MetTeL2: Towards a Tableau Prover Generation Platform

Dmitry Tishkovsky, Renate A. Schmidt, Mohammad Khodadadi*

The University of Manchester, UK

Abstract

This paper introduces MetTeL2, a tableau prover generator producing Java code from the specification 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 generated implementations.

1 Introduction

Building a platform for automatically generating provers from the definition of a logic is a 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 was introduced in the 1950s by Beth and Hintikka [8, 16]. With origins in the work of Gentzen in the 1930s [13] and thoroughly studied by Smullyan in the 1960s [29], it has become one of the most popular deduction approaches in automated reasoning. Tableau methods in numerous forms exist for various logics and many implementations of tableau provers are available.

Based on the 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. The tableau synthesis framework introduced in [27] 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 implementations of the tableau calculi as tableau provers.

As a step towards this goal we have implemented a tool, called MetTeL2, for automatically generating code of a tableau prover from user-defined specifications of a syntax and a set of tableau rules for a logical theory. The syntax and tableau rule specification languages of MetTeL2 are designed to be as simple as possible for the user and to be as close as possible to the traditional notation used in logic and automated reasoning textbooks. At the moment the syntax specification language is limited to multi-sorted propositional languages with finitary connectives. The tableau calculus specification language covers different types of tableau calculi that fit the traditional representation of tableau rules of the form $X_0/X_1 \mid \cdots \mid X_m$, where the X_i denote finite sets of expressions of the given logical theory. X_0 is a set of premises and $\{X_1, \ldots, X_m\}$ is a finite set of branches of the rule. Many labelled semantic tableau calculi for modal, description, hybrid and superintuitionistic logics belong to this paradigm.

MetTeL2 is complementary to the mentioned tableau synthesis framework [27]. The framework provides a theoretical foundation for sound, complete and terminating implementations of tableau procedures for a wide class of logics with first-order representable semantics and, in particular, for many logics which can be specified in MetTeL2. The scope of MetTeL2

*This research is supported by EPSRC research grant EP/H043748/1.

extends however that of tableau calculi derived in the framework and is not limited to semantic or labelled tableau calculi.

MetTeL2 is the successor of the MetTeL system [31, 1]. MetTeL is a tableau prover for a large class of propositional modal-type logics, including various traditional modal logics, dynamic modal 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 then use MetTeL as a prover for the specified calculus. Though flexible, the specification language of MetTeL 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.

The functionality of MetTeL2 considerably extends that of the MetTeL prover. MetTeL2 generates Java code for a tableau prover to parse problems in the user-defined syntax to solve satisfiability problems. In order to come closer to the vision of a powerful prover generation tool, MetTeL2 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 MetTeL2.

Compared with the previous MetTeL system, the tableau reasoning core of MetTeL2 has been completely reimplemented and several new features have been added, the most important being: dynamic backtracking [14] and conflict-directed backjumping [11, 25], and 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 MetTeL2 now allows the specification of rule application priorities thus providing a flexible and simple tool for defining rule selection strategies. To our knowledge, MetTeL2 is the first system with full support of these techniques for *arbitrary* logical syntax.

The aim of the current implementation is to provide an easy to use prover generator with basic specification languages without sophisticated meta-programming features that might overwhelm non-technical users. For technical users, the generated code consists of a thoroughly designed hierarchy of public Java classes and interfaces that can be extended and integrated with other systems.

The paper is structured as follows. We introduce the syntax and tableau specification languages of MetTeL2 in Sections 2 and 3. Section 4 describes a common scenario of using MetTeL2 as a prover generator. Details on implementation, prover generation, integrated optimisations, and features for controlling derivations are given in Sections 5, 6, and 7. We show how generated provers can be run and discuss their output in Section 8. The scope of MetTeL2 and further features are illustrated in Section 9. In Section 10 we discuss possible applications of MetTeL2 and our experience of using the system. Sections 11 and 12 give an overview of related work and further directions for development of the system.

