The Tableau Workbench

Pietro Abate [1] and Rajeev Goré [2]

Computer Sciences Laboratory
The Australian National University
Canberra ACT 0200, Australia

Abstract

The Tableau Workbench (TWB) is a generic framework for building automated theorem provers for arbitrary propositional logics. The TWB has a small core that defines its general architecture, some extra machinery to specify tableau-based provers and an abstraction language for expressing tableau rules. This language allows users to "cut and paste" tableau rules from textbooks and to specify a search strategy for applying those rules in a systematic manner. A new logic module defined by a user is translated and compiled with the proof engine to produce a specialized theorem prover for that logic. The TWB provides various hooks for implementing blocking techniques using histories and variables, as well as hooks for utilising/defining optimisation techniques. We describe the latest version of the TWB which has changed substantially since our system description in TABLEAUX 2003.

Keywords: generic tableau theorem prover, automated deduction, tableaux strategies, system description

1   Introduction

Highly optimised provers like `MSPASS` [9] and `FaCT` [8] can test formulae with hundreds of symbols within a few seconds. Generic logical frameworks like `Isabelle` [11] allow us to implement almost any logical calculus as a "shallow embedding" with an automatic search tactic. However researchers often find these tools too complex to learn, program or modify. In the middle are the `LWB` [7], `LoTReC` [6] and `LeanTAP` [3] which implement many different logics or which allow users to create their own prover. The `LWB` provides a large (but fixed) number of logics while `LoTReC` requires the logic to have a binary relational semantics. These efforts to merge efficiency with generality highlight the mismatch between the high level concepts of the tableau method and the low level details of a generic implementation of it.

If you have just invented a new tableau calculus for a logic without such semantics, then `LoTReC` and the `LWB` are not very useful. Lazy logicians, as opposed to

---

[1] Email: Pietro.Abate@anu.edu.au
[2] Email: Rajeev.Gore@anu.edu.au

real programmers, usually then extend `LeanTAP`, but there is no generic `LeanTAP` implementation that contains features like loop-checking or further optimisations.

The Tableaux Work Bench (`TWB`) is a generic framework for building automated theorem provers for arbitrary (propositional) logics. A logic module defined by the user is translated by the `TWB` into OCaml code, which is compiled with the `TWB` libraries to produce a tableau prover for that logic. Lazy logicians can encode their rules easily to implement naive provers for their favourite logics without learning OCaml. Real programmers can use the numerous optimisation hooks to produce faster provers and even tailor the system to their own requirements by writing their own OCaml libraries.

Differences with previous version.

The `TWB` has been completely re-written since our previous system description in TABLEAUX 2003 [2]. Despite following the same design principles, we concentrated on simplifying the software both from the user and the developer points of view. The user interface has been completely changed to allow a more natural specification of logical rules. Ocaml code is no longer visible in the rule specification section, despite still being available to specify side conditions and functions. The architecture has also been substantially changed to increase modularity and to reduce code complexity. We also abstracted the core routine from the concrete data types and created a library to manipulate objects such as sets, multi-sets and lists in a generic way. In details these are the major changes:

- The user interface has been re-designed to allow a more natural specification of tableau/sequent rules. We removed the functional interface of the previous version of the `TWB`, where rules were defined as part of an Ocaml expression, and defined a new syntax. We also added the sequent interface.

- The connective section, in which the user could specify the concrete syntax of the connective used to define logical formulae has been replace by a new and more flexible way for defining the syntax via grammar definitions.

- The previous version of the `TWB` did not allow the user to define new data structures without re-compiling the `TWB` library. The new `TWB` now features a new abstract data type architecture that allows to re-use different components to create complex data structures at the user level.

- The core algorithm has been re-written and simplified. In the current version it is possible to specify multiple invertible rules and to chain rules in the strategy to compose different rules together.

Architecture and Design

The core algorithm visits a tree generated by using a user-defined strategy to apply a finite collection of user-defined rules to an initial node containing a finite number of user-defined formulae. The `TWB` is organised into four main components which total roughly 4000 lines of OCaml code (loc). The core ($\sim$ 400 loc) contains the `visit` algorithm, all type definitions and the strategy interpreter. The

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

Fig. 1. Definition of the ($\wedge$) and ($\wedge_{\text{inv}}$) and (K) rule.

