OOPS: An $S5_n$ Prover for Educational Settings

Gert van Valkenhoef[1,2]

Department of Business and ICT, University of Groningen,
P.O. Box 800, 9700 AV, Groningen, The Netherlands

Elske van der Vaart[3]

Department of Artificial Intelligence, University of Groningen,
P.O. Box 407, 9700 AK, Groningen, The Netherlands

Theoretical Biology Group, University of Groningen,
P.O. Box 14, 9750 AA, Haren, The Netherlands

Rineke Verbrugge[4]

Department of Artificial Intelligence, University of Groningen,
P.O. Box 407, 9700 AK, Groningen, The Netherlands

Abstract

We present OOPS, an open source, cross-platform, easy-to-run tableau prover for $S5_n$. OOPS is aimed at education in modal logics. Thus, it has several features that enable insight into its internal workings. Specifically, OOPS allows tableaux to be visualized and can generate counter models for formulas that are not provable. Moreover, the OOPS Graphical User Interface (GUI) increases ease of use and an integrated general purpose scripting language (Lua) is used to provide convenient and powerful interactions with the OOPS tableau generator.

Keywords: System Description, Modal Logic, Epistemic Logic, $S5_n$, Tableau Methods, Education, Visualization

1 Introduction

In this paper, we describe OOPS[5], an *O*bject *O*riented *P*rover for $S5_n$. It is a tableau-based theorem prover, aimed at satisfiability checking, that is specifically designed for use in a classroom setting. Although not as efficient as highly optimized provers like MSPASS [12] and FaCT [11], or as flexible as highly extensible provers

[1] Corresponding author.
[2] Email: g.h.m.van.valkenhoef@rug.nl
[3] Email: elskevdv@ai.rug.nl
[4] Email: rineke@ai.rug.nl
[5] http://wiki.github.com/gertvv/oops

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs

like LoTReC [7] and the Tableau Workbench [1], we believe OOPS offers a unique combination of features that make it particularly suitable for educational purposes.

First, OOPS offers support for $S5_n$, the logic typically used to model human reasoning processes. As far as we are aware, there are currently no other theorem provers which do this. Furthermore, OOPS is capable of visualizing its tableau proofs, and of generating counter-models to formulas that are false. These two properties are very useful for educational proof tools, and they are shared by at least two other systems: LoTReC can display its tableau trees, and the Logics Workbench [10] can print its derivations in sequent calculi.

Another feature OOPS has in common with LoTReC, as well as MSPASS, but with few other theorem provers, is its free and convenient distribution. OOPS is packaged as a ZIP file that includes all dependencies, and once extracted, can be run simply by double-clicking the resulting oops.jar file. This is true for any operating system, provided the Java VM is available. In contrast, the Tableau Workbench must be built from source, FaCT offers binaries only for Linux and Windows, and requires Lisp otherwise, and the Logics Workbench is not open source, and can be difficult to install, due to outdated dependencies.

A fifth property that makes OOPS particularly attractive for educational use is its Graphical User Interface (GUI). Although simple, its GUI allows students to input formulas in an easy and intuitive format, or to save and load files. LoTReC, MSPASS and the the Logics Workbench also offer GUI-based access.

OOPS' final educational attribute is its integrated scripting facility. In our experience, most student projects involving theorem provers are efforts to model riddles, or game situations. This requires that students be able to build and extend theories, using loops and conditionals where necessary. Many proof tools, like the Logics Workbench, offer custom scripting languages for this purpose. OOPS, by contrast, integrates the existing Lua [6] scripting language. Lua can be used to call most of OOPS' functions, and offers a rich input language.

To summarize, our contribution is a proof system, OOPS, designed to support education in logics, specifically in multi-agent reasoning. For this purpose, OOPS features a tableau prover for $S5_n$ that is capable of visualizing its tableau proofs and counter-models for formulas that are false. Furthermore, OOPS is platform independent and easy to install. Finally, OOPS offers a Graphical User Interface (GUI) and an integrated scripting language (Lua) that enables easy but powerful interactions with the prover.

