi \quad \langle E\rangle\varphi = \bigvee_{a\in A} \langle a\rangle\varphi$$

Letting $[E]^0\varphi = \varphi$, let $[E]^k\varphi$ be $[E][E]^{k-1}\varphi$ for some $k \geq 1$. Finally, we define the common knowledge operator C as $[C]$ and its dual as $\langle C\rangle$.

2.2.1.2 Semantics

In this section, we present a version of the logic of common knowledge based on the transitive closure of the knowledge of individual agents following [Fagin et al. 1995]. van der Hoek and Meyer give an axiomatization of *LCK* considering the reflexive transitive closure of knowledge of individual agents [van der Hoek and Meyer 1997].

Definition 2.2.1. Given a finite non-empty set A of n agents and a non-empty set AP of atoms we define a K_n model as a tuple (W, R_A, V) where W is a non empty set of worlds, R_A is a

function that assigns to each agent $a \in A$ a binary relation $R_a \subseteq W \text{x} W$ and V is a valuation that assigns to each atom $p \in P$ a set $V(p) \subseteq \mathscr{P}(W)$ where p is "true", and $\mathscr{P}(W)$ is the set of all subsets of W.

Definition 2.2.2. If $M = (W, R_A, V)$ is a K_N-model and $w \in W$ then the satisfaction relation $M, w \Vdash \varphi$ of a formula φ in a world w is recursively defined as follows:

$M, w \Vdash p$	iff	$w \in V(p)$
$M, w \Vdash \varphi \vee \psi$	iff	$M, w \Vdash \varphi$ or $M, w \Vdash \psi$
$M, w \Vdash \varphi \wedge \psi$	iff	$M, w \Vdash \varphi$ and $M, w \Vdash \psi$
$M, w \Vdash \langle a \rangle \varphi$	iff	$\exists v \in W, wR_a v$ and $M, v \Vdash \varphi$
$M, w \Vdash [a]\varphi$	iff	$\forall v \in W, wR_a v$ implies $M, v \Vdash \varphi$
$M, w \Vdash \langle C \rangle \varphi$	iff	$M, w \Vdash \langle E \rangle^k \varphi$ for some $k = 1, 2, \cdots$
$M, w \Vdash [C]\varphi$	iff	$M, w \Vdash [E]^k \varphi$ for all $k = 1, 2, \cdots$

We say that a formula φ is satisfiable if there exists a model M and a world $w \in W$ such that $M, w \vDash \varphi$. A formula φ is valid if $\neg\varphi$ is not satisfiable.

As pointed out in [Fagin et al. 1995], this view of common knowledge has an interesting graph-theoretical interpretation. A state t is reachable from a state s in k steps ($k \geq 1$) if there exist states $s_0, s_1, \ldots s_k$ such that $s_0 = s$ and $s_k = t$ and for all j with $0 \leq j \leq k-1$ there exists an agent $a \in A$ such that $(s_j, s_{j+1}) \in R_a$. Then t is reachable from s exactly if there is a path from s to t in the graph. With this view, it is also easy to see a connection with other fix-point logics like CTL where path operators allow to explicitly reason about these paths in the graph.

2.2.2 Decision Procedures

In the literature, there are different decision procedures for *LCK* mainly based on the iterative interpretation of the common knowledge operator.

Jäeger and Alberucci present both a finitary Tait-style calculus using a restricted form of cut, and an infinitary cut-free calculus [Alberucci and Jäeger 2002]. On the one hand, the finitary calculus uses a form of cut that, albeit restricted to a subset of the Fisher-Ladner closure of the initial formula, makes this calculus computationally intractable. On the other hand, the infinitary calculus always takes the worst complexity case. Alberucci also gives an embedding of *LCK* into the μ-calculus [Alberucci 2002]. Kretz and Studer present a decision procedure based on deduction chains [M. Kretz 2006]. van Ditmarsch and Dyckhoff present a sequent calculus for *LCK* that is, however, not complete [van Ditmarsch and Dyckhoff 2002]. Kaneko presents an embedding of *LCK* in the game logic that gives a cut-free sequent calculus, that again, uses the iterative interpretation of the common knowledge [Kaneko 1999] and hence require an infinitary rule.

The satisfiability problem for *LCK* is strictly related to the satisfiability problem of the propositional dynamic logic *PDL* (presented in Section 2.3). Halpern and Moses derive an exponential lower bound for the satisfiability problem for *LCK* by using an embedding of *LCK* into *PDL* and proving an exponential upper bound by using a filtration argument similar to the one given by Pratt [Fagin et al. 1995; Pratt 1979a].

De Giacomo and Massacci present a labelled tableau calculus for *PDL* with converse and they claim that the calculus is sound, complete, and cut-free, if converse is not considered

[De Giacomo and Massacci 1996]. By embedding *LCK* into *PDL*, this gives a labelled cut-free tableau calculus for *LCK*. However, the soundness of this calculus has been challenged by demonstrating a counter-example [Schmidt and Tishkovsky 2003]. But this approach is however rather "overkill" since the structure of *LCK* is simpler than *PDL*: the common knowledge operator cannot be nested to build up more complex operations on relations as in *PDL*.

As pointed out in [Alberucci and Jäeger 2002] and [van Ditmarsch and Dyckhoff 2002], because of the inherent computational complexity of *LCK*, an efficient decision procedure and therefore an efficient automatic theorem prover for *LCK* is unlikely to be developed. Nonetheless, the by empirical experience with provers for temporal logics and modal logics suggest that it may be feasible to develop efficient decision procedures for *LCK* for many cases of interest. All the proposed calculi based on the iterative interpretation of the common knowledge operator have high complexity even in the average case. To the best of our knowledge there is no sound, complete, cut-free and non-infinitary calculus for *LCK*.

2.3 Propositional Dynamic Logic

Propositional Dynamic Logic (PDL), is a poly-modal logic introduced to model the evolution of the computation process by describing the properties of states reached by programs during their execution. *PDL* was introduced in [Harel and Kozen 1982; Fischer and Ladner 1979].

In *PDL* there are as many modalities as programs. For example, if B is a program which is obtained by combining atomic programs, then $\langle B\rangle\varphi$ expresses that there is an execution of B leading to a final state that satisfies φ.

PDL enjoys nice algorithmic proprieties: it is finitely axiomatizable [Harel and Kozen 1982], its model-checking problem is $PTIME$-complete [Harel 1984], its satisfiability problem is $EXPTIME$-complete [Harel 1984]. However it is strictly less expressive than the alternation free fragment of the modal μ-calculus [Harel and Kozen 1982].

Extensions of the basic *PDL*, include *CPDL*, considering a converse operator [Fischer and Ladner 1979], or $PDL-\Delta$, where programs can be repeated infinitely often [Streett 1981]. These two extensions are both strictly more expressive than basic *PDL*.

2.3.1 Syntax and Semantics

2.3.1.1 Syntax

Let A be a non-empty set of atomic programs and AP a set of propositional variables with $A \cap AP = \emptyset$. The language of PDL is constructed as follows, where $p \in AP$ and $a \in A$:

$$\begin{aligned} a &::= a_0 \mid a_1 \mid a_2 \mid \cdots \\ \pi &::= a \mid \pi_1 \cup \pi_2 \mid \pi_1;\pi_2 \mid \pi^* \mid ?\varphi \end{aligned}$$

$$\begin{aligned} p &::= p_0 \mid p_1 \mid p_2 \mid \cdots \\ \varphi &::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle\pi\rangle\varphi \end{aligned}$$

2.3.1.2 Semantics

The semantics for *PDL* is defined in the usual way through Kripke structures. Let $M = (W, R, L)$ be a Kripke structure where W is a non-empty set of worlds, R assigns to each program $a \in A$ a binary relation $R(a)$ on W and L assigns to each atomic variable $p \in AP$ the set of worlds $L(p)$ in which p is true. The extension of R to complex programs and the definition of the consequence relation $\models$ of *PDL* are defined by simultaneous induction as:

$$
\begin{array}{lll}
R(\pi_1 \cup \pi_2) & = & R(\pi_1) \cup R(\pi_2) \\
R(\pi_1; \pi_2) & = & R(\pi_1) \circ R(\pi_2) \\
R(\pi^*) & = & \text{the reflexive transitive closure of } R(\pi) \\
R(?\varphi) & = & \{(w,w) \in W^2 \mid M, w \models \varphi\}
\end{array}
$$

$$
\begin{array}{lll}
M, w \models p & \text{iff} & w \in L(p) \text{ for } p \in AP \\
M, w \models \neg\varphi & \text{iff} & M, w \not\models p \\
M, w \models \varphi_1 \wedge \varphi_2 & \text{iff} & M, w \models \varphi_1 \text{ and } M, w \models \varphi_2 \\
M, w \models \varphi_1 \vee \varphi_2 & \text{iff} & M, w \models \varphi_1 \text{ or } M, w \models \varphi_2 \\
M, w \models \langle\alpha\rangle\varphi & \text{iff} & \exists v \in W \text{ such that } (w,v) \in R(\alpha) \text{ and } M, v \models \varphi
\end{array}
$$

Let φ be a *PDL* formula and $M = (W, R, L)$ be a Kripe model. Then φ is said to be satisfiable if there is $w \in W$ with $M, w \models \varphi$. A formula φ is said to be valid if $\neg\varphi$ is not satisfiable.

2.3.2 Decision Procedures

PDL has been studied for almost 30 years. The first decision procedure in the literature is due to Fisher and Ladner where the authors give a two pass algorithm in which the first phase builds a pseudo model in the form of a graph, while the second phase model checks this graph to obtain a model [Fischer and Ladner 1979]. Pratt gives a more refined decision procedure in [Pratt 1979b]. De Giacomo and Massacci give a single pass algorithm based on Pratt's ideas for *CPDL* [De Giacomo and Massacci 1996]. Schmidt[3] proposed an implementation of this tableau calculus but appear to be unsound. The authors of `Lotrec` claim a decision procedure for *PDL* and give a implementation of it with their theorem prover [Gasquet et al. 2005].

2.4 The Modal μ-calculus and Beyond

The modal μ-calculus is an extension of modal logic with least and greatest fix-point constructors. It was first introduced by Scott and De Bakker in the late 1960's and axiomatized by Kozen in the mid 1980's [Kozen and Parikh 1983]. It subsumes dynamic and temporal logics like *PDL*, *PLTL*, *CTL*, and *CTL**. The modal μ-calculus provides us with the capability of expressing and reasoning about assertions concerning temporal proprieties of dynamic systems with potentially infinite behaviour. The semantics of the modal μ-calculus is usually based on the concept of transition systems. The modal μ-calculus falls outside the scope of this work. For a detailed survey refer to [Stirling 1988].

[3] See `http://www.cs.man.ac.uk/~schmidt/pdl-tableau/`

2.5 Discussion

Vardi emphasizes that an essential property of many interesting modal logics is the *tree model property*: that is, if a formula is satisfiable, then it is satisfiable in the root of a tree structure [Vardi 1996]. Conversely, Andreka, van Benthem and Nemeti, put forward the thesis that the *guarded fragment of first order logic* generalizes and explains most of the good properties of modal logic [Andréka et al. 1998].

On the one hand, Vardi notices that the majority of classic modal logics can be embedded in the fragment $FO2$ of first-order logic with two variables. This fragment still has the finite model property, but not the finite tree property. $FO2$ is decidable, but not as "robustly decidable" as normal modal logic. In fact, the resulting logic obtained by extending $FO2$ with fix-point operators is highly undecidable [Grädel and Otto 1998]. Therefore Vardi's conclusion is that the finite tree property is specifically the reason to explain the good algorithmic property of modal logics.

On the other hand, Andreka, van Benthem and Nemeti, suggested that the real reason for the good properties of modal logic is not the tree model property, but the fact that their first order translation can be embedded in the guarded fragment *GF* of first order logic. The guarded fragment of first order logic is of course still decidable, has the finite model property and also enjoys a generalization of the tree model property [Grädel 2003].

Recently van Benthem has shifted his position emphasising the model tree property as essential for the analysis of dynamic epistemic logics as a species of epistemic temporal logic [**?**].

In particular, in relation to fix-point logics, Grädel and Walukiewicz extended *GF* with fix-point operators [Grädel and Walukiewicz 1999]. This new logic μGF is still decidable (it is $2EXPTIME$-complete), enjoys the finite model property and moreover enjoys a generalized variant of the tree model property. The μ-calculus, as well as all the other logics mentioned in this work, can be embedded in μGF. Grädel gives a decision procedure for μGF based on parity games in [Grädel 1999].

2.6 Conclusion

In this chapter we presented different fix-point logics. Many similar decision procedures have been presented in the literature for this class of logics. The two most common approaches based on tableau-sequent methods are to either give decision procedures based on a two pass algorithm or to use a calculus with a non-finitary rule and to rely on the finite model property enjoyed by these logics for termination. Both approaches are computationally expensive as the entire model (exponential in size) must always be built to obtain an answer, even in the average case. A third avenue that we do not directly consider is to use automata. This indeed gives an optimal complexity decision procedure for checking satisfiability [Emerson and Jutla 2000] but at the expense of renouncing the simplicity of tableau-sequent methods. We know of no implementation for automata-based decision procedures for these logics. In this work we focus on decision procedures based on tableau/sequent methods.

By using a two pass decision procedure, the first pass must build the entire model, to be subsequently model-checked. Conversely, by using an infinitary calculus, the upper bound that

is computed based on a theoretical considerations, must always be considered making these methods computationally expensive. Sequent calculi with cut are equally computationally intractable even when analytic cut (often based on an extension of the Fisher-Ladner closure) is considered.

To fill this gap, we advocate a general decision procedure based on a variation of the tableau method that directly exploits the fix-point characterization of the logic considered: it is cut-free and it is space efficient. In Chapter 3, we present a one pass decision procedure for the unified branching time logic UB by using a method that extends the work in [Schwendimann 1998] for $PLTL$. As future work, we outline how it may be possible to extend this method to all the logics presented in this chapter and possibly to the μ-calculus.

The main insight behind our approach is that, by decorating tableau nodes with *additional* data structures to keep track of information related to already visited branches, we can detect isolated sub-trees and therefore discard parts of the (pseudo) model much earlier than with a two-pass procedure. Moreover by using a depth first visit, only one branch at a time is kept in memory at any time (albeit of exponential size in the worst case), which opens the possibility of parallelizing the decision procedure by exploring different branches independently.

Poor runtime performance, typical of tableaux decision procedures, can be alleviated by using a simple form of syntactic caching where identical nodes appearing in different branches of the tableau are visited only once. In this context, by “syntactic equality” of two nodes, we mean not only that the two nodes have the same characterizing set of formulae, but that they also have the same “additional” information gathered from exploring other parts of the tableau. Other more aggressive forms of caching can easily lead to unsoundness.

Part II

Towards a General Decision Procedure for Fix-Point Logics

A Single Pass Tableau Procedure for Unified Branching Time Logic

In this chapter, we focus on a single pass decision procedure for the unified branching logic UB by extending the work in [Schwendimann 1998] and adapting the two pass algorithm presented in [Ben-Ari et al. 1981; Emerson and Halpern 1985]. UB can be seen as the next step up in complexity from $PLTL$ by considering branching rather than linear, time. Thus it offers a simple platform to develop a general single pass algorithm for branching fix-point logics like CTL, PDL and LCK.

In the reminder of this chapter we first give the syntax and semantics of UB, in Section 3.2, we present a single pass tableau algorithm for UB using histories and variables. The implementation is given in Appendix 8.2.2.3. We conjecture that the calculus is sound and complete. The details of the proofs are still being worked out, but we are confident of the results.

Note. This chapter is based on work done in collaboration with Rajeev Goré.

3.1 Syntax and Semantics

We present the syntax and semantics of the logic UB as a subset of those of the logic CTL presented in Section 2.1.

Syntax

First we summarize the syntax for the logic CTL, then we give the syntax of the logic UB by defining the operators AF and EF in terms of the CTL until operator.

Definition 3.1.1. CTL Syntax. We consider a countable set AP of propositional variables and the connectives $\neg, \wedge, A, \bigcirc, U$. The set of CTL-formulae is the smallest set defined using the BNF grammar below:

$$
\begin{aligned}
p &::= p_1 \mid p_2 \mid \cdots \\
\varphi &::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bot \mid \top \\
&\quad \mid A\varphi \mid \varphi_1\ U\ \varphi_2 \mid \bigcirc\varphi.
\end{aligned}
$$

Notation. We can also define all abbreviations $\vee$, $\rightarrow$, $\leftrightarrow$, F, G in the usual way and we have $E\varphi = \neg A \neg \varphi$ (cf. Section 2.1). We use $\varphi \equiv \psi$ to mean that φ is logically equivalent to ψ.

From now on, we restrict our attention to the logic UB which is a subset of CTL. In particular we consider only the set of UB formulae defined using the *BNF* grammar below:

$$
\begin{aligned}
p &::= p_1 \mid p_2 \mid \cdots \\
\varphi &::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \bot \mid \top \\
&\quad \mid AG\varphi \mid AF\varphi \mid AX\varphi \\
&\quad \mid EG\varphi \mid EF\varphi \mid EX\varphi
\end{aligned}
$$

where

$AF\varphi := A(\top U \varphi) \equiv \varphi \vee (\neg\varphi \wedge AXAF\varphi)$

$EF\varphi := E(\top U \varphi) \equiv \varphi \vee (\neg\varphi \wedge EXEF\varphi)$

$EX\varphi := \neg \bigcirc \neg\varphi$

$AX\varphi := \bigcirc\varphi$

$AG\varphi \equiv \neg EF \neg\varphi$

$EG\varphi \equiv \neg AF \neg\varphi$

Note. We could take the view that, $EF \equiv \neg AG \neg$, $AF \equiv \neg EG \neg$ and $EX \equiv \neg AX \neg$, and give a more terse syntax in terms of just the primitive operators AG, EG, AX, $\wedge$ and $\neg$. However, the present approach makes the notation of the next section and the formulation of the tableau calculus easier.

Modal connectives are categorized as follows :

Normal eventualities: AX and EX

Compound eventualities: AG and EG

Iterated eventualities: EF and AF.

Definition 3.1.2. We say that a formula is in *negation normal form* if it is implication free and all negations appear only in front of atoms. In the remainder we assume all formulae are in negation normal form.

Definition 3.1.3. Formulae in negation normal form are classified as below:

α-formulae			β-formulae		
α	α_1	α_2	β	β_1	β_2
$AG\varphi$	φ	$AXAG\varphi$	$AF\varphi$	φ	$AXAF\varphi$
$EG\varphi$	φ	$EXEG\varphi$	$EF\varphi$	φ	$EXEF\varphi$
$\varphi \wedge \psi$	φ	ψ	$\varphi \vee \psi$	φ	ψ

Formulae $EX\varphi$ and $AX\varphi$ are neither α- nor β-rules as the can have an arbitrary number of denominators. Formulae $EF\varphi$ and $AF\varphi$ are called iterated eventualities because of their fix-point nature. Formulae $EX\varphi$ and $AX\varphi$ are called base eventualities since they demand that "eventually φ is true". We sometimes write $EV\varphi$ for iterated eventualities.

Semantics

Definition 3.1.4. A UB-model is a triple (S,R,L) where

1. S is a non-empty set of states
2. $R \subseteq S \times S$ is a total binary relation over S
3. $L : AP \to 2^S$ is an assignment of propositional variables to sets of states.

Note. "total" in definition 3.1.4 means that for every $s \in S$, there is some $t \in S$ such that sRt. In modal logic, the word "seriality" is often used to describe such relations.

Definition 3.1.5. A *fullpath* in (S,R,L) is an infinite sequence $s_0, s_1, s_2 \dots$ of states such that for each $i \geq 0$, $s_i \, R \, s_{i+1}$.

Notation. For the fullpath $b = s_0, s_1, s_2 \dots$, and any $i \geq 0$ we write:

b_i for the state s_i

$s \in b$ if for some $i \geq 0$ we have $s = b_i$

Definition 3.1.6. Let $M = (S,R,L)$ be a UB-model. The truth value $M,s \Vdash \varphi$ of a formula φ in a state $s \in S$ is recursively defined as follows:

1. $M,s \Vdash p$ iff $p \in L(s)$
2. $M,s \Vdash \neg\varphi$ iff $M,s \nVdash \varphi$
3. $M,s \Vdash \varphi_1 \vee \varphi_2$ iff $M,s \Vdash \varphi_1$ or $M,s \Vdash \varphi_2$
4. $M,s \Vdash \varphi_1 \wedge \varphi_2$ iff $M,s \Vdash \varphi_1$ and $M,s \Vdash \varphi_2$
5. $M,s \Vdash EX\varphi$ iff $\exists t \in S$ such that sRt and $M,t \Vdash \varphi$
6. $M,s \Vdash AX\varphi$ iff $\forall t \in S$, if sRt then $M,t \Vdash \varphi$
7. $M,s \Vdash EF\varphi$ iff $\exists$ fullpaths b with $b_0 = s$, $\exists t \in b$ such that $M,t \Vdash \varphi$
8. $M,s \Vdash AF\varphi$ iff $\forall$ a fullpath b with $b_0 = s$ and $\exists t \in b$ such that $M,t \Vdash \varphi$

9. $M, s \Vdash EG\varphi$ iff $\exists$ a fullpath b with $b_0 = s$ and $\forall t \in b$, $M, t \Vdash \varphi$

10. $M, s \Vdash AG\varphi$ iff $\forall$ fullpaths b if $b_0 = s$ then $\forall t \in b$, $M, t \Vdash \varphi$

Definition 3.1.7. A *structure* is a triple (S, R, L) where S is a non-empty set of states, R is a binary relation on S and L is an evaluation function that maps states to set of formulae. A *Pre-Hintikka* structure is a structure such that for all $s \in S$ the following properties hold:

Propositional Consistency :

PC 1: If a variable $p \in L(s)$, then its negation $\neg p \notin L(s)$

PC 2: If $\alpha \in L(s)$, then $\alpha_1 \in L(s)$ and $\alpha_2 \in L(s)$

PC 3: If $\beta \in L(s)$, then $\beta_1 \in L(s)$ or $\beta_2 \in L(s)$.

Local Modal Consistency :

MC 1: If $AX\varphi \in L(s)$, then $\forall t \in S$ if sRt then $\varphi \in L(t)$

MC 2: If $EX\varphi \in L(s)$, then $\exists t \in S$ such that sRt and $\varphi \in L(t)$.

Definition 3.1.8. A *Hintikka* structure is a Pre-Hintikka structure (S, R, L) with the two following properties for all $s \in S$:

EF: If $EF\varphi \in L(s)$ then $\exists$ a fullpath b with $b_0 = s$ and $\exists t \in b$ with $\varphi \in L(t)$,

AF: If $AF\varphi \in L(s)$ then $\forall$ fullpaths b if $b_0 = s$ then $\exists t \in b$ with $\varphi \in L(t)$.

Definition 3.1.9. Let φ be a formula and (S, R, L) be a Hintikka structure; we say it is a *Hintikka* structure for φ iff for some $s \in S$, $\varphi \in L(s)$.

Theorem 3.1.1. *A UB formula φ is satisfiable if there exists a Hintikka structure for φ [Ben-Ari et al. 1981].*

Note. The following two definitions enforce a fulfilment condition for EF-eventualities which is stronger than Definition 3.1.8. A fullpath b fulfills an eventuality $EF\varphi$ if there exists an $i \geq 0$ such that $\varphi \in L(b_i)$ **and** all states from b_0 to b_i have $EF\varphi$ in their label. That is, b is a witnessing EF-path. Intuitively, fullpath b does not fulfill the eventuality $EF\varphi$ if φ has no witness on b, or every witness for φ on b is preceded by a point on b whose label does not contain $EF\varphi$ meaning that b "steps off" and stops being an EF-path before reaching the witness for $EF\varphi$. For AF-formulae, b can never "step off", so the following definition for AF-formulae is no stronger than Definition 3.1.8.

Definition 3.1.10. Let (S, R, L) be a pre-Hintikka structure. Let nfl be a function of type $(formula, fullpath) \rightarrow boolean$ defined as follows.

$$nfl(\phi, b) = \begin{cases} true & \text{if } \phi = EV\varphi \text{ and } \forall n \geq 0\ [\varphi \in L(b_n) \Rightarrow \exists j\ .\ 0 \leq j \leq n, EV\varphi \notin L(b_j)] \\ false & \text{otherwise} \end{cases}$$

Definition 3.1.11. Let (S,R,L) be a pre-Hintikka structure. The set $open(S,R,L)$ of "open eventualities" is defined as:

$$\begin{array}{rcl} open(S,R,L) & = & \{AF\varphi \mid \exists s \in S \,.\, AF\varphi \in L(s) \\ & & \quad \& \ \exists \text{ a fullpath } b \text{ with } b_0 = s \text{ and } nfl(AF\varphi, b) = true\} \\ & \cup & \{EF\varphi \mid \exists s \in S \,.\, EF\varphi \in L(s) \\ & & \quad \& \ \forall \text{ fullpaths } b \text{ with } b_0 = s \text{ , } nfl(EF\varphi, b) = true \\ & & \quad \& \ \exists \text{ a fullpath } b \text{ with } b_0 = s \\ & & \quad \& \ \forall i \geq 0 \,.\, EF\varphi \in L(b_i) \ \& \ \varphi \notin L(b_i)\} \end{array}$$

The 2nd and 3rd conjuncts for $EF\varphi$ ensure that not every fullpath from S can "step off". That is, there must be at least one "procrastinator" that **never** witnesses φ.

Definition 3.1.12. Let (S,R,L) be a pre-Hintikka structure. A procrastinator is a path b in (S,R,L) such that there exists $s \in b$ and a formula $\varphi \in L(s)$ such that $\varphi \in open(S,R,L)$.

Theorem 3.1.2. *Let (S,R,L) be a pre-Hintikka structure. If $open(S,R,L) = \emptyset$ then (S,R,L) is a Hintikka structure.*

Proof. For a contradiction, assume a pre-Hintikka structure (S,R,L) with $open(S,R,L) = \emptyset$ is not a Hintikka structure. Since it is a pre-Hintikka structure, it must fail either conditions **EF** or **AF** in Definition 3.1.8.

Then there exists some $s \in S$ such that:

NotAF: $AF\varphi \in L(s)$ and $\exists$ a fullpath b with $b_0 = s$ such that $\forall t \in b \,.\, \varphi \notin L(t)$; OR

NotEF: $EF\varphi \in L(s)$ and $\forall$ fullpaths b with $b_0 = s$ and $\forall t \in b$, $\varphi \notin L(t)$.

Case NotAF. Since (S,R,L) is a pre-Hintikka structure, $AF\varphi \in L(s)$ and $c_0 = s$ and $\forall t \in c \,.\, \varphi \notin L(t)$ together imply that $nfl(AF\varphi, c) = true$ since the antecedent in Definition 3.1.10 always fails. Therefore, $AF\varphi \in open(S,R,L)$ contradicting $open(S,R,L) = \emptyset$.

Case NotEF. Since (S,R,L) is a pre-Hintikka structure, $EF\varphi \in L(s)$ and $\varphi \notin L(s)$ implies $EXEF\varphi \in L(s)$. Hence there exists a fullpath b^1 with $b^1_0 = s$ and $EF\varphi \in L(b^1_1)$. By **NotEF** we must have $\varphi \notin L(b^1_1)$. Hence $EXEF\varphi \in L(b^1_1)$. By repeating this reasoning we can construct a "procrastinator" fullpath $\pi = b_0, b^1_1, b^2_2, \cdots$ such that $\varphi \notin L(b^i_i)$ for all $i \geq 1$ and $\varphi \notin L(b_0)$. Hence, $nfl(EF\varphi, \pi) = true$ meaning that $EF\varphi \in open(S,R,L)$ and contradicting $open(S,R,L) = \emptyset$.

