Elaborations on OOPS

Project Report

Gert van Valkenhoef

Supervisor: Rineke Verbrugge

June 26, 2008

Multi-agent Systems Group
Artificial Intelligence
University of Groningen

Contents

1 Introduction

This report describes OOPS, an Object Oriented Prover for $S5_n$. $S5_n$ is a particular instance of modal epistemic logics (Halpern and Moses, 1992), that serves as the de facto standard model of idealized knowledge for many applications in AI and computer science. See e.g. Van der Hoek and Meyer (2004) for an introductory text on epistemic logic and its applications.

OOPS is a Java implementation of the **ELtap** proof system (De Boer, 2006) and it was originally conceived as a final project for the course Multi-Agent Systems at the University of Groningen (Van der Vaart and van Valkenhoef, 2007). OOPS is a tableaux prover, because such systems stand appealingly close to the semantics of the logical system (see section 3).

The creation of OOPS was inspired by the Logics Workbench (LWB) (Heuerding et al., 1996), in its role as a computational aid for the Multi-Agent Systems course. The Logics Workbench has several shortcomings and therefore we sought to create an alternative. Specifically, these shortcomings were lack of portability, the antiquated libraries it depends on, and the somewhat dysfunctional user interface. Furthermore, the LWB source code is not freely available, thus these shortcomings can't easily be fixed.

Therefore, when creating OOPS, we wanted to create a proof system that is well documented, portable and for which the source code is freely available under an open source license. Furthermore, since its initial application is as a tool for students in the Multi-Agent Systems course taught in Groningen, we wanted the prover to support $S5_n$ and to provide as much insight into its workings as possible. For that purpose, the current project extended OOPS to support the visual inspection of the generated tableau and the generation of a counter-model from the tableau.

The original project (Van der Vaart and van Valkenhoef, 2007), despite delivering a working implementation of the prover, did not include proofs of any of the formal properties of the system, nor was the performance thoroughly tested. Furthermore, a huge list of desirable features were identified, but not yet implemented.

The current project, entitled 'Elaborations on OOPS', aims to resolve most of these shortcomings. Therefore, the important formal properties (soundness, completeness, correctness and complexity) of the system are proven. The implementation is thoroughly described and improved and extended in several ways. Finally, the performance of OOPS is empirically tested; several versions of OOPS are compared to another proof system.

This report is organised as follows: first, the syntax and semantics of the system $S5_n$ are briefly introduced (section 2): the reader is assumed to be familiar with $S5_n$, but the notation must be unambiguous. In section 3, semantic tableaux are introduced as a means of satisfiability checking for $S5_n$ and specifically, the system used by OOPS is described. The soundness and completeness of this system is proven in section 4. Details of the

implementation (section 5) are then given, including a precise description of the core algorithm and several peripheral functionalities. The following section (section 6) shows that the implementation is correct, that is, that it corresponds precisely to the proof system described in section 3. The computational complexity is analyzed in section 7, the practical implications of which are then tested empirically (section 8). Finally the project as a whole is discussed and evaluated (section 9) and further work is identified (section 10).

Note that in this document, $\boxtimes$ is used as the QED symbol, in order to avoid confusion with the modal operator $\Box$.

2 $S5_n$: Syntax & Semantics

This section describes the syntax and semantics of $S5_n$, in the minimal sense that is required for the proofs of the formal properties of `OOPS`. In actuality, `OOPS` supports the language introduced below, in definition 2.1 and the abbreviations of table 1. Some further extensions are possible, but discussion will be deferred until they become relevant.

2.1 Syntax

Definition 2.1. *The logical* language $\mathcal{L}_n(\Phi)$ *with* n *agents (named* $1, \ldots, n$*) and a nonempty, possibly countably infinite, set of primitive propositions* Φ*, is the least set of formulas for which:*

- *If* $p \in \Phi$*, then* $p \in \mathcal{L}_n(\Phi)$
- *If* $\varphi \in \mathcal{L}_n(\Phi)$*, then* $(\neg\varphi) \in \mathcal{L}_n(\Phi)$
- *If* $\varphi, \psi \in \mathcal{L}_n(\Phi)$*, then* $(\varphi \wedge \psi) \in \mathcal{L}_n(\Phi)$
- *If* $\varphi \in \mathcal{L}_n(\Phi)$ *and* $1 \leq i \leq n$*, then* $(\Box_i\varphi) \in \mathcal{L}_n(\Phi)$

Although the above is the minimal language needed for soundness and completeness results, a number of convenient abbreviations may be used (see table 1). In addition, parentheses will be omitted whenever readability permits it.

Definition 2.2. *The* size $|\varphi|$ *of a formula* $\varphi \in \mathcal{L}_n(\Phi)$ *is its length over the alphabet* $\Phi \cup \{\neg, \wedge, (,), \Box_i, \ldots, \Box_n\}$.

Note that the agent subscripts are part of the operator according to the above definition and hence do not count towards the size of the formula.

Definition 2.3. *The* modal depth *(or* depth*)* $dep(\varphi)$ *of a formula* $\varphi \in \mathcal{L}_n(\Phi)$ *is defined as:*

$\varphi \vee \psi$	$\neg(\neg\varphi \wedge \neg\psi)$
$\varphi \rightarrow \psi$	$\neg(\varphi \wedge \neg\psi)$
$\varphi \leftrightarrow \psi$	$\neg(\varphi \wedge \neg\psi) \wedge \neg(\neg\varphi \wedge \psi)$
$\Diamond_i \varphi$	$\neg\Box_i\neg\varphi$
$\bot$	$p \wedge \neg p$
$\top$	$\neg\bot$

Table 1: Abbreviations for formulas in $\mathcal{L}_n(\Phi)$

- $dep(p) = 0$ *for* $p \in \Phi$
- $dep(\neg\psi) = dep(\psi)$
- $dep(\varphi \wedge \psi) = max(dep(\varphi), dep(\psi))$
- $dep(\Box_i\psi) = 1 + dep(\psi)$ *where* $1 \leq i \leq n$

The depth of a formula φ corresponds to the levels of nesting of the box operators in φ. Clearly, $dep(\varphi) < |\varphi|$ for all formulas φ.