`data-type` library ($\sim$ 700 loc) implements support data structures. The `tableau` library ($\sim$ 800 loc) provides the machinery to implement tableau-based provers. The `syntax` library ($\sim$ 1700 loc) allows a user to define specific logic modules. Finally the command line interface ($\sim$ 250 loc) provides a way to compile and execute the logic modules on input formulae.

2 Basic Notions

In this section we outline some well-known tableau principles which we used to specify logical rules in the TWB using a familiar notation. We will detail the TWB code in Section 3.

Tableau rules are of the form $\dfrac{n}{d_1 \ldots d_m}$ where n is the numerator, while $d_1 \ldots d_m$ is a list of denominators. The numerator contains one or more distinguished formula schemas called *principal formulae* and one or more formulae schema called *side formulae*. We use meta-variables $A, B, P, Q, \cdots$ to represent principal formulae and X, Y, Z for, possibly empty, set of side formulae. Note that this notation is just for describing the intended rule semantic clearly. As we shall see later, in the TWB, principal formulae are specified using an identifier in curly braces, while meta variables are specified using upper-case identifiers.

Operationally, tableau rules can be seen as functions to transform a node into a list of nodes during the construction of a graph. We say that a rule is *applicable* if we can *partition* the contents of the current node to instantiate the meta-variables in the numerator. The *denominators* of a rule represent actions to expand the graph by creating new nodes according to these instantiations.

The repeated application of rules gives rise to a tree of nodes, with children obtained from their parent by instantiating a rule. To capture sequent calculi as well as tableau calculi, we view this process as a search for a proof where all leaves are "closed". But in general, for tableau calculi with cyclic proofs, there are really three notions: *success*, *failure* and *undetermined*: we stay with the dichotomy "closed" as success (of proof search) and "open" as failure (of proof search) for now to simplify the exposition. A numerator pattern like Z is *unrestricted* since its shape does not restrict its contents in any way. Conversely, the pattern $\Box X$ is *restricted* to contain only $\Box$-formulae. The ($\wedge$)-rule in Figure 1 is seen as instructions to partition the contents of a node N to instantiate $A \wedge B$ to a single $\wedge$-formula from N and to instantiate Z to all the *other* formulae so that $Z = N - (A \wedge B)$ where "$-$" is the "subtraction" operator appropriate to our containers. In particular, if N is a set then there are no "hidden contractions". Similarly, the (K)-rule from Figure 1 instructs us to partition N into three disjoint parts: a single $\Diamond$-formula

$\Diamond P$; a container $\Box X$ of $\Box$-formulae; and a container Z of all *other* formulae. The intended interpretation of the (K)-rule is invariably that Z should not contain any $\Box$-formulae since this ensures completeness of the rule. We therefore further assume that the partitions specified by the numerator are *maximal*.

Currently in the TWB, we forbid "purely structural" numerators like $X; Y$ which non-deterministically partition the node N into two arbitrary disjoint containers X and Y (even though such numerators may be useful for linear logic), and numerators like $\Box X; \Box Y; Z$ since both can lead to an exponential number of possible different partitions.

2.1   Non-determinism

One of the main challenge when designing a generic tableau/sequent prover is to identify and to remove the non-determinism implicit in the pen-and-paper formulation of tableau systems. In general, we can identify three forms of non-determinism associated with three fundamental aspects of a generic tableau-based decision procedure:

Node-choice: the algorithm determines which leaf becomes the current node;

Rule-choice: the algorithm determines which rule to apply to the current node;

Instance-choice: a heuristic procedure determines the order in which to explore the different instantiations of the current node created by the chosen rule.

The first form of non-determinism is resolved in the TWB by using a depth first visit function that deterministically selects the left most node as candidate. The second source of non-determinism is more complex and it arises from the lack of guidance when applying logical rules. For example, in classical propositional logic, where all rules are invertible, the order in which rules are applied is not important. However 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. That is, the linear (K)-rule shown contains some "implicit branching" since we may need to backtrack over the different choices of principal formulae until we find a closed branch.

