MetTeL^2: Towards a Prover Generation Platform

Dmitry Tishkovsky, Renate A. Schmidt, and Mohammad Khodadadi

School of Computer Science, The University of Manchester, UK
{dmitry,schmidt,khodadadi}@cs.man.ac.uk

Abstract This paper introduces MetTeL^2, a tableau prover generator producing Java code from the specifications of a logical syntax and a tableau calculus. It is intended to provide an easy to use system for non-technical users and allow technical users to extend the implementation of generated provers.

1 Introduction

Building a platform for automatically generating provers from the definition of a logic is a very challenging task. As the problem of generating a deduction calculus from the definition of a logic is highly undecidable, the best that we can hope for is technology for solving the problem for certain restricted cases. The tableau method introduced in the 1950s by Beth and Hintikka based on the work of Gentzen in the 1930s and thoroughly studied by Smullyan in the 1960s has become one of the most popular deduction approaches in automated reasoning. Tableau methods in various forms exist for various logics and many implemented tableau provers exist. Based on this collective experience in the area our recent research has been concerned with trying to develop a framework for synthesising tableau calculi from the specification of a logic or logical theory. The tableau synthesis framework introduced in [6] effectively describes a class of logics for which tableau calculus synthesis can be done automatically. This class includes many modal, description, intuitionistic and hybrid logics. Our long-term goal is to synthesise not only tableau calculi but also implemented tableau provers.

As a step towards this goal we have implemented a tool, called MetTeL^2, for automatically generating an implemented tableau prover from the specification of a set of tableau rules provided by the user. MetTeL^2 is the successor of the MetTeL tableau prover [8,1]. MetTeL is a tableau prover for a large class of propositional modal-type logics, including various traditional modal logics, dynamic model logics, description logics, hybrid logics, intuitionistic logic and logics of metrics and topology. It does already allow users to specify their own tableau calculi and use MetTeL as a prover for the specified calculus. The specification language of MetTeL, though flexible, is based on a *fixed* set of logical operators common to the mentioned logics. This means there is no facility in the specification language to allow the user to define their own set of logical operators unrelated to operators of modal-type logics.

Like MetTeL, LoTREC [5] and the Tableaux Work Bench (TWB) [2] are generic tableau provers for modal-type logics. The systems differ however in

various ways, for example, in the kind of tableau approach used, the specification language provided, the way blocking is performed and configured, and the possibilities to control the way the search performed.

The three systems do not produce code for a prover but rather act as virtual machines that perform tableau derivations. On the one hand, it is not possible to accommodate within a virtual machine all imaginable requirements for new provers without giving the user appropriate flexibility in the specification language. On the other hand, any specification language necessarily restricts the user, which is useful, since it also reduces the number of potential specification errors. This dilemma can be resolved by producing prover code that is ready for possible modifications by an experienced user who may wish to tailor the prover for better performance, for a particular application and other purposes not envisaged by the prover developers.

MetTeL^2 is the successor of the MetTeL prover and considerably extends its functionality. MetTeL^2 generates Java code for a tableau prover to parse problems in the user-defined syntax and solve satisfiability problems. In order to come closer to the vision of a powerful prover generation tool, MetTeL^2 is equipped with a flexible specification language for users to define their logic or logical theory with syntactic constructs as they see fit. Thus no logical operators are predefined in MetTeL^2.

The generated tableau provers can be tuned further. Addressing the needs of an advanced user, an API of the tableau core engine is designed to accept user-defined tableau expansion strategies implemented as Java classes. The user is allowed to modify the code, for example, by implementing their own strategies for controlling the way the search is performed (search strategies, expression queues, etc) or tableau rules which might interface with other provers.

Compared with the previous MetTeL system, the tableau reasoning core of MetTeL^2 has been completely reimplemented and several new features have been added, the most important being: dynamic backtracking and conflict-directed backjumping, ordered forward and backward rewriting for operators declared to be equality and equivalence operators. There is support for different search strategies. The tableau rule specification language in MetTeL^2 now allows the specification of rule application priorities thus providing a flexible and simple tool for defining rule selection strategies. To our knowledge, MetTeL^2 is the first system with full support of these techniques for *arbitrary* logical syntax.