Thus for any $EV\varphi \in open(S,R,L)$ there is at least one "procrastinator". Conversely, if we have a pre-Hintikka structure in which there is no "procrastinator" for $EV\varphi$, then $EV\varphi$ must be fulfilled. The tableau calculus presented in the next section just tracks "procrastinators" up towards the root of the tableau. If no "procrastinator" survives the trip, then the pre-Hintikka structure built from the tableau is a Hintikka structure, and hence a model for the formulae in the root.

3.2 A One-pass Tableau Algorithm for UB

A tableau is a systematic search for a model. The tableau will be constructed as a tree of nodes with each node labelled with a set of formulae that is derived from the original formulae in the root. The structure built by the tableau procedure is associated with a pre-Hintikka structure. If the set of unfulfilled eventualities computed by the procedure at the root node is empty, this structure gives a UB-model since the associated pre-Hintikka structure is actually a Hintikka structure.

The main difference with traditional calculi for non-classical logics is that our decision procedure stores additional information with each node. Thus, each tableau node consists of a set of formulae, a set of histories and a set of variables. A history is a mechanism for collecting extra information during proof search and passing it from parents to children. Histories are therefore modified from numerator to denominators in some rules. A variable is a mechanism to propagate information from children to parents. Variables are modified from denominators to the numerator during backtracking.

More formally:

Definition 3.2.1. A node is a triple $\Gamma :: H :: V$ where Γ is a set of formulae, H is a set of histories, and V a set of variables that are synthesized.

Definition 3.2.2. A tableau for a set Γ is a tree of nodes with root $\Gamma :: H :: V$ where the children of a node are obtained by an application of a rule to the node: we say that a rule is associated with the parent node and conversely, a node is associated with the numerator of a rule. A node is terminal if it is associated with an application of a terminal rule. We say that the tableau is *expanded* if each leaf node is obtained by an application of a terminal rule. A node is saturated if no static rule is applicable to it.

Definition 3.2.3. Every node of an expanded tableau is a *pre-state*. A literal is an atom or a negated atom. A *state* of an expanded tableau is a node of the form $EX\Gamma; AX\Delta; \Lambda$ where Λ contains only literals. If the set $s = EX\Gamma; AX\Delta; \Lambda$ is a state, then the set $\{\varphi\} \cup \Delta$ with $\varphi \in \Gamma$ is a *core* associated with the $EX\varphi$-successor of state s. Also, since each branch can be "open" or "closed" or "loop", the expressions "not open" and "closed" are *not* synonymous. Similarly "not closed" and "open" are also *not* synonymous.

Note. Branching rules are applied depth-first and left-to-right with respect to the rule definition: that is, the left branch is always explored first and if it is not Open, the right branch is explored. Transitional rules are applied depth-first and left-to-right with respect to the rule definition: that is, the left branch is always explored first and if it is not Closed, the right branch is explored. Linear rules are treated as re-writing rules.

Definition 3.2.4. A node in a UB-tableau is a triple $\Gamma :: H :: V$ where:

Γ is a set of formulae as usual,

H consists of two histories called Fev and Br where

Fev is a set of (eventuality) formulae fulfilled between two consecutive applications of (EX)-rules,

Br is a list of pairs, of the form $(Core, Fev)$ where the first component of the pair is a set representing the core that led to a state, and the second component is a set of the iterated eventualities that were fulfilled by this branch in creating that state out of the $Core$,

V consists of one variable called uev where uev is a list of pairs (φ, n) where φ is an AF or EF formula and n is an integer. This variable tracks "procrastinators"

UB-tableau rules are characterized as follows:

Terminal Rules: (id), $(block)$

Linear Rules: $(\wedge)$, (AG), (EG), (D)

Universal Branching Rules: (EF), (AF), $(\vee)$

Existential Branching Rules: (EX)

Static Rules: $(\wedge)$, $(\vee)$, (AG), (EG), (EF), (AF)

Transitional Rule: (EX).

Notation. We use $Br[j]$ to denote the jth element $(Core_j, Fev_j)$ of the list Br. We use, respectively, $Br[j].core$ and $Br[j].fev$ for the first component $Core_j$ and the second component Fev_j of the pair $Br[j]$. We use the symbol "." to append one element to the front of a list. We use Λ to stand for a set containing only literals and/or constants. To focus on the "important" parts of the rule, we use "..." for the "un-important" parts which remain unchanged in passing from node to node e.g. $(\Gamma :: \cdots :: \cdots)$. In the following we use the symbol "=" to assign values to histories or temporary information holders and ":=" to assign values to variables.

3.2.1 The Rules

Linear Rules.

$$(\wedge)\ \frac{\varphi \wedge \psi\ ;\ \Gamma :: \cdots}{\varphi\ ;\ \psi\ ;\ \Gamma :: \cdots} \qquad (D)\ \frac{\Gamma :: \cdots}{EX\top\ ;\ \Gamma :: \cdots}\ EX\varphi \notin \Gamma$$

$$(AG)\ \frac{AG\varphi\ ;\ \Gamma :: \cdots}{\varphi\ ;\ AXAG\varphi\ ;\ \Gamma :: \cdots} \qquad (EG)\ \frac{EG\varphi\ ;\ \Gamma :: \cdots}{\varphi\ ;\ EXEG\varphi\ ;\ \Gamma :: \cdots}$$

The $(\wedge)$ rule is standard and (D) enforces seriality. The (AG) and (EG) rules capture the axioms $AG\varphi \leftrightarrow \varphi \wedge AXAG\varphi$ and $EG\varphi \leftrightarrow \varphi \wedge EXEG\varphi$ respectively.

Universal Branching Rules.

$$(EF)\ \frac{EF\varphi\ ;\ \Gamma :: Fev, Br :: uev}{\varphi\ ;\ \Gamma :: \{EF\varphi\} \cup Fev, Br :: uev_1 \mid EXEF\varphi\ ;\ \Gamma :: Fev, Br :: uev_2}$$

$$(AF)\ \frac{AF\varphi\ ;\ \Gamma :: Fev, Br :: uev}{\varphi\ ;\ \Gamma :: \{AF\varphi\} \cup Fev, Br :: uev_1 \mid AXAF\varphi\ ;\ \Gamma :: Fev, Br :: uev_2}$$

$$(\vee)\ \frac{\varphi \vee \psi \,;\, \Gamma :: \cdots :: uev}{\varphi \,;\, \Gamma :: \cdots :: uev_1 \mid \psi \,;\, \Gamma :: \cdots :: uev_2}$$

where in the (EF), (AF) and $(\vee)$ rules :

$$\begin{array}{rcl} f(x,y) & = & \{i \mid (x,i) \in y\} \\ UEF & = & \{(EF\psi,n) \mid f(EF\psi,uev_1) \neq \emptyset \ \&\ f(EF\psi,uev_2) \neq \emptyset \\ & & \quad \&\ n = min\ (f(EF\psi,uev_1)\ \cup\ f(EF\psi,uev_2))\ \} \\ UAF & = & \{(AF\psi,n) \mid f(AF\psi,uev_1) \neq \emptyset \ \&\ f(AF\psi,uev_2) \neq \emptyset \\ & & \quad \&\ n = max\ (f(AF\psi,uev_1)\ \cup\ f(AF\psi,uev_2))\ \} \\ uev & := & \begin{cases} uev_1 & \text{if } uev_2 = \{(\texttt{false},_)\} \\ uev_2 & \text{if } uev_1 = \{(\texttt{false},_)\} \\ UEF \cup UAF & \text{otherwise} \end{cases} \end{array}$$

The (EF) and (AF) rules capture the fix-point nature of the EF and AF modalities respectively: that is, $EF\varphi \leftrightarrow \varphi \vee EXEF\varphi$ and $AF\varphi \leftrightarrow \varphi \vee AXAF\varphi$. The (EF) rule adds $EF\varphi$ to Fev in the left denominator indicating that this denominator fulfils the eventuality $EF\varphi$. The rule (AF) behaves in a similar way. The $(\vee)$ rule is standard except for the computation of the uev.

The criteria for computing uev are as follows:

- if either branch is "closed" then keep the other uev. If both are "closed" then the $uev = uev_1 = \{(\texttt{false},_)\}$.
- if either uev_1 or uev_2 is empty because it has no "procrastinators", then the numerator is empty because $UEF = UAF = \emptyset$, hence $UEF \cup UAF = \emptyset$.
- if both are not "closed" and both contain "procrastinators", then $UEF \cup UAF$ constructs their "intersection" (sic!).

The crucial point is to simply ignore the notion of "open" and "fulfilled".

Existential Branching Rule. The following rule (EX) is a traditional modal (K) rule written in an unusual way: the first n branches continue to explore the proof tree for all eventualities that meet the side condition $\ddagger(i)$ since the core $\{\varphi_i\} \cup \Delta$ of the ith child **does not** yet appear in Br. The $(n+1)^{st}$ branch contains all eventualities that are blocked: they fail this side condition because their cores **do** appear in Br.

The only applicable rule to the $(n+1)^{st}$ denominator is the $(block)$-rule. Each denominator, $1 \leq i \leq n$, updates Br by adding $(\{\varphi_i\} \cup \Delta, Fev)$ to the head of Br so that descendant (state) nodes that would recreate this core by applying the (EX)-rule fail the side-condition. Each denominator, $1 \leq i \leq n$, also empties its Fev. Since the first n denominators are children of the (EX) node, emptying Fev "resets" this history to now track the iterated eventualities which are fulfilled between the new children and the next (EX) node. The $(n+1)^{st}$ denominator keeps Fev since it is required by the $(block)$-rule. By the strategy (described later) and the rule (D), the set $\{EX\varphi_1, \ldots, EX\varphi_n\} \cup EX\Gamma \neq \emptyset$, but if $\{EX\varphi_1, \cdots EX\varphi_n\} = \emptyset$, then the (EX)-rule is not applicable.

$$(EX)\ \frac{\begin{array}{c} EX\varphi_1\ ;\ldots;EX\varphi_n\ ;\ EX\Gamma\ ;\ AX\Delta\ ;\ \Lambda \\ ::\ Fev, Br\ ::\ uev \end{array}}{\begin{array}{ccccc} \varphi_1\ ;\ \Delta & & \varphi_n\ ;\ \Delta & & EX\Gamma\ ;\ AX\Delta\ ;\ \Lambda \\ ::\ \emptyset, Br_1\ ::\ uev_1 & \|\cdots\| & ::\ \emptyset, Br_n\ ::\ uev_n & \| & ::\ Fev, Br\ ::\ uev_{n+1} \\ \ddagger(1) & & \ddagger(n) & & \dagger \end{array}}$$

with :

$n \geq 1$
$\ddagger(i)$ is the condition $\forall j\ .\ j \leq len(Br) \Rightarrow \{\varphi_i\} \cup \Delta \neq Br[j].core$
$\dagger$ is the (blocking) condition $\forall \psi \in \Gamma\ .\ \exists j \geq 0$ such that $\{\psi\} \cup \Delta = Br[j].core$

and where for $1 \leq i \leq n$:

$Br_i = (\{\varphi_i\} \cup \Delta, Fev).Br$

$l = len(Br) - 1$

$UEV = \bigcup_{j=1}^{n+1} uev_j$

$$uev := \begin{cases} UEV & \text{if } (\texttt{false}, _) \notin UEV \text{ and } \forall(_, n) \in UEV\ .\ n \leq l \\ \{(\texttt{false}, 0)\} & \text{otherwise} \end{cases}$$

We set *uev* to $\{(\texttt{false}, l)\}$ meaning that this branch is closed, either because some child of the (EX) rule is closed, that is $\{(\texttt{false}, _)\} \in UEV$ or because there exists a child that contains an iterated eventuality that loops **strictly** lower (ie. $\exists(_, n) \in UEV.n > l$). In the latter case, this means that there exists an iterated eventuality in that child which cannot be fulfilled in this tableau by following a path starting at that child. These are aspects of the completeness proof to come.

Note. The (EX) rule is an unusual rule compared with classic tableau systems for modal logic with loops (e.g. $S4$). The reason for this difference is the fix-point nature of the logic UB. In $S4$ all loops are "good", meaning that a loop in the tableau always corresponds to a loop in the model and therefore the search procedure does not need to perform any action when a loop is detected. Conversely, in UB, because of the presence of "procrastinators", the search procedure needs to modify the value of the variable *uev* to track the eventualities that have "procrastinators" in the current model. The calculus caters for this important difference by applying the $(block)$ rule to all eventualities that have been blocked and synthesizing the value of the variable *uev* in the $(block)$ rule (below).

Terminal Rules.

$$(id)\ \frac{EX\Gamma\ ;\ AX\Delta\ ;\ \Lambda\ ::\ Fev, Br\ ::\ uev}{Stop}\ \{\neg p; p\} \subseteq \Lambda$$

where $uev := \{(\texttt{false}, len(Br))\}$

$$(block)\ \frac{EX\Gamma\ ;\ AX\Delta\ ;\ \Lambda\ ::\ Fev, Br\ ::\ uev}{Stop}\ \{\neg p; p\} \not\subseteq \Lambda \text{ and } \dagger$$

where † is the condition $\forall \psi \in \Gamma \,.\, \exists j \geq 0 \,.\, \{\psi\} \cup \Delta = Br[j].core$, and

$$
\begin{array}{rcl}
Cores & = & \{\{EF\varphi\} \cup \Delta \mid EF\varphi \in \Gamma\} \\
UAF & = & \{\ (AF\varphi, i) \mid \exists\, c \in Cores, \exists\, i \text{ such that } c = Br[i].core \\
 & & \quad \&\, \forall j \,.\, i \leq j \leq len(Br) \Rightarrow AF\varphi \in Br[j].core \;\&\; AF\varphi \notin Br[j].fev\ \} \\
UEF & = & \{\ (EF\varphi, i) \mid \exists\, i \text{ such that } \{EF\varphi\} \cup \Delta = Br[i].core \\
 & & \quad \&\, \forall j \,.\, i \leq j \leq len(Br) \Rightarrow EF\varphi \in Br[j].core \;\&\; EF\varphi \notin Br[j].fev\ \} \\
uev & := & \{(\psi, n) \in UAF \cup UEF \mid \psi \notin Fev\}
\end{array}
$$

Note. The $(block)$-rule and the (id)-rule are mutually exclusive since their side-conditions cannot be jointly true. The $(block)$-rule and the (EX)-rule are also mutually exclusive for the same reason.

The $(block)$ rule is applicable only if no other rules are applicable and checks whether there are blocked eventualities in a fully saturated node. Thus, for each eventuality, there exists an element in Br with index i that corresponds to the core that would be generated from the current node by creating an (EX)-successor $\{\varphi\} \cup \Delta$ for the set $\{EX\varphi\} \cup AX\Delta$. The $(block)$ rule sets *uev* to be the set of all blocked iterated eventualities for which the branch from the target of their loop to the current leaf is an unfulfilling "procrastinator". Then an iterated eventuality $EV\psi$ has an entry in the set *uev* if the following three conditions are simultaneously true:

- The node $\{EV\varphi\} \cup \Delta$ is equal to $Br[i].core$ for some index i.
- $EV\varphi \in Br[j].core$ for all j , $i \leq j \leq len(Br)$
- $EV\varphi \notin Br[j].fev$ for all j , $i \leq j \leq len(Br)$.

The first condition is the blocking condition, the second condition makes sure that the branch we are considering is a "procrastinator" for the iterated eventuality $EV\varphi$: that is, $EV\varphi$ is present in every core from the loop point to the leaf. The third condition makes sure that $EV\varphi$ is not fulfilled on the branch from the loop point to the leaf. If the set $EX\Gamma \cup AX\Delta$ contains no iterated eventuality then *uev* is the empty set. More importantly, if every blocked eventualities is fulfilled between the loop point and the leaf (without "stepping off") the set UEV is also empty since there are no "procrastinators".

Remark. In traditional tableau, "Open" usually means "satisfiable" because the decision procedure ensures that if no more rules are applicable, then all eventualities are fulfilled and every node is consistent. However, because of the fix-point nature of UB, in our tableau calculus, "Open" does not necessarily mean UB-Satisfiable. Nodes are marked "Closed" by setting their *uev* equal to $\{(false, len(Br))\}$ using information that is synthesized from the leaves on backtracking.

3.2.2 The Search Strategy

The tableau calculus we present uses a standard decision procedure (or strategy) where all static rules are applied first to saturate the node and then the transitional rule (EX) is applied once. When no other rules are applicable, the $(block)$ rule checks whether we stop because:

(a) we have found a "procrastinator" loop where a node repeats, but there are eventualities on this branch that cannot be fulfilled even in an infinite number of iterations of this loop;

(b) we have found a "good" loop where a node repeats and all eventualities are fulfilled in the branch;

Note that since UB enjoys seriality, there are no dead ends, that is every leaf of the tableau is either an instance of the (id)- or $(block)$- rule.

Proposition 3.2.1 (Termination)**.** *A* UB*-tableau for a node* $\varphi :: Fev, Br :: uev$ *always terminates.*

Proof. The tableau can contain only formulae from the Fisher-Ladner closure $Fl(\varphi)$ which is finite. Since no rule creates a formula outside $Fl(\varphi)$, there are only a finite number of different nodes possible. In particular there are only a finite number of different possible cores. Consider any branch. The systematic procedure works by breaking up formulae with all static rules and then applying the (EX) rule. Then we have two possibilities: (1) we apply the (id) or the $(False)$ rule and stop, (2) we apply the $(block)$-rule and loop because the same core reappeared. Since this holds for every branch and there are no infinite branching rules, the procedure must terminate.

3.2.3 Example

We show a fully expanded tableau for the tautology $AF\phi \rightarrow AF\phi$. Note that the tableau procedure first negates and then transforms the formula in nnf. Therefore the procedure will attempt to build a tableau for the formula $EG\neg\phi;AF\phi$.

$$\dfrac{\begin{array}{c} EG\neg\phi;AF\phi \\ :: Fev=\emptyset, Br=[\,] \\ :: uev=\{(\texttt{false},0)\} \end{array}}{\begin{array}{c} \neg\phi;EXEG\neg\phi;AF\phi \\ :: Fev=\emptyset, \\ :: uev=\{(\texttt{false},0)\} \end{array}}\ EG$$

$$\dfrac{\begin{array}{c} \neg\phi;EXEG\neg\phi;AF\phi \\ :: Fev=\emptyset, \\ :: uev=\{(\texttt{false},0)\} \end{array}}{\dfrac{\begin{array}{c} \neg\phi;\phi;EXEG\neg\phi \\ :: Fev=AF\phi, Br=[\,] \\ :: uev=\{(\texttt{false},0)\} \end{array}}{Stop}\ Id \qquad \begin{array}{c} \neg\phi;AXAF\phi;EXEG\neg\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},0)\} \end{array}}\ AF$$

$$\dfrac{\begin{array}{c} \neg\phi;AXAF\phi;EXEG\neg\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},0)\} \end{array}}{\begin{array}{c} EG\neg\phi;AF\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},0)\} \end{array}}\ EX$$

$$\dfrac{\begin{array}{c} EG\neg\phi;AF\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},0)\} \end{array}}{\begin{array}{c} \neg\phi;EXEG\neg\phi;AF\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},0)\} \end{array}}\ EG$$

$$\dfrac{\begin{array}{c} \neg\phi;EXEG\neg\phi;AF\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},0)\} \end{array}}{\dfrac{\begin{array}{c} \neg\phi;\phi;EXEG\neg\phi \\ :: Fev=AF\phi, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(\texttt{false},1)\} \end{array}}{Stop}\ Id \qquad \dfrac{\begin{array}{c} \neg\phi;AXAF\phi;EXEG\neg\phi \\ :: Fev=\emptyset, Br=[(EG\neg\phi;AF\phi)] \\ :: uev=\{(AF\phi,0)\} \end{array}}{Stop}\ Loop}\ AF$$

The procedure first applies all invertible rules and then the non-invertible rule EX. The application of the axiom *Id* sets the variable *uev* equal to $\{(\texttt{false}, 1)\}$. The loop rule conversely assigns to the variable *uev* the set $\{(AF\phi, 0)\}$. The variable *uev* is then modified according to the side function of the *AF* rule (see page 3.2.1) and propagated toward the root. The tableau is closed because the set *uev* is not empty.

3.3 Soundness and Completeness

The tableau calculus presented in this chapter has been implemented with the TWB and tested with over 10000 randomly generated theorems (see Appendix 8.2.2.3). This empirical evidence make us confident of its correctness. Nonetheless the full details of the proofs are rather involved and complex. We conjecture that the logic is sound and complete, but we have not completed the work. The proof is a significant generalization of the ideas in [Schwendimann 1998; Ben-Ari et al. 1981; Emerson and Halpern 1985] and can be potentially extended further to other calculi for fix point logics such as LCK, PDL, etc.

Conjecture 3.3.1. If $\mathscr{T}$ is an expanded tableau for φ with root $\varphi :: Fev, Br :: uev$, with $uev = \emptyset$ then φ is satisfiable.

Conjecture 3.3.2. If a formula φ is UB-satisfiable then there exists an expanded tableau φ with root $\varphi :: \cdots :: uev$ where $uev = \emptyset$.

3.4 Two pass decision procedure

A decision procedure for deciding satisfiability of temporal logic UB and CTL is presented in [Ben-Ari et al. 1981; Emerson and Halpern 1985]. It is based on a two pass algorithm that first built a pseudo-model and then, in a second pass, prunes it of all classically and modally inconsistent paths. This method has been applied to different temporal and epistemic logic and it is similar to Pratt's method for deciding satisfiability of *PDL*.

In more details, the first pass aims to build a pre-Hintikka structure by using a tableau-like procedure. The size of this structure is always $\leq 2^{|p|}$ where p is the initial formula and the time required to build this structure is exponential in the size of the initial formula.

The second pass considers the pre-Hintikka structure and applies a bottom-up procedure to prune all inconsistent nodes. First all nodes containing classical inconsistencies are removed from the pre-Hintikka structure. Then the procedure focus on paths - sequences of nodes in the pre-hintikka structure - that contains eventualities that are not fulfilled. By using a marking algorithm, nodes as part of a path, are examined from the leaves to the root and are removed if they contain an eventuality that is not fulfilled in the current path.

If at the end of the second pass, the resulting structure is not empty (because all nodes were removed) then the structure is a model for the initial formula.

3.5 Extensions to Other Fix-point Logics

The calculus presented in Section 3.2 potentially lays the foundation to give calculi for other fix-point logics such as CTL, LCK, PDL. In the following we detail the extension to CTL. The extensions for LCK and PDL are similar.

CTL. The extension to CTL is straightforward, as the only difference from UB is the until operator (here we omit the rule for the operator before dual of until). All other rules are similar, except for (AF), (EF) and $(block)$ rules which are modified similarly as follows:

$$(EU)\ \frac{E(\varphi U\psi),\Gamma :: Fev; Br :: uev}{\psi,\Gamma :: EF\varphi.Fev; Br :: uev_1 \mid \varphi, EXE(\varphi U\psi),\Gamma :: Fev; Br :: uev_2}$$

$$(AU)\ \frac{A(\varphi U\psi),\Gamma :: Fev; Br :: uev}{\psi,\Gamma :: AF\varphi.Fev; Br :: uev_1 \mid \varphi, AXA(\varphi U\psi),\Gamma :: Fev; Br :: uev_2}$$

The new conditions to compute the value of the variable uev as follows:

$$\begin{array}{rcl}
f(x,y) & = & \{i \mid (x,i) \in y\} \\
UEF & = & \{(E(\varphi U\psi),n) \mid f(E(\varphi U\psi),uev_1) \neq \emptyset \ \&\ f(E(\varphi U\psi),uev_2) \neq \emptyset \\
 & & \quad \&\ n = min\ (f(E(\varphi U\psi),uev_1)\ \cup\ f(E(\varphi U\psi),uev_2))\ \} \\
UAF & = & \{(A(\varphi U\psi),n) \mid f(A(\varphi U\psi),uev_1) \neq \emptyset \ \&\ f(A(\varphi U\psi),uev_2) \neq \emptyset \\
 & & \quad \&\ n = max\ (f(A(\varphi U\psi),uev_1)\ \cup\ f(A(\varphi U\psi),uev_2))\ \} \\
uev & := & \begin{cases} uev_1 & \text{if } uev_2 = \{(\texttt{false},_)\} \\ uev_2 & \text{if } uev_1 = \{(\texttt{false},_)\} \\ UEF \cup UAF & \text{otherwise} \end{cases}
\end{array}$$

$$(block)\ \frac{EX\Gamma\,;\,AX\Delta\,;\,\Lambda :: Fev, Br :: uev}{Stop}\ \{\neg p; p\} \not\subseteq \Lambda$$

The new block rule conditions are:

$$\begin{array}{rcl}
Cores & = & \{\{E(\varphi U\psi)\} \cup \Delta \mid E(\varphi U\psi) \in \Gamma\} \\
UAF & = & \{\ (A(\varphi U\psi),i) \mid \exists\, c \in Cores, \exists\, i \text{ such that } c = Br[i].core \\
 & & \quad \&\ \forall j\ .\ i \leq j \leq len(Br) \Rightarrow A(\varphi U\psi) \in Br[j].core \\
 & & \quad \&\ A(\varphi U\psi) \notin Br[j].fev\ \} \\
UEF & = & \{\ (E(\varphi U\psi),i) \mid \exists\, i \text{ such that } \{E(\varphi U\psi)\} \cup \Delta = Br[i].core \\
 & & \quad \&\ \forall j\ .\ i \leq j \leq len(Br) \Rightarrow E(\varphi U\psi) \in Br[j].core \\
 & & \quad \&\ E(\varphi U\psi) \notin Br[j].fev\ \} \\
uev & := & \{(\psi,n) \in UAF \cup UEF \mid \psi \notin Fev\}
\end{array}$$

LCK. Since LCK is a multi-modal logic, adapting our technique requires to add one non-invertible (K)-like rule for each agent $\langle a\rangle$ and adding an expansion rule for the common knowledge operator $\langle C\rangle$ (cf. Section 2.2) as:

$$\langle a \rangle \ \frac{\langle a \rangle \varphi; [a]\Delta; \Lambda}{\varphi; \Delta}$$

$$(\langle C \rangle) \ \frac{\langle C \rangle \varphi, \Gamma :: Fev; Br :: uev}{\langle E \rangle \varphi :: \langle C \rangle \varphi . Fev; Br :: uev_1 \mid \langle E \rangle \langle C \rangle \varphi, \Gamma :: Fev; Br :: uev_2}$$

The rule for $\langle E \rangle$ and $[E]$ are just re-writing rules capturing their definitions as abbreviations in Section 2.2.

PDL Extending this method to *PDL* will be equally easy where the $\star$ operator will be handled as follows where π is an arbitrary program:

$$(\pi^*) \ \frac{\langle \pi^* \rangle \varphi, \Gamma :: Fev; Br :: uev}{\langle \pi \rangle \varphi :: \langle \pi^* \rangle \varphi . Fev; Br :: uev_1 \mid \langle \pi \rangle \langle \pi^* \rangle \varphi, \Gamma :: Fev; Br :: uev_2}$$

We are not sure whether extensions to more complex logic such as CTL^* or the μ-calculus are possible.