2 Language specification

The language of MetTeL2 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 [27]. We give a simple 'non-logical' example for describing and comparing lists to illustrate how the language of a logical theory can be defined in MetTeL2.

```
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 the declaration of the sorts and definitions of logical operators in 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 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. Several declarations of connectives for the same sort are equivalent to a joint statement which is composed from these declarations by means of the operator |. For example, the two statements for formula are equivalent to the following statement:

```
formula elementInequality = '[' element '!=' element ']' | listInequality = '{' list '!=' list '}';
```

Another example we consider is three-valued Łukasiewicz logic Ł$_3$. The syntax of the logic includes the standard connectives $\bot$, $\top$, $\neg$, $\vee$, $\wedge$, and $\rightarrow$. Truth values are considered over three-element Łukasiewicz algebra over the set $\{0, \frac{1}{2}, 1\}$. The elements of the algebra are naturally ordered. The algebra operations correspond to the connectives of the logic and are defined for any elements a and b from the algebra as follows [20, 17, 30].

$$\begin{array}{lll} \bot \stackrel{\text{def}}{=} 0 & a \wedge b \stackrel{\text{def}}{=} \min\{a, b\} & \neg a \stackrel{\text{def}}{=} 1 - a \\ \top \stackrel{\text{def}}{=} 1 & a \vee b \stackrel{\text{def}}{=} \max\{a, b\} & a \rightarrow b \stackrel{\text{def}}{=} \min\{1, 1 - a + b\} \end{array}$$

The syntax specification for the logic Ł$_3$ with respect to the three-valued Łukasiewicz algebra is the following:

```
specification Lukasiewicz3;
syntax Lukasiewicz3{
    sort valuation, formula;
    valuation true = 'T' formula | unknown = 'U' formula | false = 'F' formula;
    formula true = 'true' | false = 'false';
    formula negation = '~' formula;
    formula conjunction = formula '&' formula;
    formula disjunction = formula '|' formula;
    formula implication = formula '->' formula;
}
```

Thus, in the specification we denote truth value 0 as `F` ('false'), truth value $\frac{1}{2}$ as `U` ('unknown'), and truth value 1 as `T` ('true'). In order to indicate that a formula ϕ has truth value a in the Łukasiewicz algebra we prepend ϕ with a, that is **`U (p -> q)`** means that the formula **`p -> q`** has 'unknown' truth value.

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. Roughly speaking (see Section 7), smaller priority values imply a rule is applied earlier.

Turning back to the examples 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] $| {L0 != L1} priority 2$;
```

The first two rules are closure rules since the right hand sides of / are empty. They reflect that inequality is irreflexive. The last rule is a branching rule. 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 contextual position within the rule and not their symbolic representation.

A tableau calculus for the three-valued Łukasiewicz logic is given in [17]. The following is its specification in MetTeL2 syntax.

```
T false /  priority 0$;                    U false /  priority 0$;
U true /  priority 0$;                     F true /  priority 0$;
T P  F P /  priority 0$;                   T P  U P /  priority 0$;
U P  F P /  priority 0$;                   U P  F P /  priority 0$;
T ~P / F P  priority 1$;   U ~P / U P  priority 1$;   F ~P / T P  priority 1$;
T (P & Q) / T P  T Q  priority 2$;         F (P & Q) / F P  $|  F Q  priority 1$;
U (P & Q) / T P  U Q $|  U P  T Q  $|  U P  U Q  priority 3$;
T (P |  Q) / T P $| T Q  priority 2$;      F (P |  Q) / F P  F Q  priority 1$;
U (P |  Q) / F P  U Q $|  U P  F Q  $|  U P  U Q  priority 3$;
F (P −> Q) / T P  F Q  priority 1$;        U (P −> Q) / U P  F Q  $|  T P  U Q  priority 2$;
T (P −> Q) / T Q  $|  F P  $|  U P  U Q  priority 3$;
```

The first eight rules are closure rules which detect contradictions between truth values of formulae. All of them have the highest priority (priority value 0). The rest of the rules reflect the truth tables for the connectives of the logic (in the Łukasiewicz algebra) and are given priorities proportional to their branching factors. This means that the higher the branching factor of a rule, the less often the rule is applied in tableau derivations.

