JTabWb: a Java framework for implementing terminating sequent and tableau calculi

Mauro Ferrari[1], Camillo Fiorentini[2], Guido Fiorino[3]

[1] DiSTA, Univ. degli Studi dell'Insubria, Via Mazzini, 5, 21100, Varese, Italy
[2] DI, Univ. degli Studi di Milano, Via Comelico, 39, 20135 Milano, Italy
[3] DISCO, Univ. degli Studi di Milano-Bicocca, Viale Sarca, 336, 20126, Milano, Italy

Abstract. JTabWb is a Java framework for developing provers based on terminating sequent or tableau calculi. It provides a generic engine which performs proof-search driven by a user-defined specification. The user is required to define the components of a prover by implementing suitable Java interfaces. The implemented provers can be used as standalone applications or embedded in other Java applications. The framework also supports proof-trace generation, LaTeX rendering of proofs and counter-model generation.

1 Introduction

JTabWb is a Java framework for developing provers based on terminating sequent or tableau calculi. The project originated as a tool for experimenting and comparing on the same ground different calculi and proof-search strategies for intuitionistic propositional logic. Now it is an advanced general framework which can be used to implement different logics and calculi. It can be used either to implement provers as stand-alone Java applications or as APIs to be integrated in other Java applications. Differently from other frameworks like [1, 4, 5], in JTabWb the user can specify all the components of a prover including: formulas, proof-search nodes, rules and strategies. This allows one to easily implement provers for different logics and different calculi (sequent-style or tableau-style calculi). Its main limitation is that all the components are provided as Java classes, hence the user is expected to be experienced with the language. The object oriented nature of the target language and the compositionality of the framework supports the reuse of the components of a prover. This allows one to easily develop different variants of a prover, so to compare different implementations of data structures (formulas, sequents,...) and different strategies. The framework also provides support for generation of proof-traces (histories of proof-search), LaTeX rendering of proofs and counter-model generation. JTabWb and some provers for intuitionistic propositional logic implemented in it are available at `http://www.dista.uninsubria.it/~ferram`.

Concept	Interface	Main methods
Formula	_AbstractFormula	String format()
Node-set	_AbstractNodeSet	String format()
Strategy	_Strategy	_AbstractRule nextRule(_AbstractNodeSet, IterationInfo)
Prover	_Prover	_Strategy getStrategy()
Engine	Engine	Engine(_Prover, _AbstractNodeSet) ProofSearchResult searchProof()
Rules supertype	_AbstractRule	String name()
Regular rule $\frac{\sigma}{\sigma_1 \mid \ldots \mid \sigma_n}\,\mathcal{R}$	_RegularRule	int numberOfConclusions() _AbstractFormula mainFormula() Iterator<_AbstractNodeSet> iterator()
Clash-detection rule A function associating with any node-set a value in {SUCCESS, FAILURE}	_ClashDetectionRule	ProofSearchResult checkStatus() _AbstractNodeSet premise()
Branch-exists rule $\frac{\sigma}{\sigma_1 \parallel \ldots \parallel \sigma_n}\,\mathcal{R}$	_BranchExistsRule	int numberOfBranchExistsConclusions() _AbstractFormula mainFormula() Iterator<_AbstractNodeSet> iterator()
Meta-backtrack rule A function associating with a node-set an enumeration of rule instances	_MetaBacktrackRule	_AbstractNodeSet premise() int totalNumberOfRules() Iterator<_AbstractRule> iterator()

Fig. 1. Basic concepts and their implementation

2 Basic notions and their implementation

JTabWb provides a generic *engine* that searches for a proof of a goal driven by a user-defined prover. In particular the engine searches for a proof of the goal visiting the proof-search space in a depth-first fashion; at any step of the search, the engine asks to the strategy component of the prover the next rule to apply. The user defines the prover by implementing the interfaces modeling the logical components of the proof-search procedure in Fig. 1. For every component we indicate the corresponding interface and its main methods.