3.5.1 Algorithmic Aspects

The decision procedure based on the tableau calculus for UB described in the previous section is obviously a depth first search of the proof tree that is step-by-step generated by the applications of the tableau rules, with a backtracking strategy required to synthesize the variable component in each node.

Schwendimann shows that is possible to use the back-jumping optimization in his calculus for $PLTL$. It should also be possible to adapt simplification to these logics to speed up the search procedure. Other optimization techniques require further careful investigation.

Part III

The Tableau WorkBench

Chapter 4

A General Tableau Prover

This part consists of Chapters 4-8 in which we present the Tableaux Work Bench (TWB), a generic framework designed to implement tableau-based automated theorem provers. In Chapter 4 we give our motivations to develop the TWB and we outline a number of issues related to the design of generic theorem provers. In Chapter 5 we give a detailed description of the TWB design. In Chapter 6 we present the TWB syntax to define tableau provers from a user's perspective. In Chapter 7, we give examples of calculi implemented with the TWB and experimental results. Finally, we conclude in Chapter 8 with a comparison with other theorem provers and an outline of future work.

4.1 Introduction

Automatic theorem provers for non-classical logics can be classified into two main classes, direct and translation provers:

Direct provers are based on different methods, such as resolution [Nivelle et al. 2000] or tableau [Goré 1999] to directly perform deduction on specific logics. Theorem provers such as FaCT [Horrocks and Patel-Schneider 1998], FaCT++ [Tsarkov and Horrocks 2006], RACER [Haarslev and Möller 2001] use the tableau algorithm and have been successfully employed by the description logic community to reason with considerably large knowledge bases. Provers like the LWB [Heuerding 1996] and Lotrec [Gasquet et al. 2005] have been engineered as general provers and offer algorithms for multiple logics. Provers like KSAT [Giunchiglia and Sebastiani 1996] use a version of the Davis-Putnam procedure modified for modal logic.

Translational provers utilise the correspondence between first-order logic and modal logics. They use a first order theorem prover, like SPASS [Weidenbach et al. 2002] or Vampire [Riazanov and Voronkov 2002] and a rather complex translation to a subset of first order logic that guarantees complete reasoning. The most well known example of a translational theorem prover for modal logic is MSPASS [Hustadt and Schmidt 2000].

Outside this classification, there exists proof editors, that are not specifically designed for modal logic, but which offer rudimentary search facilities. Among other existing proof editors, we mention xpe [Mouri 2001], JAPE [Bornat and Sufrin 1997] and PESCA [Ranta 2000].

On the other side of the spectrum, interactive theorem provers, like `Isabelle` and `Coq`, based on the Curry-Howard isomorphism between typed λ-calculi and intuitionistic logic can be used to give decision procedures for non-classical logic too. A detailed comparison with other provers and the `TWB` is given in Chapter 8.

4.2 Motivations and Target Audience

Theorem provers for non-classical logics have now matured dramatically in the last decade. On the one hand, highly optimised and specific theorem provers can test formulae with hundreds of symbols within a few seconds [Haarslev and Möller 2001]. On the other hand, the scientific community has put great effort into making this processing and expressive power available to a non-technical audience through simplified user interfaces and by theorem prover frameworks like the `LWB` [Heuerding et al. 1995] that implement a large number of logics.

Highly optimised direct theorem provers utilise many different optimisation techniques to speed up proof search in particular logics, while translational provers utilise translations into first-order logic and fast first-order theorem provers. However these avenues are not always viable when experimenting with logics that are not directly implemented: for example, `FaCT` cannot handle logics with an intuitionistic base; and although the `LWB` can handle intuitionistic logic, it can handle only a fixed collection of logics.

The `LWB` has been engineered to be easy to extend thanks to its modular design and polished API [1]. However the knowledge of a programming language (C/C++) is a pre-requisite to write a new module, making it unpalatable to non-technical/non-programmer researchers.

`MSPASS` gives a sound and complete prover for any first-order definable logic, but *a priori* it gives a decision procedure only for the ones that fall into decidable fragments of first-order logic like the two-variable fragment, or the guarded fragment. Indeed, `MSPASS` has over 128 flags, and it is not at all obvious how to obtain a decision procedure for a particular first-order definable logic using `MSPASS`.

On the other side of the spectrum, generic automated tableau-based provers like `Blast_tac` [Paulson 1999] and `Lotrec` [Gasquet et al. 2005] provide facilities for experimenting with new tableau calculi. As far as we are aware, the only system which allows a user to experiment with different optimisation techniques, different proof-search strategies and different tableau calculi together is `Lotrec`, which we discuss in detail in Section 8.1. In particular, all theorem provers cited above, with the exception of `Lotrec` do not allow a user to **easily** experiment with different history/dynamic-blocking mechanisms which are important techniques to implement theorem provers for many non-classical logics.

The Tableaux WorkBench (`TWB`) is a generic framework based on tableau algorithms, specifically designed for expressing and combining new tableau rules into an underlying tableau proof (and disproof) engine. It provides a simple user-interface and facilities to specify decision procedures and to experiment with optimisations techniques. The main motivations to create the `TWB` framework were:

1. To create an easy to use tool to allow a non-technical audience to experiment with non-classical logics;

[1] Application Program Interface: Routines that extend a language's functionality.

2. To create a modular framework to experiment with and benchmark different calculi and optimisation techniques;

3. To explore and expand the potential of tableau methods;

4. To experiment with decision procedures for fix-point logics.

The target audience of the TWB divides into three main categories:

First the TWB can be use to experiment with new logic systems by those who do not have the technical skills to modify existing theorem provers such as the LWB. The TWB is meant to be a tool to quickly mechanise logical systems with minor modifications from their "pen and paper" formulation. The interface of the TWB, albeit limiting for skilled users, is designed to be intuitive and close as possible to the traditional way of specifying logic.

Second, the TWB can be used as an aid to teach automated reasoning[2]. The similarity between the syntax of the TWB and the syntax found in the literature [Goré 1999], gives educators more time to concentrate on automated reasoning aspects without distracting students with complicated programming techniques.

Third, the TWB can be used by expert users, with programming skills, to build custom theorem provers. Since the TWB is built on top of the OCaml programming language, it is easy to access and modify existing data structures and to customise the default proof search algorithm. Moreover, the TWB has a modular design and each module offers a "side effect free" API in terms of high-order functions, making it easy to modify specific components without compromising the soundness of the overall architecture.

4.3 Anatomy of a General Tableau Theorem Prover

Tableau calculi and all other variants like labelled tableau [Massacci 2000], semantic tableau [Fitting 1983], sequent calculi [Gentzen 1935], hypersequents [Avron 1996], etc. can be characterised by their common underlying algorithm. From an abstract point of view, *proof search* in this context consists of building a *proof tree* using a finite set of *rules*. In this section we analyse a number of issues to consider when designing a general tableau-based theorem prover. Most of the considerations in this section will clarify the design criteria that guided the development of the TWB.

4.3.1 Tableau Methods

Tableau calculi can be viewed as algorithms that search for models of a given formula (the sequent method, on the other hand, views the proof tree construction not as model building, but as proof-search). Tableau methods are one of the most flexible calculi for automated reasoning. Modularity, efficiency and simplicity make tableau methods palatable to cleanly specify logical systems from a proof theoretical perspective. Conversely, from an automated reasoning point of view, tableau methods can be easily mechanised by making the search algorithm deterministic. Having the possibility to control and fine tune the search procedure gives to

[2]Currently, the TWB does not provide tools to implement first order logics. We will extend the user interface to support quantifiers as future work (cf. Section 8.2)

tableau algorithms greater flexibility to explore and to implement a large range of problems. In particular:

Modularity: Tableau methods have the convenient property of being modular and make it easy to adapt one logic to another by just modifying a few rules and the decision procedure [Fitting 1983]. For example, tableaux for basic modal logic can be built on top of tableaux for *CPL* by simply adding rules to handle modal connectives.

Efficiency: Tableau methods are reasonably efficient. Naive tableaux are deemed to be inefficient for large problems, but experimental evidence show that they are as efficient as other methods if adequate optimisation techniques are employed [Massacci 1998; Massacci 2000; Horrocks 1997]. To this end, it is essential to remove non-determinism from the decision procedure, so as to minimise backtracking.

Simplicity: Tableau methods allow a very natural formulation of logical systems and offer a straightforward avenue to generate readable proofs and (counter) models.

From an automated reasoning perspective, there are a number of desirable elements that are important when implementing decision procedures based on the tableau method:

Tactic Language: Naive tableau methods do not specify the order in which rules are applied during proof-search. A tactic language is important to complement and extend a tableau calculus specification and therefore to design efficient decision procedures. By removing the inherent non-determinism of tableau methods, the search space can be greatly reduced, making problems with high complexity tractable on the average case.

Histories: Tableau calculi do not always ensure termination of the decision procedure. In particular, tableau calculi that do not enjoy the analytical sub-formula propriety can run into cycles. Different methods have been developed to re-gain termination in such cases. Histories or blocking techniques, among others, have proved particularly successful [Howe 1998; Horrocks 1997; Heuerding et al. 1996].

Variables: Similar to histories, where information is passed top-down, variables can be used in tableau procedures to propagate information bottom-up regarding already explored branches during the search procedure. In conventional tableau calculi, we can imagine that the information regarding the "status" of a branch (being open or closed) is stored in a variable passed from the leaves to the root.

Note. sequent-based procedures and tableau-based procedures can essentially be seen as two different sides of the same coin. In the following, we focus on tableau methods, but it is easy to see that this choice is more related to the author's preference than to any technical reason.

4.3.2 Mechanizing Tableau Methods

In our context, tableau rules can be generically seen as functions to transform a node into a list of nodes during the construction of a graph. Thus, a rule *numerator* can be seen as a pattern that matches formulae in the current node by using *formula schema* as pattern identifiers.

We say that a rule is *applicable* if the numerator of the rule matches the current node. The *denominators* of a rule represent actions to expand the graph. The numerator contains one or more distinguished formula schemas called *principal formulae* and one or more *side formulae*. We use *meta-variables* $A, B, P, Q, \cdots$ to represent principal formulae and X, Y, Z for, possibly empty, sets of side formulae. We also write $A; X$ to mean $\{A\} \cup X$. In the following we consider a generic tableau algorithm that is not tied to any specific logic. We start by formally defining a rule.

Definition 4.3.1. A rule is a tuple (C, D, B) where:

C is a function that checks if the rule is applicable to a numerator,

D is a function that transforms a node into a list of (denominator) nodes,

B is a function to check if the proof tree that results from the current node respects a logic specific backtracking condition.

In the literature, tableau rules are often written as:

$$\frac{n}{d_1 \ldots d_m}$$

where n is the numerator, while $d_1 \ldots d_m$ is a list of denominators. Both n and each d_i is a formula schema. Formula schemas are used in the search procedure to *partition* the current node and to associate formula instances to meta-variables.

Below, in the in the $(\wedge)$-rule shown on the hand side left, the numerator specifies a partition of the current node which instantiates the meta-variables A and B respectively to the first and second conjunct of the principal formula $A \wedge B$ while instantiating the meta variable Z to all other formulae in the node. For the (K)-rule on the right, the numerator specifies a partition of the current node which instantiates the meta-variable P to the immediate sub-formula of $\Diamond P$, instantiates X to the set of immediate sub-formulae of all $\Box$-formulae and to Z all other formulae in the node. Note that since all $\Box$-formulae are matched by the schema $\Box X$, this implies that the set Z will be $\Box$-free.

$$(\wedge)\ \frac{A \wedge B; Z}{A; B; Z} \qquad \textbf{(K)}\ \frac{\Diamond P; \Box X; Z}{P; X}$$

We now define the rule graph as in [Brotherston 2005].

Definition 4.3.2. Let $\mathscr{L}$ be a logic and $\mathscr{S}$ be the set of all sets of well-formed formulae in $\mathscr{L}$. Let $\mathscr{R}$ denote a set of rules. Then we define a *rule graph* by (V, s, p, r), where:

- V is a set of vertices (or nodes),
- $s : V \to \mathscr{S}$ is a labelling function which maps each vertex to a set of formulae,
- $r : V \to \mathscr{R}$ is a labelling function which maps each vertex to a rule
- $p : V \to V^n$ is a labelling function which maps a vertex to a set of vertices. We write $p_j(v)$ for the jth component of $p(v)$;

- for all $v \in V$, the component $p_j(v)$ is defined when $r(v)$ is a rule with m denominators, $1 \leq j \leq m$, and $\dfrac{s(v)}{s(p_1(v))\dots s(p_m(v))}$ is an instance of rule $r(v)$.

A *path* in a rule graph is a sequence $v_0 j_0 v_1 j_1 \cdots$ such that for each $i \geq 0$, $v_{i+1} = p_{j_i}(v_i)$.

Definition 4.3.3. A rule graph (V, s, p, r) is a *proof tree* if there is a distinguished node $v_0 \in V$ such that for all $v \in V$ there is a unique path from v_0 to v. We call v_0 the *root* of the tree. A proof tree is a *derivation tree* if the root is a set of formulae respecting a logic-specific condition.

Definition 4.3.4. Let $\mathscr{S}$ be a set of well formed formulae and $\mathscr{R}$ a set of rules. A *decision procedure* is a sequence of steps that builds a finite proof tree by applying rules from $\mathscr{R}$ to an initial set of formulae in $\mathscr{S}$.

Rules build a proof tree from an initial node containing a finite set of formulae. Traditionally, logicians focus on the existence of a derivation tree rather than on the means by which this derivation is found. Conversely, automated reasoners focus their attention to design algorithms to search for a derivation tree as efficiently as possible. From a theoretical point of view, it is therefore not important to specify the order in which rules are applied as long as there exists a sequence of rule applications which builds a derivation tree. But from a practical point of view, specifying a particular order can dramatically speed up the search and construction of a proof tree. Moreover, since different rule sequences can produce different derivation trees, an efficient decision procedure should always try to build the smallest derivation tree.

We are mainly interested in building efficient decision procedures based on tableau methods. It is therefore important for us to classify rules using aspects related to the mechanization of tableau calculi rather than semantic aspects. A decision procedure can be characterised by two mutually linked procedures: one that builds the proof tree by applying rules, and one that visits this proof tree searching for a derivation tree. We classify rules as follows:

Linear rules: the backtracking condition is true if **the** denominator respects a logic specific condition.

Universally branching rules: the backtracking condition is true if and only if the proof trees that stem from **each** denominator respect a logic specific condition. We use “|” to separate the denominators of such rules.

Existentially branching rules: the backtracking condition is true if and only if the proof tree that stems from **some** denominator respects a logic specific condition. We use “||” to separate the denominators of such rules.

Conditionally branching rules: the backtracking condition is true if and only if the proof tree that stems from **the/each/some** denominator respect(s) a logic specific condition. We use “|||” to separate the denominators of such rules.

In the literature, rules are also respectively classified as α-rules (linear rules), β-rules (universally branching rules), and π-rules (existentially branching rules) [Fitting 1983]. For example, in *CPL*, the $(\wedge)$-rule is a linear rule since the tableau for the numerator is closed if and only if the tableau for the (only) denominator is closed. The $\vee$-rule:

$$(\vee)\ \frac{A \vee B; Z}{A; Z | B; Z}$$

is a universally branching rule since the tableau for the numerator is closed if and only if the tableau for both (all) denominators are closed. The deterministic (K)-rule from the basic modal logic K (note that the pattern matching here implies that the set Z is diamond- and box- free):

$$\textbf{det-K}\ \frac{\Diamond P_1 \ldots \Diamond P_n; \Box X; Z}{P_1; X || P_2; X || \ldots || P_n; X}$$

is an example of an existentially branching rule since the numerator is closed if and only if the tableau for some (one) denominator is closed.

Note. The double bar "||" intuitively means: "If the sub-tableau $P_i; X$ is open, then explore the sub-tableau for $P_{i+1}; X$". This is opposite to the single bar "|" traditionally used to specify the $(\vee)$-rule : "if the first branch is closed, then explore the second branch".

Conditionally branching rules are a generalization of universally and existentially branching rules: the U-rule for $PLTL$ (see Section 7.1.2) is an example.

4.3.3 Removing Non-Determinism

In the literature, the specification of logical calculi often hides a great deal of non-determinism which is commonly accepted by logicians. However, to implement an efficient theorem prover, it is necessary to make the decision procedure deterministic. In general, we can identify three forms of non-determinism associated with three fundamental aspects of a generic decision procedure for tableau methods:

Node-choice: the visit algorithm determines which leaf node (amongst many) to expand.

Rule-choice: the strategy algorithm determines which rule (amongst many) to apply to the current node.

Formula-choice: a heuristic procedure determines which principal formula (amongst many) in the current node is chosen.

Formula-choice is also characterised as *don't care* non-determinism while node-choice as *don't know* non-determinism. A *don't care* non-deterministic choice is an arbitrary choice of one among multiple possible continuations for a computation, all of which return the same result. A *don't know* non-deterministic choice is one choice among multiple possible continuations, which do not necessarily return the same result. For completeness in a tableau calculus, it is often sufficient to assume that all choices are *don't know* non-deterministic. However, a deterministic algorithm based on this assumption will be extremely inefficient since it has to consider all possible continuations for all choices. These two forms of non-determinism are related to the traditional notion of invertibility [3].

[3] A rule is invertible if whenever the numerator has a derivation so do/does all/some denominator(s).

$$\textbf{(K)}\ \frac{\Diamond P; \Box X; Z}{P; X} \qquad \textbf{(det-K)}\ \frac{\Diamond P_1 \ldots \Diamond P_n; \Box X; Z}{P_1; X || P_2; X || \ldots || P_n; X} \qquad \textbf{(rec-K)}\ \frac{\Diamond P; \Box X; \Diamond Y; Z}{P; X \mid\mid \Box X; \Diamond Y}$$

Figure 4.1: Alternative definitions of the K rule for basic modal logic.

4.3.3.1 Formula-choice.

A *don't care* non-deterministic choice is usually related to optimisation techniques based on logic-specific considerations used to reduce the size of the search space. Heuristics such as MOMS [Freeman 1995] (Maximum number of Occurrences in disjunctions of Minimum Size) or iMOMS [Freeman 1995] (inverted MOMS) order the formulae in the node to always choose the less/more constrained disjunct, which, in principle, should lead to an earlier clash.

Other more specialized heuristics such as Maximise-Jumps [Horrocks and Patel-Schneider 1998] are designed to work with optimisation techniques like back-jumping. The Maximise-Jumps heuristics is designed to choose the formula for which the maximum back-jumping identifier is minimal and therefore if it leads to a clash, will maximise the effect of the back-jumping procedure.

4.3.3.2 Node-choice.

The traditional formulation of the rule K for the basic modal logic, as shown on the left of Figure 4.1, assumes implicit backtracking on the choice of principal diamond formulae since there may be other diamond formulae in the set Z. To mechanise this kind of non-determinism, we need to make explicit how we "recover" from an unsuccessful choice and how we proceed in the proof search. That is, the implicit semantics is that if $\Diamond P_1, \cdots, \Diamond P_n \in Z$ and if the tableau for $(P;X)$ is open, the search procedure will try the (K)-rule on all other diamond formulae in the set Z before continuing further. Thus, the (K)-rule embeds a *don't know* non-deterministic choice, where selecting different principal formulae can lead to different results.

A fairly intuitive way to make the (K)-rule appear more deterministic is to make the backtracking explicit using existential branching as shown in (**det-K**) (center in Figure 4.1). Alternatively, we can write the (K)-rule by iterating over all other possibilities as in the right formulation (**rec-K**) in Figure 4.1, and define an appropriate search strategy. In Section 6.1.3 we show how to express *don't know* choices in the `TWB`.

4.3.3.3 Rule-choice

The third source of non-determinism in tableau methods arises from the lack of guidance when applying logical rules. Tableau methods are typically non-deterministic in nature. By specifying a finite set of rules, they say what can be done, not what must be done. In theory, a naive algorithm could just try all possible combinations of logical rules until one successful sequence is found. In other words, a deterministic algorithm can be implemented to backtrack over all possible rules, and therefore to explore the entire search space by brute force. However, when efficiency is important, such an approach should be avoided and a *strategy-based procedure* should be adopted instead.

To obtain an efficient mechanical procedure, tableau rules need to be coupled with a *strategy* which controls the order in which these rules are applied. For example, in *CPL* where all rules are invertible, since the order in which rules are applied is not important, specifying a strategy that applies all linear rules and the axiom first, and then all branching rules, can potentially shorten the proof tree. The problem is however different in basic modal logic, where not all sequences of rule applications ensure a solution since the (K)-rule is not invertible.

The problem becomes even more complex when dealing with tableau-sequent systems for non-classical logics with more than one non-invertible rule (for example intuitionistic logic as in [Dyckhoff 1992]). In this situation, at any given choice point, after all invertible rules are applied, we are forced to guess which non-invertible rule to apply, and eventually to undo this decision if it does not lead to the construction of an acceptable proof tree. Consequently, if the proof tree obtained from the application of the first rule of a sequence of non-invertible rules does not respect a logic-specific condition, the entire proof tree generated from that rule application must be discarded. To recover from this wrong choice, the proof must be re-started using the next non-invertible rule available in the sequence.

The notion of *tactic* is fundamental to define such strategies. Tactical languages originated in the work of the Edinburgh LFC [Paulson 1987] and then were refined in `Isabelle` [Paulson 1993]. The work on tactic languages for specifying search strategies follows a long history of work in the theorem proving and functional programming communities [Felty 1993].

The tactic language used in the `TWB` is loosely based on the tactic language *Angel* [Martin et al. 1995]. Angel[4] is a general-purpose tactic language that is not tailored to any particular proof tool; it assumes only that rules transform nodes into a list of nodes. Although the language was originally intended to support goal-directed proof, it is to be much more widely applicable as a language in which general transformations, such as complex re-writing rules, can be described.

4.4 Design Criteria

We summarise the main design criteria we adopted in the implementation of the `TWB`.

Generality: The `TWB` has been designed to be a tool for a non technical audience. As such, we did not assume any domain specific knowledge, leaving to the user full freedom to use the `TWB` as a generic framework to experiment with arbitrary logics. This decision does not come without penalties. The `TWB` cannot compete with specialized theorem provers for speed. Although we have optimised low level data structures and algorithms, it is impossible to address domain specific optimisations is a generic way.

Extensibility: The `TWB` implements a specialised high-level language tailored to specify decision procedures for non-classical logics. The `TWB` also offers to more experienced and demanding users, a number of "hooks" to tweak and to specialise the default behaviour. Moreover, since the `TWB` is built on the top of the OCaml compiler, the confident user

[4] The language is named Angel because it makes tactics "angelically non-deterministic". That is, a compound tactic will fail only if there is no possible path from input to output, or in other words, dead-end paths are covered by backtracking. The angelic style removes the need for the tactic programmer to code backtracking explicitly.

can also access the entire underlying programming language, adding and modifying basic data structures or to write new heuristics.

Modularity: The TWB design addresses this criteria at two different levels. On the one hand, the core of the prover is built following a strictly modular approach. It is therefore possible to replace components without compromising the overall architecture and to customise the TWB to implement different flavours of tableau calculi or more efficient data structures. On the other hand, the user is free to compose logic modules, and reuse components via the standard OCaml module system. Moreover, the tedious task of compiling modules and resolving dependencies is delegated to a small separate application, making this process transparent to the user.

Language: We have chosen OCaml to develop the TWB for two main reasons. Firstly because OCaml is a functional language. Functional languages are known to be concise and programs written in such languages are therefore easier to maintain and more resistant to problems related to memory management and runtime errors [Hudak and Jones 1994]. Secondly, because OCaml has a broad community of users, OCaml has an impressive number of third party libraries available and it has already been used to implement other successful theorem provers such as Coq, Hol light and NuPrl. Languages like Prolog, Mercury, Haskell, Java and C++ were also considered but deemed less suitable.

Functional Style: OCaml is an hybrid language which mixes elements of imperative programming with its functional basis. To minimise coding errors and to reduce maintenance time, the TWB core infrastructure was developed in a purely functional style. However, since this choice could lead to performance penalties, imperative programming style was used to implement a number of data structures and control algorithms.

Portability: To be accepted by a wider public, it is essential for a system to be portable to different platforms. Currently the TWB runs on modern operating systems such as Gnu/Linux and MacOsX. Moreover, the TWB has virtually no dependencies[5] on third-party software outside the OCaml compiler itself, therefore making it easy to install and port to other platforms.

4.5 Development Techniques, Tools and Availability

The TWB has been developed loosely following the Extreme Programming methodology [Beck 2000]. We started with “the simplest thing that could possibly work” and then refined our requirements incrementally during the years as our understanding of the problem gave us more insights and challenges. We released only version 2.0 of the TWB [Abate and Goré 2003] in 2004. This re-factoring process is continuing in the development version of the TWB. The OCaml programming language lends itself very easily to this methodology, allowing to very quickly test ideas and to produce working prototypes that can be later re-factored into the final

[5]The TWB depends on an external library, ExtLib that is distributed with the TWB. We plan to remove this dependency by re-implementing the subset of this library that is currently used.

product, or discarded. Several Honours students used the TWB to develop user interfaces and to experiment with temporal and intuitionistic logics using version 2.0.

The tools used for the development of the TWB are standard open source development tools. The TWB is available for downloading from the website

`http://twb.rsise.anu.edu.au`

Moreover, the Darcs[6] repository of the stable and development version can be found at

`http://twb.rsise.anu.edu.au/Repository`

[6] Darcs is a versioning control system similar to CVS. See `http://www.darcs.net/`

Chapter 5

Overview of the TWB

"Great design": It's the ineffable quality that certain incredibly successful products have that makes people fall in love with them despite their flaws.

Joel Spolsky

The TWB is organised into four main components: core, data-type, tableau and syntax. In the core library, we define all type definitions and the strategy implementation. The data-type library gives the implementation of support data structures. The tableau module provides the machinery to implement tableau-based provers and the syntax library provides an easy way to access the TWB library and to define specific logic modules. We now give a detailed outline of the core library and an overview of the other libraries. This description is related to the stable version of the TWB (August 2006). The development version is discussed in Section 8.2.

5.1 The Core Algorithm

The core algorithm of the TWB is a procedure to visit a tree that is generated by the repeated application of a finite set of rules to an initial node containing a finite number of formulae. This visit procedure is composed of two *mutually recursive functions*. The first function selects a new rule to be applied to the current node while the second function traverses the tree generated by the application of the rule, by recursively calling the visit procedure. In Figure 5.1, we show a snapshot of the visit procedure by focusing on a node in the proof tree (the *current node*).

Note. Figure 5.1 assumes a depth-first, left-to-right visit procedure. This is however not a pre-requisite as it is possible to implement different visit procedures (cf. Section 5.2.5).

We can isolate four basic components of the visit procedure: rule conditions, rule application (actions), branch conditions and rule backtrack (see also Section 6.1.3).

Rule condition: this function is used by the strategy procedure. It checks if a rule is applicable to the current node and if the side conditions of the rule are satisfied. The result of this function is a partition of the current node according to the patterns specified in the numerator of the selected rule.

Rule application: this function accepts a rule that has been selected and applies it to the current node in order to expand the proof tree. The result of this function is a list of nodes, each identifying a new branch in the proof tree.