2 Language specification

The language in MetTeL^2 for specifying the syntax of a logical theory, is in line with the many-sorted object specification language of the tableau synthesis framework defined in [6]. We now give a simple 'non-logical' example for describing and comparing lists to illustrate how the language of a logical theory can be defined in MetTeL^2.

```
specification lists;
syntax lists{
   sort formula, element, list;
```

```
    list empty = '<>' | composite = '<' element list '>';
    formula elementInequality = '[' element '!=' element ']';
    formula listInequality = '{' list '!=' list '}';
}
```

The first line starting with the keyword **specification** defines **lists** to be the name of the user-defined logical language. The **syntax lists{...}** block consists of a declaration of the sorts and definitions of logical operators in a simplified BNF notation. Here, the specification is declared to have three sorts. For the sort **element** no operators are defined. This means that all **element** expressions are atomic. The second line defines two operators for the sort **list**: a nullary operator **<>** (to be used for the empty list) and a binary, infix operator **<..>** (used to inductively define non-empty lists). **composite** is the name of the operator **<..>**, which could have been omitted. The next two lines define how expressions of sort **formula** can be formed. For example, the line **formula listInequality = '{' list '!=' list '}';** defines an inequality operator on lists, while the previous line defines an inequality operator on elements (note the difference in notation via the brackets). This means formulae can be two types of inequality expressions. The first mentioned sort in a declaration, in our case **formula**, is the *main sort* of the defined language.

3 Tableau calculus specification

The tableau rule specification language of MetTeL2 is loosely based on the tableau rule specification language of MetTeL, but extends it in significant ways. The premises and conclusions of a rule are separated by **/** and each rule is terminated by **$;**. Branching rules can have more than two sets of conclusions and are separated by **$|** symbols. Premises and conclusions are expressions in the user-defined logical language. Additionally, the user can annotate a rule with a *priority value*. The default priority value of any rule with unspecified priority is 0. Smaller priority values imply a rule has higher priority.

Turning back to the example of the previous section, tableau rules for list comparison might be defined as follows.

```
[a != a] / priority 0$;
{L != L} / priority 0$;
{<a L0> != <b L1>} / [a != b] $| {L0 != L1} priority 2$;
```

As the parsing of rule specifications is context-sensitive the various identifiers (**a**, **L**, **L0**, etc) are recognised as symbols of the appropriate sorts. Thus *sorts* of identifiers are distinguished by their context and not their case. The first two rules are closure rules since the right hand sides of the **/** are empty. They reflect that inequality is irreflexive. The last rule is a branching rule.

4 Using MetTeL2

The binary version of MetTeL2 is distributed as a **jar**-file and requires Java Runtime Environment, Version 1.6.0 or later. MetTeL2 can be called from the command line as follows.

tableau.rule.delimiter	Terminator of tableau rules. Default: **$;**
tableau.rule.branch.delimiter	Separator between branches in branching rules. Default: **$|**
tableau.rule.premise.delimiter	Separator of premises and conclusions in rules. Default: **/**
branch.bound	An expression for computing an apriori bound on the maximal number of expressions in a branch. Default: empty, this means the feature is disabled

Figure 1. Properties accepted by MetTeL2.

```
>java −jar mettel2.jar [−i <sf>] [−t <tf>] [−d <od>] [−p <pf>]
```

A file with the syntax specification can be given using the **−i** option. A file with the specification of the tableau rules can be given with the **−t** option. If the **−t** option is specified MetTeL2 attempts to do everything for the user by generating Java code, compiling it and producing a final executable **jar**-file of the prover. In this case, Java Development Kit, Version 1.6.0 or later is required.

The directory where the generated code is placed can be given using the **−d** option.

With the **−p** option the user can specify the name of a standard Java property file where a currently small number of properties can be configured. Figure 1 lists the properties currently supported by MetTeL2. In order to be able to handle logics with eventualities a non-standard feature to realise the 'avoid huge branch strategy' (cf. [4,7]) is the **branch.bound** property. For example, this line in the Java property file

```
branch.bound = ((int)(java.lang.Math.pow(2,%l)))
```

configures the generated prover so that any branch is discarded once it contains more than $2^{\%l}$ expressions, where **%l** is the parameter for the length of the input expression.

5 Prover generation

The parser for the specification of the user-defined logical language is implemented using the ANTLR parser generator. The specification is parsed and internally represented as an abstract syntax tree (AST). The internal ANTLR format for the AST is avoided for performance purposes. The created AST is passed to the generator class which processes the AST and produces the following files: (i) a hierarchy of Java classes representing the user-defined logical language, (ii) an object factory class managing the creation of the language classes, (iii) classes representing substitution and replacement, (iv) an ANTLR grammar file for generating a parser of the user-specified language and the tableau language, (v) a main class for the prover parsing command line options and initiating the tableau derivation process, and (vi) JUnit test classes for testing the

parsers and testing the correctness of tableau derivations. In the current state, for testing purposes, most of the classes related to the derivation process are combined in a separate library. In future versions, more and more classes from this library and their extensions will migrate to the generated parts. This will allow to produce faster provers tailored for particular application areas.