Formulas are the basic elements of the formal system at hand; one can define formulas of any kind, e.g., propositional, first-order or modal formulas, but also "formulas with a sign" or "labelled formulas". The data structure storing formulas during proof-search is modeled by a *node-set*. E.g., in the case of a sequent calculus, node-sets represent *sequents* $[\Gamma \Rightarrow \Delta]$ where Γ and Δ are finite sets or multisets of formulas. Formulas and node-sets only require the implementation of a method format(), which is invoked by the engine to print detailed information about the proof-search process.

JTabWb models four kinds of rules: *regular*, *clash-detection*, *branch-exists* and *meta-backtrack*. *Regular* and *clash-detection* rules directly correspond to

the rules of a calculus; *branch-exists* and *meta-backtrack* rules are meta-rules to encode the proof-search strategy. All the rules have _AbstractRule as a common supertype.

Regular rules directly correspond to the usual formalization of rules in tableau calculi. A regular rule has the form displayed in Fig. 1: $\mathcal{R}$ is the name of the rule, σ is the *premise* of $\mathcal{R}$ and $\sigma_1, \ldots, \sigma_n$ ($n \geq 1$) its *conclusions*. We use a double line to represent rules of the framework so to distinguish them from the rules of the sequent calculus we use in the examples.

A rule $\mathcal{R}_s$ of a sequent calculus can be mapped to a regular rule $\mathcal{R}$ writing $\mathcal{R}_s$ bottom-up (the conclusion of $\mathcal{R}_s$ becomes the premise of $\mathcal{R}$). As an example, let us consider the rule for left disjunction of **G3i** [6]

$$\frac{[A, \Gamma \Rightarrow H] \qquad [B, \Gamma \Rightarrow H]}{[A \vee B, \Gamma \Rightarrow H]} \vee L$$

This rule is represented in our framework by the regular rule:

$$\frac{[A \vee B, \Gamma \Rightarrow H]}{[A, \Gamma \Rightarrow H] \quad | \quad [B, \Gamma \Rightarrow H]} \vee L$$

The *main formula* of a regular rule is the formula put in evidence in the premise of the rule (e.g., $A \vee B$ in the above example). An instance of a regular rule is an object implementing the _RegularRule interface. Conclusions of a regular rule are returned as an enumeration of objects of type _AbstractNodeSet. To represent an enumeration of objects we use the Java Iterator interface: it defines the method next() to get the next element in the enumeration and the method hasNext() to check whether the enumeration contains more elements or not.

Clash-detection rules model rules without conclusions corresponding to the end-points of a derivation (closure rules of tableau calculi and axiom rules of sequent calculi). We represent such a rule by a function $\mathcal{CD}$ that, given a node-set σ, returns SUCCESS if σ is an end-point of the derivation, FAILURE otherwise. As an example, the axiom rules of **G3i**

$$\frac{}{[A, \Gamma \Rightarrow A]}\ \text{Ax} \qquad\qquad \frac{}{[\bot, \Gamma \Rightarrow H]}\ L\bot$$

correspond to the following clash-detection rule:

$$\mathcal{CD}([\Gamma \Rightarrow A]) = \begin{cases} \mathsf{SUCCESS} & \text{if } \bot \in \Gamma \text{ or } A \in \Gamma \\ \mathsf{FAILURE} & \text{otherwise} \end{cases}$$

An instance of a clash-detection rule is an object that implements the interface _ClashDetectionRule providing the checkStatus() method encoding the corresponding $\mathcal{CD}$ function.