Definition 2.4. *A formula ψ is a* subformula *of φ if either:*

- $\psi = \varphi$
- $\varphi = \neg\varphi'$ *and ψ is a subformula of φ'*
- $\varphi = \varphi' \wedge \varphi''$ *and ψ is a subformula of φ' or φ''*
- $\varphi = \Box_i\varphi'$ *and ψ is a subformula of φ'*

$Sub(\varphi)$ is the set of all subformulas of φ. $Sub^+(\varphi)$ is the set of all subformulas of φ and their negations.

It turns out that the length of the formula φ forms an upper bound on the number of subformulas of φ: $|Sub(\varphi)| \leq |\varphi|$.

2.2 Semantics

The semantics are defined in the customary way, that is, through possible world semantics (Halpern and Moses, 1992). The formal framework used is that of *Kripke structures*, in which the notion of 'possible worlds' is captured by an accessibility relationship between worlds.

Definition 2.5. *A* Kripke structure for n agents is a tuple $M = (S, \pi, R_1, \ldots R_n)$, where S is a set of *states* or *possible worlds*, π is a truth assignment to the primitive propositions of Φ for each state $s \in S$ ($\pi(s) : \Phi \mapsto \{0, 1\}$) and R_i is a binary relation on the states of S for $i = 1, \ldots, n$. The *size* of a structure M is the number of states in S. Structures with infinite size are allowed. Let $\mathcal{M}_n$ be the set of all Kripke structures with n agents.

Definition 2.6. *A formula φ is true (satisfied) in a state s in structure M, written $(M, s) \models \varphi$, according to the following:*

- $(M, s) \models p$ *for* $p \in \Phi$ *iff* $\pi(s)(p) = 1$
- $(M, s) \models \varphi \wedge \psi$ *iff* $(M, s) \models \varphi$ *and* $(M, s) \models \psi$
- $(M, s) \models \neg\varphi$ *iff* $(M, s) \not\models \varphi$
- $(M, s) \models \Box_i\varphi$ *iff* $(M, t) \models \varphi$ *for all* t *such that* $(s, t) \in R_i$.

Definition 2.7. *Let $M = (S, \pi, R_1, \ldots, R_n)$ be any Kripke structure and let $\mathcal{M}$ be any class of Kripke structures. The following may be defined regarding satisfiability and validity of formulas:*

- φ *is valid in* M *(M is a model of φ), written* $M \models \varphi$, *if* $(M, s) \models \varphi$ *for all* $s \in S$
- φ *is valid with respect to* $\mathcal{M}$, $\mathcal{M} \models \varphi$, *if* $M \models \varphi$ *for all* $M \in \mathcal{M}$
- φ *is satisfiable in* M *if* $(M, s) \models \varphi$ *for some state* $s \in S$
- φ *is satisfiable in* $\mathcal{M}$ *if* φ *is satisfiable in some* $M \in \mathcal{M}$

It is not difficult to show that φ is valid in M (respectively $\mathcal{M}$) iff $\neg\varphi$ is not satisfiable in M (respectively $\mathcal{M}$). The following theorem (Halpern and Moses, 1992) captures some of the formal properties of $\models$:

Theorem 2.8. *Let $\mathcal{M}_n$ denote the class of all Kripke structures with n agents. For all formulas $\varphi, \psi \in \mathcal{L}_n(\Phi)$, structures $M \in \mathcal{M}_n$ and agents $i = 1, \ldots, n$:*

1. *if φ is an instance of a propositional tautology, then $M \models \varphi$*
2. *if $M \models \varphi$ and $M \models \varphi \rightarrow \psi$, then $M \models \psi$*
3. *$M \models (\Box_i\varphi \wedge \Box_i(\varphi \rightarrow \psi)) \rightarrow \Box_i\psi$*
4. *if $M \models \varphi$ then $M \models \Box_i\varphi$*

The above theorem shows that by using Kripke structures, there are certain constraints on the inferences that are valid in the system. These can be captured in the axiom system K_n, which is known to be a sound and complete axiomatization with respect to $\mathcal{M}_n$ (Halpern and Moses, 1992):

Definition 2.9. *The axiom system K_n consists of two axioms:*

(A1) *All instances of tautologies of the propositional calculus*

(A2) $(\Box_i\varphi \wedge \Box_i(\varphi \rightarrow \psi)) \rightarrow \Box_i\psi$ $i = 1, \ldots, n$

And two rules of inference:

(R1) *From* $\vdash \varphi$ *and* $\vdash \varphi \rightarrow \psi$ *infer* $\vdash \psi$ *(modus ponens)*

(R2) *From* $\vdash \varphi$ *infer* $\vdash \Box_i \varphi$ *(generalization)*

A formula φ *is* K_n provable*, denoted* $K_n \vdash \varphi$ *if:*

- φ *is an instance of one of the axioms*
- φ *follows from* K_n *provable formulas by one of the inference rules*

The definition of provability relative to an arbitrary axiom system $\mathcal{S}$ *is defined analogously. We say a formula* φ *is* $\mathcal{S}$-consistent *if* $\neg\varphi$ *is not* $\mathcal{S}$*-provable.*

Definition 2.10. *The following* additional axioms *are often used to arrive at a useful characterization of the modal operator:*

(A3) $\Box_i \varphi \rightarrow \varphi$, $i = 1, \ldots, n$

(A4) $\Box_i \varphi \rightarrow \Box_i \Box_i \varphi$, $i = 1, \ldots, n$

(A5) $\neg\Box_i \varphi \rightarrow \Box_i \neg\Box_i \varphi$, $i = 1, \ldots, n$

(A6) $\neg\Box_i \bot$, $i = 1, \ldots, n$

Definition 2.11. *The axiom system* $S5_n$ *is characterized as* K_n *with the extra axioms* A3*,* A4 *and* A5*.*

It is possible to define several properties on the relations R_i that restrict the possible Kripke structures we consider.

