MULTITAB: An automated theorem prover for three-valued logic (Bachelorproject)

Rogier de Weert, s1985779, p.r.de.weert@student.rug.nl
L.C. Verbrugge*

February 17, 2016

Abstract

Automatic theorem provers are systems designed to solve formal logical inferences. The prover (MULTITAB) discussed in this paper is designed to solve inferences in five different multiple-valued logics, of which four are 3-valued logics, and one is a 4-valued logic. MULTITAB is based on the tableau method, and uses a strategy to solve inferences by creating the smallest possible tableau for any inference. Different settings for the creation of the tableau and visualization of it were tested on correctness and speed. Each setting was tested for each logic on 16 formal logical inferences. These results were then combined to form groups for each settings and groups for each logic. These combined groups were used in an ad-hoc analysis using Wilcoxon signed rank tests, leading to significant differences in speed found between logic groups and between setting groups. *MULTITAB* finds solutions in less than a tenth of a second.

1 Introduction

Logical studies usually have propositional logic and classical first-order logic [1] as their subject, the first being the basis to most of the other logics we know of, and the second being the logic thought of as closely related to human reasoning. Multiple-valued logics, like First Degree Entailment (FDE), Priest's logic of paradox (LP) and the Mix Logic (RM_3)[2, chap. 8] are not studied extensively outside of universities. Artificial intelligence students, for example, learn of multiple-valued logic, but usually focus more on propositional and first-order logic further in their (student) career. Evidence for this is that there currently are very few automated proof systems for 3-valued logics available for free, but a whole lot of proof systems for propositional and classical logic are available for free [3]. In multiple-valued logics there are (as the name indicates) multiple truth values. For the logic of paradox (LP) and the mix logic (RM_3) there are three truth values, indicating true (1), false (0) and both (b). For Kleene's logic (K_3) and Łukasiewicz' logic ($Ł_3$) there are three truth values, indicating true (1), false (0) and neither (n). *Both* indicates that a formula is true and false at the same time, whereas *neither* indicates that a formula is neither true nor false. FDE has four truth values: true (1), false (0), both (b) and neither (n). LP, RM_3, K_3 and $Ł_3$ are three-valued logics, and FDE is a four-valued logic. Having the possibility to assign more than two truth values to a formula gives rise to more possibilities in for example programming, but multiple-valued logics are also studied in automated reasoning [4] and decision making, symbolic model-checking [5], neural networks [6] and the delay testing of modern chips [7, 8]. All of these uses require knowledge and comprehension about the truth values of formulas, within reasonable time.
The research described in this paper features a tableau prover for $RM3$, $Ł_3$, K_3, LP and FDE that is capable of visualizing its tableau proof [9], as well as giving counterexamples when an infer-

*University of Groningen, Institute of Artificial Intelligence

ence is not valid. The main question this paper answers is: How will an automated theorem prover for $RM3$, Ł$_3$, K_3, LP and FDE based on the tableau method perform when tested on speed and correctness?
The design of the prover was inspired by other provers, like OOPS [10], MOLTAP [11] and The Tableau Workbench [12]. The prover will be built, trained and optimized on a set of logical inferences that can be valid or not in each of the aforementioned logics, obtained from several sources. For this training set, see Appendix A. Testing on speed and correctness will be done on a different set of logical inferences.
This paper further gives a technical description of the prover, as well as test results and a discussion about known problems and further work.

2 Model implementation

In this and the next sections, a brief description of the tableau method used in *MULTITAB* will be given. Then, the input language and handling are described. Finally, a technical model description of the implementation in Java is given, as well as testing methods used on *MULTITAB*.

2.1 tableaux for logical inferences

MULTITAB, implemented in Java, uses a specific form of the tableau method to solve formal logical inferences. This method consists of constructing a proof tree using certain tableau rules. A completely extended proof tree is created when all possible rules are applied to all nodes in all branches.
In the described logics (FDE, LP, K_3, RM_3 and Ł$_3$), each formula has a sign attached to the formula. All assumptions are labelled with a plus, '+' (1 in *MULTITAB*), and the conclusion is labelled with a minus, '-' (-1 in *MULTITAB*). A branch in the proof tree is closed when a certain closing condition has been met. These closing conditions, where A is a formal logical formula, are:

1. $A, +$ and $A, -$ exist on the same branch;
2. $\neg A, +$ and $A, +$ exist on the same branch;
3. $\neg A, -$ and $A, -$ exist on the same branch.

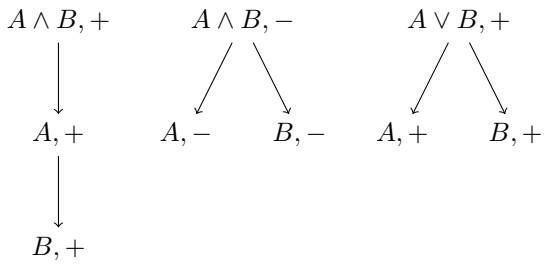
Branches in FDE close only when the first closing condition is met, branches in K_3 and Ł$_3$ close when the first or second closing condition is met, and branches in LP and RM_3 close when the first or third closing condition is met. When all branches in a proof tree are closed according to the condition of a logic, the inference is valid for that logic. A branch is considered open when no more rules can be applied to the formulas on the branch that have not been applied yet and no closing condition is met on any of the formulas on the branch. When a branch is open, the inference is invalid. A counterexample can then be read of the open branch. This counterexample will be based on the collection of all the literals on the open branch with a + sign. Note that no loops can occur in any of the tableaux generated by this method, since in any of the tableau rules a formula will be rewritten either into a formula with the same length, or split into two or three formulas that are all smaller than the original.

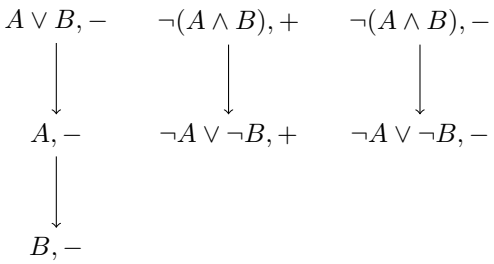
2.2 Soundness and completeness

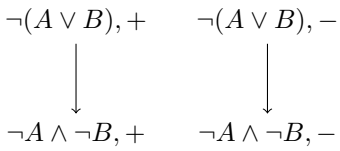
FDE, LP, K_3, Ł$_3$ and RM_3 tableaux generated by the described methods are sound and complete with respect to the relational semantics of each distinct logic, because the tableau rules are sound and complete for each logic [2, chap. 8.7.1-8.7.9; 8.10.4, solutions L. Barson]. *MULTITAB* exactly follows the rules of each of the different tableau methods, and thus the system generates sound and complete tableaux for each different logic.