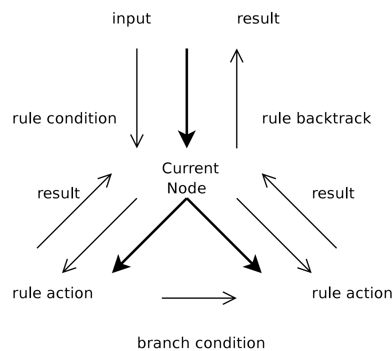


Figure 5.1: Algorithm Overview

Branch condition: this function is executed during the visit of the sub-tree rooted at the current node. Assume we have a branch condition for each branch generated by the rule application, and assume further we have n branches. Then the branch condition related to a particular branch i is executed, after visiting branch i, to determine if the sub-tree created so far (including all the already visited branches) satisfies a logic-specific condition. If this is the case, the visit explores the next branch (if any), otherwise, the visit is interrupted and the procedure backtracks.

Rule backtrack: these functions are executed after all branches (or a number of them according to the branch condition) are visited. The backtrack function synthesises the result of the visit of the sub-tree rooted at the current node and passes it up to the parent.

5.2 Core Library Modules

The core library defines the architecture of the prover using abstract data types and functors [1]. Figure 5.2 shows an overview of the structure of the core library, including the main methods of each class and the relation between each class and basic data type. Each module is described in detail in the following sections. The core library is composed of the following modules:

Sequence: defines a generic lazy list library;

Tree: defines the proof tree data type;

Node: defines the proof tree node;

Rule: defines the interface for specifying rules;

Strategy: defines and implements the strategy language;

Visit: defines and implements the visit algorithm of the proof tree.

[1] A functor is a function from modules to modules, used in functional languages to build parametrised modules.

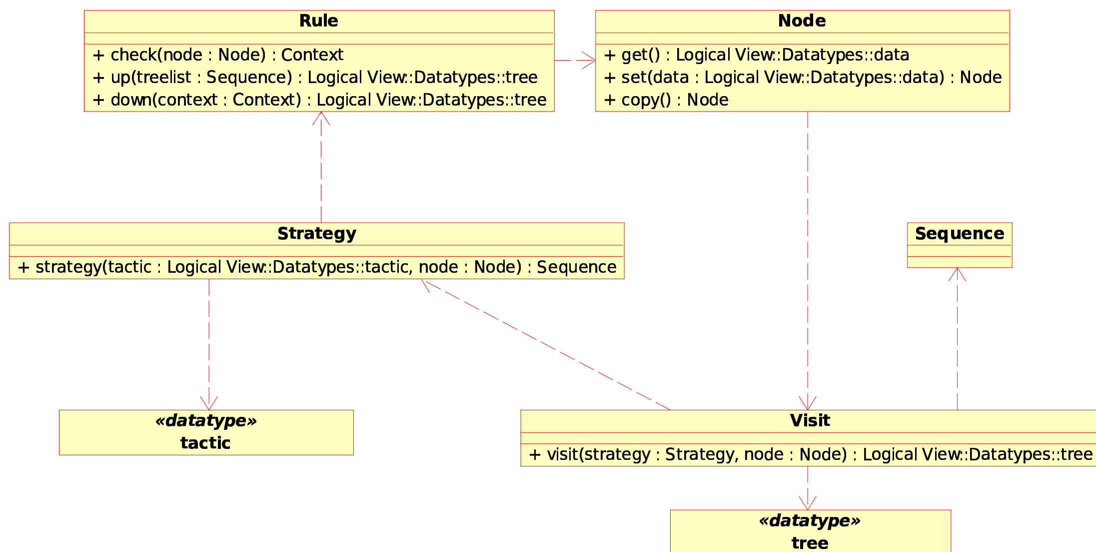


Figure 5.2: Core Library Overview

Note. In this work, we slightly abuse the notation of UML diagrams. Strictly speaking, only the *rule* and *node* data structures are real objects, in the sense of Object Oriented Programming (OOP), while the *strategy*, *visit* and *sequence* are OCaml modules (collection of functions). Despite this difference, we can think of OCaml modules as singleton object instances of a class type [Gamma et al. 1995; Vicente and Wagner 2003]. In fact, since all objects in the core libraries are implemented as purely functional objects[2], this difference is purely stylistic.

5.2.1 The Sequence Module

OCaml is by design an eager language. To avoid wasteful computations, we need to provide a lazy data structure to construct the proof tree. In the following, we assume familiarity with lazy evaluation strategies: for an overview of lazy data structures see [Okasaki 1996] and [Paulson 1991]. The *Sequence* module (*Seq*) in Listing 5.1, is used to implement a backtracking monad in the visit algorithm (cf. Section 5.2.5). The *Sequence* module implements a `MonadPlus` monad as described in Section 1.4 with the usual operators: `return`, `bind`, `mzero`, `mplus`.

In addition to these, we have other operators that are used in the strategy module to implement the monad *MState* (a detailed overview of this monad is in Section 5.2.4). These operations are formalised in [Kiselyov et al. 2005] and [Hinze 2000] to overcome limitations of the basic list monad when used to implement efficient backtracking algorithms. In detail, we add the following combinators to the standard list monad:

`guard` ***b*****:** if *b* is true then force the execution of a monadic operation, else, the result of the monadic operation is `mzero`.

[2] A purely functional object is an object in the sense of OOP, where all its visible fields are immutable, and therefore side-effect free. See [Chailloux et al. 2000] for details.

`determ` *m*: given a computation *m*, return the first successful computation, if any. This operation is similar to the cut operator in Prolog.

`msplit` *m*: determines whether a given computation *m* fails or succeeds at least once. If the computation succeeds, then the function "splits" the computation into the first successful result and the rest of the computation. If the computation fails, then it cannot be split. Operationally we can think of `msplit` as running the input computation looking for the first successful choice, providing a sort of "lookahead" of length one.

`ifte` *t th el*: if the computation of *t* succeeds with at least one result then (`ifte` *t th el*) behaves like (`bind` *t th*), else, (`ifte` *t th el*) is equivalent to *el*. The construct `ifte` is equivalent to the Prolog soft-cut [Clocksin and Mellish 1987] and Mercury's If-Then-else construct [Henderson et al. 1996]. The construct `ifte` is implemented using `msplit`.

5.2.2 The Node Module

A node is a generic container used to encapsulate arbitrary data structures (like sets, multi-sets, sequences, etc.) of type *elt*. A node can be seen as a vertex in the rule graph in Definition 4.3.2. The signature of the node is as follows:

```
class node : 'elt ->
    object ('node)
        method get  : 'elt
        method set  : 'elt -> 'node
        method copy : 'node
    end
```

The node class has only three generic methods:

get: returns the node data;

set: "replaces" the node data;

copy: returns a fresh copy of a node;

Listing 5.1: Backtracking Monad

```
type 'a m
type 'a excp = Nothing | Just of ('a * 'a m)
val return : 'a -> 'a m
val bind   : 'a m -> ('a -> 'b m) -> 'b m
val mzero  : 'a m
val mplus  : 'a m -> 'a m -> 'a m
val guard  : bool -> unit m
val determ : 'a m -> 'a m
val msplit : 'a m -> 'a excp m
val ifte   : 'a m -> ('a -> 'b m) -> 'b m -> 'b m
```

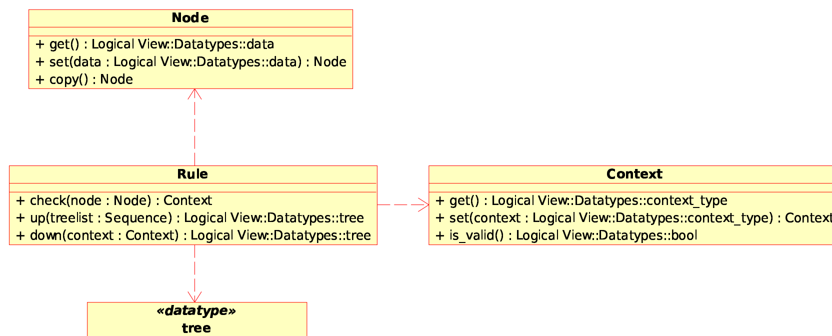


Figure 5.3: Rule Module

Since the node implementation is supposed to be side-effect free, the `set` method effectively constructs a new node instead of modifying the node data structure (*elt*) in place.

Note. The operation **set** could possibly construct a large object. While this is not advisable when designing efficient algorithms, we believe that the benefit of retaining a "pure" node object and the clever memory management of OCaml offsets this cost.

5.2.3 The Rule Module

The rule module implements the interface to define logical rules in the `TWB`. A rule can be seen as a re-writing function that is applied if a pre-condition is met. The class diagram in Figure 5.3 shows the relation of the rule object with other core data structures. In particular, the rule module depends on the Context module that defines a generic data-structure interface to store the intermediate state of a rule application.

The rule class in Listing 5.2 has three methods:

check: This method tests whether the rule is applicable to a node by pattern matching the node and checking the side conditions. It returns an object of type `ct` as in Listing 5.3;

down: This method applies the rule to extend the current node into a list of new nodes using the result of the `check` method;

up: This method specifies actions to be performed during backtracking, after all denominators are explored.

Listing 5.2: Rule class signature

```
type tree = node Tree.tree
class virtual rule :
  object
    method virtual check : node -> context
    method virtual down  : context -> tree
    method virtual up    : tree Llist.llist -> tree
  end
```

The class type *ct* stipulates a generic interface for the context object. In particular, the method `is_valid` is invoked during the execution of the **check** method and specifically be the strategy language interpreter (cf. Section 6.1.4), to check if the current context object should be considered or discarded. At the core level, the rule context data-structure is unspecified. In the tableau library we will see that the rule context is implemented (cf. Section 5.4.3) as a triple composed of a lazy sequence, a hash table and an object of type `node` (cf. Section 5.4.1).

Listing 5.3: Context class signature

```
class type ['celt] ct =
  object('ctx)
    method get      : 'celt
    method set      : 'celt -> 'cxt
    method is_valid : bool
  end
```

5.2.4 The Strategy Module

A tactic is used to guide the proof search as explained below. The Strategy module implements an interpreter for the tactic language. The language used in the `TWB` is loosely related to the tactic language employed in theorem provers like `Isabelle` (for a formal definition of a tactic language refer to [Martin et al. 1995]). We now describe the tactic language in terms of the data-type in Listing 5.4, which is a simplified version of the real data-type.

The most basic tactic is a simple rule application. The tactic `Rule` is composed of two functions: *check* and *apply*. The first function checks if the rule is applicable to the current node, while the second applies the rule to transform the current node into a list of new nodes. There are two possible outcomes when applying a rule tactic to a node: if the rule is applicable to the node, that is, the *check* function returns `true`, then the rule is applied, producing a list of new nodes; else the rule does not match the current node, and the application of the function fails. Two other basic tactics are `Skip` and `Fail`. The trivial tactic `Skip` always succeeds, while the tactic `Fail` always fails.

Listing 5.4: The tactic language

```
type check = ('state -> bool)
type apply = ('state -> 'state)
type 'state tactic =
    |Rule of (check * apply)
    |Seq  of 'state tactic * 'state tactic
    |Alt  of 'state tactic * 'state tactic
    |Cut  of 'state tactic
    |Mu   of 'state tactic
    |Skip
    |Fail
```

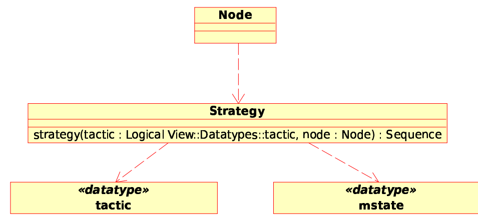


Figure 5.4: Strategy Module

Tactics can be combined in two ways: in alternation or in sequence. The sequential composition $\texttt{Seq}(t_1,t_2)$ of two tactics is written as $t_1;t_2$. This tactic first applies t_1 and then applies t_2 to the outcome of t_1. If either t_1 or t_2 fails so does the combined tactic.

The tactic $\texttt{Alt}(t_1,t_2)$ combines two tactics in alternation and is written as $t_1 \mid t_2$. That is, it first applies t_1. If t_1 fails, then it applies t_2, else it behaves like t_1. If both t_1 and t_2 fail, the combined tactic fails. An extension of the alternation tactic is discussed in Section 8.2.2.2.

The tactic `Cut` t behaves like t, but locally restricts the action of alternation inside t. If t does not contain alternation, then `Cut` t behaves like t. Otherwise it returns the first successful tactic application from t; if a subsequent tactic application after `Cut` t fails, then the whole tactic fails, that is, alternatives within t are not re-explored. For example, let $t = Cut(t_1 \mid t_2)\ ;\ t_3$ be a tactic and assume t_1 succeed, but t_3 fails. Then the tactic t will fail without trying tactic t_2.

The tactic `Mu` implements the fix-point combinator $\mu X \circ tac(X)$ where X is a variable and $tac(X)$ is a tactic in which the variable X may occur as though it were itself a tactic. The tactic $\mu X \circ tac(X)$ behaves as $tac(X)$, but where each occurrence of X behaves as though it were $\mu X \circ tac(X)$.

In the following we use the short cut t^* for $\mu X \circ (t;X|Skip)$. With t^*, first the tactic t is applied. If the tactic t fails then the combined tactic t^* behaves like the tactic t. Else the combined tactic behaves like t^*.

Figure 5.4 shows the dependencies of the strategy module. In the real implementation, the tactic language interpreter depends on the *MState* monad explained shortly. In particular the signature of the interpreter is as follows:

```
type m = MState.res MState.m Seq.llist
val strategy : tactic -> node -> m
```

The type m is a (lazy) sequence of state monads each containing the result of the computation of the strategy. The strategy function accepts a *tactic* (that is, an expression of the tactic language) and the initial node and returns the list of executions of the strategy. Since the strategy interpreter is interleaved with the visit function, each result of the strategy function (cf. `MState.res MState.m`) is itself a function that contains the result of the application of a rule to the input node and the "rest of the computation". In turn, since the rest of the computation is dependent on the result of the application of the selected rule, it is itself a partial function that accepts a node and returns a sequence of type m. In order to achieve this computation we propose the `MState` monad.

Listing 5.5: The MState monad

```
type res          = rule * context
type continuation = Cont of (node -> res m llist)
and csstack       = continuation llist
and state         = csstack

type 'a m = state -> 'a * state
val return : 'a -> 'a m
val bind   : 'a m -> ('b -> 'b m) -> 'b m
```

MState Monad

The module `MState` in Listing 5.5 implements a monad for dynamic continuation-passing style (CPS) by combining a state monad with a CPS monad. A similar monad can be found in [Biernacki et al. 2005]. The implementation of this monad follows an operational approach where continuations are represented as a list of control-stack frames composed of list concatenation. A different approach would be to represent continuations with continuation-passing functions and compose them by continuation-passing composition. Refining this idea, we could also have used a CPS monad to abstract this behaviour and plug it into the `MState` monad. For an overview of delimited continuations see [Danvy and Filinski 1990] and [Felleisen 1988].

5.2.5 The Visit Module

The visit module implements the core algorithm described in Section 5.1. Given a node and the state of the strategy, the recursive procedure follows this pseudo-algorithm:

1. Select a new rule using the strategy applied to the current node;
2. Build the new leaves of the proof tree according to the selected rule;
3. Call visit on all new branches (leaves) of the tree;
4. If there are no more rules, then return a leaf and backtrack;

This algorithm is implemented in the `TWB` as shown in Listing 5.6. The core function `aux_visit` takes a traversal function, a strategy continuation (`str`), the control-stack (`state`) and the current node (`node`). Following the pseudo-algorithm, if the strategy procedure (`str node`) succeeds then we apply the selected rule and we recursively invoke the procedure on the sub-tree generated by the rule application. Otherwise, if the strategy does not yield a result, we pop the control-stack (`Seq.determ(state)`), if it is not empty, and recurse using the first strategy continuation in the current state. If the control-stack is empty, this means that no more rules are applicable to the current node and we backtrack returning a leaf node (`Seq.return Leaf(node)`) of the proof tree. Note that a branch of a proof tree is explored only if necessary. Listing 5.7 contains of a depth first traversal function `dfs`. To modify the search strategy a different traversal function must be implemented.

Listing 5.6: The TWB core algorithm

```
let rec aux_visit traversal str state node =
  Seq.ifte
  (str node)
  (fun ms ->
    let ((rule,context),newstate) = ms state in
    Seq.bind (Seq.determ(newstate)) (fun (Cont cont) ->
        let tree = rule#down context in
        traversal cont (tl(newstate)) rule context tree
    )
  )
  (
    Seq.ifte
    (Seq.determ(state))
    (fun (Cont cont) ->
            aux_visit traversal cont (tl(state)) node)
    (Seq.return (Leaf(node)))
  )
```

The real implementation of the visit function (not shown here) also includes a caching mechanism to avoid re-computation of identical parts of the proof tree. Note that the caching mechanism can be customized by the user by selecting which rules should check the cache table (cf. Section 6.1.5.1), therefore effectively establishing check points in the proof search.

5.3 User Data-Types Library

The Core library defines the structure of the theorem prover. The data-type library provides concrete data types that are the foundations for the tableau library and can be extended by the user. The user data types are built hierarchically with the library divided into two sections: basic data types and composite data types. The basic data types module implements the machinery to handle formulae, tuples, triples, integers, booleans, etc. The composite data types are high level data types (sets, multi-sets, lists, . . .) that are defined on top of the basic types. Composites types are built using OCaml standard library modules [Chailloux et al. 2000]. Figure 5.5 shows the set interface and how different data-types are defined. In particular, we note that the type *list* and *set* both implement the set interface. This is done to modularise the library

Listing 5.7: Depth-First Traversal Function

```
let rec dfs str state rule context = function
  |Leaf(_) as tree ->
     Seq.return (rule#up context (Llist.return tree))
  |Tree(l) ->
     Seq.return (rule#up context
       (Seq.bind l (aux_visit dfs str state)))
```

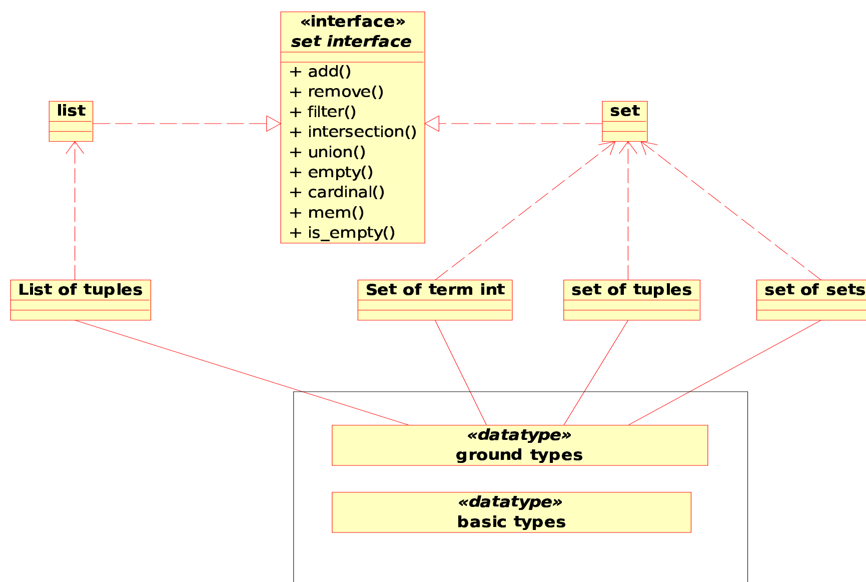


Figure 5.5: User Data Structures Hierarchy

design and minimise dependencies to non-polymorphic data structures. Other data structures in the library are implemented hierarchically by re-using pre-defined data structures and by using the OCaml standard library. A list of the pre-defined data-structures is in Section 6.1.2.

Note. The basic data types are considered user-defined in the sense that they are not tied to the core library definition. Ideally, the user should be able to extend the basic types in the logic module by using high level type declarations. At the moment, to extend the basic data types, it is necessary to re-compile the library. It is left for future work to make user data types truly extensible. See Section 8.2 for details.

Formula Map and History Map

The user data-type library also implements two ad-hoc data structures to define sets of formulae and histories, respectively, *gmap* and *hmap*. These two modules are used in the tableau module to give an implementation of the *elt* type defined in the node module in Section 5.2.2. The data structures *gmap* and *hmap* are implemented by using a hash table to associate formula schemas with a container of formula instances. The container can then be defined to be a set, multi-set or a list according to the logic specific requirements.

5.4 Tableau Library Modules

The core library and the data-type library are the foundations upon which (possibly arbitrary) families of theorem provers can be built. In this section, we present the library to implement tableau based theorem provers. The library is composed of the following modules:

NodePattern: defines the data type to specify the numerator and denominator of a rule;

Build: implements functions to build a new node given a pattern;

Partition: implements the functions to match a node given a pattern;

UserRule: implements three rule types: linear, universal and (conditional) existential.

5.4.1 The Partition Module

The Partition module defines the machinery to pattern match the numerator of a rule with the current node and to build the trailer of the proof tree. The partition algorithm, given a list of patterns as input, returns a tentative match-object in the form of a node and a substitution list.

Patterns are specified by user-defined *pattern functions* defined in the logic modules (cf. Section 6.1). Generically, a pattern function accepts a substitution list and returns a new substitution list containing the formulae that have been matched. Since pattern functions have full access to the substitution list, they can be used to implement arbitrary conditions or to arbitrarily modify formulae (cf. Section 6.1.3.3).

The implementation of the partition algorithm is based on lazy data structures to minimise the amount of computation. Since the pattern can be arbitrary, the complexity of the algorithm depends on the user-defined pattern functions, and in the worst case can be exponential in the number of objects in the node, but it is more likely to be sub-linear in the average case.

5.4.2 The UserRule Module

The rule behaviour as described in Section 4.3.2 is implemented in the UserRule module. This module defines a number of helper functions that generically specify the algorithms: to check if a rule is applicable (using the partition algorithm described above); to specify how to explore the proof tree in relation to the type of rule; and to specify how to behave on backtracking.

5.4.3 The RuleContext Module

The rule context type *ct* defined in Section 5.2.3 specifies the interface of the rule context data structures. The tableau library implements an instance of such a data-structure as a triple $(Enum, SubStList, Node)$ where:

Enum is a lazy enumeration of all possible ways of pattern-matching the numerator of the selected rule with the current node,

SubStList is a table that associates collections of formulae to meta-variables, and

Node is the old node, without formulae already pattern matched and stored in the substitution list.

The *SubStList* data-type, in particular, is built upon the data-type library by using a hash table that associates a string to a set (or list) of formulae.

component	loc
core	464
tableau library	818
syntax library	1624
data-type library	664
application (cli)	226
total	3800

logic	loc (tableau)	loc (functions)	total
CPL	18	98	126
K	21	72	93
KT	31	72	103
S4	41	72	113
PLTL	148	194	242

Table 5.1: Lines of Code

5.5 User Syntax Library

The tableau library defines the engine to build tableau based theorem provers. To make the TWB accessible to non-programmers, we provide a high level language to define tableau calculi. The user syntax library implements the parser for this language.

The syntax library is composed of two modules: user syntax and input parser. The first defines the tableau syntax as described in Chapter 6. The second defines a number of helper functions to parse input formulae according to the concrete syntax defined in the logic module.

The user syntax is defined as an extension of the OCaml programming language. To reuse the OCaml syntax and avoid re-writing the parser, we used the OCaml pre-processor camlp4. The user syntax is therefore defined using the camlp4 language and modules. The OCaml pre-processor implements an $LL(1)$ parser to extend the language with arbitrary constructs.

Currently (August 2006), camlp4 is going though a major revision. The TWB parser does not offer comprehensive error detection. We expect, as further work, to completely re-write this component by using the new version of camlp4, to make the parser more extendible and modular, and in particular, to improve the error reporting capabilities (cf. Section 8.2).

The input parser uses the same technology as above. Each theorem prover instance generated by the TWB is therefore linked with a generic $LL(1)$ parser, “configured” for the specific logic, to parse input formulae according to the concrete syntax defined in the logic module.

5.6 Summary

The TWB is very small compared with other theorem provers when considering the total number of lines of code (loc). This is the result of two design choices. The first to use a functional programming language like OCaml, and the second to minimize the use of imperative constructs. On the left of Table 5.1 we outline the number of lines of code for each module. We note that the two central modules, core and data-type, account only for less than one third of the total while the syntax library that implements the user interface to express tableau calculi accounts for almost half of the total.

On top of the TWB engine, building a new theorem prover for your favourite logics is then trivial and will not require to write more than a few hundreds lines of high-level code. On the right of Table 5.1, we show the number of loc of several calculi implemented in the TWB. Consider that for all logics, we also count several support functions (such as those to transform the input into negation normal form), and we have counted these multiple times, once for

each module even if the code is often shared. Moreover, as future work, we plan to increase modularity by sharing logical rules from different modules, so as to reduce duplication. For example, the tableau calculi for *CPL* account for 18 loc while the nnf procedure for 98, for a total of 126. We have to point out that a metric based on locs does not offer a rigorous measure to compare different implementations. In fact, to produce these results, we wrote our tableau rules on one line, when possible, and removed blank lines. The result is still readable, but users might want to write rules on different lines and using a different “typographic” style. In Appendix 8.2.2.3 we show the prover for *CPL*. We sometimes we this “compressed” format in the figure that follows, merely to save space.

Chapter 6

Using the TWB

"Simple things should be simple and hard things should be possible."

Alan Key (HP Labs.)

In this chapter, we describe the syntax of the TWB user interface. The TWB framework implements a collection of libraries to define automatic tableau-based theorem provers. To make the TWB engine more accessible to non-programmers, we present an abstraction layer to design new calculi without the need for learning the complex syntax of a programming language. In this section we present the syntax for specifying tableau based calculi. Other calculi like sequent, labelled tableau, etc. will be added in further work (cf. Section 8.2).

6.1 Defining the Calculus for a Logic

A tableau calculus is defined in the TWB by writing a *logic module* (eg. `pc.ml` or `k.ml`). A logic module is composed of four main sections:

Connectives: defines the concrete syntax of formulae;

Histories: defines the set of variables and histories used in the tableau algorithm;

Tableau Rules: defines the collection of tableau rules;

Strategy: defines the systematic proof-search procedure.

A logic module is automatically translated into OCaml code and then compiled and linked to the TWB libraries. The resulting object is an automatic theorem prover that accepts formulae with a syntax as specified in the connectives section and which outputs the derivation tree.

We tried to give a syntax that is as close as possible to the syntax used in the literature to specify tableau calculi. On the one hand, this approach allows non-technical users to specify most basic modal logics using a high level language. On the other hand, this approach gives access to the full underlying programming language thereby allowing the experienced user great flexibility. By compiling rules to native code, it is possible to run the prover at full speed, bypassing any intermediate abstraction layer. Moreover it simplifies the design of the underlying abstract machine, and ultimately reduces the TWB to a set of high order functions.

Note. In the following BNF specifications, we use UIDENT to specify identifiers that begin with an upper case character, LIDENT to specify lower case identifiers and OCaml expression to identify a section of code specified according to the OCaml syntax for (simple) expressions. We use String, Int, etc to identify the corresponding OCaml data types.