In the rest of this paper, we first offer a technical description of OOPS, including its proof system, its formal properties, its implementation in Java, and the details of its input language (Section 2). This is followed by a more detailed description of OOPS' educational properties (Section 3), as well as a worked-out example (Section 4). We conclude with a discussion of current limitations and future work (Section 5).

[6] http://www.lua.org/

2 Technical description

In this section, we will first give a brief description of the tableau method used in OOPS and summarize the formal properties (soundness, completeness, complexity) of the system. Then, we explain how the tableau system is implemented in Java. Finally, we describe the input language for modal formulas that is provided by OOPS.

2.1 OOPS tableaux

The OOPS tableau system for $S5_n$ is a Java [8] implementation of the proof system **ELtap** [4,3]. **ELtap**, in turn, draws on [5] and [2]. [9] provides a good review of tableau methods for modal logics. Here, we summarize how tableaux are formalized in OOPS. For a complete description, see [15].

When we construct a tableau, we do so with the aim of creating a Kripke model in which a formula φ is satisfied. This is done by assuming φ is true and then systematically working out the implications of this assumption. Specifically, OOPS generates a collection of branches that each represent alternative ways of working out these implications. A *branch* B consists of a number of nodes. A *node* γ is a combination of a *formula* ψ and a *label* σ. A label is a systematically chosen name for a world in the Kripke model that is being constructed. Branches are created and expanded by the application of *rules* to existing nodes on a branch. A *rule* R consists of:

- A precondition $\mathrm{pre}(R)$, written above a horizontal bar, which is a node containing variables;
- A postcondition $\mathrm{post}(R)$, written below the horizontal bar, which is a list of nodes containing variables. There are two types of postconditions:
  - Linear: add nodes to the current branch, written top-to-bottom;
  - Branching: create a number of new branches, written left-to-right, separated by a vertical bar.
- Zero or more constraints, which restrict the values variables may take.

The rules employed by OOPS are given in Table 1. A rule R is applicable to a branch B, if there is a node $\gamma \in B$ that matches $\mathrm{pre}(R)$, such that no constraints are violated and R has not previously been applied to γ.

In the case of modal rules, after the precondition $pre(R)$ has matched, we may either need to create a new world (add a label part) or to match existing worlds (labels). In the first case, we create a label part that is uniquely identified, through a function $\ulcorner\cdot\urcorner$ that encodes formulas as valid label parts. In the latter case, the postcondition will contain a variable as a placeholder for one of the label parts. Such a postcondition must be applied to all labels that its label matches to.

A branch B is *closed* if there is a label σ and a formula ψ, such that both $(\sigma, \psi) \in B$ and $(\sigma, \neg\psi) \in B$. A branch B is *open* if it is not closed and no more rules can be applied to it. Thus, an open branch corresponds to a successful attempt to satisfy φ, whereas a closed branch corresponds to a failed attempt to satisfy φ. Specifically, for an open branch B, the labels determine the set of worlds and the accessibility relations in the corresponding Kripke model. For each label, the set of formulas given for that label determines the valuation in the corresponding world