The problem is even more complicated in tableau-sequent calculi with more than one non-invertible rule: for example intuitionistic logic as in [4]. 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.

In the TWB we address all these issues by coupling tableau rules with a *strategy* to control the order in which rules are applied. In particular the strategy we employ

$$(\vee) \quad \frac{A \vee B; Z}{A; Z | B; Z} \qquad (\text{det-K}) \ \frac{\diamond P_1; \cdots; \diamond P_n; \Box X; W}{P_1; X || \ldots || P_n; X} \qquad (\text{rec-K}) \ \frac{\diamond P; \Box X; \diamond Y; W}{P; X \ || \ \Box X; \diamond Y}$$

Fig. 2. Universally and Existentially Branching rules.

uses the following operators given current node N:

- The basic strategy is a rule and it is identified by a rule name (eg. Id). A rule succeeds if it is applicable;

- The strategies $Skip$ and $Fail$ are two basic strategies that respectively, always succeed or always fail. The strategy $Skip$ is the same as writing a rule that is always applicable and does nothing. The strategy $Fail$ is the same as writing a rule that is never applicable.

- Deterministic alternation, written as $t_1 ! t_2$, is the strategy that behaves as t_1 if the first rule of t_1 is applicable otherwise behaves as t_2. If the strategy t_1 is selected, the operator ! does not backtrack to choose t_2.

- Alternation. The strategy $t = t_1 || t_2$, is the strategy that first tries to apply t_1 to N. If the result of the visit function, using the strategy t_1 is successful, then t behaves as t_1. Otherwise $t = t_2$.

- Sequencing. The strategy $t = t_1; t_2$ succeeds if both t_1 and t_2 are applicable. Fails otherwise. In particular (tN) is equal to $(t_2(t_1 N))$.

- Star. The strategy t^* behaves as the expression $\mu X.(t; X | Skip)$, that is, the strategy t is applied as much as is possible, and then returns the result of the computation when it fails. Intuitively, if the strategy t succeeds m times the result will be $t_1(t_2(...(t_m N)...))$ where each incarnation t_i is t itself.

The third form of non-determinism is the aforementioned choice of one rule instance (partition) of the current node from potentially many rule instances (partitions) of the current node. As we saw, the different instances are obtained by choosing different formulae as the principal formulae. This nondeterminism can be resolved by using optimisation techniques based on logic-specific considerations used to reduce the size of the search space. For example, if the chosen rule is an $(\vee)$-rule, heuristics such as MOMS [5] (Maximum number of Occurrences in disjunctions of Minimum Size) or iMOMS [5] (inverted MOMS) order the disjunctions (formulae) in the node to always choose the less/more constrained disjunct, which, in principle, should lead to an earlier clash. By default, the TWB currently uses a lexicographic ordering of the potential principal formulae and simply chooses the next formula in this ordering.

2.2 Controlling Backtracking.

The search procedure is an attempt to find a "closed" tableau, so it must recover from unsuccessful branches that do not close. For rules with "implicit branching", the best recovery action depends upon the type of non-determinism embodied in

the different rule instances.

One one hand, the ($\wedge$)-rule from Figure 1 is invertible in many tableau calculi: in that case, it embodies a *don't care* non-determinism for formula choice since we are free to choose *any* conjunction from the current node as the principal formula of this rule instance. If the chosen instance does not lead to a closed denominator, then there is no need to backtrack over the other different conjunctions in the current node since invertibility guarantees that none of them will lead to a closed denominator. Operationally, it is currently not feasible to automatically detect whether a rule is invertible in a given tableau calculus, so one way to declare a rule as invertible is by writing it as ($\wedge_{\mathrm{inv}}$) using a double-line separator as shown in Figure 1.

On the other hand, in the (K)-rule, the implicit branching is essential since different choices may give different results, thus embodying a form of *don't know* non-determinism. For example, consider the different choices of principal $\diamond$-formula for a node containing both $\diamond p$ and $\diamond\bot$: the $\diamond p$-choice may or may not close, but the $\diamond\bot$-choice is guaranteed to close. That is, if one rule instance does not close then we must backtrack over the other possible rule instances *until we find an instance that gives a closed denominator* or all instances are found to be open, before backtracking further.

