

# System Description: The Tableaux Work Bench

Pietro Abate and Rajeev Goré

Australian National University, Canberra, Australia,

[Pietro.Abate|Rajeev.Gore]@anu.edu.au      <http://cs1.anu.edu.au/~abate/twb>

**Abstract.** The Tableaux Work Bench (TWB) is a meta tableau system designed for logicians with limited programming or automatic reasoning knowledge to experiment with new tableau calculi and new decision procedures. It has a simple interface, a history mechanism for controlling loops or pruning the search space, and modal simplification.

## 1 Introduction

Theorem provers for classical propositional modal logics have matured dramatically to the point where formulae with hundreds of symbols can be tested within a few seconds. Direct theorem provers like FaCT [HPS98] and LWB [Heu96] utilise many different optimisation techniques to speed up proof search in particular logics, while translational provers like MSPASS [HS00] utilise fast first-order theorem provers like SPASS. But these avenues are not always viable: FaCT cannot handle logics with an intuitionistic base; although the LWB can handle intuitionistic logic, it can handle only a fixed collection of logics; and although MSPASS gives a sound and complete prover for any first-order definable logic, *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 many flags, and it is not at all obvious how to obtain a decision procedure for a particular first-order definable logic using MSPASS.

While generic tableau-based provers like `Blast_tac` [Pau99] 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` [FdCFG01], which we discuss in Section 6.

Existing proof editors like `xpe` [Mou01], `JAPE` [BS97] and `PESCA` [NvP01] provide only rudimentary proof search facilities (e.g. iterative deepening for `PESCA`), while `blobLogic` [How02] contains a fixed collection of tableau rules.

The Tableaux Work Bench (TWB) is a generic meta-tableau system 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 incorporate or design optimisations and specific decision procedures. By dividing high level and low level optimisations, the user can concentrate on algorithmic aspects related to his or her tableau calculus, leaving to the developer more

complex and generic performance issues about the underlying prover. The TWB includes a generic history mechanism, facilities to perform simplification, to handle global assumptions and to implement more complex optimisation procedures. Currently, the TWB does not offer proof-editing capabilities. The TWB cannot possibly compete with FaCT, MSPASS or LWB in speed, but its versatility should be of use to the TABLEAUX community since there is no restriction to modal calculi, although currently, they must be propositional.

## 2 Generic Tableau Algorithm

The rules that can be specified with the TWB are those which respect the analytic super-formula property [Gor99], allow history mechanisms [Heu98,How98] and permit global assumptions [Fit83]. At the moment it is not possible to specify calculi that require two pass tableau algorithms (a la PLTL), infinitary calculi, or explicit geometric relational properties like weak-directedness. However using the history mechanism and clever calculi design, it is possible to handle most well-known modal calculi.

The TWB is based on a purely syntactic tableau algorithm. The user can specify the proof-search strategy governing the order of rule applications. Each step in the depth-first search corresponds with a rule application where a rule is selected if its pattern matches the formulae in the current node. Once a rule is selected and executed, the algorithm recursively continues the proof tree exploration until a closed tableau is found or it is not possible to apply any rules (in this case the branch is open). The only axiom embedded in the system is  $\frac{X:A;\neg A}{\perp}$ . However this axiom can be “turned off” if it is not necessary in the calculus (eg.: para-consistent logic).

## 3 User Interface

The TWB is designed to be easy to use, flexible and extensible. Negated Normal Form (NNF) is available but not mandatory, and the user can program his or her own rewriting system for normal forming; see the manual [Aba03].

**Connective Definitions** The TWB has a number of hard-wired symbols to express connectives with one, two or three arguments. To add a new symbol to the language, at the moment, it is necessary to edit the source code of the lexer and symbols’ parser, but in future releases, new connectives will be defined via the user interface itself.