Double Negation Rule	$\neg\neg$ $\sigma \quad \neg\neg\varphi$ $\sigma \quad \varphi$	
Conjunctive Rules	$\wedge_\wedge$ $\sigma \quad \varphi \wedge \psi$ $\sigma \quad \varphi$ $\sigma \quad \psi$	$\wedge_\vee$ $\sigma \quad \neg(\varphi \vee \psi)$ $\sigma \quad \neg\varphi$ $\sigma \quad \neg\psi$
Disjunctive Rules	$\vee_\wedge$ $\sigma \quad \neg(\varphi \wedge \psi)$ $\sigma \quad \neg\varphi \mid \sigma \quad \neg\psi$	$\vee_\vee$ $\sigma \quad \varphi \vee \psi$ $\sigma \quad \varphi \mid \sigma \quad \psi$
Possibility Rules (where $i \neq j$)	$\boldsymbol{M}_\Box$ $\sigma.k_i \quad \neg\Box_j\varphi$ $\sigma.k_i.\ulcorner\neg\varphi\urcorner_j \quad \neg\varphi$	$\boldsymbol{M}_\Diamond$ $\sigma.k_i \quad \Diamond_j\varphi$ $\sigma.k_i.\ulcorner\varphi\urcorner_j \quad \varphi$
Possibility Rules∗	$\boldsymbol{M}_\Box\boldsymbol{*}$ $\sigma.k_i \quad \neg\Box_i\varphi$ $\sigma.\ulcorner\neg\varphi\urcorner_i \quad \neg\varphi$	$\boldsymbol{M}_\Diamond\boldsymbol{*}$ $\sigma.k_i \quad \Diamond_i\varphi$ $\sigma.\ulcorner\varphi\urcorner_i \quad \varphi$
Basic Necessity Rules (where $i \neq j$)	$\boldsymbol{K}_\Box$ $\sigma.k_i \quad \Box_j\varphi$ $\sigma.k_i.h_j \quad \varphi$	$\boldsymbol{K}_\Diamond$ $\sigma.k_i \quad \neg\Diamond_j\varphi$ $\sigma.k_i.h_j \quad \neg\varphi$
Basic Necessity Rules∗	$\boldsymbol{K}_\Box\boldsymbol{*}$ $\sigma.k_i \quad \Box_i\varphi$ $\sigma.h_i \quad \varphi$	$\boldsymbol{K}_\Diamond\boldsymbol{*}$ $\sigma.k_i \quad \neg\Diamond_i\varphi$ $\sigma.h_i \quad \neg\varphi$
Special Necessity Rules (where $i \neq j$)	$\boldsymbol{T}_\Box$ $\sigma.k_i \quad \Box_j\varphi$ $\sigma.k_i \quad \varphi$	$\boldsymbol{T}_\Diamond$ $\sigma.k_i \quad \neg\Diamond_j\varphi$ $\sigma.k_i \quad \neg\varphi$
Special Necessity Rules∗	$\boldsymbol{R}_\Box\boldsymbol{*}$ $\sigma.k_i \quad \Box_i\varphi$ $\sigma \quad \varphi$	$\boldsymbol{R}_\Diamond\boldsymbol{*}$ $\sigma.k_i \quad \neg\Diamond_i\varphi$ $\sigma \quad \neg\varphi$

Table 1
Tableau Extension Rules (see Section 2.2 for an explanation of how OOPS applies these rules.)

in the Kripke model. A tableau for φ is closed if all branches are closed, otherwise it is open.

Now, for any proof system, it is important that its proofs correspond exactly to the semantics of the logic. The proof system used by OOPS has been shown to be both sound and complete for $S5_n$ [15]. Furthermore, in the same work, the implementation (Section 2.2) was shown to correspond to the formal description of the proof method. Unfortunately, this work also shows that the algorithm used by OOPS needs exponential time in the worst case, whereas satisfiability for $S5_n$ is known

Connective	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$	$\Box_i$	$\Diamond_i$
OOPS Symbol	~	&	|	>	=	#_i	%_i
Precedence	1	2	3	4	4	1	1

Table 2
OOPS Connectives.

to be PSPACE-complete [9]. However, we believe that for educational purposes the functionality offered by OOPS (see Section 3) easily makes up for this shortcoming. Moreover, the implementation of these features does not depend on the specific proof algorithm used. Thus, as future work, the current algorithm may be replaced by one that is in PSPACE.

2.2 Implementation

In order to ensure the exhaustive, but non-redundant application of Table 1's rules, OOPS employs two data structures: the *match queue* and the *necessities list*. Whenever a node is added to the current branch, we attempt to match every possible rule to that node. The resulting matches are placed on the match queue. Now, the Basic Necessity rules (Table 1) pose a specific problem: the postcondition may apply to labels that have not been generated yet. To address this, partially matched postconditions of these rules are stored in the necessities list. Whenever a new label is generated, any matches from this list to the new label are added to the match queue. These data structures are specific to a branch, i.e., when a new branch is created, it receives a copy of the current match queue and necessities list.

For reasons of efficiency, the match queue is a priority queue and rules can be given a numeric priority value, which specifies the order in which matches are applied to the tableau. In this way, we may define a strategy to close branches as soon as possible. For example, it is preferable to execute all possible non-branching propositional rules before attempting to execute any other rules.