Traditional tableau calculi do not envisage any form of communication between the branches of a tableau, but operationally we wish to allow such communication. For traditional explicitly branching rules like ($\vee$), it is possible to add decorations that allow inter-branch communication. But this is not possible with the branches implicit in the choice of principal formula of the (K) rule. We therefore allow the denominators of a rule to be separated by a new type of separator $||$ which captures "existentially branching": the numerator is closed if **some** denominator is closed. This is dual to the "universally branching" separator $|$ used for explicitly branching rules like ($\vee$): the numerator is closed if **all** denominators are closed.

Figure 2 shows two ways of "determinising" the implicit backtracking in the (K)-rule using existential branching. By our maximality constraint on numerator patterns, the rule (det-K) instructs us to partition the numerator into n principal $\diamond$-formulae, a container $\square X$ for all the $\square$-formulae, and a container W for all the other formulae. The horizontal double-lines tells us to commit to this partition and the $||$ separator tells us that the numerator is closed if some denominator is closed. This rule is actually difficult to implement since the parameter n is effectively a free variable which must be instantiated. But we can determinise even this aspect by using a recursive version called the (rec-K)-rule from Figure 2 instead which can be read as: choose a principal formula $\diamond P$, put all other $\diamond$-formulae in $\diamond Y$, put all $\square$-formulae in $\square X$, put all non-boxed and non-diamond formulae in W, commit to this partition, and close the numerator if some denominator closes. By repeatedly applying (rec-K) to the right denominator, we can step through the same partitions as the (det-K) rule, without having to worry about the value of n since the (rec-K) rule will fail to apply when the right denominator contains no $\diamond$-formulae because $\diamond P$ cannot be empty. We need a strategy to ensure that no other rules are applied between these consecutive applications of (rec-K) but it is easy to capture such a

Instance	Branching			Linear
choice	Universal "|"	Existential "||"	Conditional "|||"	
Backtrack "--"	$(\vee)$	-	-	(K) $(\wedge)$
Commit "=="	$(\vee_{\text{inv}})$	(rec-K)	-	$(\wedge_{\text{inv}})$

Table 1
Rule Classification

strategy using the strategy language.

¿From a general perspective, the two types of branching can be characterised as conditionally branching: "explore different denominators until some user-defined condition becomes true", with both universal and existential branching as instances.

Our resulting rule classification is shown in Table 1. Linear rules have one denominator, while branching rules have two or more which are universal or existential related. Each rule can either commit or not commit to the partition (instantiation) selected first by the formula-choice heuristic (by being invertible or non-invertible). If the rule commits then only this first partition is considered while expanding the denominator(s). If the rule does not commit then the denominator(s) will be re-instantiated for each partition generated by the application of the rule numerator to the current node until an instantiation is found that closes the numerator. With this rule classification it is easier to mechanize tableau rules as specified in the literature whilst removing non-determinism. Rules for classical modal logic can be characterized as shown. In Table 1 we indicate with the symbol "-" combinations that cannot be expressed in the TWB using a single rule. Nonetheless these can be expressed by using two rules "chained" in the strategy with the sequence operator.

3 Defining the Calculus for a Logic

In this section we detail the TWB syntax using as a running example a tableau calculus for S4 from [10] and we will assume the reader is familiar with it. A logical module in the TWB is composed of three main parts: the grammar definition, the logical rules and the strategy.

3.1 Defining the grammar

The keyword CONNECTIVES in the TWB introduces a list of strings separated with a semicolon ";". Each string can be used as a connective in the grammar definition (below) and it will become a reserved keyword in the module. Connectives can be of variable length, can be composed by a mix of numbers, symbols and characters.

```
CONNECTIVES [ "~";"&";"v";"->";"<->";"<>";"[]" ]
```

The keyword GRAMMAR introduces the specification of the grammar used in the module to write formula expressions and formula schemas. A grammar is composed of a sequence of productions separated by a double semicolon ";;". Two mandatory

productions are `formula` and `expr`. A third pre-defined production that can be
omitted is `node`.

