# The Tableau WorkBench

Pietro Abate[1] and Rajeev Goré[2*]

[1] The Australian National University
Canberra ACT 0200, Australia
[2] The Australian National University and NICTA
Canberra ACT 0200, Australia
`http://twb.rsise.anu.edu.au` `[Pietro.Abate|Rajeev.Gore]@anu.edu.au`

**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 like backjumping, simplification and caching. We describe the latest version of the TWB which has changed substantially since our system description in TABLEAU 2003.

**Introduction and Motivation.** Highly optimised provers like `MSPASS` [HS00] and `FaCT` [HPS98] can test formulae with hundreds of symbols within a few seconds. Generic logical frameworks like `Isabelle` [Pau93] allow us to implement almost any logical calculus as a "shallow embedding" with an automatic search tactic. But researchers often find these tools too complex to learn, program or modify. In the middle are the `LWB` [Heu96], `LoTReC` [GHLS05] and `LeanTAP` [BP95] 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. 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 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.

[*] NICTA is funded by the Australian Government's Dept of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

**Programming Language and Style.** We wrote the TWB in OCaml because functional languages are more concise than imperative languages, and functional programs are easier to maintain and more resistant to memory management problems and run-time errors [HJ94]. OCaml also has been used to implement other generic theorem provers like Coq, Hol light and NuPrl. To minimise coding errors and reduce maintenance time, the TWB core infrastructure uses a purely functional style. But to avoid some of the ensuing performance penalties, some data structures use imperative style. Currently the TWB runs on Gnu/Linux and MacOsX and has virtually no dependencies on third-party software outside the OCaml compiler, making it easy to install/port to other platforms.

**Core Algorithm and Architecture.** 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 [AG06]. The TWB is organised into four main components which total roughly 4000 lines of OCaml code (loc). The core (464 loc) contains the visit algorithm, all type definitions and the strategy interpreter. The data-type library (664 loc) implements support data structures. The tableau library (818 loc) provides the machinery to implement tableau-based provers. The syntax library (1624 loc) allows a user to define specific logic modules. Finally the command line interface (226 loc) provides a way to compile and execute the logic modules on input formulae.

**Defining the Calculus for a Logic.** A tableau calculus is defined via a *logic module* like s4.ml in Appendix 1. Due to space limitations, we describe only the features used in Appendix 1 and assume familiarity with the history-based calculus from [HSZ96].
CONNECTIVES: A connective is either a constant or is a triple with an OCaml type constructor (redundant in future releases), a concrete syntax specification with "_" indicating arity and argument order, and a binding strength "Zero", "One" or "Two" with "Two" being the weakest. In Appendix 1, the connectives of $S4$ are defined as usual.
HISTORIES and Variables: Variables must begin with a lower-case letter and histories with an upper-case letter. Histories are used to pass information top-down from numerator to denominators 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 from the leaves to the root by a default variable called status. The default value for a data container is an empty instance of that data container. In Appendix 1, the histories DIAS (diamonds) and UBXS (unboxes) are both sets of formulae with the empty set as default value.
TABLEAU and 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.
Id: The simplest tableau rule in Appendix 1 has a name Id and a numerator pattern which partitions the current node into three disjoint parts: a formula $A$; its negation $\sim A$; and an (unrestricted) container $Z$. 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 separator 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.

`False`: For conciseness, rules can be written on one line as shown by the rule `False`.

`And`: The `And` rule is written to highlight that rules can 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.

`Or`: The `Or` rule uses a traditional numerator with an unrestricted pattern `X` but uses universal branching to create a left child containing $A$ and a right child containing $B$. But it also uses the `SIMPLIFICATION` procedure from the library `Kopt.simpl` to simplify every occurrence of $A$ and $B$ in `X` to `Verum` in the left and right child respectively.

`T`: 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(.,.)` provided in the library `Twblib`. 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 [HSZ96]).