6.1.1 Defining Connectives

The connectives section defines the syntax of the formulae used in the rule definitions (as formula schema) and as input formulae. The connectives section is defined according to the following BNF grammar:

```
connectives  ::=  "CONNECTIVES" conn list "END"
conn         ::=  conn_type "," syntax "," assoc ";"
                  | UIDENT "," "Const" ";"
conn_type    ::=  UIDENT
syntax       ::=  string
assoc        ::=  "Zero" | "One" | "Two"
```

Thus a connective in the TWB is either a constant or is defined as a triple where:

conn_type: the first component is an OCaml type representing a formula as defined in Section 5.3,

syntax: the second component specifies the concrete syntax, where the underscore "_" is interpreted as a meta-variable occurrence,

assoc: the third component is a binding strength "Zero", "One" or "Two" with "Two" the weakest.

For example, in Listing 6.1, we define four binary connectives And, Or, Imp, and DImp, a unary connective Not and two constants, Verum and Falsum.

Listing 6.1: Connectives for *CPL*

```
CONNECTIVES
  DImp, "_<->_", Two;
  Imp,  "_->_",  One;
  And,  "_&_",   One;
  Or,   "_v_",   One;
  Not,  "~_",    Zero;
  Verum, Const;
  Falsum, Const;
END
```

Note. The parser allows us to specify unary and binary connectives, and in principle, it can be modified to specify more exotic formula shapes (see Section 8.2). But because of restrictions in the OCaml lexer, symbols like [] or [F], and constants like *True* and *False* cannot be defined via the connectives definition.

6.1.2 Defining Histories and Variables

Histories and variables can be used in the `TWB` to decorate tableau nodes with additional data structures and to maintain additional global information during the proof search. Histories are used to pass information top-down and variables to pass information bottom-up. Histories are commonly used to implement blocking techniques by passing information about already visited nodes from parents to leaves. At any stage in the proof search, at any node, it is then possible to decide to stop the procedure if a cycle in the path that leads to the current node is detected.

Variables are used to pass information about already explored sub-trees. For example, the traditional notion of a tableau being “Open” or “Closed” can be seen as information that is carried from the leaves to the root by a variable. In fact, the `TWB` uses a variable named `status` to carry this information. Histories and variables are defined using the following BNF grammar:

histories	::=	”HISTORIES” history list ”END”
history	::=	”(” identifier ”:” type ”:=” expr [default] ”)” ”;”
identifier	::=	LIDENT | UIDENT
type	::=	UIDENT ”of” type list
		| UIDENT
expr	::=	ocaml expression
default	::=	”default” expr

Thus, history identifiers always begin with the first character capitalized, while variable identifiers are always lower case. Each history and variable is associated with its own data container. The pre-defined data containers and their respective types are shown in Table 6.1. For example, in Listing 6.2, we declare a history `Diamonds` and two variables `uev` and `n`. The history `Diamonds` and the variable `uev` are associated with a new object of type `Set.set` by using the OCaml keyword `new` to instantiate a new object (ie. `new Set.set`). The variable `n` is associated with a new object of type `Singleton.set` which is a container of only one element.

Histories and variables can also be associated with a default value that is an OCaml expression of the correct type. Listing 6.2 shows an example where we do not set a default value for `Diamonds` and `uev` but we set a default value for `n`. The default value for singleton elements can be left unspecified. The default value for other data containers is just an empty instance of that data container.

Listing 6.2: Histories and variables

```
HISTORIES
  (Diamonds : Set of Formula := new Set.set);
  (uev      : Set of Formula := new Set.set);
  (n        : Int := new Singleton.set default 0)
END
```

Container	Type
Mtlist.listobj	List of Formula
Listofsets.listobj	List of (Set of Formula)
Set.set	Set of Formula
Singleton.single	Int | String | Formula | ...
Setofsets.set	Set of (Set of Formula)
Setoftermint.set	Set of (Formula * Int)
Setoftupleset.set	Set of (Set of Formula * Set of Formula)
Listoftupleset.listobj	List of (Set of Formula * Set of Formula)

Table 6.1: Pre-Defined Data Containers

Note. The handling of data containers in the stable version is very inflexible as it offers only a fixed set of data types and requires the `TWB` library to be recompiled to add new ones. In Section 8.2 we outline our ideas to address this problem in future versions.

The variable `status` is a special case. If `status` is not declared in the history section, the `TWB` will instantiate it implicitly with type `String` and default “Open” and use it to determine the result of the visit of a branch in the proof tree. If it is declared, the `TWB` will consider it as any other variable definition, but the user will be responsible for determining when to backtrack and how to propagate the value of the variable upon backtracking. Not specifying the backtracking behaviour, in relation to the variable `status`, does not interfere with the visit procedure, but it can lead to the exploration of the entire proof tree.

6.1.3 Defining Rules

Tableau rules define the actions to be executed during the proof search. Rules are declared in the tableau section of a logic module according to the following BNF grammar:

tableau	::=	”TABLEAU” rule list ”END” [”(cache)”]
rule	::=	numerator
		separator
		denominator list
		side_condition
		action
		branch_condition
		backtrack

Numerator. The numerator of each rule specifies a partition of the current node in the proof tree. A numerator is composed of a list of formula schemas separated by semicolons. Formulae schemas are parsed according to the connectives declared in the connectives section. We say that a set is *qualified* if it is specified by a formula schema (eg. $A \rightarrow B$). We say a set is *anonymous* if it is specified by just a meta-variable (eg. X).

Assume we have already declared the connectives *Box* and *Dia* and f is a function of type $formula\ list \rightarrow bool$ and g a function of type $formula \rightarrow bool$. Instead of giving a BNF

grammar for the formula schema, we give a number of examples to illustrate the syntax.

$\{Dia\ P\}$**:** match a non-empty principal formula;

$(Dia\ P)$**:** match a possibly empty principal formula;

$Box\ X$ **:** match a possibly empty qualified set;

$f\ (Box\ X)$ **:** match a possibly empty qualified set that meets the condition expressed by function f;

$g\ (\{Dia\ X\})$ **:** match a non-empty principal formula that meets the condition expressed by function g;

X **:** match a possibly empty anonymous set.

Note. Rules in the `TWB` can specify a wide range of patterns, and partition the current node in different ways. The user is responsible for reducing the complexity of the pattern matching as the `TWB` has no in-built facilities to detect computationally expensive patterns.

Separator. A rule is defined as invertible if the separator between the numerator and denominators is a non interrupted sequence of at least two "=" symbols and a rule is defined as not-invertible if the separator is a sequence of at least two "-" symbols. Intuitively, invertible and not-invertible rules capture the usual notion of invertibility. The `TWB` does not check that the rule is really invertible. It assumes that the user has proved an inversion lemma independently. Moreover, we also abuse this notation in the recursive definition **KRec** of the (K)-rule, where a non interrupted sequence of "=" symbols is used to specify that the rule must not automatically backtrack, rather than specify that the rule is invertible in the traditional sense.

Denominator. In general, a rule can have a list of denominators related by a branching operator. Each denominator specifies the shape of the root of one of the branches that extend the tableau and is composed of a list of formula schemas. However their interpretation is different: whereas in the numerator definition, formula schemas partition the current node - therefore acting as patterns - in the denominator definition, formula schema, are interpreted as instructions to build new formulae from the ones associated to meta-variables in the numerator.

We give examples to illustrate the syntax of the formula schemas used in the denominator. In the following we assume that A is a meta-variable that was pattern-matched to the list of formulae $[\varphi_1;\varphi_2;\cdots;\varphi_n]$ in the numerator and $Hist$ is a history of type `Set of Formula` declared in the history section that currently contains a list of formulae $[\psi_1;\psi_2;\cdots;\psi_m]$. Further assume that h is a generic function of type $formula\ list \rightarrow formula\ list$.

A**:** adds the list $[\varphi_1;\varphi_2;\cdots;\varphi_n]$ to a child of the current node;

$Box\ A$**:** adds the list $[Box\ \varphi_1;Box\ \varphi_2;\cdots;Box\ \varphi_n]$ to a child of the current node;

$h\ (Box\ A)$**:** adds the list $[h(Box\ \varphi_1);h(Box\ \varphi_2);\cdots;h(Box\ \varphi_n)]$ to a child of the current node;

$Hist$**:** adds the list of formulae $[\psi_1;\psi_2;\cdots;\psi_m]$ to a child of the current node;

Open, *Close* **or** *Stop* respectively specify that the current branch is open, closed or that the procedure must stop. When the variable `status` is not defined in the connectives section, the *Open* and *Close* directive will also implicitly set the value of the variable `status` respectively to “Open” or “Close”. The *Stop* directive does not affect the value of the `status` variable.

In the following sections, we give a number of examples of tableau rules. We assume that the following connectives have been declared in the connectives section: `&`, `Box`, `Dia`, $\rightarrow$, `v`. Rules can be written on one or more lines because carriage returns and spaces are not interpreted by the parser. In this section, for clarity, we will write rules on multiple lines.

6.1.3.1 Linear Rules

The simplest rule is an invertible, linear rule that breaks down a connective and leaves all other formulae intact.

```
RULE And
{ A & B }
=========
  A ; B
END
```

This example defines an invertible rule that pattern matches the current node with *A*&*B* as the (non-empty) principal formula, replaces *A*&*B* with *A* and *B* and leaves all other formulae in the node intact to create the denominator.

Note. Since the numerator does not contain anonymous sets, the pattern match is partial, meaning that all formulae in the current node that are not matched by the numerator, appear unchanged in the denominator. Thus this is effectively a “re-writing” rule. If no formula of the form *A*&*B* is in the current node, the rule fails.

To reduce the overhead associated with the number of formulae in a node that are all candidates for a linear application of a rule, it is also possible to specify no principal formula, and break all connectives of all formulae matching a formula schema at once.

```
RULE T
  Box X
=========
    X
END
```

This example defines an invertible rule that pattern matches the current node with the possibly empty formula schema *Box X*. If multiple *Box* formulae are present in the current node, they will all be reduced at once. If no formula of the form *Box X* is on the current node, the rule fails. All other formulae are left intact.

It is possible to specify an arbitrary number of principal formulae as in the rule *Mp* (below). If the same meta-variable (*A* is the example) is used in different principal formulae, the TWB will attempt to unify with all other instances of the same variable in other principal formulae: that is, the pattern `{ A } ; { A` $\rightarrow$ `B }` will match the formulae `a ; a` $\rightarrow$ `b` , but not the formulae `a ; c` $\rightarrow$ `b` since `a` does not unify with `c`.

```
RULE Mp
{ A } ; { A -> B }
==================
      B
END
```

This example defines an invertible rule that selects from the current node two non-empty principal formulae, leaving all other formulae in the node intact. The first principal formula A must also unify with the A in the formula schema $A \rightarrow B$. Also A and $A \rightarrow B$ do not appear in the denominator. If the specified pattern does not match any formula in the current node, the rule fails.

To stop the proof search in the TWB, we use the directives *Close* or *Open* or *Stop*. These are interpreted by the TWB engine to change the value of the variable `status`, stopping the visit procedure and triggering backtracking. For example, the *Id* rule in *CPL* stops the search procedure if a positive and a negative occurrence of the same formula is detected.

```
RULE Id
{ A } ; { ~ A }
================
  Close
END
```

This example defines an invertible rule with two principal formulae. If the numerator pattern-matches the current node, the procedure will stop and backtrack, setting the variable `status` to "Close".

6.1.3.2 Branching Rules

The Rule *Or* below, specifies a universal branching rule (see Section 4.3.2). By using the symbol "|", the TWB machinery implicitly defines a branching condition that is checked upon backtracking after the visit of the left branch. If the value of the `status` variable returned by the left branch is "Close", then the right branch will be explored. Otherwise the right branch is not explored and the visit procedure will backtrack.

```
RULE Or
{ A v B }
==========
  A | B
END
```

This example defines an invertible rule with a non-empty principal formula $A \vee B$. The rule then has two denominators composed of, respectively, A and B, but with all formulae other then $A \vee B$ left intact.

The (K)-rule shown below defines an existential branching rule. Since the rule is declared as non-invertible, the TWB machinery will handle backtracking implicitly trying all different partitions. In this particular case the implicit branch condition will force the next (implicit) branch to be explored only if all the previously explored branches were open (eg. the `status` variable is set to "Open"). Note that in the (K)-rule we explicitly specify the set Z as an anonymous set to explicitly remove it from the denominator. The default behaviour of the TWB is to leave all formulae non explicitly matched by any pattern intact in the denominator(s).

```
RULE K
{ Dia A } ; Box X; Z
--------------------
  A ; X
END
```

This example defines a non-invertible rule that partitions the current node into three disjoint sets so that the set Z is box-free. The set Z is explicitly discarded, and *Box X* is "unboxed" to add X to the denominator.

As discussed in Section 4.3.3, the (K)-rule can also be expressed as follows:

```
RULE KRec
{ Dia A } ; Box X; Dia Y; Z
==========================
  A ; X || Box X; Dia Y

END
```

This example defines an invertible rule with existential branching. Two denominators are defined: the left one by the set $A;X$ and the right one by all the other Box- and Diamond-formulae except *Dia A*. The set Z is discarded since it will be *Box*-free and *Dia*-free.

The definition of the (**KRec**)-rule effectively removes the non-deterministic choice (in this particular case) by allowing for the strategy to recursively re-applying the same rule until a closed branch is found or no more diamonds are available. The use of the double bar "||" sets an implicit condition that is checked on backtracking to make sure that the second branch is explored only if the value of the `status` variable in the first branch is "Open".

6.1.3.3 Rules with Side-Conditions, Branch-Conditions and History Actions

A rule can be associated with two sets of conditions: rule conditions and branch conditions. Rule conditions, specified by the keyword `COND`, define a list of side conditions to be checked after a node has been pattern-matched as specified in the numerator. If all the side conditions are true, then the tentative partition is accepted. Otherwise the partition is not considered and the pattern matching algorithm proceeds until a successful partition is found. Conversely, branch conditions, specified by the keyword `BRANCH`, are executed on backtracking to check whether to explore the next branch (if there is one) or to backtrack further. It is possible to specify a list of branch conditions, one for every branch and each set of branch conditions will have access to all variables returned by any branch already explored. An attempt to access a variable related to a branch yet to be visit will generate a run-time error.

In order to modify histories, we can declare a list of actions associated with each rule. If no actions are specified, all declared histories will be passed unmodified from parents to leaves. History actions are identified by the keyword `ACTION` using the following BNF grammar:

```
rule_action    ::=  "ACTION" "[" branch_action list "]" "END"
branch_action  ::=  "[" action list "]"
action         ::=  UIDENT ":=" function ";"
function       ::=  LIDENT "(" function list ")"
                    | UIDENT
```

We give an example to illustrate the use of side conditions, branch conditions and history actions. Assume we have declared two histories, `Boxes` and `Diamonds`, both of type `Set of Formula`. We give a recursive version of Heuerding's rule for the logic $S4$ below [Heuerding et al. 1996]. In Listing 6.3 we give the `TWB` implementation of it. In the following we associate the histories Π and Σ respectively with the histories `Diamonds` and `Boxes` in Listing 6.3. The functions `notin` and `add` are OCaml functions defined in a support library and behave as follows:

`notin(Dia A, Diamonds)`: checks if the formula `Dia A` $\in$ `Diamonds`;

`add(Box X,Boxes)`: adds the set of formulae identified by `Box X` to the history `Boxes`.

$$S4 \; \frac{\Diamond\varphi;\Diamond\Delta;\Box\Gamma;\Lambda \; :: \; \Pi,\Sigma}{\varphi;\Gamma \; :: \; \{\Diamond\varphi\}\cup\Pi\cup\Diamond\Delta\,,\,\Sigma \quad || \quad \Diamond\Delta;\Box\Gamma \; :: \; \{\Diamond\varphi\}\cup\Pi\,,\,\Sigma} \; \Diamond\varphi\notin\Pi$$

Listing 6.3: S4 Rule a là Heuerding

```
RULE S4
  { Dia A } ; Dia Y ; Box X ; Z
  ==============================
     A ; X || Dia Y ; Box X

COND notin(Dia A, Diamonds)

ACTION [
     [ Diamonds := add(Dia A,Diamonds);
       Diamonds := add(Dia Y,Diamonds)];

     [ Diamonds := add(Dia A,Diamonds) ]
]

END
```

The tableau version of the original sequent calculus along with the full `TWB` implementation[1] is given in Section 7.1.1.

6.1.3.4 Rules with Synthesized Variables

Each node has a number of variables defined by the user associated to it. The value of these variables is modified by using user defined actions that are defined in the `BACKTRACKING` section of a rule definition. Variables can be manipulated on backtracking, after all children of a branching rule are visited according to the branch conditions. During the visit, the partial results of the exploration of the sub-tree below the current node are stored on the stack in a list and can be accessed by the branch conditions to determine when to stop the visit procedure. The syntax to defined these function is the same as in the `ACTION` section. In particular, variables can be accessed with the following syntax:

`lid@`*int*: where `lid` is the variable identifier and *int* is an integer identifying the branch number (starting from 1). The result is a single element.

`lid@last`: where `lid` is the variable identifier and `last` is a keyword to identify the result of the previously visited branch. The result is a single element.

[1] This rule is slightly different from the ($S4H$) rule in Section 7.1.1 in the handling of the *Box* formulae.

`lid@all`: where `lid` is the variable identifier and `all` is a keyword to identify all variables for all branches. In this case the result is a list.

The syntax `lid@last` is particularly useful when the number of branches in not known in advance as in the specification of the (K)-rule with implicit backtracking (cf. page 74).

For example, in Listing 6.4, we show a version of the *Or* rule that modifies the value of a variable bj using the function `mergelabel`. This version of the *Or* rule is used to implement a version of the back-jumping algorithm and is described in Section 7.1.3.3.

Listing 6.4: Implementation of Back-jumping

```
RULE Or
          { A v B }
=================================
fixlabel(Idx,A)  |  fixlabel(Idx,B)

ACTION     [[ Idx := inc(Idx) ]; [ Idx := inc(Idx) ]]
BRANCH     [ backjumping(Idx,bj@1) ]
BACKTRACK [ bj := mergelabel(bj@all,status@last) ]
END
```

Thus, each denominator calls the function `fixlabel` which accepts a history `Idx` and a formula `a` and "fixes" the label of `a` by adding the index stored in the history `Idx`. In this example the history `Idx` is used to associate each disjunction to a unique index. In each action, the value of the history `Idx` is incremented. Since the value of `Idx` is stored on the stack, the result of the left action will increment `Idx` by one, while the right action will increment `Idx` by a value related to the number of disjunctions expanded in the visit of the left branch.

6.1.4 Defining the Strategy

A strategy specifies the order in which rules are applied during the proof search. The tactic language adopted in the TWB has the following syntax: The basic tactic is a rule. It succeeds if the rule is applicable to the current node, otherwise it fails. Tactics can be combined in two ways: in alternation $t_1 \mid t_2$ or in sequence $t_1;t_2$. A tactic can be repeated until it fails, that is, until no more tactic rules are applicable: we write $(t)^*$. See Section 5.2.4 for a more detailed description. The following BNF grammar defines a strategy in the TWB.

```
strategy    ::= "STRATEGY" ":=" tactic_def
tactic      ::= "tactic" "(" tactic_def ")"
tactic_def  ::= UIDENT
              | tactic_def "|" tactic_def
              | tactic_def ";" tactic_def
              | tactic_def "*"
              | LIDENT
```

A tactic is an OCaml expression that can be bound to identifiers as usual by using the OCaml `let` construct. A strategy is a TWB directive that is declared by the keyword `STRATEGY`.

In the following example we specify a simple strategy for the basic modal logic K. Assume rules *Id*, *False*, *And*, *Or* and K are all declared in the `TABLEAU` section. The strategy specifies that we first apply all the invertible rules, until no such rules are applicable to the current node, then we apply the K rule once, then we start over again. This definition is interpreted by the strategy function (cf. Section 5.2.4) and repeated for every branch of the proof tree according to the visit algorithm. In the following example `saturate` is an OCaml identifier bound to a tactic.

```
let saturate = tactic ( (Id | False | And | Or)* )
STRATEGY := ( ( saturate | K )* )
```

This strategy definition is equivalent to the following "inline" definition:

```
STRATEGY := ( ((Id | False | And | Or)* | K)* )
```

In certain situations, it is essential to apply a rule (or a set of rules) if no other (set of) rules are applicable. For example, if we want to build a counter model for a propositional logic formula from a failed attempt to build a closed tableau, we need to collect all the propositional atoms in the last non-contradictory node after all connectives have been broken and all rules have been applied. Such a rule can be seen as a *meta-rule* as its role is not to break connectives, but to execute an action outside the tableau procedure. For example, assume we have declared a rule to collect propositional atoms named $Collect$. Then the following strategy will execute this rule once when no other rules are applicable.

```
STRATEGY := ( (Id | False | And | Or)* | Collect )*
```

6.1.5 Formula Manipulation

The `TWB` defines the keyword `term` to allow us to manipulate formulae (according to the concrete syntax defined in the connectives section) as OCaml expressions. Terms can also be pattern matched as any other OCaml data type. This is achieved by extending the OCaml grammar to seamlessly manipulate formulae otherwise extraneous to the language. Alongside the standard OCaml syntax, we added the following extensions to manipulate formulae:

Atoms: An atom is declared either using an identifier with the first letter capitalized (eg. P, Q, P0, ...), or by using the syntax $p(i)$, where p is a lower case identifier and i is an OCaml expression of type integer. For example, $p(1)$.

Identifiers: An identifier has the same meaning as an OCaml identifier. Terms can be bound to identifiers using the `let` operator and can be used within term expressions.

`TWB` **terms:** Terms are used as expressions or patterns. In particular, we can use an OCaml function to build terms within a `term`: by writing `term ( a v [f ()])`, the resulting term is one where the first disjunct is given by the OCaml identifier a, while the second disjunct is the result of the OCaml expression $f()$.

Substitution: If `p` is an identifier bound to a `TWB` term, and `a` and `c` are `TWB` terms then the expression `a{p/c}` is a new term where all occurrences of `p` in `a` are simultaneously substituted with `c`.

For example, the following function, given a formula *a* and an integer *n*, returns a new formula of the form $\Box ... \Box a$ with *n* occurrences of $\Box$. That is, a function call to `mbox term(Falsum) 2` will return a term of the form `Box Box Falsum`.

```
let rec mbox t = function
    |0 -> t
    |n -> mbox (term ( Box t )) (n-1)
```

In Listing 6.5, we give a function to compute the modal depth of a formula to illustrate the use of pattern matching when working with `TWB` terms. We use `term ( F )`, where `F` is a pattern that matches `TWB` terms, to specify an OCaml pattern. We use the standard pattern matching mechanism and `max` *n m* is the function that returns the maximum of *n* and *m*.

Listing 6.5: Function to Compute the Modal Depth of a Formula

```
let rec modaldepth = function
    |term ( Verum )    -> 0
    |term ( Falsum )   -> 0
    |term ( Dia a )
    |term ( Box a )    -> (modaldepth a) + 1
    |term ( ~ a )      -> modaldepth a
    |term ( a & b )
    |term ( a v b )
    |term ( a -> b )
    |term ( a <-> b ) -> max ( modaldepth a ) ( modaldepth b )
    | _                -> failwith "modaldepth"
```

The `NEG` and `PP` directives can be used in the `TWB` to declare two rewrite functions with type $formula \rightarrow formula$, respectively to specify a negation function and a pre-processing function used by the prover. The negation function is necessary because the `TWB` does not make any assumptions about the syntax of the logic and therefore about the semantic meaning of each connective. An example of a negation formula is below. See Appendix 8.2.2.3 for an example of a negation normal form procedure.

```
let neg = function term ( a ) -> term ( ~ a )
NEG := neg
```

The pre-processing hook provided (`PP`) can be used to re-write the input formula into a normal form. For example, in order to reduce branching and to shorten the proof tree, the theorem prover `leanTAP` [Beckert and Posegga 1995] orders the input formula by counting the number of disjunctions. `leanTAP` accomplishes this by pre-processing the input formula and re-ordering conjunctions and disjunctions by considering the number of disjunctions and the built-in Prolog lexicographic ordering.

6.1.5.1 Caching

The TWB implements a caching mechanism to avoid re-computation of identical parts of the proof tree. The user can decide to enable caching for a restricted number of rules, by specifying the keyword (cache) in the rule definition. By default the caching mechanism is enabled, but only the nodes generated by rules that are marked by the (cache) keyword will be indexed in the cache table. This is done to minimize the memory footprint of the proof to give the user the flexibility of selectively enable caching only for "important" rules. For example, in the rule below, we specify that each denominator of a *K* rule must be stored in the cache.

```
RULE K
{ Dia A } ; Box X ; Z
-----------------------
  A ; X
END (cache)
```

The form of caching implemented in the TWB is of course not optimal as it is based on syntactic equality and it does not exploit logic-specific characteristics. Moreover, to maximise the effect of the caching mechanism, the user should normalize the input formulae via a normal form procedure (cf. Section 6.1.5).

6.1.5.2 Exit Functions

It is possible to define a custom exit function to return a human-readable output for the prover. By default, the exit function returns the value of the implicit variable status which can be "Close" or "Open". However, this can be overwritten by the user, for example, when the decision procedure is not binary or to execute an arbitrary function at the end of the visit procedure or to customise the output of the search procedure. In Listing 6.6, we define an exit function that inspects the value of the variable status and returns a custom string. The exit function is declared by the directive EXIT.

Listing 6.6: Example of the exit function

```
let exit = function
    |"Close" -> "The tableau is Closed"
    |_       -> "The tableau is Open"

EXIT := exit (status@1)
```

6.2 Building and Using a Theorem Prover for a Calculus

In this section we present the TWB command line interface, a minimalistic application to run provers built within the TWB. The current interface is very basic and will be improved in future work (cf. Section 8.2.

6.2.1 Building the Prover

The TWB is built on top of the OCaml programming language and it uses the OCaml tool-chain to compile a logic module into an application. In particular, a logic module is parsed by using a Camlp4 [de Rauglaudre 2003] syntax extension (cf. Section 5.5), that extends the OCaml language, and defines the syntax of the TWB. Camlp4 is the ocaml preprocessor and it is integrated into the OCaml tool-chain.

To save the user from the tedious task of compiling a logic module by hand, we wrote a small application, `twbcompile`, that given a logic file, resolves its dependencies with other modules and compiles and links it to the TWB library. Different programs from the OCaml toolchain are required: `ocamlopt` is the OCaml compiler; `ocamldep` is a utility to query an object about its dependencies; `camlp4o` is the OCaml preprocessor. The compilation process has four steps:

camlp4o: the logic module is translated into OCaml source code;

ocamlopt: the source code is compiled into an object;

ocamldep: all dependencies are resolved, generating all objects;

ocamlopt: all objects are linked together with the TWB library to generate the executable.

6.2.2 Executing the Prover

The result of the compilation of a logic module is a command line program (the *prover*). The prover (`pc` is the program generated by the compilation of the logic module for pc.ml). It accepts formulae from standard input, or from a file, and returns the result of the proof. The input file is composed of one formula per line with comments identified by the symbol `#`. Figure 6.1 shows a typical session, where the formula `a → b & c` is passed as input from the command line to a prover for *CPL*. Note that the given formula is negated by default.

By default, each prover accept a number of arguments to define its behaviour. Table 6.2 shows the basic flags accepted by a generic prover. The prover outputs the result of the proof and details about the (user) time and the total number of rule applications.