- `node`: this production defines the shape of a tableau/sequent node. By default,
  a tableau node is composed of a set of formulae. It also possible to specify other
  shapes such as `set => singleton` where the node is a sequent with a set on the
  left and a singleton element on the right.
- `expr`: this production defines the outer most expressions used in the definition
  of tableau rules. Generally an expression is just a formula but it can also be
  decorated with a label or with another arbitrary data structures.
- `formula` : this production defines the shape of the formula definition to be used
  in the tableau rules.

```
GRAMMAR
formula :=
     ATOM | Verum | Falsum
   | formula & formula    | formula v formula
   | formula -> formula  | formula <-> formula
   | [] formula | <> formula
   | ~ formula
;;
expr := formula ;;
END
```

The grammar accepted by the TWB is a loose LL(1) grammar. Conflicts are
not handled automatically and the definition of ambiguous grammars can lead to
unspecified results. In the grammar definition we use these three conventions: lower-
case identifiers in a production are interpreted as grammar identifiers for a produc-
tion; upper-case identifiers are interpreted as constants (eg. `Verum` and `Falsum`);
and the keyword `ATOM` identifies an anonymous atom in the grammar.

3.2   Defining histories and variables

The TWB has the infrastructure to add histories and variables to tableau/sequent
rules. Histories are used to pass information top-down from numerator to denomi-
nators and are typically used to implement blocking techniques. Variables are used
to pass information bottom-up from already explored sub-trees to parents. The
traditional notion of a branch being "Open" or "Closed" is carried implicitly from
the leaves to the root by a default variable called `status`.

Operationally, a history or a variable is an object that must be instantiated by
the user when defining a logic module. The TWB library defines the interface for
such objects in terms of an Ocaml functor. The TWB data types library defines
three generic data structures that can be assembled by the user to define complex
histories or variables. A functor is a function that given a module of type Element-
TYPE returns a module of type TwbSet.S . The ElementTYPE type module has
the following signature:

```
module type ValType =
    sig
        type t
        val to_string : t -> string
        val copy : t -> t
    end
```

Similarly, a set of set of formulae is declared by instantiating the type `t` as *FormulaSet.set*, the function `to_string s` as *s#to_string* and the function `copy` `s` as *s#copy* (where # is just the OCaml syntax to invoke an object method). The three basic functors to build such data structures are: TwbSet.Make (defines a set of elements), TwbMSet.Make (defines a multi set of elements) and TwbList.Make (defines a list of elements). To add a history of type "set of formulae" we need to define a module FormulaSet by using the functor TwbSet.Make and pass to it the type `formula` as ElementTYPE as shown below.

```
module FormulaSet = TwbSet.Make(
    struct
        type t = formula
        let to_string = formula_printer
        let copy s = s
    end
)
```

Once the module is defined, we add it to the tableau/sequent node using the following syntax. The keyword `HISTORIES` expects a sequence of history declarations. Each such declaration consists of three elements. The first element is an upper-case identifier (e.g. UBXS) naming the history used in the tableau rules, followed by a colon. The second element is the type of the history (e.g FormulaSet.set), followed by an `:=` operator. The third element is the default value for the history (e.g. a new empty object of type FormulaSet.set).

```
HISTORIES
  UBXS : FormulaSet.set := new FormulaSet.set ;
  DIAS : FormulaSet.set := new FormulaSet.set
END
```

Variables are declared similarly with the following syntax. The only difference is that the identifier used for variables is always lower-case.

```
VARIABLES
  status     : String := "Open"
END
```

3.3 Defining tableau rules

The collection of rules are enclosed within keywords `TABLEAU` and `END`. Rules are specified via the `RULE` and `END` keywords with principal formulae enclosed in braces. For conciseness, rules can be written on one line by omitting line-breaks before and

after the horizontal line.

The simplest tableau rule has a name `Id` and a numerator pattern which partitions the current node into three disjoint parts: an atom a; its negation $\sim a$; and an (unrestricted) container Z. Note that atoms are specified using a lower case identifier like a, while sets of formulae using an upper-case identifier like Z.