The generated Java classes for syntax representation and algorithm for rule application follow same paradigm as in the old MetTeL system [8].

MetTeL2 implements two general techniques for reducing the search space in tableau derivations: dynamic backtracking and conflict directed backjumping. Dynamic backtracking avoids repeating the same rule applications in parallel branches by keeping track of rule applications common to the branches. Conflict-directed backjumping derives conflict sets of expressions from a derivation. This causes branches with the same conflict sets to be discarded. Since MetTeL2 is a prover generator, dynamic backtracking and backjumping needed to be represented and implemented in a generic way completely independent of any specific logical language and tableau rules. Particularly tricky were the computations of conflicting sets for backjumping as these are closely tied to rules of a tableau calculus. To the best of our knowledge, MetTeL2 is the first system which implements these techniques in a generic way for any logical syntax and any calculus.

The provers generated by MetTeL2 come with support for ordered backward and forward rewriting with respect to equalities appearing in the current branch. In the language specification equality expressions can be identified with one of the in-built keywords **`equality`**, **`equivalence`** or **`congruence`**. For example, the line **`formula equivalence = formula '<->' formula;`** in the logic specification defines the binary operator **`<->`** and the keyword equivalence signals that reasoning with this operator should be realised by rewriting. Each Java class representing a tableau node keeps a rewrite relation which is completed with respect to all equality expressions appearing in a branch. Once an equality expression is added within a tableau node, backward rewriting is applied. This means the rewrite relation is rebuilt with respect to the newly added equality, and all expressions of the node are rewritten with respect to the rewrite relation. Forward rewriting (with respect to the current rewrite relation) is applied to all new expressions added to the branch during the derivation.

The core tableau engine MetTeL2 provides various ways for controlling derivations. The default search strategy is depth-first left-to-right search which is implemented as a MettelSimpleLIFOBranchSelectionStrategy request to the MettelSimpleTableauManager. Also the breadth-first search is implemented as a MettelSimpleFIFOBranchSelectionStrategy and can be used after a small modification in the generated Java code. A user can also implement their own search strategy and pass it to MettelSimpleTableauManager.

The rule selection strategy can be controlled by specifying priority values for the rules in the tableau specification. Rules with the same priority values are iterated sequentially. To ensure fairness all applicable rules within the same priority group are queried for applications an equal number of times. Preference

is given to rules from groups with smaller priority values. Again the user could implement their own rule selection strategy and modify the generated code.

Blocking in tableau derivations can be implemented as variants of the unrestricted blocking rule [6]. Consider, for example, the following declarations which might be part of the language specification for a description (or hybrid) logic.

```
sort concept, individual;
concept at = '@' individual concept | negation = '~' concept;
concept equality = '[' individual '=' individual ']';
```

This defines respectively the sorts **concept** and **individual** and two operators **@** and **~**. The last line defines an equality operator **=** on individuals which is handled by rewriting. The unrestricted blocking rule can now be defined by the following tableau rule.

```
@i p @j q / [i = j] $| ~[i = j] $;
```

The purpose of the two premises here is domain predication so that the variables **i** and **j** are instantiated at rule application (cf. [6]), because symbols that do not occur in premise positions are not instantiated. In essence the rule causes individuals occurring in expressions of the form **@i p** to be systematically set equal, if this does not lead to a model on the left, then the right branch is explored. The idea is to find small/finite models. The unrestricted blocking rule ensures termination of sound and complete tableau calculus in case the specified logic has the finite model property (cf. [6,7]).

6 Using the generated provers

The generated prover **jar**-file can be run via the command line as follows.
>java −jar <prover_name>.jar [−i <if>] [−o <of>] [−t <tf>]
<prover_name> is the name of the syntax specification. An input file **<if>** can be specified via the **−i** option. If the option is not specified then input is expected from standard input. The input file must contain a list of expressions of main sort (the *first* specified sort in the syntax specification) separated by space characters. The prover will output the result to the file **<of>** if the option **−o** is given or to the standard output stream otherwise. With the **−t** option the user can specify a file with an alternative definition of a tableau calculus. If the option is omitted then the calculus specified at generation is used. If no tableau calculus was specified at generation then a tableau calculus definition must be provided now, which can be done with the **−t** option.

The provers return the answers **Satisfiable** or **Unsatisfiable**. If the answer is **Unsatisfiable** and the prover is able to extract the input expressions needed for deriving the contradiction they are printed. If the answer is **Satisfiable** then all the expressions within the completed open branch are output as a **Model**.