Definition 2.12. Conditions on the possibility relation R*:*

- R *on a set* S *is* reflexive *if* $(s, s) \in R$ *for all* $s \in S$
- R *is* transitive *if, for all* $s, t, u \in S$*, if* $(s, t) \in R$ *and* $(t, u) \in R$ *then* $(s, u) \in R$
- R *is* symmetric *if, for all* $s, t \in S$*, if* $(s, t) \in R$*, then* $(t, s) \in R$

Let $\mathcal{M}_n^{rst}$ *be the class of Kripke structures for which the accessibility relations* R_i, $i = 1, \ldots, n$*, are reflexive, symmetric and transitive.*

Theorem 2.13. *The system* $S5_n$ *is a sound and complete axiomatization with respect to* $\mathcal{M}_n^{rst}$ *(Halpern and Moses, 1992).*

Theorem 2.13 shows that every structure $M \in \mathcal{M}_n^{rst}$ is a model of $S5_n$. The converse, that every model M of $S5_n$ is also in $\mathcal{M}_n^{rst}$, does not hold. However, we can easily find an equivalent model that is in $\mathcal{M}_n^{rst}$:

Theorem 2.14. *If M is a model of $S5_n$, then so is its reflexive, symmetric, transitive closure M^{rst}. Moreover, $M^{rst} \in \mathcal{M}_n^{rst}$ and M and M^{rst} are equivalent (they satisfy exactly the same formulas) (Halpern and Moses, 1992).*

In addition, the following shows that $S5_n$ is decidable:

Theorem 2.15. *If φ is $S5_n$ consistent, then φ is satisfiable in a structure $M \in \mathcal{M}_n^{rst}$ with at most $2^{|\varphi|}$ states, where every primitive proposition $p \in \Phi$ which is not a subformula of φ is false at every state (Halpern and Moses, 1992). Hence, the model is finitely representable, even if Φ is countably infinite.*

3 Tableaux

The `OOPS` tableaux system for $S5_n$ is based on the proof system **ELtap** (De Boer, 2006), which in turn draws on Fitting and Mendelsohn (1999) and Beckert and Gore (1997). `OOPS`' handling of labels is more rigid than **ELtap**'s and is inspired directly by Beckert and Gore (1997). Unlike that work, however, `OOPS` does not reduce modal logic to predicate calculus, thus avoiding the complications that arise there.

The definition of a tableau used here is taken from Halpern and Moses (1992), section 6.3. Following that paper, `OOPS` is considered to generate a *pre-tableau* for a formula φ, from which a true *tableau* for φ may be derived.

Definition 3.1. *A* propositional tableau *is a set T of formulas such that:*

1. *if $\neg\neg\psi \in T$, then $\psi \in T$*
2. *if $\psi \wedge \psi' \in T$, then both $\psi, \psi' \in T$*
3. *if $\neg(\psi \wedge \psi') \in T$, then either $\neg\psi \in T$ or $\neg\psi' \in T$*
4. *it is not the case that both ψ and $\neg\psi$ are in T for some formula ψ*

T is a propositional tableau for φ *if T is a propositional tableau and $\varphi \in T$.*

Theorem 3.2. *The propositional formula φ is satisfiable iff there is a propositional tableau for φ (Halpern and Moses, 1992).*

Definition 3.3. *A K_n* tableau *is a tuple $T = (S, L, R_1, \ldots, R_n)$, where S is a set of states, $R_1, \ldots, R_n$ are accessibility relations and L is a* labeling function *that associates with each state $s \in S$ a set $L(s)$ of formulas such that:*

1. *$L(s)$ is a propositional tableau*
2. *if $\Box_i\psi \in L(s)$ and $(s,t) \in R_i$, then $\psi \in L(t)$*

3. *if* $\neg\Box_i\psi \in L(s)$*, then there exists a* $t \in S$ *with* $(s,t) \in R_i$ *and* $\neg\psi \in L(t)$

$T = (S, L, R_1, \ldots, R_n)$ *is a* K_n *tableau for* φ *if* T *is a* K_n *tableau and* $\varphi \in L(s)$ *for some state* $s \in S$*.*

Definition 3.4. *An* $S5_n$ tableau *is a* K_n *tableau that satisfies in addition:*

4. *if* $\Box_i\psi \in L(s)$*, then* $\psi \in L(s)$

5. *if* $(s,t) \in R_i$ *then* $\Box_i\psi \in L(s)$ *iff for all* $(t,u) \in R_i^{rst}$*,* $\psi \in L(u)$ *(where* R_i^{rst} *is the reflexive, symmetric, transitive closure of* R_i*)*

Condition (5) replaces the much simpler formulation of Halpern and Moses (1992), which states 'if $(s,t) \in R_i$ then $\Box_i\psi \in L(s)$ iff $\Box_i\psi \in L(t)$'. The reformulation is necessary because the tableau method used by Halpern and Moses (1992) relies on full direct carry-over of all modal formulas for the relevant agent for the construction of a reflexive, symmetric, transitive model, whereas `OOPS` expresses this constraint by restrictions on the labels that are generated (modalities for one agent only add one level of labels, see theorem 4.1), as is explained in section 3.1.

Theorem 3.5. *A formula* φ *is* $S5_n$ *satisfiable iff there is a* $S5_n$ *tableau for* φ*.*

Proof. The proof (Halpern and Moses, 1992) proceeds by showing that from a model in which φ is satisfied, a tableau for φ can trivially be derived and from a $S5_n$ tableau for φ, an $S5_n$ model can be generated that satisfies φ. ⊠

3.1 `OOPS` pre-tableaux

An `OOPS` pre-tableau is a collection of branches. A branch consists of a number of nodes. Each node is made up of a label and a formula. New branches are created by the application of rules to existing nodes on a branch. These terms are introduced more formally below.

Definition 3.6. σ *is a* label *(for a* $\mathcal{L}_n(\Phi)$ *formula) if:*

- $\sigma = 0$ *(*σ *is the empty label)*
- $\sigma = 0.\alpha_0$ *(*σ *is the top label)*
- $\sigma = \tau.k_i$*, where* τ *is a label,* $k \in \mathbb{N}$ *and* $1 \leq i \leq n$

The length *of a label is the number of dots it contains.*

Informally, we may think of labels as representing worlds, except for the empty label, which does not represent any world, but allows for a more concise representation of OOPS' rules (that is, the notation for labels is more involved than strictly necessary, but this simplifies the notation for rules and corresponds directly to the implementation).