The rules are implemented in such a way that they are easily replaceable by a different ruleset. Moreover, the tableau generator allows the generation process to be monitored. This enables the decoupled implementation of such features as tableau visualization and counter-model construction (see Section 3).

2.3 Input language

OOPS employs an input language for formulas implemented using the SableCC [6] compiler generator for Java. Propositions are input as strings of characters and digits, starting with a lowercase character. Agent identities are represented by natural numbers. OOPS uses the widely understood infix notation for logical formulas. Table 2 shows OOPS ascii equivalents for different logical operators, as well as their precedences; lower numbers indicate stronger bindings.

In addition to this, the language allows the input of variables as placeholders for either (sub-)formulas or agent identities. This is useful in the definition of rules and allows one to create template formulas that can be instantiated in different ways, by substitution. Variables are strings of characters and digits that start with an

uppercase character.

3 Functionality

In this section, we highlight a number of features of OOPS that we believe are important in an educational setting. Even though other systems may share some of OOPS' features, there is no other system that possesses all of them.

3.1 Integrated Scripting

In order for a theorem prover to be truly useful, it is not sufficient to be able to answer 'true' or 'false' given an input formula. Rather, our experience has shown that students will need to formulate and extend theories. Doing this by hand by editing a single large formula quickly becomes unmanageable. Moreover, we want to have a powerful toolbox to assist us in the formulation of larger theories. This toolbox should include general programming constructs such as loops and conditionals. Some other tools, such as the Logics Workbench, provide custom scripting languages for this purpose. The advantage of this approach is that the language can be tailored specifically to common usage of the prover. The disadvantage, however, is that developing a custom language is costly. Therefore, that the resulting language is likely to be lacking in expressive power. Furthermore, the user has to learn a language that has no application outside of the prover and for which support (i.e., documentation, user community and bug fixes) may be limited.

To address these concerns, OOPS integrates the general-purpose scripting language Lua. Lua has been designed specifically to be an embeddable language and is widely used both as an extension language and as a front-end for libraries written in other languages. Thus, Lua enables us to define an environment that is tailored to the needs of theorem proving, while avoiding the concerns associated with implementing a custom language. See [13] for a good introduction to programming in Lua. Currently, most of OOPS' functionality is available from Lua and more extensive support is being worked on.

The above outlines our reasons for integrating OOPS with Lua. Now we briefly describe how OOPS can be used through its Lua interface. All OOPS methods are encapsulated in the oops namespace. The basis for interaction with OOPS through Lua is the *theory* concept. A theory is, simply put, a collection of formulas. The following example code creates a theory and adds a formula to it:

```
th = oops.Theory()
th:add("#_1 p")
```

We define a number of operations on theories: checking of consistency, provability of a formula within a theory and satisfiability of a formula within a theory:

```
print(th:consistent())              -- true
print(th:provable("#_2 #_1 p"))     -- false
print(th:satisfiable("~#_2 p"))     -- true
```

where -- starts a comment, here used to indicate the output produced by the **print** statement. Now, to aid in the construction of theories, we allow the explicit creation of formulas, on which we have defined the operation of substitution. For example:

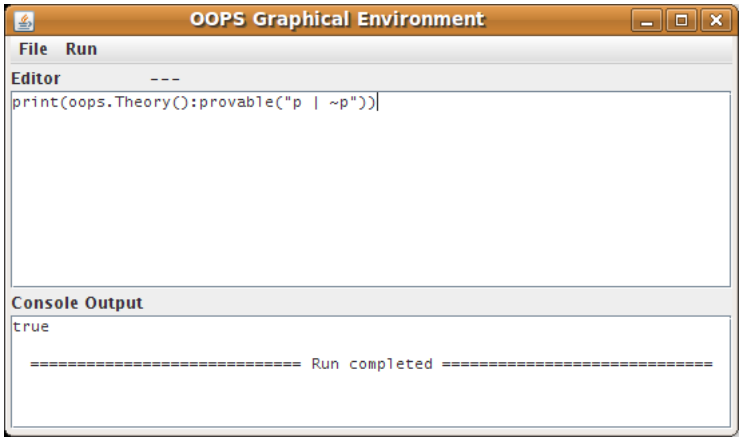

Figure 1. The OOPS Graphical User Interface.

```
th = oops.Theory()
f = oops.Formula("#_A V")
for i=1,4,1 do
        th:add(f:substitute({V = "p | q"}, {A = i}))
end
print(th)
```

which expresses that each of the agents $1, 2, 3$ and 4 'knows' $(p \vee q)$. The resulting output is:

```
[#_4 (p | q), #_3 (p | q), #_1 (p | q), #_2 (p | q)]
```

This completes our description of how OOPS is called from Lua. It must be noted that Lua is a very powerful and complete language and that much more can be achieved than is suggested by the above examples. For example, command-line interaction with the user is readily available through Lua.