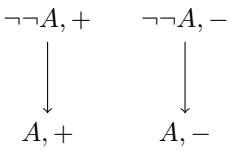
2.3 Tableau rules

The general tableau rules for FDE, LP, K_3, RM_3 and Ł$_3$ are [13]:

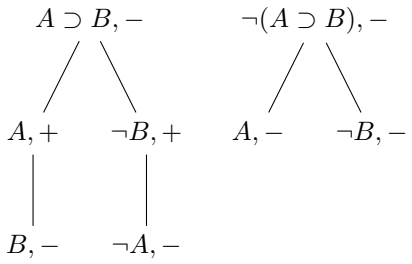


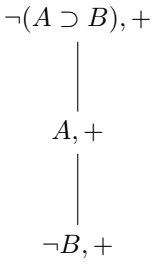




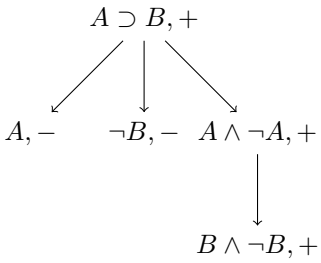


Extra tableau rules for both RM_3 and Ł$_3$ are [13]:

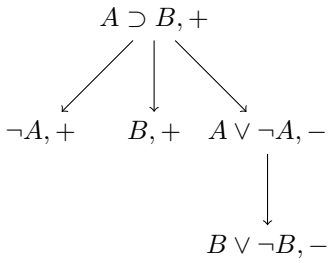




The different extra implication rule for RM_3 is [13]:



The different extra implication rule for Ł$_3$ is [13]:



3 Technical model description

MULTITAB is split in two different Java programs, because this clearly illustrates the differences between two of the methods used: the first method stops when a first counterexample is found, and the second method always searches the complete proof tree for (all) counterexamples. These programs however are similar to each other, and therefore will be treated as one program. Where there are differences, this will be remarked.

3.1 Input

The input of *MULTITAB* is given in four arguments. The first argument should be a String with the assumptions of the inference, separated internally by commas. When there are no assumptions, an empty String suffices. The second input argument should also be a String, this time holding the conclusion of the inference. When there is no conclusion, an empty String suffices. In both strings, logical symbols should be replaced by the symbols as shown in Table 1. *MULTITAB* cannot handle $\leftrightarrow$, and therefore $A \leftrightarrow B$ should be written as $(A \supset B) \wedge (B \supset A)$, and input in *MULTITAB* as

Logic symbol	$\neg$	$\wedge$	$\vee$	$\supset$
Input symbol	$\sim$	&	/	>

Table 1: Input mode for logical symbols

Input	1	2	3	4	5
Logic	FDE	LP	K_3	RM_3	Ł$_3$

Table 2: Integer input for choice of logic

"$(A > B)\&(B > A)$". The use of classical logical notation is mandatory, as *MULTITAB* only recognises this. The third argument should be an integer ranging between 1 and 5, telling *MULTITAB* in which logic the inference should be solved. Table 2 shows which number corresponds to which logic. The fourth argument should be an integer, 0 or 1, where 1 lets *MULTITAB* print its proof tree, and 0 does not. This final argument is used for visualizing the tableau proof or not.

3.2 Data Class: Expression

The data class *Expression* class holds a formula (as a String) and the sign (as an integer of 1 or -1) attached to this formula. Getters and setters allow other classes and methods to use the data in this class. Further in this section, when a formula is mentioned, the String of a specific Expression class representing the logical formula is meant. When further in this section a sign is mentioned, this refers to the integer value of 1 or -1 representing the sign (+ or -) of the formula in the same Expression class.

3.3 Class: Main

The *Main* class first initializes a timer, and then splits the first argument (the assumptions) by the commas. Each of these parts is made into an Expression class with 1 as a sign and added to an ArrayList, called the to-do list. The second argument (the conclusion) is entered into this same to-do list as an Expression class with -1 as a sign. Then this to-do list, together with two empty ArrayLists (the literals list and the tree list), the starting time, the logic indicator (third input-argument) and a boolean indicating visualization (fourth input-argument made into a boolean) is made into the Branch class. The first difference between the two programs is seen here. The program that searches the complete proof tree for counterexamples also includes a zero-value to the Branch class. If all Branch classes are done computing, and this value is still zero when returned, the inference is valid, if not the inference is invalid. The program that stops when a counterexample is found stops the system on the spot (in the Branch class), so this value is not needed.

3.4 Class: Branch

The *Branch* class is the actual computing class of *MULTITAB*. First, the general framework of this class will be described, and afterwards the functions. Starting with a filled to-do list, an empty literals list and an empty tree list, this class will look for a counterexample as fast as possible by applying depth-first search in the proof tree while simultaneously building this proof tree. The proof tree is built by applying rules to the Expressions in the to-do list, adding new Expressions to the to-do list and deleting the Expressions that were processed from this to-do list and placing them on the tree lists and in the case of an Expression being a literal also placing the Expression on the literals list. It does this by looping several times over the to-do list, and also by creating new instances of Branch classes when a tableau rule indicates a split in the proof tree. Whenever new Branch classes are instantiated, the current class stops applying rules to its to-do list.
During every loop, while the to-do list has Expressions in it and no splitting rule has been applied to an Expression, the Expressions in the literals list and the to-do list are first evaluated with the done-function. This function tests all Expressions in these lists for the closing conditions matching the logic in which the class is operating. When no closing condition is met, the to-do list is evaluated by the rules-function. This function chooses the best rule and applies this to the to-do list, this strategy and the rules are explained later. Afterwards, when the to-do list is empty (the branch is open) or a closing condition has been met (the branch is closed), this class prints itself, only when the visualisation boolean is true, and only itself. The tableau proof is thus not printed completely as a whole, but each branch is printed independently. If a branch is open

and has not split, the counterexample is printed selectively of the literals list.
Here another difference between the methods arise. When using the first method (stop on first counterexample), the to-do list is empty and no closing condition has been met (the branch is open), *MULTITAB* prints its counterexample and then stops completely. When using the second method (search for all counterexamples) and the to-do list is empty and no closing condition has been met, the counter indicating validness is increased. This means that when the complete proof-tree has been searched, the counter is not zero anymore, and the inference is considered to be invalid. Counterexamples are given at each open branch.