The label $0.\alpha_0$ (which may be written '1' for brevity) is the 'top' world, which we will want to satisfy φ. The constant α may be taken to be any natural number (by convention, we may pick 0), the subscript 0 is not identical to any of the agents $1 \ldots n$. The label $\tau.k_i$ corresponds to a world accessible for agent i from the world that corresponds with τ.

Definition 3.7. *A* node *(or labeled formula) γ is defined as $\gamma = \sigma\ \varphi$, where σ is a label for the $\mathcal{L}_n(\Phi)$ formula φ.*

Definition 3.8. *A* branch *B is either:*

- *$\emptyset$ (the empty branch)*
- *A tuple (B', γ) where B' is a branch and γ is a node.*

For convenience, a branch may be seen as the set of nodes it contains. The size *of a branch, $|B|$ is the number of nodes it contains.*

Define the labeling function *$L'(B, \sigma) = \{\psi \mid \sigma\ \psi \in B\}$ and the* label function *$l(B) = \{\sigma \mid \exists_\psi(\sigma\ \psi \in B)\}$*

Definition 3.9. *A branch B' is an* extension *of a branch B (B is a sub-branch of B') if:*

- *$B' = B$ or*
- *$B' = (B'', \gamma)$ and B'' is an extension of B.*

The above defines the *structure* of the OOPS pre-tableau. What remains is to show how a pre-tableau for a formula φ may be built. This proceeds by the execution of rules on a branch, as defined below. However, first the concept of a rule must be introduced and several notations need to be extended in order to be able to express rules.

Although the language $\mathcal{L}_n(\Phi)$ (definition 2.1) is convenient for the proofs of formal properties of the proof system, the rules are expressed in a more elaborate language that includes the abbreviations of table 1 and allows not only agent constants and primitive propositions, but also variables for which a formula or agent may be substituted:

Definition 3.10. *The language $\mathcal{L}_n^+(\Phi)$ with n agents (named $A = 1, \ldots, n$), a non empty set of primitive propositions Φ, a set of agent variables V_a and a set of formula variables V_f, is the least set of formulas for which:*

- *If $\varphi \in \mathcal{L}_n(\Phi)$, then $\varphi \in \mathcal{L}_n^+(\Phi)$*

- *If* φ *an abbreviation of* $\psi \in \mathcal{L}_n^+(\Phi)$*, then* $\varphi \in \mathcal{L}_n^+(\Phi)$
- *If* $v \in V_f$*, then* $v \in \mathcal{L}_n^+(\Phi)$
- *If* $\varphi \in \mathcal{L}_n^+(\Phi)$ *and* $a \in V_a$*, then* $\Box_a\varphi \in \mathcal{L}_n^+(\Phi)$

$\mathcal{L}_n(\Phi)$*is defined in definition 2.1 and abbreviations are listed in table 1.*

The language $\mathcal{L}_n^+(\Phi)$is sufficiently powerful to express formulas as used in the tableau rules defined below. Labels (definition 3.11) are also extended by the possibility of all or part of the label being a variable:

Definition 3.11. σ *is a* label *(for a* $\mathcal{L}_n(\Phi)$ *formula) if:*

- $\sigma = 0$ *(*σ *is the empty label)*
- $\sigma = 0.\alpha_0$ *(*σ *is the top label)*
- $\sigma = \tau.k_i$*, where* τ *is a label,* $k \in \mathbb{N}$ *and* $1 \leq i \leq n$

The length *of a label is the number of dots it contains.*

Definition 3.12. σ *is a* label containing variables *if:*

- $\sigma = 0$ *(*σ *is the empty label)*
- $\sigma = 0.\alpha_0$ *(*σ *is the top label)*
- $\sigma = \mu$*, where* μ *is a* label variable
- $\sigma = \tau.k_i$*, where* τ *is a label (containing variables) and* $k \in \mathbb{N}$ *or* k *is a* world variable *and* $1 \leq i \leq n$ *or* i *is an* agent variable
- $\sigma = \tau.{}^{\ulcorner}\varphi^{\urcorner}{}_i$*,* τ *and* i *as in the previous case and* $\varphi \in \mathcal{L}_n^+(\Phi)$

Definition 3.13. *A* rule R *consists of*

- *A precondition* $pre(R)$*, written above a horizontal bar, which is a node containing variables;*
- *A postcondition* $post(R)$*, written below the horizontal bar, which is a list of nodes containing variables. If these nodes are written top-to-bottom, the rule is linear. If the nodes are written left-to-right, the rule splits;*
- *Zero or more constraints, which restrict the values variables may take.*

The tableau rules used by OOPS are listed in table 2 . In the definition of these rules, $\ulcorner \cdot \urcorner$ is an arbitrary 1-1 function from $\mathcal{L}_n^+(\Phi)$ into $\mathbb{N}$. Note that there are also rules for formulas that are not in the language according to definition 2.1 and that these rules can, in fact, easily be derived by writing the pre- and postconditions in standard form and applying the 'fundamental' rules, those that involve only operators in the language. Therefore, in what follows only the fundamental rules will be considered. The derived rules are included for comprehensiveness, as the concrete implementation of OOPS includes them as well.