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.

```
>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 source code, compiling it and producing a final executable **jar**-file of the prover. In this case, Java Development Kit,

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.

Version 1.6.0 or later is required. The directory where the generated Java source 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 currently a 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., [10, 28]) 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. **%l** is the only pattern variable available in the current MetTeL2 implementation but we plan to introduce several other patterns, e.g., pattern variables that reflect the number of atomic expressions of each sort in the input. The property file is also reserved for other non-standard features, flag settings, and definitions that may be required for advanced tuning of the prover generation process.

All the options are optional. 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 the default values are used.

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 version, 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 the production of faster provers tailored for particular application areas.

The generated Java classes for syntax representation and algorithms for rule application follow the same paradigm as in the previous MetTeL system [31]. All generated Java classes for the syntax representation are specialisations of the basic MettelExpression interface. For efficiency reasons, each kind of expression is represented as a separate Java class, which is not parameterised by operators. At runtime, the creation of expression objects is managed by means of a factory pattern generated as a specialisation of the base interface MettelObjectFactory and ensures that each expression is represented by a single object. Each generated expression class implements several methods. The two most important are: (i) a method for matching the current object with the expression object supplied as a parameter. This method returns the substitution unifying the current expression with the parameter. (ii) The second method returns an instance of the current expression with respect to a given substitution.

Every node of the tableau is represented as a tableau state object comprising of a set of formulae associated with the node and methods for manipulating the formulae and realising rule applications.

The application of rules is implemented as follows. Every rule is applied within a tableau state. A tuple of formulae from the set of active formulae associated with the tableau state is selected and the formulae in the tuple are matched with the premises of the chosen rule. Since matching is computationally expensive, it is performed only once for any given formula and each premise of a rule. This is achieved by maintaining sets of all the substitutions obtained from matching the selected formula with the rule premises. All the selected formulae are discarded from the set of active formulae associated with the rule. If the tuple of the selected formulae match the premises of the rule, the resulting substitution object is passed to the conclusions of the rule. The final result of a rule application is a set of branches, which are sets of formulae obtained by applying the substitution to the conclusions of the rule.

6 Built-in optimisations

MetTeL2 implements two general techniques for reducing the search space in tableau derivations: dynamic backtracking [14] and conflict directed backjumping [11, 25]. 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. Although dynamic backtracking and backjumping are known for some time and have been used in many tableau provers they are usually defined for a particular tableau procedure which is based on some fixed syntax and fixed tableau calculus. The implementation in MetTeL2 is especially involved for the case of backjumping where calculations of conflicting sets are closely tied to the rules of the 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 performance achieved by these optimisations has not been specially tested, but, due to these optimisations, the total execution time of the generated provers on examples developed for testing the MetTeL2 generator and the generated provers decreased more than 100-fold.

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 built-in keywords **equality**, **equivalence** or **congruence**. For example, the declaration formula **equivalence** = formula '<->' formula; in the logic specification defines the binary operator <->. 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 completed with respect to all equality expressions appearing in a branch. Since only ground expressions are allowed in a branch, the rewrite relation operates only on ground expressions. Every equality expression is oriented by a lexicographic path ordering $\prec$ based on the order of creation of atomic expressions in the branch. Hence, any (ground) rewrite system based on any such ordering $\prec$ is terminating. Thus, if an equality expression $\alpha \leftrightarrow \beta$ appears on the branch one of the rewrite rules $\alpha \xrightarrow{\mathcal{R}} \beta$ or $\beta \xrightarrow{\mathcal{R}} \alpha$ is added to the rewrite relation depending on whether $\beta \prec \alpha$ or vice versa.

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.

In future we plan to provide an implementation of a completion procedure for added equalities which takes a care about dependencies of derived rewrite rules for more efficient backtracking.

7 Controlling derivations and blocking

The core tableau engine of 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. Breadth-first search is implemented as a MettelSimpleFIFOBranchSelectionStrategy request 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. In future more search strategies will be implemented (e.g., strategies with iterative deepening) and the choice of strategy will be configurable at the generation stage.

The rule selection strategy can be controlled by specifying priority values for the rules in the tableau calculus specification. The rule selection algorithm checks the applicability of rules and returns a rule that can be applied to some expressions on the current branch according to the rule priority values.

First, the algorithm selects a group of rules with the same priority value. Selection of a group with higher priority value is made only if no rules with smaller priority values are applicable. That is, if several rules are applicable preference is given to rules from groups with smaller priority values.

Second, rules with the same priority values are checked for applicability sequentially. Usually, fair application of rules is a necessary condition to achieve completeness of a tableau derivation. An application of rules during derivation is *fair* if every rule which is applicable to some expressions in a branch is eventually applied to these expressions or a contradiction is detected beforehand. To ensure fairness for rules within the same priority group all rules within the group are checked for applicability an equal number of times. For example, given a single closure rule, to achieve that the closure rule is applied immediately after any new information is added to a branch the user can assign to the closure rule the priority value 0 and to all other rule values higher than 0, e.g., 1. If, however, several rules are assigned the same priority value

as the closure rule it can happen that several tableau expansion rules are applied before the branch is checked for contradictions via the closure rule. Furthermore, if the closure rule has a priority value strictly greater than other rules, then the branch is checked for contradiction via the closure rule only after it has been fully expanded, i.e., no expansion rule is applicable to it. More subtle control for application of the closure rules can be also achieved, e.g., to ensure that the closure rules are applied only after particular expansion rules. On the side we remark that it is not necessarily a good idea to give closure rules highest priority in a group by themselves as then performance may degrade because contradiction testing dominates the search.

Again the user could implement their own rule selection strategy and modify the generated code.

To achieve termination for semantic tableau approaches some form of blocking is usually necessary. To generate a prover with blocking the user can specify a blocking rule similar to the unrestricted blocking rule from [28] as one of the rules of the tableau calculus. If the definition of the rule involves equality operators then rewriting is triggered (see above), and, based on the results in [27, 28], the blocking rule can be used to achieve termination for logics with the finite model property.

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, on application of the rule, the variables i and j are instantiated by individuals which are present in the current tableau node (cf. [27]), 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 to be equal. If this does not lead to a model in the left subtableau, then the right branch is explored.

The idea of unrestricted blocking rule is to ensure termination of sound and complete tableau calculus in case the specified logic has the finite model property (cf. [27, 28]) and find finite models. The first of the two termination conditions in [27, 28] is automatically true because the generated provers are equipped with ordered rewriting. The second termination condition can be satisfied by using appropriate priority values for tableau rules of the tableau calculus. Currently, it is not yet possible to emulate standard blocking techniques such as subset and equality blocking but by varying the specification of the blocking rule it is possible to perform blocking more selectively [4, 18].

8 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 the standard input. The input file must contain a list of expressions of the 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 generated 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**.

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 **<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})]
```