The distinction between *invertible* and *non-invertible*[1] rules has a crucial role in the definition of a proof-search procedure, since non-invertible rules introduce backtrack points in proof-search. E.g., the rules of **G3i** for right disjunction

[1] We adopt the formalization of invertible rule of [6].

```
public class Rule_Right_Or implements _BranchExistsRule {
  private Sequent premise;
  private Formula or_formula;
  public Rule_Right_Or(Sequent sequent, Formula or_formula) {
    this.premise = sequent; this.or_formula = or_formula;
  }
  ...
  public Iterator<_AbstractNodeSet> iterator() {
    LinkedList<Sequent> list = new LinkedList<Sequent>();
    Formula[] disjuncts = or_formula.getSubformulas();
    for(int i=0 ; i < disjuncts.length; i++){
      Sequent conclusion = premise.clone();
      conclusion.removeRight(or_formula);
      conclusion.addRight(disjuncts[i]);
      list.add(conclusion);
    }
    return list.iterator();
  }
}
```

Fig. 2. Implementation of the rule $\vee R_i$ of **G3i**

$$\frac{[\Gamma \Rightarrow A_i]}{[\Gamma \Rightarrow A_1 \vee A_2]} \vee R_i \quad i \in \{1, 2\}$$

are not invertible. Indeed, it could be the case that $[\Gamma \Rightarrow A_2]$ is provable while $[\Gamma \Rightarrow A_1]$ is not. Hence, searching for a proof of $[\Gamma \Rightarrow A_1 \vee A_2]$, we have to try both the rules; if the construction of a proof for $[\Gamma \Rightarrow A_1]$ fails, we have to reconsider the premise (*backtrack*) and try the other way. The two rules above can be formalized in our framework by means of a *branch-exists rule*. A branch-exists rule $\mathcal{R}$ with premise σ and conclusions $\sigma_1, \ldots, \sigma_n$ $(n \geq 1)$ means that σ is provable iff *at least one* of the σ_i is provable. An instance of a branch-exists rule is an object implementing the _BranchExistsRule interface. The iterator method returns the conclusions of the rule as an enumeration of objects of type _AbstracNodeSet. As an example, in Fig. 2 we describe an implementation of the rules $\vee R_i$ of **G3i**.

A *calculus* **C** is a finite set of regular rules, clash-detection rules and branch-exists rules. A **C**-*tree* π is a tree of node-sets such that, if σ is a node of π with $\sigma_1, \ldots, \sigma_n$ as children, then either there exists a regular-rule of **C** having σ as premise and $\sigma_1, \ldots, \sigma_n$ as conclusions, or $n = 1$ and there exists a branch-exists rule of **C** having σ as premise and σ_1 as one of its conclusions. A **C**-*proof* is a **C**-*tree* π such that, for every leaf σ of π, there exists a clash-detection rule $\mathcal{CD}$ of **C** such that $\mathcal{CD}(\sigma) = \mathsf{SUCCESS}$.

To define proof-search strategies we need to encode another kind of backtracking arising from the application of non-invertible rules. Let us consider the non-invertible rule $\rightarrow L$ of **G3i**

$$\frac{[A \to B, \Gamma \Rightarrow A] \qquad [B, \Gamma \Rightarrow H]}{[A \to B, \Gamma \Rightarrow H]} \to L$$

If we are searching for a proof of a sequent σ containing more than one implication in the left-hand side, we must try all the possible instances of the rule $\to L$ to assert the provability status of σ. To express this situation we use a *meta-backtrack* rule, which is a function $\mathcal{MB}$ associating with a node-set σ an enumeration of rule instances having σ as premise (the non-invertible rules applicable to σ). We remark that a meta-backtrack rule is not a rule of the calculus but a meta-rule to encode the proof-search strategy in presence of non-invertible rules.

The *strategy* is a function that, taken the current goal of the proof-search (a node-set) and the current state, returns the next rule to apply in the proof-search. The method nextRule(_AbstractNodeSet goal,IterationInfo info) of the _Strategy interface is a callback method invoked by the engine when it needs to determine the next rule to apply in the proof-search. The engine invokes this method providing as arguments the current goal of the proof-search and a bunch of data describing the last move performed by the engine. E.g., the method getAppliedRule() of IterationInfo returns the rule applied by the engine in the last move; in many cases this is the only data needed to choose the next rule to apply to goal. For instance, this is an high-level description of the strategy for a terminating sequent calculus for intuitionistic propositional logic (as, e.g., the calculus **LSJ** described in [2]).

```
_AbstractRule nextRule(_AbstractNodeSet sequent,
                       IterationInfo info) {
  if (an invertible rule r is applicable to sequent)
    return r;

  _AbstractRule lastAppliedRule = info.getAppliedRule();
  if (lastAppliedRule is not a clash-detection rule)
    return the clash-detection rule for sequent;

  if (non invertible rules r1...rN are applicable to sequent)
    return the meta-backtrack rule collecting r1...rN
  return null
}
```

A *prover* is an object implementing the _Prover interface which provides the getStrategy() method and some other methods that are not essential to our discussion.