The semantics for rules are defined below. For the definitions, the notion of a 'substitution' is also needed:

Definition 3.14. *For the current purposes, a* substitution *may be viewed as the union of four relations:*

- *From formula variables to formulas not containing variables;*
- *From agent variables to agents;*
- *From 'world' variables to* $\mathbb{N}$*;*
- *From label variables to labels not containing variables.*

A substitution s *may be applied to a node* N *by replacing any* x *that occurs in* N *and* $(x, y) \in s$ *by* y*. The result of this operation is denoted* $s(N)$*. This definition can be extended to multiple nodes and thus to rules in the obvious manner.*

Definition 3.15. *A rule* R *is* applicable *to a node* N *if and only if there is a substitution* s *that satisfies the constraints and for the variables in the precondition* $pre(R)$*, such that* $s(pre(R)) = N$*.*

Definition 3.16. *A rule* R *is* executable *on a branch* B*, if and only if it is applicable to some node* $N \in B$*, with substitution* s *and there is a substitution* $s' \supseteq s$ *that satisfies the constraints and such that labels occurring in the postcondition also occur on the branch:* $s'(l(post(R))) \subseteq l(B)$*.*

The combination of the rule R *with the substitution* s' *that makes the rule* R concrete *(i.e.,* $s'(R)$ *does not contain variables) is called an* instantiation *of the rule* R*.*

Definition 3.17. *The* execution *of an instantiation of rule* R *with substitution* s' *on a branch* B *is defined by one of two cases:*

- *If the rule is linear, the nodes* $\{N_0, \ldots, N_n\} = s'(post(R))$ *are added to the current branch, giving the extension* $B' = ((\ldots(B, N_0), \ldots), N_n)$*;*
- *If the rule splits, the nodes* $\{N_0, \ldots, N_n\} = s'(post(R))$ *generate* n *extensions:* $B_0 = (B, N_0), \ldots, B_n = (B, N_n)$*.*

$\neg\neg =$ $\dfrac{\sigma \quad \neg\neg\varphi}{\sigma \quad \varphi}$		*Double Negation Rule*
$\wedge_\wedge =$ $\dfrac{\sigma \quad \varphi \wedge \psi}{\begin{array}{ll}\sigma & \varphi \\ \sigma & \psi\end{array}}$	$\wedge_\vee =$ $\dfrac{\sigma \quad \neg(\varphi \vee \psi)}{\begin{array}{ll}\sigma & \neg\varphi \\ \sigma & \neg\psi\end{array}}$	$\wedge_\rightarrow =$ $\dfrac{\sigma \quad \neg(\varphi \rightarrow \psi)}{\begin{array}{ll}\sigma & \varphi \\ \sigma & \neg\psi\end{array}}$	*Conjunctive Rules*
$\vee_\wedge =$ $\dfrac{\sigma \quad \neg(\varphi \wedge \psi)}{\sigma \quad \neg\varphi \mid \sigma \quad \neg\psi}$	$\vee_\vee =$ $\dfrac{\sigma \quad \varphi \vee \psi}{\sigma \quad \varphi \mid \sigma \quad \psi}$	$\vee_\rightarrow =$ $\dfrac{\sigma \quad \varphi \rightarrow \psi}{\sigma \quad \neg\varphi \mid \sigma \quad \psi}$	*Disjunctive Rules*
where $i \neq j$	$M_\Box =$ $\dfrac{\sigma.k_i \quad \neg\Box_j\varphi}{\sigma.k_i.{\ulcorner\neg\varphi\urcorner}_j \quad \neg\varphi}$	$M_\Diamond =$ $\dfrac{\sigma.k_i \quad \Diamond_j\varphi}{\sigma.k_i.{\ulcorner\varphi\urcorner}_j \quad \varphi}$	*Possibility Rules*
	$M_\Box* =$ $\dfrac{\sigma.k_i \quad \neg\Box_i\varphi}{\sigma.{\ulcorner\neg\varphi\urcorner}_i \quad \neg\varphi}$	$M_\Diamond* =$ $\dfrac{\sigma.k_i \quad \Diamond_i\varphi}{\sigma.{\ulcorner\varphi\urcorner}_i \quad \varphi}$	*Possibility Rules∗*
where $i \neq j$	$K_\Box =$ $\dfrac{\sigma.k_i \quad \Box_j\varphi}{\sigma.k_i.h_j \quad \varphi}$	$K_\Diamond =$ $\dfrac{\sigma.k_i \quad \neg\Diamond_j\varphi}{\sigma.k_i.h_j \quad \neg\varphi}$	*Basic Necessity Rules*
	$K_\Box* =$ $\dfrac{\sigma.k_i \quad \Box_i\varphi}{\sigma.h_i \quad \varphi}$	$K_\Diamond* =$ $\dfrac{\sigma.k_i \quad \neg\Diamond_i\varphi}{\sigma.h_i \quad \neg\varphi}$	*Basic Necessity Rules∗*
where $i \neq j$	$T_\Box =$ $\dfrac{\sigma.k_i \quad \Box_j\varphi}{\sigma.k_i \quad \varphi}$	$T_\Diamond =$ $\dfrac{\sigma.k_i \quad \neg\Diamond_j\varphi}{\sigma.k_i \quad \neg\varphi}$	*Special Necessity Rules*
	$R_\Box* =$ $\dfrac{\sigma.k_i \quad \Box_i\varphi}{\sigma \quad \varphi}$	$R_\Diamond* =$ $\dfrac{\sigma.k_i \quad \neg\Diamond_i\varphi}{\sigma \quad \neg\varphi}$	*Special Necessity Rules∗*

Table 2: Tableau Extension Rules

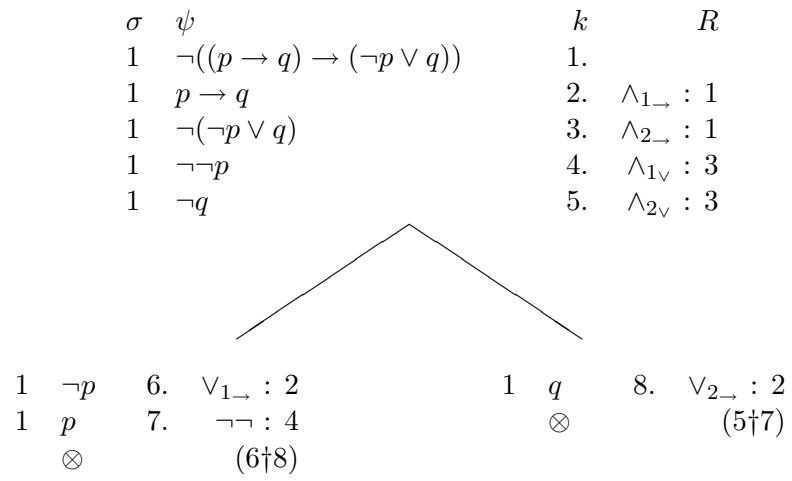

Figure 1: Tableau Proof of $(p \to q) \to (\neg p \vee q)$

Every instantiation of a rule R that has been applied to a branch B may not subsequently be applied to any extension of B. That is, redundant execution is avoided (however, these instantiations are still called executable, in accordance with the above definition, even if their execution would be redundant).