3.4.1 Function: done

The function *done* evaluates the list it has been given for any closing condition met (according to the current logic). All Expressions in the list are compared with each other, and when a closing condition is met, this function returns *true*. If the input list was the to-do list, only the first closing condition is evaluated. The Expressions that meet this closing condition are removed from the to-do list and added to the tree list and literals list.

3.4.2 Function: rules

The function *rules* operates in two parts. If in any part of this or in following sections "operator" occurs, this refers to the operator that determines the rule to apply on a certain Expression. This operator is in many cases the head connective of a formula, mostly when the head connective is a binary connective. But in the cases of unary connectives, this operator differs from the actual head connective. First, the operator of each Expression in the to-do list is searched for by implementation of headConnective(). If there is no such operator, this means the Expression is a literal, and literalRule() is applied to the Expression and this function stops. Otherwise each Expression is given a strategy number, based on its operator, the current logic and its sign through multiple switches. The lowest strategy number indicates the rule to actually apply, and only this rule is applied to one of the Expressions with this strategy number. Table 3 illustrates how different operators, signs and logics lead to different strategy numbers and rules.
When an operator is encountered that is not in Table 3, the first and last character of the formula are deleted, because the formula was in between parentheses and this function stops. This will be the case when *rules()* encounters the bottom row of Table 4. There is also one special case, indicated with the asterisk (row 3 of Table 3). This is an exception case where the operator is (, the logic is RM_3 or Ł$_3$, the sign is '-' and the inner operator (the operator of the part in the parentheses) is >. In this case the Expression is given a strategy number of 4, since giving it a strategy number of 2, as for all the other cases in which the operator is (, would mean that a splitting rule would be favored over a rule where the Branch is not split. Since not splitting is always favored over splitting, to decrease the size and complexity of a proof-tree, this case is given 4 as strategy number. The strategies are given these numbers to keep the complexity of the proof-tree as low as possible. The strategy numbers 1, 2 and 3 lead to rules that do not split the Branch, whereas the strategy numbers 4, 5, 6 and 7 lead to rules that split the Branch into two new Branches, and the strategy number 8 leads to a rule that splits the Branch into three new Branches, greatly increasing the complexity of the proof-tree.

3.4.3 Function: headConnective

This Function returns the position of the operator of a formula by counting the parentheses in that formula. If there are no parentheses, the first occurrence of a binary connective (>, & or /) is the operator. If there is no binary connective in this case, 1 is returned (indicating the second position of the String). If there are one or two blocks of parentheses (with possible blocks of more parentheses inside it) and a binary connective (>, & or /) outside of that, the first occurrence of such connective is the head connective. If there is one block of parentheses and no binary connective (>, & or /) outside of that, the second position of the string is returned. If all of the above are not applicable, also the second position of the String is returned. This default option is shown in the three bottom rows in Table 4. Table 4 gives some examples of formulas, the return integer of headConnective() and the corresponding operator. The last row is especially interesting, here the return value is 1, which points

Operator	Sign	Logic	Strategy number	Rule to apply
$\sim$	+ or -	all	1	Double Negation rule
(	+ or -	all	2	deMorgan
$(*$	-	RM_3 or Ł$_3$	4	deMorgan
&	+	all	3	noSplitRule
&	-	all	5	splitRule
/	+	all	5	splitRule
/	-	all	3	noSplitRule
$>$	+ or -	FDE, LP or K_3	6	implyNoRule
$>$	+	RM_3 or Ł$_3$	8	implyPlusRule
$>$	-	RM_3 or Ł$_3$	7	implyMinRule

Table 3: Strategy choosing for rules to apply

to a character that is not an operator in Table 3. *rules()* will thus remove the outer parentheses of this inference.

Formula	Return value	Operator
A/B	1	/
$\sim\sim A\&(B > C)$	3	&
$(A\& \sim A) > B$	6	$>$
$(A/B)\&(D/\ C)$	5	&
$\sim\sim (A/B)$	1	$\sim$
$\sim ((\sim A/B)\&C)$	1	(
$(A\& \sim A)$	1	A

Table 4: Examples of formulas, return values and the corresponding operator

3.4.4 Function: literalRule

The *literalRule* function deletes the Expression from the to-do list, and adds it to the tree list and the literals list

3.4.5 Function: negnegRule

The *negnegRule* function removes the Expression from the to-do list, removes the first two characters from the formula (the double negation) and adds this new formula, with the same sign as a new Expression to the to-do list.

3.4.6 Function: noSplitRule

The *noSplitRule* function deletes the Expression from the to-do list, adds it to the tree list, splits the formula in a part before the operator (A) and a part after the operator (B), and adds these new Expressions (with formulas A and B, with the same sign as the input Expression) to the to-do list.

3.4.7 Function: splitRule

The *splitRule* function deletes the Expression from the to-do list, adds it to the tree list, then, splits the formula in a part before the operator (A), and a part after the operator (B). Two new Branch classes are created: both are complete strong copies of the current Branch class, and to one of the to-do lists the new Expression with formula A and the same sign as the Input Expression is added, and to the other the new Expression B and the same sign as the input Expression is added. The current Branch is now split.

3.4.8 Function: deMorgan

First, the *deMorgan* function deletes the Expression from the to-do list and adds that Expression to the tree list. Then it determines the operator of the (sub)formula in between the parentheses of the given Expression, and splits this new formula in two parts: the part before the operator (A), and the part after it (B). Now a switch on the operator determines the next action.
If the operator is &, the new Expression with formula $\sim A/ \sim B$ and the same sign as the input Expression is added to the to-do list. If the operator is /, the new Expression with formula $\sim A\& \sim B$ and the same sign as the input Expression is added to the to-do list. If the operator is $>$, a new switch is encountered, this time evaluating the logic. If the

logic is FDE, LP or K_3, the new Expression with formula $\sim (\sim A \vee B)$ and the same sign as the input Expression is added to the to-do list. If the logic is RM_3 or Ł$_3$, a new switch evaluating the sign is encountered. If this sign is positive (1), two new Expressions (with formulas A and $\sim B$ and positive signs) are added to the to-do list. If the sign is negative, two new Branch classes are created, both are complete strong copies of the current Branch class, and to one of the todo-lists the new Expression with formula A and a negative sign (-1) is added, and to the other the new Expression with formula $\sim B$ and a positive sign (1) is added. The current Branch is now split.