Figure 6.2 shows the trace of the proof. For each rule application, we display the name of the rule (ie. `And`) followed by a *flow identifier*. A flow identifier is a tuple of integers used to maintain a parent-child relationship between two nodes in the proof tree. For example, when a linear rule is applied, and *i* is the index associated to the parent, the flow identifier will be of the form `i → i+1`. When a branching rule is applied, the prover will generate two (or more,

```
> echo "a -> b & c" | ./pc
Proving: ~ ( a -> b & c )
Time: 0.00
Result: Open
Total rule applications: 2
```

Figure 6.1: Simple TWB session.

`--nopp`	Disable the pre-processor function specified in the logic module
`--noneg`	Disable the negation function specified in the logic module
`--trace`	Print the proof trace proof trace
`--time`	Set a timeout for the current session. If the input file contains more then one formula, each proof will have the same timeout
`--verbose`	Print additional information regarding the proof
`--nocache`	Disable the caching mechanism

Table 6.2: Runtime flags.

depending on the cardinality of the rule) flow identifiers, one for each branch. Consequently, if an `Or` rule with index *i* is applied, the first branch will have rule identifier `i` $\rightarrow$ `i+1`, while the second branch will have rule identifier `i` $\rightarrow$ `i+d+1`, where *d* is the identifier of the last rule applied on the first branch before backtracking. Flow identifiers are used to navigate the proof tree more easily and to provide spacial information to display the proof tree by using a GUI (see Section 8.2 for more details about the GUI).

```
> echo "a v ~ a" | ./pc -trace
Proving: ~ ( a v ~ a )
And ( 0 -> 1 )
((~ a))
(a)

Id ( 1 -> 2 )

Time: 0.00
Result: Close
Total rule applications: 2
```

Figure 6.2: Session with trace enabled.

Case Studies and Experimental Results

"You're bound to be unhappy if you optimise everything."

Donald Knuth

In this chapter we analyse four case studies to highlight the versatility and expressive power of the `TWB`. In Section 7.1.1 we give the implementation of the basic modal logic $S4$ and in Section 7.1.2 the propositional linear temporal logic $PLTL$. In Section 7.1.3 we show how to implement a number of known optimisation techniques. In Section 7.1.4, we show how we implemented the benchmark suite presented in [Heuerding and Schwendimann 1996] using the `TWB` framework. Lastly, in Section 7.2, we show empirical results obtained by comparing several optimisations for a tableau prover for the logic $S4$ and a comparison of the `TWB` with the `LWB`. In this chapter we assume familiarity of the decision procedures and techniques used.

7.1 Case Studies

7.1.1 Basic modal logic S4

We now give an implementation of a tableau calculus for the basic modal logic $S4$. We use a tableau version of the sequent calculus from [Heuerding et al. 1996].

In the following we use two histories, Π for the diamond history and Σ for the box history. Intuitively, the diamond history contains certain diamond formulae found in the branch from the root to the current node. The box history contains all the boxes found in the branch from the root to the current node.

We assume that the input formula is in negation normal form. The propositional rules of the calculus are standard. We assume also that, when not explicitly noted, histories are passed from parents to children un-changed. A node in the tableau is always of the form $\Gamma :: \Pi, \Sigma$, where Γ is a set of formulae and Π and Σ are the diamond and box histories. We use "$\ldots$" to denote un-important parts of the node which remain unchanged in passing from numerators to denominators:

Listing 7.1: S4 Connectives

```
CONNECTIVES
  And,   "_&_",   Two;
  Or,    "_v_",   Two;
  Imp,   "_->_",  One;
  DImp,  "_<->_", One;
  Not,   "~_",    Zero;
  Dia,   "Dia_",  Zero;
  Box,   "Box_",  Zero
END

HISTORIES
  (DIAMONDS : Set of Formula := new Set.set);
  (BOXES    : Set of Formula := new Set.set)
END
```

$$(\bot)\ \frac{\varphi; \neg\varphi\ ::\ \ldots}{\bot} \qquad (\wedge)\ \frac{\varphi \wedge \psi\ ::\ \ldots}{\varphi; \psi\ ::\ \ldots} \qquad (\vee)\ \frac{\varphi \vee \psi\ ::\ \ldots}{\varphi\ ::\ \ldots \mid \psi\ ::\ \ldots}$$

The modal tableau rules shown below are an adaptation of Heuerding's sequent rules [Heuerding et al. 1996]. In particular the (T)-rule is applied to a principal formula $\Box\varphi$ only if its sub-formula φ is not in the history Σ. If $\varphi \notin \Sigma$, the denominator is created from the current rule by deleting $\Box\varphi$ and adding φ. The formula φ is added to the history Σ and the diamond history Π is emptied.

Since the Kripke models for the logic $S4$ are transitive, box formulae always accumulate. In the calculus, the (T)-rule is applied only if the immediate sub-formulae φ of the box formula $\Box\varphi$ under consideration is "new" (not present in the box history) in the branch, thus exploiting transitivity. Intuitively, if a box formula is new in a branch, we force the procedure "to recompute", by forgetting (empting Π) any information about the diamond formulae found so far.

Similarly, the $S4$ rule is applied only if its diamond formula $\Diamond\varphi$ is not present in the diamond history Π. If $\Diamond\varphi \notin \Pi$, then the formula φ and the content of the box history Σ is added to the denominator. The context Δ is discarded and the formula $\Diamond\varphi$ is the diamond history Π. The box history Σ is unchanged.

$$(T)\ \frac{\Box\varphi\ ::\ \Pi, \Sigma}{\varphi\ ::\ \emptyset\,,\ \{\varphi\} \cup \Sigma}\ \varphi \notin \Sigma \qquad (S4)\ \frac{\Diamond\varphi; \Delta\ ::\ \Pi, \Sigma}{\varphi; \Sigma\ ::\ \{\Diamond\varphi\} \cup \Pi\ ,\ \Sigma}\ \Diamond\varphi \notin \Pi$$

Note. The rules described in this section assume a decision procedure that first applies all classical rules and the (T)-rule until they are no longer applicable; and then which applies the modal rule $S4$. Therefore, in the $(S4)$-rule, the set Δ will consist of all diamond formulae in the node except $\Diamond\varphi$ and literals only. In particular, since the (T)-rule removes all box formulae from the node and stores their immediate sub-formulae in the box history Σ, then the set Δ will be box-free.

First we declare a number of connectives and the two histories *BOXES* and *DIAMONDS* in Listing 7.1. The rules in Listing 7.2 are a simple transcription of the tableau rules above. The strategy definition is standard: we execute all *CPL* rules and the (T)-rule first and then the $(S4)$-rule.

Listing 7.2: S4 Rules

```
TABLEAU
  RULE S4
  { Dia P } ; Z
  ------------
  P ; BOXES

  COND notin(Dia P, DIAMONDS)
  ACTION [ DIAMONDS := add(Dia P,DIAMONDS) ]
  END

  RULE T
  { Box P } == P
  COND notin(P, BOXES)
  ACTION [
        BOXES := add(P,BOXES);
        DIAMONDS := emptyset (DIAMONDS)]
  END

  RULE Id
  { A } ; { ~ A } == Close
  END

  RULE And
  { A & B } == A ; B
  END

  RULE Or
  { A v B }
 ==========
    A | B
  END
END

STRATEGY := ( (False|Id|And|T|Or)* | S4 )*
```

In particular we use three functions to handle histories:

add(formula list, history): accepts a list of formulae and a history object and returns the history object with the formula list added to it.

emptyset(history): returns an empty history object.

In Listing 7.2 we declared the $S4$ rule as not-invertible by using “–” lines, and therefore the backtracking associated with the choice of the diamond formula is

implicit. Conversely, the rule *S4H* in Listing 7.3 treats the backtracking explicitly.

We note that our adaptation of the calculus is not as efficient as the original sequent version from Heuerding [Heuerding et al. 1996]. The reason is that the $(S4)$-rule does not share information among its branches. Consequently, it is possible for the same diamond formula to be "reduced" along different branches of the tableau. Clearly this is not necessary, hence wasteful from a computational point of view. A more efficient version of the $(S4)$-rule can be written as follows, where we make the recursion step explicit and we ensure that the first branch does not "reduce" any diamonds in the set Δ since they will be considered instead in the second branch.

$$S4H \;\; \frac{\Diamond\varphi;\Diamond\Delta;\Lambda \;::\; \Pi,\Sigma}{\varphi;\Sigma \;::\; \{\Diamond\varphi\}\cup\Pi\cup\Diamond\Delta \,,\, \Sigma \,||\, \Diamond\Delta \;::\; \{\Diamond\varphi\}\cup\Pi \,,\, \Sigma} \; \Diamond\varphi\notin\Pi$$

We show the implementation of the rule *S4H* in Listing 7.3. Note that we use $==$ to declare this rule "invertible". This is a consequence of the fact that by using this recursive definition, we effectively determinised the $(S4H)$-rule. The strategy guarantees that the rule is applied after all *CPL* rules.

Listing 7.3: S4 Rule a lá Heuerding

```
RULE S4H
  { Dia A } ; Dia Y ; Z
  ==============================
     A ; BOXES || Dia Y

COND notin(Dia A, DIAMONDS)

ACTION [
     [ DIAMONDS := add(Dia A,DIAMONDS);
       DIAMONDS := add(Dia Y,DIAMONDS)];

     [ DIAMONDS := add(Dia A,DIAMONDS); ]
]

END
```

7.1.2 Propositional linear temporal logic

In this section, we describe an implementation of the decision procedure for $PLTL$ from [Schwendimann 1998]. We give a brief description of the calculus to illustrate the `TWB` implementation but we assume familiarity with the syntax and semantics of $PLTL$ (cf. Section 2.1). For a full overview of the decision procedure refer to [Schwendimann 1998]. In order to describe the decision procedure we first give a few definitions:

Lists: We use $*$ for the concatenation of lists and [] for the empty list. If M is an empty list then we write $len(M)$ for the length of M and $M[i]$ for the ith element of M with $1 \leq i \leq len(M)$. If M is a list of tuples, then we write $M[i]_j$ to denote the projection to the jth element of tuple $M[i]$.

Node: A node is a triple $(\Gamma, (Ev, Br), (n, uev))$ also written $\Gamma \ :: \ Ev, Br \ :: \ n, uev$ where:

Γ is a set of formulae in negation normal form.

Ev is a set of formulae in negation normal form representing the currently satisfied eventualities.

Br is a list of pairs representing the current branch with the ith pair corresponding to the Γ and Ev parts of the ith-state on this branch.

n is a natural number indicating the "earliest" state reachable from the current one via a loop.

uev is a set of eventuality formulae in negated normal form. It represents the unfulfilled eventualities of the current branch (loop).

(Ev, Br) are histories, while (n, uev) are variables.

According to the above definitions $\Gamma \ :: \ Ev; Br \ :: \ n, uev$ is the extended notion for an abstract node. With this machinery in place, the rules of $PLTL$ are given in Figure 7.1. Note that if "..." appears in the same position in the numerator and the denominator(s) of a rule, then we mean that the corresponding parts are the same. These rules are from [Schwendimann 1998].

The branching used in $PLTL$ is neither universal nor existential, but both branches must always be explored. This is due to the fact that it is impossible to determine the status of a given branch in isolation: rather, a branching node can only be declared "open" or "closed" when everything below it has been explored.

In the following we give the `TWB` implementation of the tableau rules for $PLTL$. This implementation is a refinement of the work conducted in [Thornton 2004] using version 2.0 of the `TWB`. All functions associated with the calculus are written in OCaml within the `TWB` framework and provided in the Appendix 8.2.2.3

First we declare the connectives and histories used in the calculus according to the $PLTL$ syntax (cf. Section 2.1) in Listing 7.4 in the usual way. Note that `X` is a connective for the connective $\bigcirc$ in the $(Next)$-rule in Figure 7.1.

Terminal rules:

$$(false)\ \frac{false;\Gamma\ ::\ Ev,Br\ ::\ n,uev}{n := len(Br)\ \ uev := \{false\}}$$

$$(contr)\ \frac{\varphi;\neg\varphi;\Gamma\ ::\ Ev,Br\ ::\ n,uev}{n := len(Br)\ \ uev := \{false\}}$$

$$(loop)\ \frac{\Delta;\bigcirc\Sigma\ ::\ Ev,Br\ ::\ n,uev}{}$$

where in $(loop)$ there exists an i, $1 \leq i \leq len(Br)$, such that:

1. $\Delta;\bigcirc\Sigma = Br[i]_1$
2. $n := i-1$ and $uev := \{\varphi\ U\ \psi \mid \varphi\ U\ \psi \in \Sigma$ and $\psi \notin (\cup_{j=i+1}^{len(Br)} Br[j]_2 \cup Ev)\}$.

α-rules:

$$(\alpha)\ \frac{\alpha;\Gamma\ ::\ \ldots\ ::\ \ldots}{\alpha_1;\alpha_2;\Gamma\ ::\ \ldots\ ::\ \ldots}$$

β-rules:

$$(\vee)\ \frac{\varphi \vee \psi;\Gamma\ ::\ Ev,Br\ ::\ n,uev}{\varphi;\Gamma\ ::\ Ev,Br\ ::\ n_1,uev_1 \mid \psi;\Gamma\ ::\ Ev,Br\ ::\ n_2,uev_2}$$

$$(U)\ \frac{\varphi U \psi;\Gamma\ ::\ Ev,Br\ ::\ n,uev}{\psi;\Gamma\ ::\ Ev \cup \{\psi\},Br\ ::\ n_1,uev_1 \mid \varphi;\bigcirc(\varphi\ U\ \psi);\Gamma\ ::\ Ev,Br\ ::\ n_2,uev_2}$$

where in $(\vee)$ and (U):

1. $n := min(n_1, n_2)$.
2. $m = len(Br) - 1$.

$$uev := \begin{cases} \emptyset & if\ uev_1 = \emptyset\ or\ uev_2 = \emptyset, \\ \{false\} & if\ n_1 > m\ and\ n_2 > m\ (and\ uev_1 \neq \emptyset,\ uev_2 \neq \emptyset), \\ uev_1 & if\ n_1 \leq m\ and\ n_2 > m\ (and\ uev_2 \neq \emptyset), \\ & or\ if\ uev_2 = \{false\}, \\ uev_2 & if\ n_2 \leq m\ and\ n_1 > m\ (and\ uev_1 \neq \emptyset), \\ & or\ if\ uev_1 = \{false\}, \\ uev_1 \cap uev_2 & otherwise. \end{cases}$$

Next time rule:

$$(Next)\ \frac{\Delta;\bigcirc\Sigma\ ::\ Ev,Br\ ::\ \ldots}{\Sigma\ ::\ Ev := \emptyset, Br * ((\Delta;\bigcirc\Sigma),Ev)\ ::\ \ldots}$$

Figure 7.1: The rules in the calculus *PLTL* from [Schwendimann 1998]

Listing 7.4: PLTL Connectives and Histories

```
CONNECTIVES
  DImp,        "_<->_", Two;  And,        "_&_",  One;
  Or,          "_v_",   One;  Imp,        "_->_", One;
  Until,       "_Un_",  One;  Before,     "_Bf_", One;
  Falsum,      Const;   Verum,      Const;
  Not,         "~_",    Zero;
  Next,        "X_",  Zero;
  Generally,   "G_",  Zero;
  Sometimes,   "F_",  Zero
END

HISTORIES
  (Ev      : Set of Formula := new Set.set) ;
  (Br      : List of (Set of Formula * Set of Formula)
                 := new Listoftupleset.listobj);
  (uev     : Set of Formula := new Set.set);
  (status  : String := new Set.set);
  (n       : Int := new Set.set default 0)
END
```

Then, in Listing 7.5, we give the terminal rules. All rules are straightforward except the (*Loop*)-rule. The (*Loop*)-rule application is subject to a side condition to detect loops. If the side condition of the (*Loop*)-rule is true (we are in a loop), then the procedure will stop and backtrack. The (*Loop*)-rule also computes the value of the variables `n` and `uev`. The functions used in Listing 7.5 have the following meaning:

loop_check(xa, xb, z, br): implements the side condition of the (*loop*)-rule in Figure 7.1. It accepts the content of the node via *xa*, *xb*, *z* and checks if the node is already in the list *br*. If this is the case it returns *false*, meaning the rule is not applicable. *true* otherwise.

setclose(): returns a set containing the term *Falsum*.

setclosen(Br): returns an integer that is equal to the length of *Br*.

setuev(xa, xb, z, ev, br): returns a set containing the *U*-formulae that are not satisfied on the sub-branch from the looping point to the current (leaf) node. This computes the value of *uev* for the loop rule as in Figure 7.1.

setn(xa, xb, z, br): computes the index *n* of the loop rule.

In Listing 7.6 we give the linear (α) rules. The functions used in Listing 7.6 are the following:

emptyset(h): returns *h* with all its elements removed.

push(xa, xb, z, ev, br): accepts the content of the node in $fl1$, $fl2$, $fl3$, a set of eventualities in *ev* and a list *br* and returns the list *br* with the tuple $(fl1 \cup fl2 \cup fl3, ev)$ appended to it.

Listing 7.5: PLTL Terminal Rules

```
RULE False
Falsum == Stop
BACKTRACK [ uev := setclose(); n := setclosen (Br) ]
END

RULE Contr
{ A } ; { ~ A } ==  Stop
BACKTRACK [ uev := setclose(); n := setclosen (Br) ]
END

RULE Loop
{ X A } ; X B ; Z == Stop
COND [ loop_check(X A, X B, Z, Br) ]
BACKTRACK [
    uev := setuev(X A, X B, Z, Ev, Br);
    n   := setn (X A, X B, Z, Br)
]
END
```

Listing 7.6: PLTL Linear Rules

```
RULE Next
{ X A } ; X B ; Z
================
    A ; B

ACTION [
    Ev := emptyset(Ev);
    Br := push(X A, X B, Z, Ev, Br)
]
END

RULE And { A & B } == A ; B END
RULE Ge { G A } == A ; X (G A) END
RULE Before {A Bf C} == nnf (~ C) ; A v X (A Bf C) END
```

A possible immediate optimisation for the branching rules is to exploit the condition enforced by the function that computes the value of the variable *uev* in the β-rule in Figure 7.1. According to Section 4.3.2, we classify this type of branching as conditional branching, where the second branch is explored only if a certain condition is met. In particular, the second branch is never explored if, according to the function to establish the value of the variable *uev* in the β-rule, the variable $uev_1 = \emptyset$. In this case the result of the exploration of the second branch would be irrelevant.

Listing 7.7: PLTL Branching Rules

```
RULE Until
          { C Un D }
=============================
    D ||| C ; X ( C Un D )

ACTION      [ Ev := add(D, Ev) ]
BRANCH      [ not_emptyset(uev@1) ]
BACKTRACK [
    uev := beta(uev@1, uev@2, n@1, n@2, Br);
    n   := min (n@1, n@2)
]
END

RULE Or
{ A v B }
=========
 A ||| B

BRANCH      [ not_emptyset(uev@1) ]
BACKTRACK [
    uev := beta(uev@1, uev@2, n@1, n@2, Br);
    n   := min (n@1 , n@2)
]
END
```

In Listing 7.7 we give the optimized branching (β) rules. Both rules implement a conditionally branching rule (via the ||| symbol) where the condition makes sure that the second branch is not explored if the variable `uev` is empty. The functions used in Listing 7.7 are the following:

beta(uev1, uev2, n1, n2, Br): implements the function for computing the value of *uev* via clause 2 of the β-rule in Figure 7.1.

min(n1, n2): returns the minimum of $n1$ and $n2$.

not_emptyset(uev): checks if the set *uev* is empty.

The strategy used in Schwendimann's calculus is to apply the terminal rules first. If these fail then it applies all invertible rules and then the $(Next)$-rule once. This strategy is enforced in our implementation as follows:

```
let terminal = tactic ( (Id | False | Loop) )
let saturate = tactic ( (And | Before | Ge | Or | Until)* )
STRATEGY := ( terminal | saturate | Next)*
```

7.1.3 Optimisations for Tableau-based Theorem Provers

This section outlines several techniques used to optimise calculi for non-classical logics and their implementation in the `TWB` framework.

Different empirical studies have been carried out, particularly in the domain of *knowledge representation* proving in the last decade the usefulness of optimisation techniques in to tableau methods [Horrocks 1997; Haarslev and Moller 2001; Hustadt and Schmidt 1998].

We can classify different optimisation techniques on two levels:

Logic level optimisations are independent of the machinery used in the theorem prover and focus on the abstract theoretical aspects of the logic under consideration. Simplification, normal forms, semantic branching, back-jumping, blocking methods, etc. are examples of logic level optimisations.

Machine level optimisations make assumptions about the implementation of the theorem prover and its low level data structures: e.g. caching, efficient data structure optimisations, lazy unfolding, etc

In the design of the `TWB`, we had to trade efficiency for generality. Other theorem provers, FACT [Horrocks and Patel-Schneider 1998] or the LWB [Heuerding et al. 1996], to name a few, adopt a different approach designing highly optimised theorem provers for a restricted number of logics. However, despite its generality, the `TWB` still allows the user to experiment with different optimisation techniques. Thus, the `TWB` can be used as a test-bench to compare different approaches and to develop new optimisation techniques. We present empirical results obtained by comparing the optimisations given in this section in Section 7.2.1.

7.1.3.1 Simplification

Simplification is a well known technique which helps to reduce both the length and number of branches in the proof tree (for references see [Massacci 1998]). Simplification is particularly useful to exploit redundancy in formulae by recording inconsistencies during the proof search. The procedure of simplifying φ with ψ is substantially a re-writing procedure employing two stages:

$$\varphi[\psi] \rightarrow_{simp} \varphi' \rightarrow_{bool} \varphi''$$

In the first stage we replace all occurrences of the formula ψ in φ with $\top$; in the second stage we eliminate the propositional constant $\top$ and $\bot$ using standard boolean reductions. Simplification subsumes other techniques like DPLL [Davis and Putnam 1960], Boolean constraint propagation [McAllester 1990] and KE [D'Agostino and Mondadori 1994].

Since simplification is logic specific, in the `TWB`, the simplification procedure is implemented as an OCaml function (`simpl` in Listing 7.8 and Listing 7.9), and

Listing 7.8: CPL with Simplification

```
SIMPLIFICATION := simpl

TABLEAU
  RULE Id { A } ; { ~ A } == Close END
  RULE False Falsum == Close END

  RULE And
  {A & B} ; X
  ====================
  A[B]; B[A] ; X[A][B]
  END

  RULE Or
  { A v B } ; X
 ==============================
     A ; X[A] | B ; X[B]
  END
END
```

declared by using the `SIMPLIFICATION` directive. Listing 7.8 shows a version of the tableau calculus for *CPL* that employs simplification to reduce the search space. The syntax $a[\phi]$, used in the *And* and *Or* rule in Listing 7.8, is syntactic sugar that is translated to a call to the simplification function with arguments ϕ and a in Listing 7.9. The function `boolean` used in Listing 7.9 implements a simple re-write function according to Table 7.1 (details not shown).

7.1.3.2 Semantic Branching

The naive method to satisfy a disjunction $\varphi \vee \psi$ is to explore the branch with φ first and if this branch closes, explore the branch for ψ. This is commonly referred to as syntactic branching and is rather inefficient. If the branch for φ leads to a clash, this information is lost when exploring the branch for ψ.

Semantic branching adds $\neg\varphi \wedge \psi$ to the second branch if φ leads to a clash. This makes the information that φ is unsatisfiable explicit and possibly prunes the search space because a tree in which φ is satisfiable is not tested again.

The drawback of semantic branching is that can add complex formulae to the tree which can slow down the procedure. Empirical evidence however shows that the

$\neg\varphi \vee \varphi \to \top$	$\varphi \vee \top \to \top$	$\varphi \vee \bot \to \varphi$	$\varphi \vee \varphi \to \varphi$
$\neg\varphi \wedge \varphi \to \bot$	$\varphi \wedge \top \to \varphi$	$\varphi \wedge \bot \to \bot$	$\varphi \wedge \varphi \to \varphi$
$\neg\bot \to \top$	$\neg\top \to \bot$		

Table 7.1: Rewrite rules.

Listing 7.9: Simplification procedure

```
let rec simpl phi a =
    let rec aux phi a = match a with
        |term (~ b) when b = a -> term(Falsum)
        |term (~ b)   -> term ( ~ [aux phi b] )
        |term (b & c) -> term ( [aux phi b] & [aux phi c] )
        |term (b v c) -> term ( [aux phi b] v [aux phi c] )
        |_ when phi = a -> term(Verum)
        |_ when phi = (nnf (term ( ~ a ))) -> term(Falsum)
        |_ -> a
    in
    boolean (aux phi a)
```

benefit of semantic branching often speeds up the search procedure. Moreover, coupling semantic branching with simplification usually offsets this drawback.

Implementing this optimisation in the `TWB` is trivial. The *Or* rule is modified as in Listing 7.10. Since for this example, we assume that these rules use negation normal form, it is necessary to process the formula $\neg\varphi$ to push the negation down to the atoms in the usual way. The `TWB` in general does not require the use of negation normal form.

7.1.3.3 Back-jumping

Naive backtracking algorithms often explore regions of the search space rediscovering the same contradictions repeatedly. This phenomenon is known as thrashing [Mackworth 1992]. Back-jumping is a technique to avoid un-important re-computations and therefore minimizing thrashing. The back-jumping method forces the visit algorithm to backtrack to the last branching point of the search tree which is relevant to the failure of the current branch. This is achieved by maintaining an *assumptions set* for each branch which contains all formulae which have contributed to the closure of a branch. Assumptions sets are then used to avoid the investigation of any branch which contains one of the assumptions sets since this branch is guaranteed to close. For an overview of these techniques and extensions such as dependency directed back-jumping, refer to [Dechter 1990; Stallman and Sussman 1976].

Listing 7.10: Semantic Branching

```
RULE Or
      { A v B }
==============================
 A | B ; nnf_term(~ A)
END
```

Listing 7.11: Back-jumping Rules for *CPL*

```
RULE Id
{ A } ; { ~ A } == Close
BACKTRACK [ bj := addlabel(A, ~ A) ]
END

RULE Or
      { A v B }
==========================
fixlabel(A) | fixlabel(B)

BRANCH     [ backjumping(A,bj@1) ]
BACKTRACK [ bj := mergelabel(A, bj@all) ]
END
```

To implement back-jumping in the `TWB` we first decorated every formula with a list of integers. This is done by modifying the default internal representation of formulae in the user data type library[1]. Then we declare a new variable `bj` that we use to record the assumptions sets of the formulae involved in a clash and a history `Idx` to keep track of the index associated with the rule as follows:

```
HISTORIES
(Idx : Int     := new Singleton.set default 0);
(bj  : ListInt := new Set.set default [])
END
```

The only rules in the *CPL* calculus that require any modifications are the (Id)-rule and the (Or)-rule as shown in Listing 7.11. We use four OCaml functions to handle the assumptions sets. In particular:

addlabel(formula, formula): accepts two formulae and returns the assumption set containing the labels associated with the input formulae;

fixlabel(idx, formula): accepts a formula and the index of the current (Or)-rule, and returns the same formula adding the index of the current disjunction to its assumptions set;

backjumping(idx, int list): accepts the index of the current disjunction and a variable containing the assumptions set generated by the first branch, and returns true if the index associated with the current disjunction is not in the assumptions set;

mergelabel(int list list): accepts the list of all assumptions sets and returns the union of these lists.

[1] In the further work (cf. Section 8.2) we outline how this step will be lifted to the user level in the next version of the `TWB`, removing the necessary low-level modifications that are currently necessary.