3.2 Graphical User Interface

As is discussed above, Lua provides a convenient scripting interface to OOPS. However, modern computer users do not expect to run applications from the command-line. Even if they are used to this concept, it is not always the most convenient method of interaction. Therefore, OOPS includes a very simple Graphical User Interface (GUI), in which scripts can be displayed, edited and executed (Figure 1). In addition, scripts can be loaded from and saved to a file. For those who prefer to use an external editor (e.g., there are many editors that offer syntax highlighting for Lua), a single key combination reloads a modified file from the file system. The application consists of two panels: the top panel shows the current script and the bottom panel shows the output.

Though minimal, the GUI greatly enhances the convenience with which OOPS can be used. Firstly, script and output are shown in one place, allowing for easy cross-referencing. Second, scripts are run through a single key combination (or invocation from the menu). Finally, the load, save and refresh functionalities give the user the freedom to use the integrated editor or an external editor of choice with equal convenience.

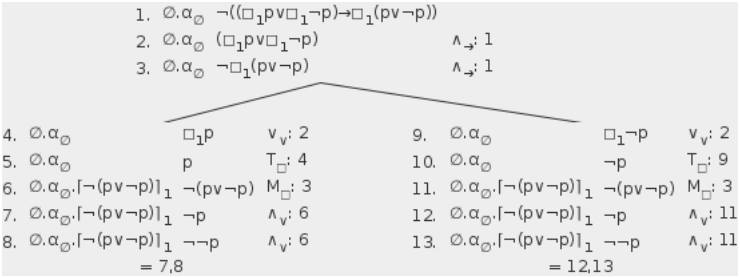

Figure 2. Visualization of the tableau that checks provability of φ, by attempting to satisfy $\neg\varphi$, where $\varphi = ((\Box_1 p \vee \Box_1 \neg p) \rightarrow \Box_1(p \vee \neg p))$. In this case, the tableau is closed (i.e. φ is provable), as indicated by the $= (m, n)$ under each branch, where m and n indicate the line numbers at which two contradicting formulas are found.

3.3 Free and Convenient Distribution

As we noted in Section 1, most current proof tools have problems related to either platform dependence, aging dependencies, lack of maintenance or difficult installation procedures. OOPS addresses these problems in several ways. First, OOPS is implemented in pure Java, which means that OOPS will run on any operating system for which a Java virtual machine is available. This is true for most operating systems available today. Second, OOPS is distributed as a ZIP file that includes all dependencies. No installation is needed, one simply extracts the ZIP file and double-clicks the resulting oops.jar file. Hence, OOPS is platform independent and easy to run, having no dependencies apart from the Java VM and what is provided in the OOPS distribution.

The concern of continued maintenance is harder to address. To ensure that OOPS can be used and extended in the future by anyone who wishes to do so, we provide the full source code [7] under the GNU General Public License (GPL). It is our hope that others will contribute extensions to OOPS.

3.4 Visualization of Tableaux

When a student is learning to work with modal logics, a prover can often give surprising results. He or she may encounter undesirable outcomes when constructing a theory and would like to be able to 'debug' the theory by inspecting the proof process. Moreover, inspecting generated tableaux may enhance understanding of tableau methods and the semantics of modal logics in general. To support this, OOPS includes a visualization module for labeled tableaux. Figure 2 shows an example of such a visualization. The tableau is drawn as a tree. In the tree, nodes are numbered in the order in which they are added (left-most on each line). After the node number, the label is shown, followed by the formula. Finally, the rule that resulted in the creation of the specific node and the node number to which the rule matched are given.