3.4.9 Function: implyNoRule

The *implyNoRule* function deletes the Expression from the to-do list, adds it to the tree list, then splits the formula in a part before the operator (A), and a part after the operator (B). It then adds a new Expression (with formula $\sim A/B$ and the same sign as the input Expression) to the to-do list.

3.4.10 Function: implyPlusRule

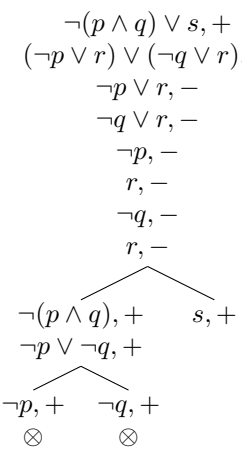
The *implyPlusRule* function deletes the Expression from the to-do list, adds it to the tree list, then splits the formula in a part before the operator (A), and a part after the operator (B). Then, three new Branch classes are created, each being strong complete copies of the current Branch class.
To the first, the new Expression with formula A with a negative sign (-1) is added when the logic is RM_3, and a new Expression with formula $\sim A$ with a positive sign (1) is added when the logic is Ł$_3$.
To the second, the Expression with formula $\sim B$ with a negative sign (-1) is added when the logic is RM_3, and the Expression with formula B with a positive sign is added when the logic is Ł$_3$.
To the third, the new Expressions with formulas $A\& \sim A$ and $B\& \sim A$, both with a positive sign (1) are added when the logic is RM_3. When the logic is Ł$_3$ the new Expressions with formulas $A/\sim A$ and $B/$ with a negative sign (-1) are added to the third Branch class' to-do list.

3.5 Complexity

As explained above, *MULTITAB* takes efficiency into account by trying to keep the complexity of the tableau low. The computational complexity of the general satisfiability and validity problem for the five logics described is however, without visualizing, in PSPACE. This is because the program, at any time during the computation process, has to have two lists in its memory. These two lists are the to-do list and the literals list of the current Branch class that is being evaluated. These lists can have at most two times as many units in it as the number of operators in the original inference. Thus, the validity problems as solved by depth-first search in tableau trees are in PSPACE, similar to those of modal logics like K. If *MULTITAB* is set to visualize its tableaux, the computational complexity could reach EXPTIME. This is because it now has to print all of its branches after evaluating them, and if each branch is split on each formula this could create an exponentially large tableau in comparison to the length of the original inference.

4 Example

To further clarify the implementation of the automatic theorem solver, a short example will be given in this section. The example will be a step-by-step run, with comments, of the model on the inference $\neg(p \wedge q) \vee s \models_{FDE} (\neg p \vee r) \vee (\neg q \vee r)$, searching for all counterexamples and visualizing its proof tree. The tableau for this inference in FDE is shown directly below.



The input arguments for the model will be " $\sim$

$(p\&q)/s$" "$(\sim p/r)/(\sim q/r)$" 1 1.
The Main class will recognize this as the logic FDE with visualization (the last two arguments), and will create two new Expressions(" $\sim (p\&q)/s$", 1) and ("$(\sim p/r)/(\sim q/r)$", -1) and put these in the todo-list. This list, together with an empty tree and an empty literals list, is sent to the Branch class. Each Expression will be noted as ("A", x) where "A" is the formula, and x is the sign.
The Branch class has not split and has Expressions in its to-do list, so it will first check if any Expression in its to-do list or literals list matches the first closing condition. Since this is not the case, it will now assign strategy numbers to each Expression in the to-do list. For this it needs the operator of each Expression. By calling headConnective(), the positions of these operators are returned and the operator can be found, as well as the strategy numbers (holding into account the logic and sign) for all Expressions. What the model now knows is shown in table 5. Since 3 is the lowest strategy number, the noSplitRule() will be executed on ("$(\sim p/r)/(\sim q/r)$", -1), meaning that this Expression will be placed on the tree-list, and the new Expressions (" $\sim p/r$", -1) and (" $\sim q/r$", -1) are put on the to-do list. This was the first loop of the model. Since the Branch has not split and there are Expressions in the to-do list, another iteration is started, by first evaluating the literals and to-do list on closing conditions, and then assigning a strategy number to each Expression in the to-do list. Table 6 now shows the knowledge of the model. Now, the noSplitRule() function will be executed on (" $\sim q/r$", -1), because the strategy numbers of both (" $\sim q/r$", -1) and (" $\sim p/r$", -1) are 3, but the former is chosen because that Expression is later in the list. The noSplitRule() will remove the Expression (" $\sim q/r$", -1) from the to-do list, add it to the tree-list and add the new Expressions (" $\sim q$", -1) and ("r", -1) to the to-do list.
On the next two iterations, the two latest added Expressions are recognized as literals and are removed from the to-do list, after which they are added to both the literals and the tree lists. The rule-selection procedure will not happen in these cases.
On the iteration after this, the to-do list looks like the one right above, but without the rightmost column. So the same procedure as before will happen, the noSplitRule() will remove the Expression (" $\sim p/r$", -1) from the to-do list, add it to the tree-list and add the new Expressions (" $\sim p$", -1) and ("r", -1) to the to-do list.
On the next two iterations, the two latest added Expressions are recognized as literals, and are removed from the to-do list, and added to both the literals and the tree lists. The rule-selection procedure will not happen in these cases. Now only the Expression with strategy number 5 still remains in the to-do list, and thus the splitRule() function will be executed on that Expression.
The splitRule() function removes the Expression (" $\sim (p\&q)/s$", 1) from the to-do list, adds it to the tree list and creates two new Branch classes that are both complete strong copies of the original one. To the first of these these new Branch classes the new Expression (" $\sim (p\&q)$", 1) is added, while to the second one, the new Expression ("s", 1) is added. The original class is now split and will not do anything anymore. The model continues with the first new Branch class. This new Branch class has not split and has one Expression in its to-do list. After evaluation of the closing conditions (none met), the model searches again for the lowest strategy number of all the Expressions in the to-do list (even if there is only one, like in the current case) using headConnective and rules(). The rules() function evaluates as follows:

Expression	(" $\sim (p\&q)$", 1)
operator position	1
operator	(
strategy number	2

The deMorgan() function applies to (" $\sim (p\&q)$", 1), deleting it from the to-do list, adding it to the tree list, and adding the new Expression (" $\sim p/\sim q$", 1) to the to-do list. This new to-do list is in the next iteration:

Expression	(" $\sim p/\sim q$", 1)
operator position	2
operator	/
strategy number	5

So, again the splitRule is executed, leading to two new Branches, of which the first is evaluated first. This new Branch only has the Expression (" $\sim p$", 1) in its to-do list , which is recognized as a literal and handled as such (removed from to-do list and

Expression	(“ $\sim (p\&q)/s$”, 1)	(“$(\sim p/r)/(\sim q/r)$”, -1)
operator position	6	6
operator	/	/
strategy number	5	3

Table 5: Knowledge of model at certain step

Expression	(“ $\sim (p\&q)/s$”, 1)	(“ $\sim p/r$”, -1)	(“ $\sim q/r$”, -1)
operator position	6	3	3
operator	/	/	/
strategy number	5	3	3

Table 6: Knowledge of model at a later step

added to literals and tree lists). Now the to-do list of this Branch is empty, but the literals list is still evaluated once more.The literals list consists of the Expressions (“ $\sim p$”, -1), (“r”, -1), (“ $\sim q$”, -1), (“r”, -1) and (“ $\sim p$”, 1). Duplicates can exist in the literals list and are ignored. Now, for the first time a closing condition is met, and so the Tree list is printed, with a neat X beneath it.
The previous splitting of a Branch led to two new Branches, of which only the first one has been evaluated. This second Branch had only the Expression (“ $\sim q$”, 1) in its to-do list , which is recognized as a literal and handled as such. Now, a closing condition is also met on the literals in this Branch, and the tree list is printed with a neat X beneath it.
Now, the second Branch of the first split is evaluated. This also only had one Expression (“s”, 1) in its to-do list. This is again recognized as a literal and handled as such. Now the to-do list of this branch is also empty, and the literals list, consisting of (“ $\sim p$”, -1), (“r”, -1), (“ $\sim q$”, -1), (“r”, -1) and (“s”, 1), is once more evaluated. Since there is no closing condition met this time, the Branch is open. Therefore the master integer called ‘eval’ is incremented with one. In the previous cases, where a closing condition was met, this was not the case. Now each member of the literals list that has a positive sign (1) is printed, showing the counterexample. In this case only $s\rho 1$ is printed.
Now, all Branches are done, and the Main class evaluates the master integer ‘eval’. This is not zero, thus the inference is invalid. This will be given by the model, as well as the time it took to give this solution in milliseconds.

5 Testing the model

Four different settings of *MULTITAB* were tested. The difference between these settings were stopping when a counterexample was encountered or not, and in visualizing the proof tree or not. The four different settings are named A1, A2, B1 and B2. Table 7 shows which name corresponds to which setting. *MULTITAB* was built, trained and optimized on a set of 22 formal logical inferences, called the training set (found in appendix A). After this, its performance was evaluated on these same inferences. The performance measure is split into two parts, correctness and speed. If *MULTITAB* gave the correct solution to an inference, it was said to be 100% correct on that inference. *MULTITAB* gives a correct solution if the inference is valid in Table 10 (in Appendix A) and the model’s solution is that the inference is valid, or if the inference is invalid in Table 10 and the model’s solution is that the inference is invalid, and it gives at least one correct counterexample. This means that the counterexample given by *MULTITAB* must really be a counterexample. If *MULTITAB* gave an incorrect solution to an inference, it was said to be 0% correct on that inference. Note that giving only one counterexample is sufficient, and therefore no difference in correctness will be recorded for a solution with one or with multiple counterexamples.
The time in which each inference was solved was measured in milliseconds by *MULTITAB* itself, using the internal clock of the system it runs on. This leads to 10 different populations (two for each logic, correctness and speed), four times. So for speed testing, there are 20 groups, and for correctness

Name	Method	Visualization
A1	Stop on first counterexample	No
A2	Stop on first counterexample	Yes
B1	Search complete proof tree	No
B2	Search complete proof tree	Yes

Table 7: Different testing populations

testing, there are 20 groups.
For speed testing these 20 groups can be combined into 4 groups for the different settings, or into 5 groups for the different logics. To truly test the performance and the reliability of *MULTITAB*, the same experiment was executed using a new, other set of 16 different formal logical inferences of which none existed in the original set. This set will be called the test set. These inferences can be found in Appendix B.
Where in any of these inferences $\leftrightarrow$ occurs, $A \leftrightarrow B$ was rewritten as $(A \supset B) \wedge (B \supset A)$. If errors now arise in the solutions of *MULTITAB*, these errors will be structural errors, since they occur because of errors in the model's code.
The results of the second experiment will be displayed and discussed. Different populations will be compared. For the results of correctness, all populations will be compared. For the results of speed, only some populations will be compared using the Wilcoxon signed rank test, since the populations are paired and non-normal distributed. The combined populations of each setting (A1, B1, A2 and B2) will be compared, leading to 6 different tests. A second comparison will be done on the combination of groups for each logic, leading to 10 different tests, since there are 5 groups to compare.

5.1 Representavity and variety

In both the training set and the test the variety between the inferences was high. This means that inferences were tested that were valid in some logic, while invalid in other logics, creating differences between the tested logics. An example of this variety is that in the test set, a lot of inferences are found that are invalid in FDE. This is because an inference that is valid in FDE is automatically valid in LP and K_3, while inferences that are invalid in FDE may be either invalid or valid in LP or K_3. This difference increases the variety in the test set. Furthermore some inferences can be found in both the training and the test set where the inference is invalid in only one logic and valid in the others, as well as inferences that are valid in only one logic and invalid in the others. Using these inferences in speed testing could show a difference of speed of the system in evaluating valid or invalid inferences. Finally having inferences that are valid in LP but invalid in RM_3, as well as inferences that are valid in K_3 but invalid in Ł$_3$, increases the variety, because this shows the difference between logics that have the same closing conditions, but different tableau rules. This is also true for inferences that are invalid in LP but valid in RM_3, as well as for inferences that are invalid in K_3 but valid in Ł$_3$. For both the training set and the test set the abovementioned broad variety of inferences, with respect to the validity of the five logics, was selected so that the results, especially about speed, would give a realistic picture of the landscape of inferences.