Listing 7.12: Support Functions for Implementing Benchmarks.

```
let rec mbox t = function
    |0 -> t
    |n -> mbox (term ( Box t )) (n-1)

let rec list2disj = function
    |[] -> term ( Falsum )
    |[h] -> h
    |h::t -> term (h v [list2disj t])

let rec range lo hi =
        if lo > hi then [] else lo :: range (lo+1) hi

let axiomD  = term (Box p(0) -> Dia p(0))
let axiomT  = term (Box p(0) -> p(0))
let axiomD2 = term (Dia Verum)
let axiomA4 = term (Box p(0) -> Box Box p(0))
let axiomB  = term (p(0) -> Box Dia p(0))
```

7.1.4 Benchmarks

In this section, we give the code to implement the benchmarks presented in [Heuerding and Schwendimann 1996]. The `TWB` offers the possibility to easily manipulate logical formulae and to write complex re-write functions (cf. Section 6.1.5). We took inspiration from the `LWB` syntax in an effort to make it easier for users to adopt the `TWB`.

As an example, we give the implementation of the benchmark named `k_d4_p` in Section 6.3 in [Heuerding and Schwendimann 1996]. In Listing 7.12 we declare three support functions and several axioms. In particular:

`mbox` puts `n` boxes in front of the formula `t`;

`list2disj` converts the list `l` of formulae into a disjunction of all its elements;

Listing 7.13: Implementation of the Benchmark k_d4_n

```
let k_d4_p_aux n i = term (
      [mbox axiomT n] v
    ~ [mbox axiomD2 i] v
    ~ [mbox axiomA4 i] v
    ~ [mbox term ( axiomA4{Dia p(0)/p(0)} ) i] v
    ~ [mbox axiomB i] v
    ~ [mbox term ( axiomB{~ p(0)/p(0)} ) i]
)

let k_d4_p n =
        nnf (list2disj (List.map (k_d4_p_aux n) (range 1 n)))
```

`range` returns a list of integers ranging from `lo` to `hi`.

Listing 7.13 implements the benchmark `k_d4_p` using the notion of substitution described in Section 6.1.5.

7.2 Experimental Results

In this section we first describe an empirical analysis of implementations of different provers for the logic *S*4 by using several optimisation techniques described in Section 7.1.3, and second, we compare a prover generated with the `TWB` framework with one from the `LWB`.

Note. We conducted our benchmarks on the following system: Hardware: Pentium 4 (2.4 Ghz), 1GB RAM, 1GB swap space; Software: Debian GNU/Linux OS, OCaml 3.09.2.

7.2.1 Comparing Optimization Techniques

Table 7.2 and 7.3 compare different optimizations for the logic *S*4. In particular we run our benchmarks using the following variation of the basic tableau prover.

$s4$ is the naive tableau prover for *S*4 as described in Section 7.1.1. The only optimization is caching;

$s4 - nocache$ is as $s4$ but without using caching;

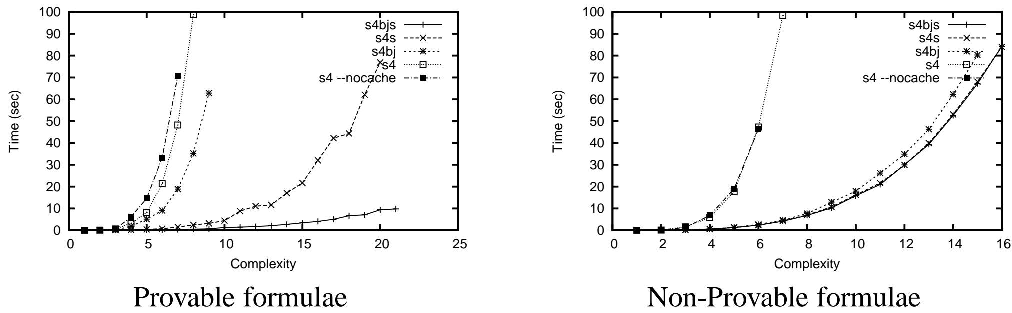
$s4s$ is as $s4$ but using caching and simplification and semantic branching;

$s4bj$ is as $s4$ but using caching and back-jumping and semantic branching;

$s4sbj$ uses all the previously mentioned optimization.

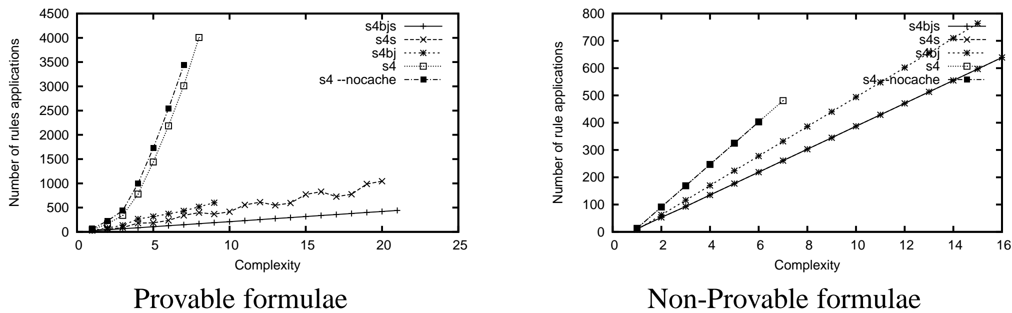
Table 7.2 shows the time measurements considering the benchmark class $s4_s5$ as in [Heuerding and Schwendimann 1996]. Conversely Table 7.3 shows the number of rule applications performed during the test. It is clear that using all optimizations pays off time-wise and by reducing the search space considerably. These results show also the advantage of using simplification and in particular when combined with back-jumping. Semantic branching, in our tests, only marginally improved performance and therefore we did not include a test for it alone.

The tables presented here are just a sample of our experiments to highlight the potential of the `TWB` to be used as a test-bench to experiment with a range of optimization techniques. It is important to note that the amount of work involved in implementing these optimization techniques in the `TWB` is minimal as described in Section 7.1.3. Our results confirm those in the literature. For an analysis of these techniques and empirical evaluation see [Hustadt and Schmidt 1998; Hustadt and Schmidt 2002; Heuerding and Schwendimann 1996].



Provable formulae   Non-Provable formulae

Table 7.2: S4_S5 (time)



Provable formulae   Non-Provable formulae

Table 7.3: S4_S5 (rules)

7.2.2 Comparison with the LWB

The TWB has been engineered to be a generic and flexible framework, rather than to produce highly optimised theorem provers. Nonetheless, it is important to compare it against other well established theorem provers. Since the TWB itself makes few or no logic-specific assumptions, the duty to make logic-specific optimisations is left to the user.In the following, we compare the TWB with the LWB using the benchmark suite presented in [Heuerding and Schwendimann 1996]. These benchmarks have been re-implemented in the TWB framework (see Section 7.1.4).

Tables 7.4, 7.5 and 7.6 in this section show, for each class, how many formulae of each set could be solved. For each class, we generated all formulae with complexity up to 21 and we set a timeout of 100 seconds. We used the same set of formulae for each prover. For the LWB we recorded as a failed attempt to prove a formula either if the prover timed out after 100 seconds or, in certain cases, if the prover failed with an error (`Error : lwb stack too small`). In the latter case we did not investigate the reason for such error. For these benchmarks, we used an optimized version of our tableau calculi for K, KT and $S4$ that uses simplification, semantic branching, back-jumping and caching.

The results are very encouraging. In particular, the results in Table 7.4 show that the TWB can compete with the LWB in terms of pure speed for the logic K. In our tests, the LWB used considerably more memory then the TWB for certain classes of

class	twb	lwb
k_branch_p	6	15
k_d4_p	**11**	10
k_dum_p	**18**	16
k_grz_p	**21**	17
k_lin_p	**21**	14
k_path_p	**16**	14
k_ph_p	4	7
k_poly_p	**19**	11
k_t4p_p	**21**	10

class	twb	lwb
k_branch_n	4	11
k_d4_n	**17**	6
k_dum_n	19	21
k_grz_n	21	21
k_lin_n	**21**	5
k_path_n	**14**	12
k_ph_n	6	7
k_poly_n	20	21
k_t4p_n	**21**	8

Table 7.4: Benchmarks comparing the `TWB` and `LWB` for modal logic *K*

class	twb	lwb
kt_45_p	**10**	9
kt_branch_p	5	15
kt_dum_p	17	18
kt_md_p	4	21
kt_grz_p	21	21
kt_path_p	5	13
kt_ph_p	4	7
kt_poly_p	3	21
kt_t4p_p	7	7

class	twb	lwb
kt_45_n	**9**	8
kt_branch_n	3	11
kt_dum_n	14	14
kt_md_n	5	5
kt_grz_n	21	21
kt_path_n	4	11
kt_ph_n	5	7
kt_poly_n	2	2
kt_t4p_n	2	8

Table 7.5: Benchmarks comparing the `TWB` and `LWB` for modal logic *KT*

class	twb	lwb
s4_45_p	8	21
s4_branch_p	14	15
s4_grz_p	**21**	18
s4_ipc_p	21	21
s4_md_p	8	19
s4_path_p	4	8
s4_ph_p	3	7
s4_s5_p	21	21
s4_t4p_p	4	21

class	twb	lwb
s4_45_n	5	17
s4_branch_n	4	11
s4_grz_n	21	21
s4_ipc_n	21	21
s4_md_n	6	7
s4_path_n	3	6
s4_ph_n	4	7
s4_s5_n	16	21
s4_t4p_n	1	21

Table 7.6: Benchmarks comparing the `TWB` and `LWB` for modal logic *S*4

formulae. We also note that when caching is not used, the `TWB` runs by using no more then 32MB of stack space, and virtually no heap space. As further work, we plan to optimize the caching mechanism to limit memory usage. Conversely Table 7.6 shows that the `LWB` is more efficient for the logic $S4$. The poor performances of the `TWB` for $S4$ can be attributed to the fact we do not use any further optimisation to exploit logic-specific properties. It is left to future work to explore this avenue.

Chapter 8

Related and Future Work

In this chapter, in Section 8.1 we compare the TWB with other similar theorem provers in detail. In Section 8.2, we outline ideas that we want to explore in the immediate future and we set a long term plan for the development of the TWB.

8.1 Related Work

Direct provers like LWB [Heuerding 1996], FaCT [Horrocks and Patel-Schneider 1998] or RACER [Haarslev and Möller 2001] are clearly superior if they handle the sought-after logic, and the LWB, in particular, handle intuitionistic logic and a long list of particular modal logics. But programming a new calculus into the above mentioned provers is difficult for anyone except their authors. FACT is written in C[1] and FACT++ [Tsarkov and Horrocks 2004] is a re-implementation of FACT in C++. Both provers deal with a specific family of description logic. The LWB is written in C++. Writing a new logic module with the LWB, despite its modular and particularly clean architecture, requires knowledge of its programming language and familiarity with the intricacies of the memory management model of the underlying operating system. The LWB API to build new logical modules has many caveats related to the internal representation of formulae, making the task of defining a new module error prone, in particular to non expert programmers. Moreover, modifying existing modules in the LWB can be also a daunting task considering the number of lines of code necessary to implement *CPL* and all related support functions, 26K, or the module for *K*, 6K. Additionally, the LWB, by design is tailored to implement propositional logic and would require substantial modifications to incorporate first order reasoning. Conversely, the TWB does not have such restrictions and implementing first order logic would not require any considerable modifications to the proof engine (for example [**?**]).

Translational provers like MSPASS [Hustadt and Schmidt 2000] are equally superior to the TWB in terms of performance if the logic is first-order definable and falls into a decidable fragment like the two-variable fragment or the guarded fragment. MSPASS can be used with highly efficient theorem provers for first order logic like

[1]The original prototype of FACT was written in Lisp

Vampire [Riazanov and Voronkov 2002] or SPASS [Weidenbach et al. 2002]. Vampire has been proved to be one of the fastest provers for first-order logic in recent years. SPASS can even handle "second-order" logics like **G** and **Grz** by using a non-standard translation into first-order logic that mimics the traditional tableau calculi rules for these logics. But, *a priori*, MSPASS does *not* provide a decision procedure for a given decidable first-order-definable modal logic. The SCAN algorithm [Gabbay and Ohlbach 1992] for second-order quantifier elimination can often find first-order equivalents for many second-order relational conditions. But once again, this does not, *a priori* lead to a decision procedure unless the first-order equivalents fall into a decidable subset of first-order logic.

Blast_tac [Paulson 1999], is an Isabelle tactic to perform classical reasoning. It provides fairly basic facilities for designing new rules, and even allows certain rules to be marked as "un-doable" (non-invertible), but it does not allow history mechanisms or further optimisation techniques like simplification. To be fair, Blast_tac deliberately trades completeness for versatility since it is designed to be used in an interactive setting like Isabelle and "completeness is hardly relevant to interactive proofs" [Paulson 1999].

Interactive theorem provers like Isabelle [Paulson 1993] are not directly comparable with the TWB. Even if it is possible to potentially encode in Isabelle all logics that we can implement in the TWB, we think that the overhead involved in learning these complex theorem provers makes it more practical, for a non-technical user, to adopt the TWB.

The TWB is closest to lotrec, a generic theorem prover for non-classical logics [Gasquet et al. 2005]. lotrec allows users to define logical connectives, production rules, and a search strategy. While lotrec is based on labelled semantic tableau, it departs from their traditional formulation in treating a tableau as a rooted acyclic graph, rather than as a tree, of nodes which represent worlds in the Kripke semantics and edges that represent the accessibility relations. Nodes in the graph are labelled with sets of formulae and edges can be labelled with any terms. lotrec provides two types of rules: structural rules and propagation rules. Structural rules are used to create new nodes and edges in the graph while propagation rules are responsible for removing or adding formulae to nodes. This separation makes it easy to implement logics with geometric frame proprieties. lotrec and the TWB have a similar syntax for strategy definitions: both allow users to program the order of how rules are applied.

The main difference between the TWB and lotrec is that lotrec allows the user to *explicitly* refer to the underlying relational semantics, whereas the TWB (currently) does not. For example, the weak-directness frame-conditions $(\forall x, y, z. \exists w. xRy \wedge xRz \Rightarrow xRw \wedge yRw)$ for the logic **S4.2** can be coded explicitly in lotrec by referring explicitly to "formulae" like $R(x, y)$ in the rules. In the TWB it must be coded implicitly using a particular form of cut on super-formulae [Goré 1999]. Thus lotrec is biased towards semantics while the TWB is biased towards proof theory.

In fact the `lotrec` execution model is different from that of the `TWB`: `lotrec` uses a global data structure to store the reachability relation, while in the `TWB` it is implicit and global data structure (such as histories) are controlled by the user.

8.2 Future Work

In this dissertation, we described the stable version of the `TWB` (August 2006). A new version is in preparation. We now give an overview of the major changes we are working on, as well as a number of ideas for future releases.

8.2.1 Short Term Improvements

The development version is mainly geared to make the `TWB` more modular, by decoupling any specific implementation from the core via specialized modules. This strict separation of concerns will transform the `TWB` into a generic platform to develop arbitrary theorem provers.

The stable version of the `TWB` has been engineered to implement tableau based theorem provers. Consequently, the strategy and the visit module in the core library are tied to the idea of building a tree using a recursive algorithm and using the strategy to guide the exploration of the search tree.

New releases of the `TWB` will remove this restriction in order to implement other decision procedures, such as resolution procedures, sequent calculi (in this case only minor modifications to the actual algorithm will be necessary), hypersequents, etc. In Figure 8.1 we show a diagram with the new architecture. Two main modifications to the actual architecture are needed.

First, the core will be transformed into an empty shell containing only generic data types and the definition of the overall architecture. Each prover type will be defined in terms of this generic architecture and contain the specific implementation (in terms of Object Oriented Programming, we can think of the core as a generic interface and of each prover as the implementation of this interface).

Second, we need to abstract the basic type system. At the moment, basic data types are declared in the user data type library. This is far from optimal as to add a new basic type such as a new type constructor to represent multi-modal logics, it is necessary to recompile the entire library. The new version will remove this problem by shifting the data type declaration to the logic modules and transform the user data type library into a collection of generic containers such as lists, sets, ordered sets, etc. which the experienced user can use to build more complex ones.

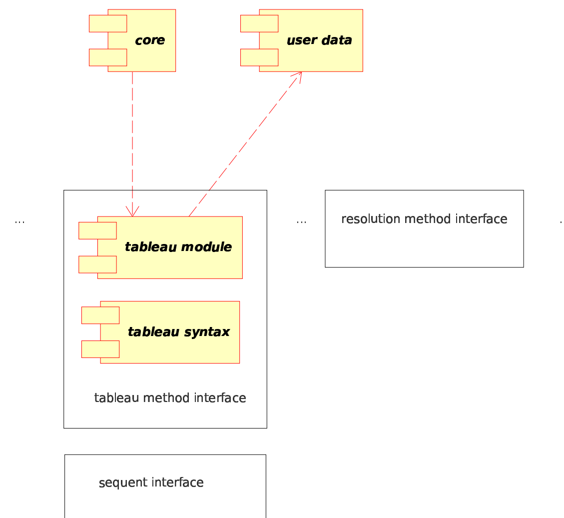


Figure 8.1: `TWB` development version architecture.

8.2.2 Long Term Improvements

8.2.2.1 Alternative formalisms

The `TWB` provides a generic platform to develop different theorem provers. At present, mainly due to time constraints, we only implemented the syntax and machinery to design standard tableau calculi. However, there are two natural extensions that can be implemented within the `TWB` with only minimal modifications: labelled tableaux and sequents.

Labelled tableau will necessitate the introduction of additional, external and global data structures. It would be even possible, for example, to simulate the `lotrec` algorithm [Castilho et al. 1997]. Sequent calculi will require no major modifications apart for a new syntax to write rules in sequent style.

The `TWB` framework could also be used in principle to experiment with resolution techniques. This avenue would, of course, require us to write a complete new visit function that operated iteratively instead of recursively, and to use the `TWB` in a radically different way. It should also be possible to implement $DPLL/Davis-Putnam-Logemann-Loveland$ algorithms.

8.2.2.2 Strategy

The strategy language and visit algorithm in the stable version of the `TWB` does not allow us to define a decision procedure where all choices are "don't know" non-deterministic in an intuitive way. For example, it is not possible to declare all *CPL* rules as non invertible and generate all possible derivation trees. While this

is not a problem in classical logic, it becomes a restriction when implementing non-classical logics where two rules are non-invertible.

For example, if we consider tableau-sequent calculi intuitionistic logics, there are two avenues to solve this problem. Firstly, it is possible to replace all non-invertible rules with one new rule: this will often serve the purpose, but at the expense of modularity and clarity. During the development of the `TWB`, several Honours students used the `TWB` to implement such logics. We came to the conclusion that this approach has too many drawbacks. In particular this approach, while handling logics with more than one invertible rule that use histories and variables, poses an unnecessary burden on the user and complicates the implementation of the logic considerably.

The second solution to this problem is to introduce a conditional alternation operator. The current alternation operator fails if the rule is not applicable. This type of failure is local. In order to determine if a successful rule application generates a "successful" sub-tree, we need to take into account a more global notion of failure.

We say that a tactic fails *locally* if no rules are applicable to a node n. A tactic fails *globally* if the tactic does not fail locally, but the proof tree generated by applying the tactic to node n does not respect a logic-specific condition (for example, if the tableau did not close after a certain number of steps). A conditional alternation tactic t, written as $(t_1 \mid c \mid t_2)$, fails if both tactic t_1 and t_2 fail locally, or if t_1 or t_2 fail globally with respect to the condition c.

Conversely, implementing multi-modal logics should not require any major modifications to the engine. For example, to implement the normal modal logic with two diamond, `Dia1` and `Dia2` is only necessary to "compose" the individual K rules for each modality with the following rule:

```
RULE Compose
Dia1 A1 ; Dia2 A2 ; Box1 X1 ; Box2 X2
=======================================
    Dia1 A1 ; Box1 X1 || Dia2 A2 ; Box2 X2

COND [ not_emptylist (Dia1 A1) ; not_emptylist (Dia2 A2) ]
END
```

Then the strategy for the product logic will simply be as follows, where $K1$ and $K2$ are two instances of the (K)-rule as defined in Section 6.1.3.

```
let saturate = tactic ( (Id|False|And|Or)* )
STRATEGY ( saturate ; Compose | (K1 | K2) )*
```

It is left to further work how to handle an arbitrary number of modalities.

A other possible improvement to the strategy module is to amend the actual definition of the MState monad to use the Call/cc operator explicitly instead of hiding this operation within the visit function. This will enhance modularity and make the code more elegant and maintainable.

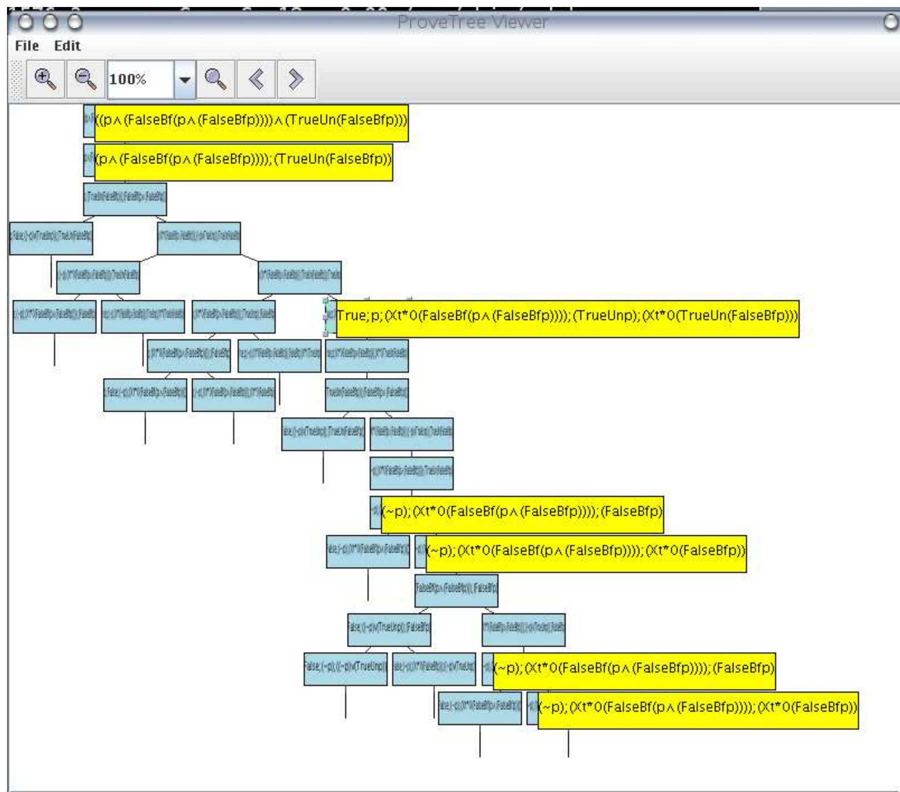


Figure 8.2: TWB 2.0 Gui Prototype Screenshot

8.2.2.3 Graphical User Interface for the TWB

The TWB interface is command line based. It allows the user to explore the proof tree trace, but it is too simple to be truly effective in analysing the proof tree. To take the TWB "mainstream" and therefore make it really easy for non-technical people to use, we envision three components:

Web interface: Different well established theorem provers offer a simple web interface to run the prover without going to the trouble of installing any software. We conducted preliminary studies to build a user interface which is possible to run via the Firefox web browser[2], based on the XUL language [Deakin 2006]. Such an interface will allow the user to select a logic module and to prove logical formulae. The interface should also provide facilities to explore the proof tree interactively and to extract counter models.

Proof Tree Viewer: One key characteristic of the tableau method is its user friendliness in terms of the possibility of exploring the tableau tree. Interactive proof editors have facilities to explore the derivation tree. However, when the tree becomes too wide (as with EXPTIME temporal logics) a naive

[2] http://www.mozilla.com/firefox/

approach makes such tools unusable. We are working on a prototype to explore a proof tree interactively via the web. Figure 8.2 shows a snapshot of an early prototype of the proof tree viewer (no longer available).

By using a client-server architecture we can decouple the generation of the proof tree (carried out by the `TWB`) with its display. Since displaying the whole tree at once is not feasible or useful if the tree is too big, the user will have the possibility of selecting the sub-tree to focus on and to apply different filters to isolate only relevant information: for example, to show only transitional rule applications. The tree viewer will be integrated with the web interface, possibly as a Firefox extension[3].

Model Viewer: The possibility of generating a counter-model is also a very important characteristic of the tableau method. Apart from the technicalities involved in generating a counter-model from an open tableau, to display the model and to correctly represent the graph is a challenging problem in itself involving graph theory and graph visualization. Ideally, we would like to provide a way to overlay the model over the proof tree, hiding and showing different components and having the possibility of zooming at different levels: for example, at the lower zoom resolution, only the model with atoms that are true or false at every world will be displayed, and at the highest level of zoom, it would be possible to visualize each formula and its decomposition in the tableau. We did not realize any prototype of this third component.