In order to test validity of a formula in the three-valued Łukasiewicz logic $Ł_3$ we have to run the prover two times with truth values `U` and `F` for the formula (cf. [17]). For example, to show that **p −> (q −> p)** is a theorem in $Ł_3$ we run the prover two times for each expression **U p −> (q −> p)** and **F p −> (q −> p)** using the following command.

```
>java −jar Lukasiewicz3.jar
```

In the case if the input is **U p −> (q −> p)** we get the following output from the prover:

```
Unsatisfiable.
Contradiction: [( U ( p −> ( q −> p ) ) )]
```

For the input **F p −> (q −> p)** we also obtain a contradiction:

```
Unsatisfiable.
Contradiction: [( T p ), ( F p ), ( F ( p −> ( q −> p ) ) ),
( F ( q −> p ) )]
```

Thus, there is no interpretation of variables **p** and **q** in the three element Łukasiewicz algebra that makes the truth value of the formula **p −> (q −> p)** different from `T`. That is, **p −> (q −> p)** is a theorem of $Ł_3$.

9 Illustration of scope and further features

MetTeL2 is designed for propositional logics and not supposed to deal with first-order languages. Nevertheless, the syntax specification language of MetTeL2 has enough expressive

power to represent languages of first-order theories with a finite number of predicate and functional symbols. Predicate and functional symbols of such a theory can be defined as connectives of formula sort and term sort respectively. For example, the following is a specification of a language of a first-order theory which contains a constant c, a unary function symbol f, a binary function symbol g, a unary *constant* predicate symbol P, and a binary *constant* predicate symbol Q.

```
specification fotheory;
syntax SomeFOTheory{
    sort formula,term,var;
    term var = '!' var;
    term c = 'c';
    term f = 'f' '(' term ')';
    term g = 'g' '(' term ',' term ')';
    formula true = 'true' | false = 'false';
    formula P = 'P' '(' term ')';
    formula Q = 'Q' '(' term ',' term ')';
    formula negation = '~' formula;
    formula conjunction = formula '&' formula;
    formula disjunction = formula '|' formula;
    formula implication = formula '->' formula;
    formula equivalence = formula '<->' formula;
    formula universalQuantifier = 'forall' var formula;
    formula existentialQuantifier = 'exists' var formula;
}
```