6 Results

The automatic theorem prover was **100% correct** on all inferences, on all four different settings, in all five logics. Figure 1 shows the means and standard error of all different testing populations in the speed tests. These populations are grouped by logic. Figure 1 is meant purely for visualization of the results. Figure 2 shows the means and standard error of each different settings, by combining the data of all the logics per setting. Table 8 shows the different test results for these comparisons. Figure 3 shows the means and standard error of each different logic, by combining the data of all the settings per logic. Table 9 shows the different test results for these comparisons.

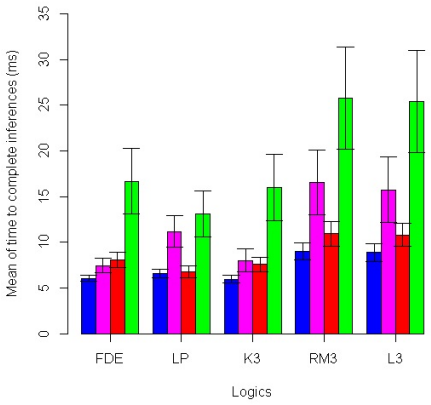


Figure 1: Means and SE of all different populations, Each bar represents a different setting in the order A1 (blue), A2 (magenta), B1 (red), B2 (green)

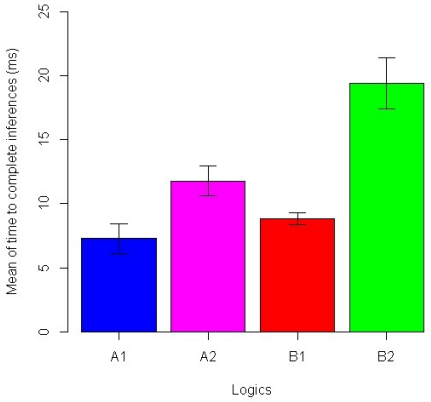


Figure 2: Means and SE of different settings, Each bar is the combination of all the logics (FDE, LP, K_3, RM_3 and $Ł_3$) per setting

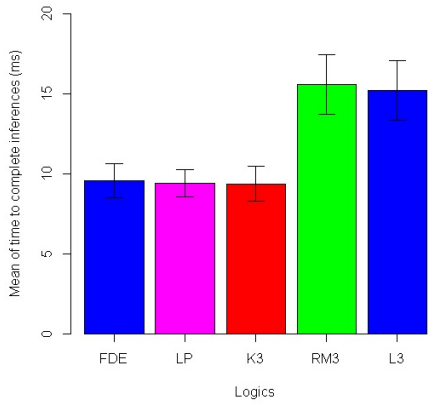


Figure 3: Means and SE of different logics. Each bar is the combination of all the setting groups (A1, A2, B1, B2, see Table 7) per logic

7 Discussion

The results show that *MULTITAB*, in any setting, is always correct. If anyone would want to solve a formal logic inference in FDE, LP, K_3, $RM3$ or $Ł_3$, they should use *MULTITAB*. Even more so because *MULTITAB* is fast. The longest time it took to solve an inference was 68 ms, which is less than a tenth of a second. As for the settings, the results show us that *MULTITAB* is significantly faster when not visualising (as opposed to visualizing), for both methods used. Searching for all counterexamples is also significantly slower than searching for only one counterexample, in both cases of visualizing or not. A thought on these results is that visualizing takes longer just because the process of printing on screen takes time. The result that searching for all counterexamples is significantly slower than stopping the search on the first counterexample is not really surprising. This significant result can probably also be found when applying this method by hand. The idea there is that stopping on the first counterexample, and therefore also stopping drawing the tree, is faster than drawing the complete proof tree, with all the counterexamples.

If users are interested in a complete proof tree and all counterexamples in one of the logics discussed, slowest setting (search for all counterexamples and

	A2	B1	B2
A1	V= 0 p= $5.02 * 10^{-14}$	V= 76 p= $1.279 * 10^{-6}$	V= 0 p= $1.058 * 10^{-14}$
A2		V= 19335 p= 0.001261	V= 85 p= $1.286 * 10^{-9}$
B1			V = 0 p= $1.045 * 10^{-14}$

Table 8: Test results of Wilcoxon signed rank test on different setting groups

	LP	K_3	RM_3	Ł$_3$
FDE	V= 452 p= 0.3801	V= 187 p= 0.4961	V= 18 p= $9.377 * 10^{-10}$	V= 54.5 p= $2.911 * 10^{-9}$
LP		V= 469 p= 0.8439	V= 64.5 p= $4.807 * 10^{-8}$	V= 143.5 p= $2.403 * 10^{-7}$
K_3			V= 12 p= $6.637 * 10^{-10}$	V= 13.5 p= $1.593 * 10^{-9}$
RM_3				V= 377 p= 0.3034

Table 9: Test results of Wilcoxon signed rank test on different logic groups

visualize, B2) should be used. If the user is however only interested in the validity of an inference, the much faster setting of stopping on the first counterexample and not visualizing is preferred.

The results for the different logics are not really surprising. The speed of *MULTITAB* is not significantly different when the logics FDE, LP and K_3 are compared with one other, nor when the logics RM_3 and Ł$_3$ are compared with each other. A thought on this is that these logics use approximately the same rules internally, and only differ on closing conditions. There are however significant differences in speed between RM_3 on the one hand and FDE, LP and K_3 on the other hand as well as significant differences in speed between Ł$_3$ on the one hand and FDE, LP and K_3 on the other hand. This means that *MULTITAB* is significantly slower in solving inferences in RM_3 and Ł$_3$ than in the other three logics. A thought on this is that RM_3 and Ł$_3$ have more, and more difficult *implication* rules that greatly increase the complexity and duration of the solution.