`S4`: Finally, the `S4` rule chooses a principal formula `Dia 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 `Dia P` $\notin$ `DIAS` specified via `COND` is true. Its `ACTION` directive constructs its child's history `DIAS` by adding `Dia P` to the current history `DIAS` but leaves `UBXS` intact. It does not commit to the choice of principal formula since it uses a separator 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.

**Defining the Strategy.** A strategy specifies the order in which rules are applied during proof search. The `TWB` tactic language is inspired by `Isabelle` but we omit details due to space limitations. The strategy from Appendix 1 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.

**Benchmarks.** The `TWB` was engineered for generality and flexibility, but it is important to compare it against other well established theorem provers. Since the `TWB` makes few or no logic-specific assumptions, optimisations are left to the user. We compared the `TWB` with the `LWB` on the benchmarks from [HS96] using a Pentium 4 (2.4 Ghz), 1GB RAM, 1GB swap space under Debian GNU/Linux OS and OCaml 3.09.2.

Table 1 shows, for each logic and class, the highest formula complexity in the increasing range 1-21 which could be solved within 100 seconds: see [HS96]. For the `LWB`, we recorded a failure either if it timed out or if it failed with an error (`Error: lwb stack too small`), which we did not investigate further. Our tableau calculi for $K$, $KT$ and $S4$ were optimised with simplification, semantic branching, back-jumping and caching. We have bolded the entries where the `TWB` is better or very close to the `LWB`.

The results are very encouraging: the `TWB` can compete with the `LWB` in terms of pure speed for the logic $K$. The `LWB` used considerably more memory than the `TWB` for certain classes of formulae. Also, when caching was not enabled, the `TWB` used less than 32MB of stack space and virtually no heap space. Conversely, Table 1 shows that the

| 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 |

| 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 |

| 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 1.** Benchmarks comparing the TWB and LWB for modal logics $K$, $KT$ and $S4$

LWB is more efficient for the logics $KT$ and $S4$, which we believe is because we used no further optimisations to exploit logic-specific properties. We leave this for future work.

## References

[AG06] P Abate and R Goré. Theory and practice of a generic tableau engine: The Tableau WorkBench. *Submitted Manuscript 15 pages*, 2006.

[BP95] B Beckert and J Posegga. lean$T^AP$: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[GHLS05] O Gasquet, A Herzig, D Longin, and M Sahade. LoTRec: Logical tableaux research engineering companion. In *TABLEAUX*, pages 318–322, 2005.

[Heu96] A Heuerding. LWBtheory: information about some propositional logics via the WWW. *Logic Journal of the IGPL*, 4:169–174, 1996.

[HJ94] P Hudak and M P Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Technical report, Yale, July 1994.

[HPS98] I Horrocks and P F Patel-Schneider. Optimising propositional modal satisfiability for description logic subsumption. *LNCS*, 1476, 1998.

[HS96] A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, and S4. Tech Report IAM-96-015, University of Bern, Switzerland, 1996.

[HS00] U Hustadt and R A Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In *Proc. TABLEAU'00*, LNAI 1847:67–71. Springer, 2000.

[HSZ96] A Heuerding, M Seyfried, and H Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In *Proc. TABLEAUX'96*, 1996.

[Pau93] L C Paulson. Isabelle: A Generic Theorem Prover. LNCS 828, Springer, 1994.

## Appendix 1: A Variant of Heuerding’s History-Based Calculus for S4

```
open Twblib           (* functions for manipulating set-histories  *)

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

HISTORIES
      (DIAS : Set of Formula := new Set.set);
      (UBXS : Set of Formula := new Set.set)
END

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 } ; x == a ; x[a] | b ; x[b] END

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

      RULE S4     { Dia p } ; z
                 ----------------
                   p ; UBXS
           COND   notin(Dia p, DIAS)
           ACTION [ DIAS := add(Dia p, DIAS) ]
      END
END

SIMPLIFICATION := Kopt.simpl (* functions for modal simplification *)
PP  := Kopt.nnf   (* PreProcess input formula using nnf function   *)
NEG := Kopt.neg   (* function to negate initial formula            *)

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