The additional connective ! is required to embed sort of variables var into the sort of terms term. The following expression is an example of a well-formed formula in the specified syntax.

```
forall x (forall y (~ P(g(f(!x),!y))) | (exists y Q(f(!y),!x)))
```

Notice that predicate symbols P and Q are constant predicate symbols. In fact, under the above approach to syntax specification, every predicate symbol of a given theory is a constant symbol. The effect of this becomes apparent in interpretation tableau rules where substitutions for these symbols will be forbidden. While the rules for Boolean connectives do not pose a problem and remain the same as for Boolean logic, the standard rules for quantifiers must be appropriately instantiated for every predicate symbol. Correct specification of a complete tableau calculus in such cases is tedious and even an impossible task for some theories.

A more promising approach to deal with first-order theories in MetTeL2 is to represent the given finitely defined first-order theory as a deductive system of formula schemes [32]. Since every deductive system in the sense of [32] is a propositional multi-modal logic, it can be naturally specified in MetTeL2.

For the curious reader we also give an example of how to specify in MetTeL2 a rule for the $\Diamond$ operator of standard modal logics. The syntax specification has to include a definition of an additional connective which corresponds to a Skolem function. For example, the syntax of a (hybrid) modal logic can include the following lines.

```
sort formula, individual;
formula singleton = '{' individual '}';
formula at = '@' individual formula;
formula diamond = '<>' formula;
individual SkolemTerm = 'f' '(' individual ',' formula ')';
```

Thus, the symbolic representation of the $\Diamond$ operator is <> and f(.,.) is a new binary connective such that f(i,p) is an individual for any individual i and formula p. In this syntax, the standard $\Diamond$ rule is as follows.

@i(<> p) / @i(<> f(i,p)) @f(i,p) p $;

That is, for every instance of the formula @i(<>p) in current tableau branch, a new individual term f(i,p) is created (where i and p are appropriately instantiated) and two formulae @i(<> f(i,p)) and @f(i,p) p are added to the branch. The formula @i(<> f(i,p)) states that the individual f(i,p) is accessible from i.

Using Skolem terms instead of newly generated individual constants avoids the need to perform again some inference steps on the same branch and in combination with ordered rewriting can considerably reduce the search space. Furthermore, Skolem terms add flexibility to tableau specification language because Skolem functions are legal not only in conclusions of tableau rules but also in their premises.

10 Application areas and experiences so far

Software to generate code for provers is useful anywhere where automated reasoning is needed. The provers generated by MetTeL2 also output models for for satisfiable problems on termination, so can be used for model generation purposes.

With MetTeL2 a quick implementation of a tableau prover can be obtained and changes can be made without programming a single line of code. Prover generation is useful for obtaining provers for newly defined logics or new combinations of logics. This is particularly pertinent to an area such as multi-agent systems where the logics are staggering complex. In ongoing work we are using MetTeL2 in combination with the tableau synthesis framework to develop provers for multi-agent interrogative epistemic logics [21]. For these logics and related dynamic epistemic logics there are almost no implemented reasoning tools. Therefore being able to generate tableau provers is very useful especially to researchers without the resources or expertise to implement automated reasoning tools themselves.

We have found MetTeL2 useful for analysing tableau calculi under development whose properties are not known yet. For example, in research conducted for [19] we used MetTeL2 to determine the refinability or unrefinability of tableau rules for a modal logic with global counting quantifiers operators. MetTeL2 can also be used to compare the effectiveness of different sets of tableau rules for the same logic. For example, with minimal effort it is possible to compare the effectiveness of standard tableau calculi with calculi following the KE approach where disjunction is handled by an analytic cut rule and a unit propagation rule.