A comparison with another tableau solver can be made at this point. One other free tableau solver was found in the pytableaux solver of Douglas Owings [14]. This system gives the user the tableau proof and validity of an inference for multiple selectable logics, including FDE, LP and K_3. Owings' system does sadly not support RM_3 and Ł$_3$. Were a user to solve inferences in these logics, the only (free) model it could use is *MULTITAB*. Another difference between *MULTITAB* and Owings' system is that Owings' system uses Polish notation, and *MULTITAB* classical notation. It is completely up to the reader to decide which model suits him or her best in this case, but most of the literature concerning tableau provers, or logical inferences in general, use the classical notation of logical inferences. Therefore, to use Owings' system a user would first have to rewrite the inference from classical to polish notation, putting work in identifying the head connectives over and over again. *MULTITAB* does this for the user, since the classical notation is mandatory as input. However, when an inference in polish notation is encountered, the user himself should rewrite the inference to classical notation to use *MULTITAB*. The speed of Owings' system was not tested or compared with the speed of *MULTITAB*.

To finalize, *MULTITAB* is always correct in five different logics and four different settings, with varying speed. The solutions are generated in less than a tenth of a second, faster than any logician can do by hand. Still, the speed of the fastest setting could

be enhanced even further. An example of this is to let *MULTITAB* check the length of Expressions before the splitting of Branches and to first evaluate the Branch with the shorter Expressions in its to-do list. Future work could further include the creation of a graphical or web user interface, to increase the usability of the solver. But for now, this automatic theorem solver is a fast and correct way for any student, teacher, academic or other user to solve formal logical inferences.

References

[1] J. Barwise, J. Etchemendy, G. Allwein, D. Barker-Plummer, A. Liu, Language, Proof and Logic, CSLI Publications, 2000.

[2] G. Priest, An Introduction to Non-Classical Logic: From If to Is, Cambridge Introductions to Philosophy, Cambridge University Press, 2008.
URL `https://books.google.nl/books?id=rMXVbmAw3YwC`

[3] L. Shaw, A feature comparison of free proof tree software (2014).
URL `http://creativeandcritical.net/prooftools/comparison-of-proof-tree-software`

[4] J. Lu, N. Murray, E. Rosenthal, A framework for automated reasoning in multiple-valued logics, Journal of Automated Reasoning 21 (1) (1998) 39–67. doi:10.1023/A:1005784309139.

[5] M. Chechik, B. Devereux, S. Easterbrook, A. Gurfinkel, Multi-valued symbolic model-checking, ACM Transactions on Software Engineering and Methodology 12 (4) (2003) 371–408. doi:10.1145/990010.990011.

[6] G. Wang, H. Shi, TMLNN: Triple-valued or multiple-valued logic neural network, IEEE Transactions on Neural Networks 9 (6) (1998) 1099–1117.

[7] S. Eggersgluess, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, R. Drechsler, MONSOON: SAT-Based ATPG for path delay faults using multiple-valued logics, Journal of Electronic Testing-Theory and Applications 26 (3) (2010) 307–322. doi:10.1007/s10836-010-5146-y.

[8] N. Takagi, A delay model of multiple-valued logic circuits consisting of min, max, and literal operations, IEICE Transactions on Information and Systems 93 (8) (2010) 2040–2047. doi:10.1587/transinf.E93.D.2040.

[9] R. Hahnle, Towards an efficient tableau proof procedure for multiple-valued logics, in: Computer Science Logic, Vol. 533, Springer, 1991, pp. 248–260.

[10] G. van Valkenhoef, E. van der Vaart, L. Verbrugge, OOPS: An $S5_n$ prover for educational settings, Electronic Notes in Theoretical Computer Science 162 (2010) 249–261. doi:10.1016/j.entcs.2010.04.018.

[11] T. van Laarhoven, MOLTAP — A modal logic tableau prover.
URL `http://twan.home.fmf.nl/moltap/index.html`

[12] P. Abate, R. Goré, The tableau workbench, Electronic Notes in Theoretical Computer Science 231 (2009) 55 – 67. doi:http://dx.doi.org/10.1016/j.entcs.2009.02.029.
URL `http://www.sciencedirect.com/science/article/pii/S1571066109000346`

[13] L. C. Verbrugge, B. Kooi, Lecture notes advanced logic, Rijksuniversiteit Groningen, 2015.

[14] D. Owings, Pytableaux Web UI.
URL `http://logic.dougowings.net`

[15] F. J. Pelletier, Seventy-five problems for testing automatic theorem provers, Journal of Automated Reasoning 2 (2) (1986) 191–216.

Appendix A: Training set

Table 10: Logical inferences in different logics, training set

#	*Inference*	FDE	LP	RM_3	K_3	Ł$_3$
1	$p \wedge q \models p$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
2	$p \models p \vee q$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
3	$p \wedge (q \vee r) \models (p \wedge q) \vee (p \wedge r)$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
4	$p \vee (q \wedge r) \models (p \vee q) \wedge (p \vee r)$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
5	$p \models \neg\neg p$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
6	$\neg\neg p \models p$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
7	$(p \wedge q) \supset r \models (p \wedge \neg r) \supset \neg q$	✓[1,5]	✓[3,5]	x[8]	✓[3,5]	x[8]
8	$p \wedge \neg p \models p \vee \neg p$	✓[1]	✓[3]	✓[6]	✓[3]	✓[7]
9	$p \wedge \neg p \models q \vee \neg q$	x[1]	✓[2]	✓[6]	✓[2]	✓[7]
10	$p \vee q \models p \wedge q$	x[1]	x[2]	x[6]	x[2]	x[7]
11	$p, \neg(p \wedge \neg q) \models q$	x[1]	x[2]	x[6]	✓[2]	✓[7]
12	$(p \wedge q) \supset r \models p \supset (\neg q \vee r)$	✓[1,5]	✓[3,5]	x[8]	✓[3,5]	x[8]
13	$q \models p \supset q$	✓[5,8]	✓[4,5]	x[4]	✓[4,5]	✓[4]
14	$\neg p \models p \supset q$	✓[5,8]	✓[4,5]	x[4]	✓[4,5]	✓[4]
15	$(p \wedge q) \supset r \models (p \supset r) \vee (q \supset r)$	✓[5,8]	✓[4,5]	✓[4]	✓[4,5]	✓[4]
16	$(p \supset q) \wedge (r \supset s) \models (p \supset s) \vee (r \supset q)$	✓[5,8]	✓[4,5]	✓[4]	✓[4,5]	✓[4]
17	$\neg(p \supset q) \models p$	✓[5,8]	✓[4,5]	✓[4]	✓[4,5]	✓[4]
18	$p \supset r \models (p \wedge q) \supset r$	✓[5,8]	✓[4,5]	✓[4]	✓[4,5]	✓[4]
19	$p \supset q, q \supset r \models p \supset r$	x[5,8]	x[4,5]	✓[4]	✓[4,5]	✓[4]
20	$p \supset q \models \neg q \supset \neg p$	✓[5,8]	✓[4,5]	✓[4]	✓[4,5]	✓[4]
21	$\models p \supset (q \vee \neg q)$	x[5,8]	✓[4,5]	x[4]	x[4,5]	x[4]
22	$\models (p \wedge \neg p) \supset q$	x[5,8]	✓[4,5]	x[4]	x[4,5]	x[4]