Since any such clash is sufficient to close the branch, we instruct this rule to commit to the first partition that matches by using a horizontal line of at least two "==" symbols. The directives `Close` or `Open` or `Stop` set the value of the default variable `status`, stopping the visit procedure and triggering backtracking. Thus the `Id` rule closes the current branch. The rule `False` closes the current branch if one occurrence of constant `Falsum` is found and does not require an unrestricted container like Z.

```
TABLEAU

  RULE Id { a } ; { ~ a } ; Z === Close END
  RULE False Falsum  === Close END
  RULE And { A & B } === A ; B END
  RULE Or  { A v B } === A | B END


  RULE T
  { [] P }
  =============
      P
  COND notin(P, UBXS)
  ACTION [
      UBXS    := add(P,UBXS);
      DIAS := emptyset(DIAS)]
  END


  RULE S4
  { <> P } ; Z
  -------------
    P ; UBXS
  COND notin(<> P, DIAS)
  ACTION [ DIAS := add(<> P,DIAS) ]
  END


END
```

The `And` rule is written to highlight the omission of unrestricted meta-variables like Z containing "all other formulae" to capture simple rewriting. It has a single (non-empty) principal formula $A\&B$, with *no accompanying unrestricted container* like `Z`. This is a signal to *replace* (rewrite) $A\&B$ with A and B *and carry all other formulae from the numerator into* the denominator. A more traditional way to write this rule would use `Z` (say) in both the numerator and the denominator.

The `Or` rule also uses rewriting but uses universal branching to create a left child containing A and a right child containing B.

The `T` rule rewrites $\Box P$ to P but only if the side-condition declared using the `COND` construct is true. This condition checks whether $P \notin$ `UBXS` using a function `notin(.,.)`. Its `ACTION` directive constructs the histories of its child by adding P to the current history `UBXS` and setting the history `DIAS` to be the empty set (because it has seen a $\Box$-formula new to `DIAS` as explained in [10]).

The `S4` rule chooses a principal formula $\diamond$ P from the numerator and creates a child containing both P and the set contained in the history `UBXS` but only if the side-condition $\diamond$ P $\notin$ `DIAS` specified via `COND` is true. Its `ACTION` directive constructs its child's history `DIAS` by adding $\diamond$ P to the current history `DIAS` but leaves `UBXS` intact. It does not commit to the choice of principal formula since it uses a horizontal line consisting of at least two "`--`" symbols, which means that if this child does not close, then this rule will backtrack over the different principal $\diamond$-formula choices.

The same rule can also being specified by making the backtracking explicit (below). The body of the rule is composed of a principal formula, a possibly empty set of diamond formulae $\diamond Y$ and an anonymous set Z. Note that now because of the presence of the schema $\diamond Y$, the set Z is diamond-formula free. The denominator has two branches: A ; $UBXS$ encodes the modal jump while $\diamond Y$ encodes a recursion step to consider all other diamonds if the first branch does not close.

```
RULE S4H
{ <> P } ; <> Y ; Z
=====================
 A ; UBXS || <> Y

COND notin(<> P, DIAS)
ACTION [ [ DIAS := add(<> A,DIAS)] ; [] ]
END
```

The numerator of the `S4H` rule is composed of a principal formula, a possibly empty set of diamond formulae $\diamond Y$ and an unrestricted set Z. Note that now because of the presence of the schema $\diamond Y$, the set Z is diamond-formula free. The denominator has two branches: A ; $UBXS$ encodes the modal jump while $\diamond Y$ encodes a recursion step to consider all other diamonds if the first branch does not close.

Note also that we now use the horizontal double line "`==`" as the rule has been determinised and no backtracking is needed to recover from a wrong choice. Moreover the double bar separator "`||`" sets up existential branching which closes the numerator if some denominator closes. Thus, the second branch is explored if and only if the result of the tableau rooted with the node $A;UBXS$ is open. The functions to manipulate histories and side conditions are explained below.