3 High-level description of the engine

We give in Fig. 3 an high-level description of the algorithm implemented by the engine to perform proof-search. An instance of the engine is built by specifying the prover and the goal; searchProof() searches for a proof of the goal driven by

the strategy specified by the prover. To simplify the presentation we assume that the data passed to the strategy only consist of the rule applied in the last step. The search space is visited in a depth-first fashion using a stack to store the information related to branch points and backtrack points. More precisely, the stack contains elements (rule,iterator), where rule is the rule that caused the push action, iterator is the associated enumeration. If rule is a regular rule, the element of the stack represents a branch point, if rule is a branch-exists or a meta-backtrack rule, the element represents a backtrack point.

The method searchProof() essentially consists of a loop; we call *iteration of the engine* an iteration of such a loop. At every iteration the state of the engine is characterized by the current goal of the proof-search (i.e., current_goal), the rule to apply in the current iteration (i.e., current_rule) and the rule selected for the next iteration (i.e., next_rule). If current_rule is a regular rule or a branch-exists rule, the engine replaces the goal with the first conclusion of the rule and determines the next rule to apply by invoking the strategy. If the applied rule has more than one conclusion, then an element e is pushed on the stack by the call push(current_rule,iterator): if current_rule is a regular rule, e is a branch point, otherwise e is a backtrack point.

If current_rule is a meta-backtrack rule, the associated enumeration iterator collects the rules to be applied to current_goal. The first rule in the enumeration (returned by the method next()) is applied and, if iterator contains one or more rules, the backtrack point (current_rule, iterator) is pushed on the stack.

If current_rule is a clash-detection rule, then the engine invokes checkStatus() to determine if the current goal is an end-point of the proof-search. If checkStatus() returns FAILURE, then the strategy selects the next rule to apply. If it returns SUCCESS, then restoreBranchPoint() searches the stack for a branch point. If such a point exists, it provides the new goal and the strategy selects the next rule to be applied; if the stack does not contain any branch point, the proof-search successfully terminates. The method restoreBranchPoint() searches the stack for a branch point, namely an element (rule,iterator), where rule is a regular rule. If such an element does not exist, it returns null; otherwise, it returns iterator.next(), representing the new goal to be proved. If iterator is empty, the branch point is removed.