1. Obtained from [2, chap. 8], page 161, problem 1, solutions L. Barson
2. Obtained from [2, chap. 8], page 161, problem 2, solutions L. Barson
3. When this inference is valid in FDE, it automatically is valid in LP and K_3, because FDE is a sublogic of these logics
4. Obtained from table in [2, chap. 7], page 126
5. FDE, LP and K_3 officially do not have $\supset$ in the language, therefor whenever $A \supset B$ was encountered, it was replaced with $\neg A \vee B$. According to [2, chap. 8.2.1], page 142
6. When $\supset$ does not occur in the inference, this inference in RM_3 is exactly the same as the inference in LP
7. When $\supset$ does not occur in the inference, this inference in K_3 is exactly the same as the inference in Ł$_3$
8. Self made tableau, validated on November 13th, 2015 by L.C. Verbrugge

Appendix B: Test set

Table 11: Logical inferences in different logics, test set

#	*Inference*	FDE	LP	K_3	RM_3	Ł$_3$
1	$\models (p \supset q) \leftrightarrow (\neg q \supset \neg p)^{9}$	x^{15}	✓^{15}	x^{15}	✓^{16}	✓^{16}
2	$\models (\neg p \supset q) \leftrightarrow (\neg q \supset p)^{9}$	x^{15}	✓^{15}	x^{15}	✓^{16}	✓^{16}
3	$\models ((p \vee q) \wedge ((\neg p \vee q) \wedge (p \vee \neg q))) \supset \neg(\neg p \vee \neg q)^{9}$	x^{15}	✓^{15}	x^{15}	x^{16}	x^{16}
4	$q \supset r, r \supset (p \wedge q), p \supset (q \vee r) \models p \leftrightarrow q^{9}$	x^{15}	x^{15}	✓^{15}	✓^{16}	✓^{16}
5	$\models (p \supset q) \leftrightarrow (\neg q \vee q)^{9}$	x^{15}	x^{15}	x^{15}	x^{16}	x^{16}
6	$\models (p \leftrightarrow q) \leftrightarrow ((q \vee \neg p)(\neg q \vee p)^{9}$	x^{15}	✓^{15}	x^{15}	x^{16}	x^{16}
7	$\models (p \vee (q \wedge r)) \leftrightarrow ((p \vee q) \wedge (p \vee r))^{9}$	x^{15}	✓^{15}	x^{15}	✓^{16}	✓^{16}
8	$\models ((p \wedge (q \supset r)) \supset s) \leftrightarrow (((\neg p \vee q) \vee s) \wedge ((\neg p \vee \neg r) \vee s))^{9}$	x^{15}	✓^{15}	x^{15}	x^{16}	x^{16}
9	$(p \wedge q) \wedge \neg p \models (p \vee q) \supset \neg p^{10}$	✓^{15}	✓^{15}	✓^{15}	x^{16}	✓^{16}
10	$\neg((p \vee q) \wedge (p \vee r)) \models \neg(p \wedge (q \vee r))^{10}$	✓^{15}	✓^{15}	✓^{15}	✓^{16}	✓^{16}
11	$p \supset (q \wedge r) \models ((r \supset \neg q) \wedge (\neg r \supset p)) \wedge p^{11}$	x^{15}	x^{15}	x^{15}	x^{16}	x^{16}
12	$p \wedge \neg p, \neg(q \vee \neg q) \models \neg(p \vee q) \wedge \neg(\neg p \vee q)^{12}$	✓^{15}	✓^{15}	✓^{15}	✓^{16}	✓^{16}
13	$\neg(p \wedge q) \vee (\neg p \wedge \neg q) \models \neg p^{13}$	x^{15}	x^{15}	x^{15}	x^{16}	x^{16}
14	$\models (p \wedge \neg(\neg p \wedge (q \vee r))) \supset (\neg(p \wedge q) \vee \neg(p \wedge r))^{11}$	x^{15}	x^{15}	x^{15}	x^{16}	x^{16}
15	$\neg(\neg p \vee q) \supset ((p \wedge \neg p) \wedge (q \wedge \neg q)) \models p \supset q^{14}$	✓^{15}	✓^{15}	✓^{15}	✓^{16}	✓^{16}
16	$\neg(p \wedge (q \vee r)), \neg(p \vee p) \wedge \neg r \models \neg r \wedge (\neg q \vee p)^{14}$	✓^{15}	✓^{15}	✓^{15}	✓^{16}	✓^{16}

9. Obtained from "Seventy-Five Problems for Testing Automatic Theorem Provers"[15]
10. Obtained from Exam Advanced Logic (June 7th, 2014), Rijksuniversiteit Groningen
11. Obtained from Resit Exam Advanced Logic (June, 2014), Rijksuniversiteit Groningen
12. Obtained from Exam Advanced Logic (April 11th, 2012), Rijksuniversiteit Groningen
13. Obtained from Exam Advanced Logic (June 16th, 2015), Rijksuniversiteit Groningen
14. Obtained from Resit Advanced Logic (July 8th, 2014), Rijksuniversiteit Groningen
15. Obtained using Doug Owings' Pytableaux Web UI [14]
16. Self made tableau, validated on December 18th, 2015 by L.C. Verbrugge