Auxiliary Functions

In the TWB we can use arbitrary OCaml functions to manipulate histories, variables and to set side conditions. In our example we use three functions to handle histories defined as follows:

```
let add (l,h) = h#addlist l
let notin (l,h) = not(h#mem (List.hd l))
let emptyset h = h#empty
```

- `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.
- `notin`(formula list, history): accepts a list of formulae and a history, and returns a boolean indicating if the list of formulae is in the history.
- `emptyset`(history): returns an empty history object.

Thus, each of the patterns like $\Diamond P$, $\Diamond Y$ and Z are lists of formulae. The function call $add(\Diamond P, DIAS)$ instantiates l to a list that contains the principal formula pattern-matched by $\Diamond P$ and instantiates h to the history object identified by $DIAS$.

The TWB implementation of sets provides a method *addlist* which implements code to add every member of l to the history object h and a method *mem* which checks whether its argument is a member of the set $DIAS$.

3.4   Defining the Strategy

The strategy specifies the order in which rules are applied during proof search. In our example, for completeness, we need to apply all classical rules first and then the S4-rule. The bang ! operator applies the first rule that is applicable. The star $*$ iterates until no rule is applicable. Since the S4 rule is the last rule, this means that all the other rules are not applicable. In other words, the strategy says "repeatedly apply the first applicable rule from the left until none are applicable, and then apply the `S4` rule, and repeat this process until no rules are applicable". This strategy is applied to every branch of the proof tree according to the visit algorithm.

```
STRATEGY :=
    let sat = tactic ( (False ! Id ! And ! T ! Or) )
    in tactic ( (sat ! S4 )* )
```

3.5   Compiling and Running the TWB

Each file in the TWB are translated in an Ocaml compilation unit. We assume that the logic rules for the modal logic S4 shown before are saved in a file s4.ml and the negation normal form function in a file s4lib.ml . To compile the logic module we use the utility **twbcompile** that is shipped with the TWB. The result of the compilation is the file **s4.twb** that is the object prover for modal logic S4. To run the file **s4.twb** we can either give it a file of formulae to prove as argument or passing a formula on the stdin channel. We refer to the TWB website for more

usage examples.

4 Conclusion and further work

In this paper we presented a new version of the Tableau WorkBench. Both the architecture and the user interface have been substantially changed from the previous version presented at TABLEAU 2003. An intermediate version of the TWB can be found in [1] where we also presented an array of benchmarks that compare the TWB with the LWB[7] showing that the TWB can compete with the LWB only if well known optimisations (such as simplification or back-jumping) are incorporated in the tableau calculus. As further work, we are working to re-implement such optimisations with the new version and to reproduce up-to-date benchmarks. The TWB can be downloaded from the website `http://twb.rsise.anu.edu.au`

References

[1] Pietro Abate. *The Tableau Workbench: a framework for building automated tableau-based theorem provers.* PhD thesis, Australian National University, 2007.

[2] Pietro Abate and Rajeev Goré. System description: The tableaux workbench (TWB). In *TABLEAUX*, Lecture Notes in Artificial Intelligence. Springer, 2003.

[3] Bernhard Beckert and Joachim Posegga. leanT^AP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[4] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, sep 1992.

[5] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Philadelphia, 1995.

[6] Olivier Gasquet, Andreas Herzig, Dominique Longin, and Mohamad Sahade. Lotrec: Logical tableaux research engineering companion. In *TABLEAUX*, pages 318–322, 2005.

[7] A Heuerding. LWBtheory: information about some propositional logics via the WWW. *Logic Journal of the IGPL*, 4:169–174, 1996.

[8] Ian Horrocks and Peter F. Patel-Schneider. Optimising propositional modal satisfiability for description logic subsumption. *Lecture Notes in Computer Science*, 1476, 1998.

[9] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference*, volume 1847 of *LNAI*, pages 67–71. Springer, 2000.

[10] Alain Heuerding, Michael Seyfried, and Heinrich Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In *Analytic Tableaux and Related Methods*, pages 210–225, 1996.

[11] L. C. Paulson. Isabelle: The Next 700 Theorem Provers. *ArXiv Computer Science e-prints*, October 1993.