The visualization is implemented as an observer on the tableau generator (see Section 2.2). The Lua code to generate Figure 2 is as follows (note that the command-line output will be **true**):

```
oops.attachTableauVisualizer()
```

[7] http://github.com/gertvv/oops

```
print(oops.Theory():provable(
    "(#_1 p | #_1 ~p) > #_1(p | ~p)"))
```

3.5 Visualization of Countermodels

In addition to being able to view the tableau, it may be helpful to be able to inspect a model that the tableau corresponds to. In case the tableau is open, the generated model will generally be more insightful, as it does not contain any redundant information. As is the case for tableau visualization, visualization of (counter-)models may enhance understanding of the semantics of modal logics.

Figure 3 is generated by the following Lua code (note that the command-line output will be **false**):

```
oops.attachModelConstructor()
print(oops.Theory():provable("#_1 p | #_1 ~p"))
oops.showModel()
```

As the reader will notice, the invocation of the model visualization is done differently from the tableau visualization. This is because we treat models as entities in their own right. In fact, the call oops.getModel() can be used to retrieve the most recently constructed model, if the last invocation of the tableau generator resulted in an open tableau. The Lua **print** function will output a textual representation of the model. However, further programmatic manipulation and inspection of models is future work.

4 Example

To illustrate the educational potential of OOPS, we present a worked-out example that we have assigned to students of multi-agent systems in the past, at the Department of Artificial Intelligence at the University of Groningen. It is inspired by Hans van Ditmarsch' Cluedo exercises [14], designed to be solved with the Logics Workbench. Our example concerns a variant of the Wise Men's Riddle, and the students' task is to check the riddle's solution, after formalizing it with OOPS. This is the variant in question, of unknown origins:

> There once was a wise queen, who was a perfect logician. For advice, she relied on three wise men, who were likewise perfect logicians. This was common knowledge among the four of them, as was the fact that none of them would ever lie or cheat.

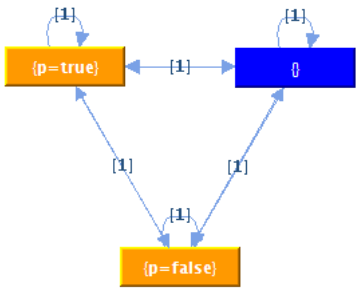

Figure 3. Visualization of a counter-model for $\varphi = (\Box_1 p \vee \Box_1 \neg p)$. The 'main' world is indicated in blue.

> One day, the queen wanted to demonstrate to her people just how wise her wise men were. She announced that she would place a hat on each of their heads, and that each of the wise men would be able to see the hats of the other two, but not his own.
>
> The queen then announced that she had three red hats and two blue hats total, and that each wise man was to determine the color of his own hat. The queen then placed the hats, and said: "Each wise man who knows the color of his hat, must now step forward."
>
> After this statement, no wise man stepped forward. So the queen repeated it, and still no wise man stepped forward. Yet, after she made her announcement a third time, all wise men stepped forward at once. What were the colors of the wise men's hats?

When we present this assignment to students, we ask them to model the riddle in steps, and to check the epistemic consequences of different hat color distributions. Eventually, this leads to an `OOPS` script that models the situation with three red hats. By taking the perspective of one of the wise men, a very minimal script is sufficient to prove that he can derive the color of his hat after observing the consequences of two announcements by the queen.

This minimal `OOPS` script is reproduced below, with informal explanations in the comments. The relevant propositions are defined as follows: Let `r1` mean that wise man 1 has a red hat, `r2` that wise man 2 has a red hat, and `r3` that wise man has a red hat. Let `b1, b2, b3` express the same relationships between wise men and hats, but for blue ones.

```
th = oops.Theory()

-- there are only two blue hats:
th:add("#_1 #_2 #_3 (((b1 & b2) > ~b3) & ((b1 & b3) > ~b2))")

-- a wise man has exactly one hat:
th:add("#_1 #_2 #_3 ((b1 = ~r1) & (b2 = ~r2) & (b3 = ~r3))")

-- a wise man sees the hats of the other two:
th:add("#_1 (#_2 b1 | #_2 r1)")
th:add("#_1 #_2 (#_3 b1 | #_3 r1)")
th:add("#_1 #_2 (#_3 b2 | #_3 r2)")

-- wise man 1 sees two red hats:
th:add("#_1 (r2 & r3)")

-- after the first announcement:
th:add("#_1 ~#_2 r2")

-- after the second announcement:
th:add("#_1 #_2 ~#_3 r3")

-- wise man 1 knows his hat is red:
print(th:provable("#_1 r1"))
```