Note that if a rule is executable on a branch, it is also executable on any extension of that branch. Therefore, tableau rules may be executed in any order. Hence, we demand only that every executable rule instantiation is eventually executed exactly once.

The construction of the `OOPS` pre-tableau is now defined. A tableau may be derived from an *open*, *exhaustively expanded* branch of the pre-tableau (definition 3.18) by the method detailed in the proof of theorem 4.2.

Definition 3.18. *Mark a branch B as* closed *if either:*

1. *For some $\sigma \in l(B)$, $L(B,\sigma)$ is* blatantly inconsistent*, that is, $\psi, \neg\psi \in L(B,\sigma)$ for some ψ. In this case, B is also called* finalized
2. *All generated labels of the form $B' = (B,\gamma)$ are marked as closed*

A branch B is open *if it is not closed and no more rules can be applied (i.e. it has been* exhaustively expanded*).*

Definition 3.19. *The pre-tableau for a formula φ is* open *if and only if a branch containing $\gamma = \emptyset.\alpha_\emptyset\ \varphi$ is open. Otherwise, the pre-tableau is* closed*.*

Figure 1 shows a closed propositional tableau for $\neg\varphi = \neg((p \to q) \to (\neg p \vee q))$, which shows that $\neg\varphi$ is not satisfiable and so that φ is provable.

4 Soundness & Completeness

Theorem 4.1. *The generation of the* OOPS *pre-tableau terminates for all* φ.

Proof. It is easy to see that both the propositional rules and the modal rules add only subformulas, or their negations, of the precondition formula to the branch. Originally, only φ is on the branch, so by induction, for any label σ on a branch B, $L'(B,\sigma) \subseteq Sub^+(\varphi)$. So if $m = |\varphi|$, the size of any label is bounded: $|L'(B,\sigma)| \leq |Sub^+(\varphi)| \leq 2m$.

Take a label σ and consider how children may be added to σ. Due to the definition of the possibility rules, any child of σ is marked as $\sigma.\lceil\psi\rceil_i$, where $\psi \in Sub^+(\varphi)$ and $1 \leq i \leq n$. So we see directly that σ has at most $2mn$ direct descendants. In fact, looking at the precondition, we see that the agent subscript is also due to its presence in a subformula of φ and we can reduce the bound to $2m$.

Take a label $\sigma.k_i \neq 0.\alpha_0$, which was created due to some formula $\neg\Box_i\psi \in L'(B,\sigma)$ by rule $M_\Box$. Note that it is not difficult to modify this analysis for $M^*_\Box$. The necessity rules will carry over formulas from σ to $\sigma.k_i$ ($K_\Box$) and from 'neighbours' $\sigma.h_i$ ($K^*_\Box$) or from $\sigma.k_i$ itself ($T_\Box$) or its descendants ($R^*_\Box$). For the possibility rule, we can easily see that $dep(\neg\psi) < dep(\neg\Box_i\psi)$. The same goes for any formula carried over from σ by $K_\Box$: $dep(\psi) < dep(\Box_i\psi)$. For any formula ψ from the neighbours $K^*_\Box$, itself ($T_\Box$) or its descendants ($R^*_\Box$), the depth $dep(\psi) < dep(\Box_i\psi)$, so any application of the possibility rules reduces the depth of formulas.

So for the label $\sigma.k_i$, all formulas imported from σ are strictly less deep than their originators from σ. The same goes for the formulas any neighbour $\sigma.h_i$ inherits from σ. The formulas $\sigma.k_i$ inherits from $\sigma.h_i$ are again strictly less deep than those in $\sigma.h_i$. According to the same argument, the formulas begotten from $\sigma.k_i$ itself and its descendants are also strictly less deep than some formula in σ. To put it differently, eventually all formulas in $L(B,\sigma.k_i)$ derive from some deeper formula in $L(B,\sigma)$, thus for all σ, $\sigma.k_i$, $dep(L'(B,\sigma.k_i)) < dep(L'(B,\sigma))$.

The length of labels, then, is bounded by the depth of nesting of modal operators: $length(\sigma) \leq dep(\varphi) + 1$. Let's call $dep(\varphi) = p$.

From the above, we get that for any branch, there can be $(2m)^p$ labels, and per label there can be $2m$ formulas, or $(2m)^{p+1}$ formulas per label (although in reality, this number will be smaller). Then there is the disjunctive rule, which may split a branch in two, which must be due to some formula in $Sub^+(\Phi)$, so the upper bound is 2^{2m}, which results in an upper bound for the size of the tree of $(2m)^p \times 2^{2m}$.

Since there is a finite upper bound on the size of the tree OOPS generates, we conclude that OOPS terminates. ⊠

Theorem 4.2. OOPS *is sound: if the* OOPS *pre-tableau for* φ *is open, then* φ *is* $S5_n$ *satisfiable.*

Proof. We construct from an open branch B, to which rules have been exhaustively applied, a tableau T with states that correspond to the elements of $l(B)$. We will prove that this tableau satisfies the properties given in definitions 3.3 and 3.4. These properties will be referred to as conditions (1) - (5). Since $\varphi \in L'(B, 0.\alpha_0)$, we then have that T is a $S5_n$ tableau for φ and according to theorem 3.5, φ is satisfiable.

Suppose B is a exhaustively expanded, open branch of the `OOPS` pretableau. Construct $T = (S, L, R_1, \ldots, R_n)$ as follows:

1. Let the set of states correspond to the labels on the branch: $S = l(B)$
2. Define the labeling function as follows: $L(s) = L'(B, s)$
3. Define the relations R_i': $(\sigma, \sigma.k_i) \in R_i'$ for all σ and k_i such that $\sigma \in S$ and $\sigma.k_i \in S$
4. The accessibility relations R_i are the reflexive, symmetric, transitive closures of the relations R_i' over S.