**Rule Definitions** The first step toward the specification of a new calculus is to define a set of rules. A rule is (up to) a six-tuple `rule(pat, act, heuristic, branching, invertibility, name)` where: `pat` is a pattern for the numerator; `act` is a pattern for the denominator(s); `heuristic` is an ordering function that affects the principal formula selection strategy; `branching` is either `All` or `Exist` to indicate whether all or only one denominator must close; `invertibility` is

either `Static` or `Trans` to indicate whether the rule is invertible and `name` is a string in quotes to describe this rule. `Heuristic`, `branching` and `invertibility` are optional and their defaults are respectively: the null function (meaning no ordering), `All` and `Static`. The various components can be specified using a `let` statement inherited from OCaml in which an internal rule name can be specified for the defined rule. For example, the classical propositional tableau ( $\vee$ ) rule is:

```
let or_r =
  let p = pat { "{A v B}"; "X" }
      and a = act { "A" ; "X" | "B" ; "X" }
  in rule (p, a, All, Static, "Or");;
```

$$\frac{A \vee B; X}{A; X | B; X}$$

The numerator consists of a principal formula enclosed in braces and a list of sets. The denominator consists of a list of branches, defined by lists of sets. We prefer this terminology rather than “premiss” and “conclusion” since these latter terms can cause confusion when using sequent calculi. The rule pattern is matched against the formulae in the current node and the rule action is executed only if the pattern is satisfied. The heuristic function can be specified in each rule and affects the selection of the principal formula of that rule. It is basically a comparison function: for example, to select the formula with higher modal weight the heuristic function could be defined by the user as:

```
let weight f1 f2 =
  let rec w = function
    |term "A & B" |term "A v B" -> w (term "A") + w (term "B")
    |term "dia A" |term "box A" -> 1 + w (term "A")
    |term "~ A" -> w (term "A")
    |term "atom a" -> 0
    | _ -> failwith "error in weight"
  in
  if (w f1) = (w f2) then 0
  else if (w f1) > (w f2) then 1
  else -1
```

The “or” rule definition above will consequently be modified to include the heuristic definition as rule (p, a, weight, All, Static, “Or”). It’s also possible to define a set of additional side conditions that must be fulfilled in order to fire a rule. For example a basic modus ponens rule on atomic formulae for the contraction-free calculus for intuitionistic logic can be coded as below where `member(x,Y)` is a user-defined function that checks if `x` is in the set `Y`.

```
let mp =
  let p = pat { "{ atom p -> B }"; "X" ; member("atom p", "X") }
      and a = act { "atom p"; "B" ; "X" }
  in rule (p, a, All, Static, "MP");;
```

Non-invertible rules invariably introduce non-determinism into a calculus and lead to choice-points in the search procedure which are explored by backtracking. In the TWB, we have chosen to specify such choice points explicitly via the “Existential” branching construct. For example the traditional ( $K$ )-rule is:

```

let k_r =
  let p = pat { "{dia A}" ; "box X" ; "dia Y" ; "Z" }
  and a =
    matchset "dia Y" with
      [] -> act { "A" ; "X" }
      | _ -> act { "A" ; "X" | "box X" ; "dia Y" ; "Z" }
  in rule (p, a, Exist, Trans, "K")

```

where `matchset` is used to define two actions: one when the set  $\diamond Y$  is empty and the other when it is not. The `Exist` appellation declares to the prover that only one of the two denominators need close in the second case.

In the example above, the `box X` part matches *all*  $\square$ -formulae in the current node, hence  $Z$  contains neither  $\square$ - nor  $\diamond$ -formulae.

**Systematic Proof Search and Strategy** Each set of rules is attached to a strategy defining the proof search procedure for the associated calculus. A strategy is defined in terms of two types of cycles (lists) containing rule names. A *\*-cycle* executes the rules in list-order until none of them are applicable. A *+cycle* executes the first applicable rule in its list only. The strategy stops if a closed tableau is detected or if no rule is applicable.