If current_rule is null, it means that the strategy failed to select a rule for current_goal in the last iteration of the engine, hence the proof-search for current_goal has failed. In this case the engine searches the stack for a backtrack point invoking restoreBacktrackPoint(). If a backtrack point exists, the engine appropriately updates its state and starts a new iteration. Otherwise, it returns FAILURE to signal that a proof for the input goal does not exist. The method restoreBacktrackPoint() searches the stack for a backtrack point, that is an element (rule,iterator), where rule is a branch-exists rule or a meta-backtrack rule. If such an element does not exist, null is returned; otherwise, the pair (rule,iterator.next()) is returned. In the latter case, if rule is a branch-exists rule, then iterator.next() is a node-set (the next goal to be proved); otherwise, rule is a meta-backtrack rule and iterator.next() is the next rule to be applied.

```
// goal and prover are the arguments of the engine constructor
current_goal = goal
strategy = prover.getStrategy()
next_rule = strategy.nextRule(current_goal,null)
while true do
    current_rule = next_rule // the only data about the last iteration
    if current_rule is a regular rule then
        iterator = current_rule.iterator()
        current_goal = iterator.next()
        next_rule = strategy.nextRule(current_goal,current_rule)
        if iterator.hasNext() then push(current_rule,iterator)

    end
    else if current_rule is a branch-exists rule then
        iterator = current_rule.iterator()
        current_goal = iterator.next()
        next_rule = strategy.nextRule(current_goal,current_rule)
        if iterator.hasNext() then push(current_rule,iterator)

    end
    else if current_rule is a clash-detection rule then
        if current_rule.checkStatus() == SUCCESS then
            current_goal = restoreBranchPoint() // restored is a goal or
                null
            if current_goal == null then return SUCCESS

            else next_rule = strategy.nextRule(current_goal,null)

        end
        else next_rule = strategy.nextRule(current_goal,current_rule)

    end
    else if current_rule is a meta-backtrack rule then
        iterator = current_rule.iterator()
        next_rule = iterator.next()
        if iterator.hasNext() then push(current_rule,iterator)

    end
    else if current_rule is null then
        r = restoreBacktrackPoint() // r is a pair
        if r == null then return FAILURE

        if snd(r) is a rule then //rule is a meta-backtrack rule
            next_rule = snd(r) // snd(r) is the next rule to try
            current_goal = fst(r).premise() // restore the goal from the
                rule
        else //rule is a branch-exists rule
            current_goal = snd(r) // snd(r) is the next sub-goal to try
            next_rule = strategy.nextRule(current_goal,current_rule)
        end
    end
end
```

Fig. 3. engine.searchProof()

4 Implemented provers and other features

The engine can be executed in verbose mode to get a detailed description of the proof-search or in trace-mode to generate a trace of the proof-search. It is possible to generate the LaTeX rendering of the **C**-trees visited during the proof-search (this only requires to provide the data rendering for node-sets and rule names). The trace of a proof-search can be used to generate the counter-model for unprovable goals. JTabWb also provides some useful support APIs (jtabwbx.* packages): two implementations of propositional formulas, a parser for propositional formulas and a command line launcher for a prover.

We have implemented several provers for intuitionistic propositional logic in the JTabWb framework. g3ibu is a prover based on the sequent calculi **Gbu** and **Rbu** [3]. lsj is a prover based on the sequent calculi **LSJ** and **RJ** [2]. Both these provers allow one to generate counter-models for unprovable sequents. Finally, jpintp provides the implementation of several well-known tableau and sequent calculi for intuitionistic propositional logic.

As for future work, we are developing a language to specify the components of a JTabWb prover and a library of formulas implementations and node-set implementations that can be used as building blocks for provers. Finally, we remark that the JTabWb can be used also to implement calculi for first-order logic and, in general, non terminating calculi. What is missing for fruitfully use these kind of calculi is a support which allows the user to directly control the engine execution. We are developing it as an interactive version of the engine.

References

1. P. Abate and R. Goré. The tableau workbench. *ENTCS*, 231:55–67, 2009.
2. M. Ferrari, C. Fiorentini, and G. Fiorino. Contraction-free linear depth sequent calculi for intuitionistic propositional logic with the subformula property and minimal depth counter-models. *Journal of Automated Reasoning*, 51(2):129–149, 2013.
3. M. Ferrari, C. Fiorentini, and G. Fiorino. A terminating evaluation-driven variant of G3i. In D. Galmiche and D. Larchey-Wendling, editors, *TABLEAUX 2013*, volume 8123 of *LNCS*, pages 104–118. Springer, 2013.
4. O. Gasquet, A. Herzig, D. Longin, and M. Sahade. LoTREC: Logical tableaux research engineering companion. In B. Beckert, editor, *TABLEAUX*, volume 3702 of *LNCS*, pages 318–322. Springer, 2005.
5. D. Tishkovsky, R.A. Schmidt, and M. Khodadadi. The tableau prover generator MetTeL2. In L. Fariñas del Cerro et al., editor, *JELIA*, volume 7519 of *LNCS*, pages 492–495. Springer, 2012.
6. A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.