Condition 1 Because the propositional rules have been exhaustively applied, we can through definition 3.1 easily claim that for all $s \in S$, $L(s)$ is a propositional tableau, for if it were not, there would be another propositional rule that could be applied, or $L(s)$ is blatantly inconsistent, which violates our assumption that B is open.

Condition 2 Say $(s, t) \in R_i$, then either $s = t$, t is an i-descendant of s, s is an i-descendant of t or s and t are siblings. This follows, because if $\sigma.k_i.h_j \in S$, $i \neq j$ for any σ, k and h because of the conditions on the rules. Now, if $\Box_i\psi \in L(s)$, because of, respectively, $K^*_\Box$ or $T_\Box$, $K_\Box$, $R^*_\Box$ and $K^*_\Box$, $\psi \in L(t)$.

Condition 3 Suppose $\neg\Box_i\psi \in L(s)$, then following $M_\Box$ or $M^*_\Box$, if $s = \sigma.k_j$, then there is a $t_1 = \sigma.k_j.\lceil\neg\psi\rceil_i \in S$ where $\neg\psi \in L(t_1)$ if $i \neq j$ or a $t_2 = \sigma.\lceil\neg\psi\rceil_i \in S$, $\neg\psi \in L(t_2)$ if $i = j$.

Condition 4 If $\Box_i\psi \in L(s)$, because of $K^*_\Box$ or $T_\Box$, depending on the form of the label s, we get $\psi \in L(s)$.

Condition 5 Take $(s, t) \in R_i$. Now suppose that $\Box_i\psi \in L(s)$. Because of the same reasoning as for condition (2), for all $u \in R$, R all states that are i-descendants, i-predecessors, i-siblings of s and s itself, $\psi \in L(u)$. This set R is also the set of all worlds i-reachable from t. So if $\neg\psi$ holds in some world v i-reachable from t, it holds in a world i-reachable from s, but then both $\psi, \neg\psi \in L(v)$, violating the assumption that T is open.

Conversely, suppose $\neg\Box_i\psi \in L(s)$. Then by $M_\Box$ or $M^*_\Box$ there is a $u \in R$, such that $\neg\psi \in L(u)$. Immediately, because $t, u \in R$, we get

that ψ is not in $L(v)$ for all $v \in R$, because if it were in $L(u)$, that would violate the assumption that T is open. Thus, condition (5) is met, because the set R is the full set of worlds in the equivalence class under R_i that includes s and t.

This shows that T is an $S5_n$ tableau and since $\varphi \in L(0.\alpha_0)$, where $0.\alpha_0 \in S$, by theorem 3.5, φ is satisfiable. ⊠

In the following proof, 'inconsistent' is used in a somewhat more liberal way to mean 'inconsistent given the structure of the pre-tableau constructed thus far'. Thus, it is not the formula in itself that is inconsistent (see definition 2.9 for this sense of 'inconsistent'), but the occurence of the formula within the given context. The first time, this use of 'inconsistent' is indicated explicitly.

Theorem 4.3. `OOPS` *is complete: if the* `OOPS` *pre-tableau for* φ *is closed, then* φ *is not* $S5_n$ *satisfiable (*$\neg\varphi$ *is provable).*

Proof. This proof is by induction of the construction of the tree. We start at the leaves of an expanded tree and work up to the root, along the way showing that if the resultant of a rule leads to an inconsistency, then that inconsistency was present before the rule was applied. Basically this means that `OOPS` constructs the *weakest* model for a formula, in the sense that only those formulas are added to the branch that are required to be true according to the semantics.

Say $B = (B', \sigma\ \psi)$ is a finalized branch, then we have that $L'(B, \sigma)$ is blatantly inconsistent, meaning that we can't create a state σ in a Kripke structure (definition 2.5) that satisfies $\psi_\sigma = \bigwedge L'(B, \sigma)$.

Now suppose that B is not a finalized branch, but marked closed. Then by definition 3.18, every generated successor $B' = (B, \sigma\ \psi)$ is marked closed. By induction hypothesis, for every such B', we have a τ such that $\psi_\tau = \bigwedge L'(B', \tau)$ is inconsistent given the structure of the pre-tableau constructed thus far.

Say B has two successors: $B_1 = (B, \tau\ \neg\psi_1)$ and $B_2 = (B, \tau\ \neg\psi_2)$ generated from some node $\tau\ \neg(\psi_1 \wedge \psi_2)$ by the disjunctive rule. Say B_1 is inconsistent in a label σ_1 and B_2 is inconsistent in σ_2. Now if $\tau \neq \sigma_1$ (σ_2), since the disjunctive rule does not create any new labels, B is also inconsistent in σ_1 (σ_2). If both B_1 and B_2 are inconsistent due to τ, $L(B_i, \tau) = L(B, \tau) \cup \psi_i$, $i = 1, 2$. Say $\chi = \bigwedge L(B, \tau)$ and $\chi_i = \chi \wedge \psi_i$, $i = 1, 2$ then by propositional reasoning $\vdash (\neg\chi_1 \wedge \neg\chi_2) \rightarrow \neg\chi$ and by the induction hypothesis, we know that χ_1 and χ_2 are inconsistent, so $\vdash \neg\chi$.

Say B has one successor $B' = (B, \tau\ \psi_i)$, $i = 1, 2$ generated by the conjunctive rule (the conjunctive rule generates two new nodes, but these can be treated independently). Say B' is inconsistent due to $\sigma \neq \tau$, then B is inconsistent due to σ. If B' is inconsistent due to τ, then $\chi_i = \chi \wedge \psi_i$ is

inconsistent and by propositional reasoning, $\vdash \neg\chi_i \rightarrow \neg\chi$, so $\vdash \neg\chi$, so B is inconsistent due to τ. The case of the double negation rule is analogous to this case.