For example, the following strategy definition specifies the usual systematic procedure for modal logic **K** where the `And` and `Or` rules are executed until they are not applicable (saturation step), and the (*K*)-rule above is executed once (transitional step): `let str = strategy [ [and_r ; or_r]* ; k_r ];;`

**History Mechanisms** The TWB also has a history facility for efficient loop-checking as part of the rule definition. For example, the traditional rule for handling reflexivity in the logic **KT** requires an implicit contraction on the principal formula to make it invertible. But this rule can then be applied *ad infinitum*, and so a starring mechanism is usually employed to stop this behaviour as shown below left. Alternatively, explicitly specifying side-conditions and actions to be executed on a history  $Z$  suffices, as shown below at right where “-” is a separator:

$$\frac{\square A; X}{A; (\square A)*, X} \quad \square A \text{ not starred} \qquad \frac{\square A; X - \square A \notin Z}{A; \square A; X - \square A \cup Z}$$

This can be coded in the TWB using the construct `- H{...}` to express histories:

```

let t_r =
  let p = pat { "{box A}"; "X" - H { isnotin("Z", "box A") } }
  and a = act { "A"; "box A"; "X" - H { add("Z", "box A") } }
  in rule(p, a, All, Static, "T Rule History");;

```

The system also allows a light version of a history, called starring, where a formula is simply starred to avoid considering it more than once, rather than making an explicit copy of it into a history.

**User-defined Functions and Simplification** Side conditions can be added to the pattern by specifying predefined functions like `isnotin` above or user-defined functions which accept a formula and return true or false. Every time a pattern is matched against a formula set, every such specified function is evaluated on every formula, and the rule is executed only if all conditions are satisfied. A user defined simplification procedure [Mas98] is used as shown below:

$$\frac{A \vee B; X}{A; X[A:=\top] \quad | \quad B; X[B:=\top]}$$

```
let or_s_r =
  let p = pat { "{A v B}"; "X" }
  and a = act { "A" ; "X"[ "A":= top ] | "B"; "X" [ "B":= top ] }
  in rule (p, a, All, Static, "Or Simpl");;
```

Global assumptions can also be used in the TWB, and can be defined statically or dynamically: in the first case they must be specified on the command line and they are present for every input formula. In the latter, the user can specify a function that accepts an input formula and returns a set of global assumptions to be used against that formula in the proof.

## 4 Experiments and Performance

The flexibility of the TWB has been tested by implementing the traditional history-based calculi for the logics **K**, **KT**, and **S4** [Heu98] in a modular fashion. We have tested the TWB using the LWB benchmarks: the TWB could solve only the first several formulae in the respective test formulae in under 50 seconds each. These times are an order of magnitude slower than the LWB and are hardly state-of-the-art. There are two basic reasons for such a difference. First, the LWB embeds many known optimisations and heuristics while the TWB currently allows only modal simplification. Second, the LWB uses more efficient data structures while the TWB currently uses naive lists.

Conversely, the TWB easily allows us to extend the calculus for **S4** into a calculus for **S4.3** and then into (the non-first-order-definable) **S4.3.1** while such an extension in the LWB is probably only possible for its authors.

## 5 Implementation

The TWB is implemented in OCaml [CMP00], a strongly typed object oriented programming language available for many architectures. The TWB can be compiled either in native form, in byte code, or run via an OCaml shell, giving total flexibility. The rules and strategy definitions effectively become part of the system itself as they are compiled in byte code and dynamically linked to the prover engine. The basic data structures of the system are lists and hash tables leaving room for future performance improvements, but the modularity of the system allows easy customisation of the internal design.

The user interface inherits its syntax from OCaml and is in fact only syntactic sugar added to the real language itself. This adds flexibility and generality because it is always possible to write more complex rule definitions without using the user interface, but with the programming language itself. In this sense the prover can be seen just as a library for tableau oriented theorem proving.

## 6 Related Work

We now compare the TWB with related work in more detail.

Direct provers like LWB [Heu96] and FaCT [HPS98] are clearly superior if they can handle the sought-after logic, and the LWB, in particular, can handle intuitionistic logic and a long list of particular modal logics. But programming a new calculus into the LWB or FaCT is difficult for anyone except their authors.

Translational provers like MSPASS [HS00] are equally superior if the logic is first-order definable and falls into a decidable fragment like the two-variable fragment or the guarded fragment. 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 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 [GO92] 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` provides fairly basic facilities for designing new rules, and even allows certain rules to be marked as “undoable” (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” [Pau99].