Concrete case studies we have undertaken with MetTeL2 include implementing unlabelled tableau calculi for Boolean logic and three-valued Łukasiewicz logic, labelled tableau calculi for standard modal logics K, KT, S4, description logic $\mathcal{ALCO}$, a hybrid logic K(E_n) with global counting operators [19], a multi-agent interrogative epistemic logic [21], and internalised tableau calculi for hybrid and description logics. We used MetTeL2 to implement a tableau decision procedure for $\mathcal{ALBO}^{\mathsf{id}}$, a description logic with the same expressive power as the two-variable fragment of first-order logic. Some of these test cases, including the Lukasiewicz3 example and the lists example from this paper (as well as an extended version of the lists example with a concatenation operator) are available at the MetTeL website [1].

11 Related work

A fast and robust method to obtain a prover for a given logic is to translate the logic into a more expressive target logic for which an automated reasoning tool is available. For instance, translation of modal and description logics into first-order logic has been extensively researched, and specialised translation approaches have been developed which use first-order resolution theorem provers [9, 22, 26] or provers for higher-order logic such as LEO II [6] (see [5] for the approach). Translation approaches require however the user is familiar with the target logic. In addition it requires knowledge of the deduction approach and prover for the target logic so that the user can appropriately use the prover and understand its output. Experience shows it is unrealistic to expect users to learn a new language, a new theory, and capabilities and flag settings of the prover being used.

Several dedicated systems exist for developing and prototyping tableau provers for modal-type logics. The Logics Workbench [15] implements a suite of generic decision procedures for propositional logic and numerous non-classical logics and includes a full programming language. In the eighties, work on developing languages for programming in non-classical logics has evolved into the generic tableau prover development platform LoTREC [12]. The Tableau Workbench (TWB) [2] provides a generic prover development platform for modal logics. Both LoTREC and the TWB give users the possibility to program their own tableau prover for modal-type logics using meta-programming languages for building and manipulating formulae, and controlling the tableau derivation process. The Logics Workbench, LoTREC, the TWB and also MetTeL differ 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. Although they have notable features compared to MetTeL2 none of them have the facility for the user to define their own set of logical operators unrelated to the built-in operators.

Support for user-definable languages and rule sets can be found in logical frameworks such as PVS [23], Isabelle/HOL [24] and Coq [7]. These are based on higher-order logic and can be used for prototyping calculi and deductive systems. There are also formal software development tools such as the KeY system [3] which allow the specification of logical theories via a taclet mechanism containing not only rule declarations, but also usage pragmatics.

All these systems are however not designed to produce executable code for a prover but rather act as virtual machines that perform derivations (in some cases interactively but often fully automatically). Even though some incorporate various support tools and are extensible, within a virtual machine it is not possible to accommodate 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. This is actually useful, since it also reduces the potential number of specification errors. MetTeL2 addresses this inescapable dilemma by generating prover code that is ready for possible modifications by an experienced user who may wish to incorporate, for example, specialised simplification routines and optimisation techniques for better performance, or who may want to incorporate the prover into a larger systems requiring automated reasoning, or add other features not supported by the prover generator, the prover engineering platform or the logical framework being used.

12 Conclusion

MetTeL2 is a prototypical system intended for experimenting with tableau calculi and prover generation for various logics. It is a small but essential step to the very ambitious goal to create a

reliable and easy to use prover generation platform which implements the automated synthesis framework [27]. In line with this goal we intend to expand the system in various ways. In particular, we will give the user more flexibility in controlling derivations by specifying various search heuristics as well as reduction orderings for the ordered rewriting at the generation stage. This will allow the user to contribute to the improvement of the generated provers. We are going to implement a selection of standard blocking mechanisms and also various generic blocking mechanisms based on specialisations of the unrestricted blocking rule. These will help to improve the performance of the generated provers, especially for non-compact logics such as temporal and dynamic logics. Finally, a comparison with other provers and the design of benchmarking suites is necessary in order to estimate the performance of generated provers and to indicate directions for further development of the prover generator.

MetTeL2 can be downloaded from [1]. A web-interface for MetTeL2 is also provided, where a user can input their specifications in syntax aware textareas and generate provers. The user can either download the generated prover as a **`jar`**-file or directly run the generated prover in the interface.

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] P. Baumgartner and R. A. Schmidt. Blocking and other enhancements for bottom-up model generation methods, 2008. Manuscript, short version published in *Proc. IJCAR 2006*.

[5] C. Benzmüller and L. C. Paulson. Multimodal and intuitionistic logics in simple type theory. *Logic Journal of the IGPL*, 18(6):881–892, 2010.