[3] `https://addons.mozilla.org/firefox/extensions/`

Appendix

Appendix

Listing 8.1: CPL tableau rules

```
CONNECTIVES
  DImp, "_<->_", Two;
  And, "_&_",  One;
  Or,   "_v_",  One;
  Imp, "_->_", One;
  Not, "~_", Zero;
  Falsum, Const;
  Verum, Const
END
TABLEAU
  RULE Id { A } ; { ~ A } == Close END
  RULE False Falsum == Close END
  RULE And { A & B } == A ; B END
  RULE Or { A v B } == A | B END
END
PP := Pclib.nnf
NEG := Pclib.neg
STRATEGY (Id|False|And|Or)*
```

Listing 8.2: PC nnf procedure

```
let rec nnf = function
    |term ( a & b ) -> term ( [nnf a] & [nnf b] )
    |term ( ~ ( a & b ) ) ->
            term ( [ nnf term ( ~ a )] v [nnf term ( ~ b )] )

    |term ( a v b ) -> term ([nnf a] v [nnf b])
    |term ( ~ ( a v b ) ) ->
            term ( [ nnf term ( ~ a )] & [nnf term ( ~ b )] )

    |term ( a <-> b ) ->
            term ( [nnf term ( a -> b )] &
                   [nnf term ( b -> a )] )
    |term ( ~ ( a <-> b ) ) ->
            term ( [nnf term ( ~ (a -> b) )] v
                   [nnf term ( ~ (b -> a) )] )

    |term ( a -> b ) -> nnf term ( (~ a) v b )
    |term ( ~ (a -> b) ) -> nnf term ( a & (~ b) )
    |term ( ~ ~ a ) -> nnf a

    |term ( ~ Atom ) as f -> f
    |term ( Atom ) as f -> f

    |term (Verum) -> term (Verum)
    |term (Falsum) -> term (Falsum)
    |term (~ Verum) -> term (Falsum)
    |term (~ Falsum) -> term (Verum)

    |f -> failwith (Printf.sprintf "nnf:%s" (Twblib.sof(f)))
```

Listing 8.3: Backjumping Functions.

```
let inc (idx) = idx + 1

let rec uniq = function
    |[] -> []
    |h::t -> h:: uniq (List.filter (fun x -> not(x = h)) t)

let addlabel (tl1,tl2) =
    match List.hd tl1,List.hd tl2 with
    |`LabeledFormula(l1,t1),`LabeledFormula(l2,t2) ->
                                                        uniq(l1@l2)
    |_ -> failwith "backjumping"

let fixlabel (idx,tl) =
    match List.hd tl with
    |`LabeledFormula(l,t) -> [`LabeledFormula(idx::l,t)]
    |_ -> failwith "fixlabel"

let backjumping (idx,intlist) = List.mem idx intlist ;;

let mergelabel (intll, status) =
    if status = "Open" then [] else uniq(List.flatten intll)
```

Listing 8.4: PLTL support Funtions.

```
let push (xa,xb,z,ev,br) =
    let set = (new Set.set)#addlist (xa@xb@z)
    in br#add (set, ev)

let termfalse = `Formula ( term ( Falsum ))
let setclose () = (new Set.set)#add termfalse
let setclosen br = br#length

let beta (uev1, uev2, n1, n2, br) =
  let m = (br#length - 1) in
  if uev1#is_empty || uev2#is_empty then (new Set.set)
  else if (List.hd uev2#elements) = termfalse then uev1
  else if (List.hd uev1#elements) = termfalse then uev2
  else if n1 > m && n2 > m then (new Set.set)#add termfalse
  else if n1 <= m && n2 > m then uev1
  else if n1 > m && n2 <= m then uev2
  else uev1#intersect uev2

let rec index n s l =
    if List.length l > 0 then
        if s#is_equal (List.nth l n) then n
        else
            if n < ((List.length l) - 1)
            then index (n+1) s l
            else failwith "index: core not found"
    else failwith "index: list empty"
```

Listing 8.5: PLTL support Funtions.

```
    let loop_check (xa,xb,z,br) =
    let (br1, br2) = List.split br#elements in
    let set = (new Set.set)#addlist (xa@xb@z) in
    List.exists (fun s -> set#is_equal s) br1

let setuev (xa,xb,z,ev,br) =
    let (br1, br2) = List.split br#elements in
    let set = (new Set.set)#addlist (xa@xb@z) in
    let i = index 0 set br1 in
    let rec buildset counter bl acc =
        if counter < ((List.length bl) - 1) then
            let bl_i = (List.nth bl counter) in
            buildset (counter+1) bl (acc@(bl_i#elements))
        else acc
    in
    let loopset = ((ev#elements) @ (buildset (i+1) br2 [])) in
    let uev =
        set#filter (function
            |`Formula term ( X ( c Un d ) ) ->
                    not(List.mem (`Formula d) loopset)
            |_ -> false
        )
    in
    uev

let setn (xa,xb,z,br) =
    let (br1, br2) = List.split br#elements in
    let set = (new Set.set)#addlist (xa@xb@z)
    in index 0 set br1
```

Listing 8.6: Tableau for the logic UB

```
CONNECTIVES
  DImp, "_<->_", Two;
  And, "_&_",  One;
  Or,  "_v_",  One;
  Imp, "_->_", One;
  ExX, "ExX_", One;
  AxX, "AxX_", One;
  ExG, "ExG_", One;
  ExF, "ExF_", One;
  AxG, "AxG_", One;
  AxF, "AxF_", One;
  Not, "~_",   Zero;
  Falsum, Const;
  Verum, Const
END
HISTORIES
  (Fev : Set of Formula := new Set.set) ;
  (Br  : List of ( Set of Formula * Set of Formula )
        := new Listoftupleset.listobj) ;
  (uev : Set of ( Formula * Int ) := new Setoftermint.set) ;
  (fev : Set of ( Formula * Int ) := new Setoftermint.set) ;
  (status : String := new Set.set)
END
open UbFunctions
open UbRewrite
TABLEAU
  RULE Id
  { A } ; { ~ A } == Stop
  BACKTRACK [ uev := setclose (Br) ]
  END

  RULE False
  Falsum == Stop
  BACKTRACK [ uev := setclose (Br) ]
  END

  RULE Axf
      { AxF P }
  ===================
   P ||| AxX AxF P

  ACTION [ [ Fev := add(AxF P, Fev) ]; [] ]
  BRANCH [ [ not_empty(uev@1) ] ]
  BACKTRACK [ uev := setuev_beta(uev@1, uev@2, Br) ]
  END
```

Listing 8.7: Tableau for the logic UB

```
RULE Exf
    { ExF P }
===================
 P ||| ExX ExF P

ACTION [ [ Fev := add(ExF P, Fev) ] ; [] ]
BRANCH [ [ not_empty(uev@1) ] ]
BACKTRACK [ uev := setuev_beta(uev@1, uev@2, Br) ]
END

RULE Exx
{ ExX P } ; ExX S ; AxX Y ; Z
=============================
    P ; Y ||| ExX S ; AxX Y

COND [ loop_check(P, Y, Br) ]
ACTION [ [
    Br  := push(P, Y, Fev, Br);
    Fev := emptyset(Fev)
] ; [] ]
BRANCH [ [ not_false(uev@1) ; not_empty_list(ExX S) ] ]
BACKTRACK [ uev := setuev_pi(uev@1, uev@2, Br) ]
END (cache)

RULE D
ExX X ; {AxX P} == ExX Verum ; AxX P
COND [ is_empty_list(ExX X) ]
END

RULE Loop
ExX X ; AxX Y == Stop
COND [ not_empty_list(ExX X) ]
BACKTRACK [ uev := setuev_loop(X, Y, Fev, Br) ]
END

RULE Or
{ A v B }
=========
 A ||| B
BRANCH [ [ not_empty(uev@1) ] ]
BACKTRACK [ uev := setuev_beta(uev@1, uev@2, Br) ]
END
```

Listing 8.8: Tableau for the logic UB

```
  RULE And A & B == A ; B END
  RULE Axg AxG P == P ; AxX AxG P END
  RULE Exg ExG P == P ; ExX ExG P END
END

let exit (uev) =
    match uev#elements with
    |(termfalse,_)::[] -> "Closed"
    |[] -> "Open"
    |_ -> "Closed"
;;

PP := nnf_term
NEG := neg_term
EXIT := exit (uev@1)

let saturation = tactic (
        (Id | False | And | Axg | Exg | Or | Axf | Exf )* )
let modal = tactic ( ( saturation | D | Exx )* )
STRATEGY ( (modal | Loop)* )
```

Bibliography

ABATE, P. AND GORÉ, R. 2003. System description: The tableaux workbench (TWB). In *TABLEAUX*, Lecture Notes in Artificial Intelligence (2003). Springer. (p. 50)

ALBERUCCI, L. 2002. *The modal μ-calculus and the Logic of Common Knowledge*. PhD thesis, University of Berne. (p. 18)

ALBERUCCI, L. AND JÄEGER, G. 2002. About cut elimination for logics of common knowledge. Technical report, Institute for Theoretical Computer Science, University of Berne, Switzerland. (pp. 18, 19)

ANDRÉKA, H., VAN BENTHEM, J., AND NÉMETI, I. 1998. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic 27*, 217–274. (p. 21)

AVRON, A. 1996. The method of hypersequents in proof theory of propositional non-classical logics. In *Logic: Foundations to Applications*, pp. 1–32. Oxford Science Publications. (p. 43)

BECK, K. 2000. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing. (p. 50)

BECKERT, B. AND POSEGGA, J. 1995. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning 15*, 3, 339–358. (p. 78)

BEN-ARI, M., MANNA, Z., AND PNUELI, A. 1981. The temporal logic of branching time. *Acta Informatica 20*, 207–226. (pp. 25, 28, 36)

BIERNACKI, D., DANVY, O., AND MILLIKIN, K. 2005. A dynamic continuation-passing style for dynamic delimited continuations. `http://www.brics.dk/RS/05/16/index.html`. (p. 60)

BOLOTOV, A. AND FISHER, M. 1999. A clausal resolution method for CTL branching-time temporal logic. *JETAI 11*, 1, 77–93. (p. 16)

BORNAT, R. AND SUFRIN, B. 1997. Jape: A calculator for animating proof-on-paper. In *Proceedings of the 14th International Conference on Automated deduction*, Volume 1249 of *LNAI* (July 1997), pp. 412–415. Springer. (p. 41)

BROTHERSTON, J. 2005. Cyclic proofs for first-order logic with inductive definitions. In *TABLEAUX* (2005), pp. 78–92. (p. 45)

CASTILHO, M., FARIÑAS DEL CERRO, L., GASQUET, O., AND HERZIG, A. 1997. Modal tableaux with propagation rules and structural rules. *Fundamenta Informaticae 32*, 3/4. (p. 104)

CHAILLOUX, E., MANOURY, P., AND PAGANO, B. 2000. *Dèveloppement d'applications avec Obective Caml*. O'Reilly. (pp. 7, 8, 55, 61)

Clocksin, W. F. and Mellish, C. S. 1987. *Programming in Prolog*. Springer-Verlag. (p. 56)

D'Agostino, M. and Mondadori, M. 1994. The taming of the cut. Classical refutations with analytic cut. *Journal of Logic and Computation 4*, 285–319. (p. 92)

Danvy, O. and Filinski, A. 1990. Abstracting control. In *LFP* (1990), pp. 151–160. ACM Press. (p. 60)

Davis, M. and Putnam, H. 1960. A computing procedure for quantification theory. *JACM 7*, 201–215. (p. 92)

De Giacomo, G. and Massacci, F. 1996. Tableaux and algorithms for propositional dynamic logic with converse. In *CADE-96*, Volume 1104 of *LNAI* (1996), pp. 613–628. (pp. 19, 20)

de Rauglaudre, D. 2003. Camlp4 - reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`. (p. 80)

Deakin, N. 2006. Xul tutorial. `http://www.xulplanet.com/tutorials/xultu/`. (p. 106)

Dechter, R. 1990. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Journal of Artificial Intelligence 41*, 3, 273–312. (p. 94)

Dixon, C. 1996. Search strategies for resolution in temporal logics. In *CADE-13* (1996), pp. 673–687. Springer-Verlag. (p. 15)

Dyckhoff, R. 1992. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic 57*, 3 (sep), 795–807. (p. 49)

Emerson, A. and Halpern, J. 1983. Sometimes and not never revisited. In *POPL* (1983), pp. 127–140. (p. 12)

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, Volume Volume B: Formal Models and Semantics. MIT Press. (p. 12)

Emerson, E. A. 1996. Automated temporal reasoning about reactive systems. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata* (1996), pp. 41–101. Springer-Verlag New York, Inc. (p. 11)

Emerson, E. A. and Halpern, J. 1986. 'Sometime' and 'Not Never' revisted: On branching versus linear time temporal logic. *JACM*, 151–178. (pp. 12, 13)

Emerson, E. A. and Halpern, J. Y. 1985. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences 30*, 1–24. (pp. 12, 16)

Emerson, E. A. and Jutla, C. S. 2000. The complexity of tree automata and logics of programs. *SIAM Journal Computation 29*, 1, 132–158. (pp. 11, 21)

Fagin, R., Halpern, J., Moses, Y., and Vardi, M. 1995. *Reasoning about Knowledge*. The MIT Press. (pp. 11, 16, 17, 18)

FELLEISEN, M. 1988. The theory and practice of first-class prompts. In *POPL* (1988), pp. 180–190. ACM Press. (p. 60)

FELTY, A. 1993. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning 11*, 1, 41–81. (p. 49)

FILLIATRE, J.-C. AND LETOUZEY, P. 2003. Functors for proofs and programs. (p. 7)

FISCHER, M. J. AND LADNER, R. E. 1979. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Science 18*, 194–211. (pp. 19, 20)

FISHER, M. 1991. A resolution method for temporal logic. In *IJCAI* (1991). (p. 15)

FISHER, M., DIXON, C., AND PEIM, M. 2001. Clausal temporal resolution. *ACM Trans. Comput. Logic 2*, 1, 12–56. (p. 15)

FITTING, M. 1983. *Proof Methods for Modal and Intuitionistic Logics*. Reidel, Dordrecht. (pp. 6, 43, 44, 46)

FREEMAN, J. W. 1995. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Philadelphia. (p. 48)

FRIEDMAN, D. P. 1988. Applications of continuations: Invited tutorial. In *1988 Principles of Programming Languages (POPL88)* (January 1988). (p. 10)

GABBAY, D. AND OHLBACH, H.-J. 1992. Quantifier elimination in second order predicate logic. In *Proc. KR-92* (1992). (p. 102)

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional. (p. 55)

GARRET, R. 2000. Lisp as an alternative to Java. `http://www.norvig.com/java-lisp.html`. (p. 7)

GASQUET, O., HERZIG, A., LONGIN, D., AND SAHADE, M. 2005. Lotrec: Logical tableaux research engineering companion. In *TABLEAUX* (2005), pp. 318–322. (pp. 20, 41, 42, 102)

GENTZEN, G. 1935. Untersuchungen über das logische schließen I and II. *Mathematische Zeitschrift 39*, 176–210 and 405–431. English translation: Investigations into logical deduction, in *The Collected Papers of Gerhard Gentzen*, M. E. Szabo (Ed), pp 68-131, North-Holland, 1969. (p. 43)

GIUNCHIGLIA, F. AND SEBASTIANI, R. 1996. Building decision procedures for modal logics from propostittional decision procedures - the case study of modal K. In *CADE-13*, Volume LNAI of *LNCS* (1996), pp. 583–598. Springer. (p. 41)

GORANKO, V. 2000. Temporal logics of computations. Notes for the European Summer School in Logic, Language and Information. (p. 12)

GORÉ, R. 1999. Chapter 6: Tableau methods for modal and temporal logics. In *Handbook of Tableau Methods*, pp. 297–396. Kluwer Academic Publishers. (pp. 41, 43, 102)

GRÄDEL, E. 1999. Decision procedures for guarded logics. In *Proceedings of 16th International Conference on Automated Deduction*, Volume 1632 of *LNAI* (1999), pp. 31–51. Springer-Verlag. (p. 21)

GRÄDEL, E. 2003. Decidable fragments of first-order and fixed-point logic - from prefix vocabulary classes to guarded logics. (p. 21)

GRÄDEL, E. AND OTTO, E. 1998. On logics with two variables. *Theoretical Computer Science*. (p. 21)

GRÄDEL, E. AND WALUKIEWICZ, I. 1999. Guarded fixed point logic. In *In Proceedings of 14th IEEE Symposium on Logic in Computer Science LICS '99* (1999). (p. 21)

HAARSLEV, V. AND MÖLLER, R. 2001. Description of the racer system and its applications. In *Proceedings International Workshop on Description Logics* (2001), pp. 131–141. (pp. 41, 42, 101)

HAARSLEV, V. AND MOLLER, R. 2001. High performance reasoning with very large knowledge bases: A practical case study. In *IJCAI* (2001), pp. 161–168. (p. 92)

HALPERN, J. AND MOSES, Y. 1985. A guide to the modal logics of knowledge and belief: preliminary draft. In *IJCAI* (1985), pp. 480–490. Morgan Kaufmann. (p. 16)

HAREL, D. 1984. Dynamic logic. In *Handbook of Philosophical Logic Volume II – Extensions of Classical Logic*, pp. 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands. (p. 19)

HAREL, D. AND KOZEN, D. 1982. Process logic: Expressiveness, decidability, and completeness. *Jornal of Computer and System Sciences 25*, 144–170. (p. 19)

HENDERSON, F., CONWAY, T., SOMOGYI, Z., AND JEFFERY, D. 1996. The mercury language reference manual. (p. 56)

HEUERDING, A. 1996. LWBtheory: information about some propositional logics via the WWW. *Logic Journal of the IGPL 4*, 169–174. (pp. iii, 41, 101)

HEUERDING, A., JÄEGER, G., SCHWENDIMANN, S., AND SEYFRIED, M. 1995. Propositional logics on the computer. In *Analytic Tableaux and Related Methods* (1995), pp. 310–323. (p. 42)

HEUERDING, A. AND SCHWENDIMANN, S. 1996. A benchmark method for the propositional modal logics k, kt, and s4. Technical Report IAM-96-015, University of Bern, Switzerland. (pp. 83, 96, 97, 98)

HEUERDING, A., SEYFRIED, M., AND ZIMMERMANN, H. 1996. Efficient loop-check for backward proof search in some non-classical propositional logics. In *Analytic Tableaux and Related Methods* (1996), pp. 210–225. (pp. 74, 83, 84, 86, 92)

HINTIKKA, J. 1962. *Knowledge and Belief*. Cornell University Press. (p. 16)

HINZE, R. 2000. Deriving backtracking monad transformers. *ACM SIGPLAN Notices 35*, 9, 186–197. (pp. 10, 55)

HORROCKS, I. 1997. *Optimising tableaux decision procedures for description logics*. PhD thesis, University of Manchester. (p. 92)

HORROCKS, I. AND PATEL-SCHNEIDER, P. F. 1998. Optimising propositional modal satisfiability for description logic subsumption. *Lecture Notes in Computer Science 1476*. (pp. iii, 41, 48, 92, 101)

HOWE, J. M. 1998. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St Andrews. (p. 44)

HUDAK, P. AND JONES, M. P. 1994. Haskell vs. Ada vs. C++ vs. Awk vs. . . . An Experiment in Software Prototyping Productivity. Technical report (July), Yale. (pp. 7, 50)

HUGHES, G. E. AND CRESSWELL, M. J. 1996. *A New Introduction To Modal Logic*. Routledge. (pp. 5, 6)

HUGHES, J. 1989. Why Functional Programming Matters. *Computer Journal 32*, 2, 98–107. (p. 6)

HUSTADT, U. AND KONEV, B. 2003. Trp++ 2.0: A temporal resolution prover. In *CADE-19*, Volume 2741 of *Lecture Notes in Artificial Intelligence* (2003), pp. 274–278. Springer. (p. 16)

HUSTADT, U. AND SCHMIDT, R. A. 1998. Simplification and backjumping in modal tableau. In *Lecture Notes in Computer Science*, Volume 1397 (1998). (pp. 92, 97)

HUSTADT, U. AND SCHMIDT, R. A. 2000. MSPASS: Modal reasoning by translation and first-order resolution. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference*, Volume 1847 of *LNAI* (2000), pp. 67–71. Springer. (pp. iii, 41, 101)

HUSTADT, U. AND SCHMIDT, R. A. 2002. Scientific benchmarking with temporal logic decision procedures. In *Proceedings of the 8th International Conference in Principles of Knowledge Representation and Reasoning* (2002), pp. 533–544. Morgan Kaufmann. (p. 16)

JANSSEN, G. 1989. Verification using temporal logic: A practical view. In *International Workshop on Applied Methods for Correct VLSI Design* (1989). North-Holland. (p. 15)

JONES, S. P., HUGHES, J., ET AL. 1999. Haskell 98: A non-strict, purely functional language. `http://haskell.org/onlinereport/`. (p. 7)

KANEKO, M. 1999. Common knowledge logic and game logic. *The Journal of Symbolic Logic 64*, 2 (jun), 685–700. (p. 18)

KISELYOV, O., CHIEH SHAN, C., FRIEDMAN, D. P., AND SABRY, A. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not. 40*, 9, 192–203. (pp. 10, 55)

KOZEN, D. AND PARIKH, R. 1983. A decision procedure for the propositional μ-calculus. *Proc. Logics of Programs* 164, 313–325. (p. 20)

KRIPKE, S. 1959. A completeness theorem in modal logic. *Journal of Symbolic Logic 24*, 1–14. (p. 3)

LAMPORT, L. 1981. 'sometimes' is sometimes 'not never'. *Seventh ACM Symposium on Principles of Programing Languages*, 174–185. (p. 12)

LANGE, M. 2002. *Games for Modal and Temporal Logics*. PhD thesis, University of Edinburgh. (p. 16)

LIANG, S., HUDAK, P., AND JONES, M. P. 1995. Monad transformers and modular interpreters. In *POPL* (1995), pp. 333–343. (p. 8)

LICHTENSTEIN, O. AND PNUELI, A. 2000. Propositional temporal logics: Decidability and completeness. *Logic Journal of the IGPL 8*, 1, 55–85. (p. 15)

M D'AGOSTINO AND D GABBAY AND R HÄHNLE AND J POSEGGA ET AL. 1999. *Handbook of Tableau Methods*. Kluwer Academic Publishers. (p. 3)

M. KRETZ, T. S. 2006. Deduction chains for common knowledge. *Journal of Applied Logic*. (p. 18)

MACKWORTH, A. 1992. Constraint satisfaction. In *Encyclopedia of Artificial Intelligence*, Volume 1, pp. 285–293. Wiley Interscience. (p. 94)

MANNA, Z. AND PNUELI, A. 1983. Verification of concurrent programs: A temporal proof system. Technical Report STAN-CS-83-967, Stanford University Department of Computer Science.

MARTIN, A., GARDINER, P., AND WOODCOCK, J. 1995. A tactic calculus - abridged version. *Formal Aspects of Computing 8*, 4, 479–489. (pp. 49, 58)

MASSACCI, F. 1998. Simplification: A general constraint propagation technique for propositional and modal tableaux. *Lecture Notes in Computer Science 1397*. (pp. 44, 92)

MASSACCI, F. 2000. Single step tableaux for modal logics: Computational properties, complexity and methodology. *Journal of Automated Reasoning*. (p. 43)

MCALLESTER, D. 1990. Truth maintenance. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Volume 2 (1990), pp. 1109–1116. AAAI Press. (p. 92)

MOGGI, E. 1989. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5-8 June 1989*, pp. 14–23. Washington, DC: IEEE Computer Society Press. (p. 7)

MOURI, M. 2001. Theorem provers with counter-models and xpe. *Bulletin of the Section of Logic 30*, 2, 79–86. (p. 41)

NIVELLE, H. D., SCHMIDT, R., AND HUSTADT, U. 2000. Resolution-based methods for modal logics. *Logic Journal IGPL*. (p. 41)

OKASAKI, C. 1996. *Purely functional data structures*. PhD thesis, University of Cambridge. (pp. 7, 55)

PAULSON, L. C. 1987. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press. (p. 49)

PAULSON, L. C. 1991. *ML for the working programmer*. Cambridge University Press. (pp. 7, 55)

PAULSON, L. C. 1993. Isabelle: The Next 700 Theorem Provers. *ArXiv Computer Science e-prints*. (pp. iii, 49, 102)

PAULSON, L. C. 1999. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science 5*, 3. (pp. 42, 102)

PRATT, V. 1979a. Models of program logics. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science* (1979), pp. 115–122.

PRATT, V. R. 1979b. Process logic: Preliminary report. In *Proc. 6th Annual ACM Symposium on Principle Of Programming Languages* (Jan. 1979), pp. 93–100. (p. 20)

PRIOR, A. 1957. *Time and Modality*. Oxford University Press. (p. 11)

RANTA, A. 2000. *PESCA–A Proof Editor for Sequent Calculus*. Department of Computing Science, Chalmers University of Technology and University of Gothenburg. (p. 41)

REYNOLDS, M. 2001. An axiomatization of full computation tree logic. *Journal of Symbolic Logic 66*, 3, 1011–1057. (p. 13)

REYNOLDS, M. 2005. Towards a ctl* tableau. In *FSTTCS*, LNCS (2005). Springer. (p. 16)

RIAZANOV, A. AND VORONKOV, A. 2002. The design and implementation of vampire. *AI Communications 15*, 2 (September), 91–110. (pp. 41, 102)

RICHARDS, B., BETHKE, I., OBERLANDER, J., AND VAN DER DOES, J. 1989. *Temporal Representation and Inference*. Academic Press. (p. 11)

SCHMIDT, R. AND TISHKOVSKY, D. 2003. A caution about taleau calculi for (c)pdl. draft manuscript. (p. 19)

SCHMITT, P. H. AND GOUBAULT-LARRECQ, J. 1997. A tableau system for linear-time temporal logic. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems* (1997), pp. 130–144. Springer-Verlag. (p. 15)

SCHWENDIMANN, S. 1998. A new one-pass tableau calculus for PLTL. In *TABLEAUX*, Volume 1397 of *Lecture Notes in Artificial Intelligence* (1998), pp. 277–291. Springer. (pp. iv, 15, 22, 25, 36, 87, 88)

SHANAHAN, M. 1997. *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT Press. (p. 11)

SISTLA, A. P., VARDI, M. Y., AND WOLPER, P. 1987. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science 49*, 217–237. (p. 16)

STALLMAN, R. M. AND SUSSMAN, G. J. 1976. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *NASA STI/Recon Technical Report N 77*.

STIRLING, C. 1987. Comparing linear and branching time temporal logics. Technical report, Dept of Computer Science, Edinburgh University.

STIRLING, C. 1988. Temporal logics for CCS. In *Proceedings of School/-Workshop on Linear Time, Branching Time and Partial Order In Logics and Models of Concurrency*, Number 354 in LNCS (1988), pp. 660–671. (p. 20)

STREETT, R. S. 1981. Propositional dynamic logic of looping and converse. In *Proceedings of ACM symposium on Theory of Computing* (1981), pp. 375–383. ACM Press. (p. 19)

THORNTON, S. 2004. Automating deduction in modal logics with fixpoint operators. (p. 87)

TSARKOV, D. AND HORROCKS, I. 2004. Efficient reasoning with range and domain constraints. In *Proc. of the 2004 Description Logic Workshop* (2004), pp. 41–50. (p. 101)

TSARKOV, D. AND HORROCKS, I. 2006. Fact++ description logic reasoner: System description. In *IJCAR* (2006). (p. 41)

VAN DER HOEK, W. AND MEYER, J. 1997. A complete epistemic logic for multiple agents - combining distributed and common knowledge. *Epistemic Logic and the Theory*. (pp. 16, 17)

VAN DITMARSCH, H. AND DYCKHOFF, R. 2002. Sequent calculi for logics with common knowledge. manuscript. (pp. 18, 19)

VARDI, M. Y. 1996. Why is modal logic so robustly decidable ? In *Descriptive Complexity and Finite Models* (1996), pp. 149–184. (p. 21)

VARDI, M. Y. 2001. Branching vs. linear time: Final showdown. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2001), pp. 1–22. Springer-Verlag. (p. 12)

VARDI, M. Y. AND WOLPER, P. 1988. Reasoning about infinite computation paths. Technical Report RJ 6209 (April), IBM TJW. (p. 16)

VERNA, D. 2006. Beating C in scientific computing applications - on the behavior and performance of Lisp, Part I. In *In Third European LISP Workshop at ECOOP* (2006). (p. 7)

VICENTE, A. AND WAGNER, E. 2003. Design patterns in ocaml. manuscript.

WADLER, P. 1992. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albequerque, New Mexico, 1992), pp. 1–14. (p. 7)

WADLER, P. 1993. Monads for functional programming. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School* (1993). Springer-Verlag. (p. 7)

WADLER, P. 2006. Functional programming in the real world. `http://homepages.inf.ed.ac.uk/wadler/realworld/`. (p. 7)

WEIDENBACH, C., BRAHM, U., HILLENBRAND, T., KEEN, E., THEOBALT, C., AND TOPIC, D. 2002. SPASS version 2.0. In *CADE-18*, Volume 2392 of

Lecture Notes in Artificial Intelligence (2002), pp. 275–279. Springer. (pp. 41, 102)

WOLPER, P. 1985. The tableau method for temporal logic: An overview. *Logique et Analyse 110-111*, 119–136. (p. 15)

WOLPER, P., VARDI, M., AND SISTLA, A. P. 1983. Reasoning about infinite computation paths. In *Proceedings of Symposium on Foundations of Computer Science* (1983). (p. 16)

WOOD, A. 2001. Structure and interpretation of computer programs. *Journal Functional Programming 11*, 2, 253–262. (p. 6)

ZANARDO, A. 1992. A note about the axioms for branching-time logic. *Notre Dame Journal of Formal Logic 33*, 2, 225–228. (p. 14)