The TWB is closest to `lotrec` [FdCFG01] in that both are generic systems that allow a user to specify new rules and strategies for experimenting with proof-search. The main differences between them are the underlying execution models. `lotrec` works at a *global* level, keeping track of all tableau nodes, and the accessibility relations among them in an *explicit* manner. For example, the weak-directedness frame-conditions for the logic **S4.2** ( $\forall x, y, z. \exists w. xRy \& xRz \Rightarrow xRw \& yRw$ ) can be coded explicitly in `lotrec` by referring explicitly to *R* in the rules. The TWB works on a local level and keeps only the information relevant to the current node, so a condition like weak-directedness must be captured implicitly using a particular form of cut on super-formulae; see [Gor99].

Whereas the TWB makes histories explicit, these must be simulated in `lotrec` by the user using the various node and edge marking techniques provided by `lotrec`. Since `lotrec` uses labels to mark the nodes it creates, it can handle the difference operator, which the TWB cannot handle. Overall, `lotrec` is biased towards semantics while the TWB is biased towards proof theory. Which you use is probably best determined by the logic in question.

## 7 Conclusion and Further Work

The TWB allows users with little or no technical background in automatic reasoning to encode their own tableau prover in a simple yet flexible manner. The code and user manual is available at <http://cs1.anu.edu.au/~abate/twb>.

The TWB is still a prototype and we envision much further work: we want to provide a sequent calculus front-end; provide facilities for calculi with “stoups”; provide facilities for hyper-sequents; allow nodes to contain multisets or lists rather than sets; allow rules which partition the side-formulae into two disjoint sets as needed in linear logic; and improve the speed of the underlying implementation.

## References

- [Aba03] P Abate. Tableaux work bench (twb) - User Manual, <http://arp.anu.edu.au/~abate/twb> 2003.
- [BS97] R Bornat and B Sufirin. Jape: A calculator for animating proof-on-paper. In W McCune (Ed), *CADE'97*, LNCS 1249:412–415, Springer.
- [CMP00] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.
- [FdCFG01] L Fariñas del Cerro, D Fauthoux, O Gasquet, A Herzig, D Longin, and F Massacci. Lotrec: the generic tableau prover for modal and description logics. In *IJCAR'01*, LNAI 2083:453-458. Springer Verlag, 2001.
- [Fit83] M. Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese Library*. D. Reidel, Dordrecht, Holland, 1983.
- [GO92] D Gabbay and H J Ohlbach. Quantifier elimination in second order predicate logic. In *Proc. KR-92*, 1992.
- [Gor99] R Goré. Chapter 6: Tableau methods for modal and temporal logics. In *Handbook of Tableau Methods*, pages 297–396. Kluwer, 1999.
- [Heu96] A Heuerding. LWBtheory: information about some propositional logics via the WWW. *Logic Journal of the IGPL*, 4(4):169–174, 1996.
- [Heu98] A Heuerding. *Sequent Calculi for Proof Search in some Modal Logics*. PhD thesis, Institute for Applied Mathematics and Computer Science, University of Berne, Switzerland, 1998.
- [How98] J. M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St Andrews, December 1998.
- [How02] C Howitt. bloblogic. <http://users.ox.ac.uk/~univ0675/blob/>, 2002.
- [HPS98] I Horrocks and P F Patel-Schneider. Optimising propositional modal satisfiability for description logic subsumption. In LNCS 1476, 1998.
- [HS00] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In *TABLEAUX 2000*, LNCS 1847:67–71, Springer, 2000.
- [Mas98] F Massacci. Simplification: a general constraint propagation technique for propositional and modal tableaux. In *Proc. TABLEAUX 98*, LNCS 1397:217-231. Springer, 1998.
- [Mou01] M Mouri. Theorem provers with counter-models and xpe. *Bulletin of the Section of Logic*, 30(2):79–86, 2001.
- [NvP01] S Negri and J von Plato *Structural Proof Theory*. CUP, 2001.
- [Pau99] L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3), 1999.