This example demonstrates the kind of assignment that can be designed with `OOPS`. It gives students the experience of using a theorem prover, and lets them experiment with different assumptions, providing insight into the formal logic that underlies a familiar riddle. This can all be done quickly and easily due to `OOPS`'s integrated scripting facility and intuitive GUI.

5 Conclusions and Further Work

In this paper, we have presented OOPS, a cross-platform, easy to install and open source tableau prover for $S5_n$. OOPS provides users with a graphical user interface, an integrated scripting language, tableau visualization and counter-model generation. We believe these features make OOPS more suited for educational use than other similar systems.

Given this, we now identify several directions for further work on OOPS. First of all, although the implementation currently allows new rule sets to be implemented relatively easily, this requires extending the Java source code and recompiling OOPS. To remedy this, we would like to implement rule sets as Lua modules and provide such modules for several logics. We would also like to implement an algorithm that allows the $S5_n$ tableau to be generated in PSPACE, as our current implementation may require an exponential amount of space.

Furthermore, we would like to have a more complete set of tools to interact with theories, formulas and Kripke models. For example, it should be possible to simplify formulas and theories. In the case of Kripke models, we would like to be able to construct and alter models ourselves and to perform operations such as model checking and bisimulation. Finally, the GUI should allow users to provide keyboard input to Lua scripts (through 'standard in') so that one can develop interactive OOPS scripts.

References

[1] Abate, P. and R. Goré, *The Tableau Workbench*, Electronic Notes in Theoretical Computer Science **231** (2009), pp. 55–67.

[2] Beckert, B. and R. Gore, *Free variable tableaux for propositional modal logics*, Automated Reasoning With Analytic Tableaux Related Methods **1227** (1997), pp. 91–106.

[3] de Boer, M., "Praktische Bewijzen in Public Announcement Logica," Master's thesis, University of Groningen, Groningen, the Netherlands (2006).

[4] de Boer, M., *KE tableaux for Public Announcement Logic*, in: B. Dunin-Kepliçz and R. Verbrugge, editors, *FAMAS'007*, 2007, pp. 56–69.

[5] Fitting, M. and R. Mendelsohn, "First-Order Modal Logic," Synthese Library **277**, Kluwer Academic Publishers, 1999.

[6] Gagnon, É., "SableCC, an Object-Oriented Compiler Framework," Master's thesis, McGill University, Montreal, Quebec, Canada (1998).

[7] Gasguet, O., A. Herzig, D. Longin and M. Sahade, *LoTReC: Logical Tableaux Research Engineering Companion*, in: B. Beckert, editor, *TABLEAUX 2005* (2005), pp. 318–322.

[8] Gosling, J., B. Joy, G. L. Steele and G. Bracha, "The Java Language Specification," The Java Series, Prentice Hall PTR, 2005, third edition.

[9] Halpern, J. Y. and Y. Moses, *A guide to completeness and complexity for modal logics of knowledge and belief*, Artificial Intelligence **54** (1992), pp. 319–379.

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

[11] Horrocks, I., *The FaCT system*, in: H. de Swart, editor, *TABLEAUX 1998* (1998), pp. 307–313.

[12] Hustadt, U. and R. Schmidt, *MSPASS: Modal reasoning by translation and first-order resolution*, in: R. Dyckhoff, editor, *TABLEAUX 2000* (2000), pp. 67–71.

[13] Ierusalimschy, R., "Programming in Lua," Lua.org, 2006, 2nd edition.

[14] van Ditmarsch, H., *Logics Workbench: Multi-agent systems* (unknown), http://www.ai.rug.nl/mas/documents/multiagent.pdf.

[15] van Valkenhoef, G., *Elaborations on OOPS – project report* (2008), http://www.ai.rug.nl/~valkenhoef/oops/elaborations.pdf.