7 Conclusion

Several test cases have been prepared for the system covering a variety of logics including Boolean logic, modal logic S4, description logics $\mathcal{ALCO}$ and $\mathcal{ALBO}^{\mathsf{id}}$ [7],

a hybrid logic with graded modalities and linear-time temporal logic, with or without capacity constraints [4]. Sample specifications with unrestricted blocking are the tableau calculi for S4, $\mathcal{ALBO}^{\mathsf{id}}$ and linear-time temporal logic (with constraints). Some of these test cases and the **`lists`** example from this paper (as well as an extended version with a concatenation operator) are available at [1].

MetTeL^2 can be download from [1]. A web-interface for MetTeL^2 is provided, where a user can input their specifications in two syntax aware textareas. The user can either download the generated prover as a **`jar`**-file or directly run the generated prover in the interface. Using the web-interface consists of three steps. The first step is defining the syntax of the logical theory. In this step the user has the chance to select one of several predefined examples. The second step is defining a tableau calculus. A proper tableau calculus will be automatically provided if the user has selected one of the predefined logics, which then can be edited or replaced. In the third step the prover is generated. These test cases are accessible through the dropdown menu in the first step.

MetTeL^2 and MetTeL are small but essential steps to the very ambitious goal to create a reliable and easy to use prover generation platform which implements the automated synthesis framework [6]. Our philosophy is slightly different from systems such as LoTREC [5] and the Key system [3], as we do not aim to provide a sophisticated meta-programming languages, but to provide easy to use systems, for the non-technical users, and for the technical user, expandable systems by allowing them to write their own Java classes and integrate them using the provided API.

References

1. MetTeL website. `http://www.mettel-prover.org`.
2. P. Abate and R. Goré. The tableau workbench. *Electronic Notes in Theoretical Computer Science*, 231:55–67, 2009.
3. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
4. D. Dixon, B. Konev, R. A. Schmidt, and D. Tishkovsky. A labelled tableau approach for temporal logic with constraints. Manuscript, submitted for publication, available at `http://www.mettel-prover.org/papers/dkst12.pdf`, 2012.
5. O. Gasquet, A. Herzig, D. Longin, and M. Sahade. LoTREC: Logical tableaux research engineering companion. In *Proc. TABLEAUX'05*, vol. 3702 of *LNCS*, pp. 318–322. Springer, 2005.
6. R. A. Schmidt and D. Tishkovsky. Automated synthesis of tableau calculi. *Log. Methods Comput. Sci.*, 7(2:6):1–32, 2011.
7. R. A. Schmidt and D. Tishkovsky. Using tableau to decide description logics with full role negation and identity. Manuscript, available at `http://www.mettel-prover.org/papers/ALBOid.pdf`, 2011.
8. D. Tishkovsky, R. A. Schmidt, and M. Khodadadi. MetTeL: A tableau prover with logic-independent inference engine. vol. 6793 of *LNCS*, pp. 242–247. Springer, 2011.

Appendix

This appendix is included purely for the benefit of the interested reviewer.

Additional information for using MetTeL2

What happens when none of the options are provided when running MetTeL2?
>java −jar mettel2.jar [−i <sf>] [−t <tf>] [−d <od>] [−p <pf>]
In the case that the **−i** option is omitted, MetTeL2 waits for a language specification from standard input. If the **−t** option is not given, MetTeL2 will not generate a **jar**-file of the prover. In this case only Java code for the prover is generated. This is useful, for example, if the user is going to amend the code with the aim of tailoring the prover for performance or defining a non-standard feature, or simply wishes to compile the code by a Java compiler provided by a different vendor. Whenever **−d** is not specified the default directory for output of Java code is the subdirectory **output** of the current directory. If the **−p** option is omitted then the default values are used.

Additional information for running a generated prover

How can a generated prover be run? Considering our list example, the user can run the prover generated from the syntax and tableau specifications in Sections 2 and 3 as follows.
>java −jar lists.jar
Since the **−i** option is not specified the prover will wait for input from the terminal. Suppose **{<a (<b L>)> != <a (<b L>)>}** is typed (and finished by pressing the **<Ctrl−D>**). The output is

```
Unsatisfiable.
Contradiction: [({(<a (<b L>)>) != (<a (<b L>)>)})]
```

For the input **{<a (<b L0>)> != <a (<b L1>)>}** the output is

```
Satisfiable.
Model: [({(<a (<b L0>)>) != (<a (<b L1>)>)}), ({(<b L0>) !=
(<b L1>)}), ({L0 != L1})]
```