[6] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II: A cooperative automatic theorem prover for classical higher-order logic. In *Proc. IJCAR'08*, vol. 5195 of *LNCS*, pp. 162–170. Springer, 2008.

[7] Y. Bertot and P. Castéran. Interactive theorem proving and program development. CoqArt: The calculus of inductive constructions. In *Texts in Theoretical Computer Science*, vol. 35. Springer, 2004.

[8] E. W. Beth. *The Foundations of Mathematics*. North-Holland, 1959.

[9] H. De Nivelle, R. A. Schmidt, and U. Hustadt. Resolution-based methods for modal logics. *Logic J. IGPL*, 8(3):265–292, 2000.

[10] C. Dixon, B. Konev, R. A. Schmidt, and D. Tishkovsky. A labelled tableau approach for temporal logic with constraints. Manuscript, `http://www.mettel-prover.org/papers/dkst12.pdf`, 2012.

[11] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, 1979.

[12] 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.

[13] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 1934.

[14] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Proc. KR'94*, pp. 226–237, 1994.

[15] A. Heuerding, G. Jäger, S. Schwendimann, and M. Seyfried. The Logics Workbench LWB: A snapshot. *Euromath Bull.*, 2(1):177–186, 1996.

[16] J. Hintikka. Form and content in quantification theory. In *Two Papers on Symbolic Logic*, vol. 8 of *Acta Philosophica Fennica*, pp. 7–55. 1955.

[17] R. Hhnle. Tableaux and related methods. In *Handbook of Automated Reasoning*, pp. 101 – 178. North-Holland, 2001.

[18] M. Khodadadi, R. A. Schmidt, and D. Tishkovsky. An abstract tableau calculus for the description logic $\mathcal{SHOI}$ using unrestricted blocking and rewriting. In *Proc. DL-2012*, 2012. To appear.

[19] M. Khodadadi, R. A. Schmidt, D. Tishkovsky, and M. Zawidzki. Terminating tableau calculi for modal logic K with global counting operators. Manuscript, `http://www.mettel-prover.org/papers/KEn12.pdf`, 2012.

[20] J. Łukasiewicz and A. Tarski. Investigations into the sentential calculus. *Logic, Semantics, Metamathematics*, pp. 38–59, 1956.

[21] Ş. Minică, M. Khodadadi, R. A. Schmidt, and D. Tishkovsky. Synthesising and implementing tableau calculi for interrogative epistemic logics, 2012. In these Proceedings.

[22] H. J. Ohlbach, A. Nonnengart, M. de Rijke, and D. Gabbay. Encoding two-valued nonclassical logics in classical logic. In *Handbook of Automated Reasoning*, pp. 1403–1486. Elsevier, 2001.

[23] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. CADE-11*, vol. 607 of *LNAI*, pp. 748–752. Springer, 1992.

[24] L. C. Paulson. *Isabelle: A Generic Theorem Prover (with a contribution by T. Nipkow)*, vol. 828 of *LNCS*. Springer, 1994.

[25] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[26] R. A. Schmidt and U. Hustadt. First-order resolution methods for modal logics. To appear in *Volume in memoriam of Harald Ganzinger*, 2006.

[27] R. A. Schmidt and D. Tishkovsky. Automated synthesis of tableau calculi. *Log. Methods Comput. Sci.*, 7(2:6):1–32, 2011.

[28] R. A. Schmidt and D. Tishkovsky. Using tableau to decide description logics with full role negation and identity. Manuscript, `http://www.mettel-prover.org/papers/ALBOid.pdf`, 2011.

[29] R. M. Smullyan. *First Order Logic*. Springer, 1971.

[30] D. Tishkovsky. On Beth property in extensions of Lukasiewicz logics. *Siberian Mathematical Journal*, 43:147–150, 2002.

[31] D. Tishkovsky, R. A. Schmidt, and M. Khodadadi. MetTeL: A tableau prover with logic-independent inference engine. In *Proc. TABLEAUX'11*, vol. 6793 of *LNCS*, pp. 242–247. Springer, 2011.

[32] D. E. Tishkovsky. On algebraic semantics for superintuitionistic predicate logics. *Algebra Logic*, 38(1):36–50, 1999.