For the case of $M_{\Box} = \tau.k_i\ \neg\Box_j\psi\ /\ \tau.k_i.n_j\ \neg\psi$, say $L'((B, \tau.k_i.n_j\ \neg\psi), \sigma)$ inconsistent due to the induction hypothesis. Now if $\sigma \neq \tau.k_i.n_j$, then B inconsistent due to σ. If $\sigma = \tau.k_i.n_j$, then any world accessible from $\tau.k_i$ where $\neg\psi$ holds, will be inconsistent ($\tau.k_i.n_j$ is the minimal world where $\neg\psi$ holds), so $\tau.k_i\ \neg\Box_j\psi$ is not satisfiable, hence B is inconsistent in $\tau.k_i$. For $M^*_{\Box}$, a similar argument holds.

For all of the necessity rules, it is true that they only add formulas to a label that are forced to be true by the Kripke semantics, so if a branch B had successor B' that is inconsistent due to a label τ, B is also inconsistent due to τ.

Thus by induction, a blatant inconsistency in some label τ is propagated upwards to $0.\alpha_0$, where we then conclude that the branch $(\emptyset, 0.\alpha_0\ \varphi)$ is inconsistent, and so is φ. ⊠

The above proofs are somewhat more involved than those in Halpern and Moses (1992), because in that work, when states (worlds or labels here) are added, all relevant epistemic formulae are copied over from the previous state. Once a new state is created, there can be no propagation of formulas from that state to previously created states. These properties make e.g. the proofs of inconsistencies above simpler, since they do not require the notion of inconsistency within the context of the tableau created thus far. To prove that the same results are obtained by `OOPS`' algorithm requires a bit more work.

Theorem 4.4. *A formula φ is $S5_n$ satisfiable if and only if the* `OOPS` *pre-tableau construction for φ returns "the pre-tableau for φ is open".*

Proof. Theorem 4.1 shows us that for any formula φ, `OOPS` terminates. Furthermore, from theorem 4.2, if the pre-tableau is open, φ is satisfiable. Theorem 4.3 shows by contraposition that if φ is satisfiable, `OOPS` will find an open pre-tableau. ⊠

5 Implementation

This section describes the implementation of `OOPS`. First the core functionalities of matching rules against formulas and generating the pre-tableau are described. Then several peripheral features are described, namely, the parser employed by `OOPS`, the visualization of pre-tableaux and the generation of counter-models from the pre-tableaux.

`OOPS` is entirely implemented in Java[1] 1.5 (Gosling et al., 2005). I

[1] http://java.sun.com/

recommend Horstmann and Cornell (2004) as a reference to Java 1.5, or Horstmann and Cornell (2007), the updated edition for Java 1.6.

All classes are contained in the package `nl.rug.ai.mas.oops`, therefore, this prefix is omitted from package names. E.g., `.formula` is taken to mean `nl.rug.ai.mas.oops.formula`. However, e.g. `java.lang` (no leading dot) still means `java.lang`.

The language $\mathcal{L}_n^+(\Phi)$ (definition 3.10), used to represent rules and formulas, is implemented by the `Formula` class hierarchy in the package `.formula`. Several operations are defined on `Formula`, for example matching against other `Formula`s and substitution for variables (section 5.1) and it supports the Visitor pattern (Gamma et al., 1994) to allow the definition of additional operations without any changes to the `Formula` class hierarchy.

Labels in their extended sense (definition 3.12), are implemented by the `Label` hierarchy in the package `.tableau`. `Label`s also support the operations of matching and substitution as well as the Visitor pattern.

Further data types listed in table 3 are implemented in classes or hierarchies of classes of approximately the same name, all in package `.tableau`.

5.1 Matching and Substitution

The processes of substitution and matching are akin to substitution and unification in logic programming (Doets, 1994, definitions 3.23 and 3.62, respectively).

However, matching is simpler than unification, in that it is a asymmetric process. When a formula $\varphi \in \mathcal{L}_n^+(\Phi)$ (left) is matched to a formula $\psi \in \mathcal{L}_n^+(\Phi)$ (right), yielding a substitution θ (definition 3.14), θ has as it's domain a subset of the formulas occurring in φ. We say that two formulas match if there is such a substitution such that $\theta(\varphi) = \theta(\psi)$.

Both processes are implemented by the `Formula` class hierarchy. The process of matching is now described informally. Then, the implementation of substitution is briefly described. Note that both operations can easily be extended to labels and thus to nodes and rules.

Matching for concrete elements (non-variables) is done by comparing the top element of the parse tree to the top element of the parse tree of the right formula. If they are not equal, there is no match (return the `null` substitution). If they are equal, let the direct subformulas match against the direct subformulas of the right formula. If these yield non-`null` substitutions, that are compatible (do not assign different values to identical variables), return their union. Otherwise, return `null`. If there are no children, simply return a substitution with an empty domain. For formula variables, return a substitution of the variable concerned to the right formula. For agent variables, analogously.

Substitution is also performed by walking through the syntax tree. Every complex syntax element first calls its direct children, yielding new (possibly

identical) children. It then makes a copy of itself, with the new children as its children. In case of variables, their value according to the substitution is returned or, if it is not in the domain of the substitution, simply the variable itself.

5.2 Tableau algorithm

Although the implementation of `OOPS` is object oriented, the actual algorithm is contained within a single class, `Tableau` in package `.tableau`. Most of the additional classes are concerned with the representation of the many different data types involved, or provide utility functions. Therefore, the algorithm description will be based on an abstract version of these representations, to be presented shortly.

In this report, the rules are always as defined in table 2. The implementation of `Tableau`, however, does not include these rules verbatim. Instead, the rules must be specified when the `Tableau` class is instantiated.

The procedure TABLEAU0 (algorithm 5.1), called `Tableau.tableau()` in the Java code, is the entrance point for code calling into `OOPS`. As input it takes a formula for which a tableau is to be constructed. Its output is the final status of the generated pre-tableau: open or closed. All other procedures described below are hidden from external code.

The algorithms often refer to a data type (see table 3), for example 'State S', which should be interpreted as "S is of the form State, as defined in table 3". Furthermore, there are a number of predicates and functions defined on these data types (see table 4) for which no explicit algorithm is specified. In the definitions, a $\langle\rangle$-list represents a sequence of fixed length, a []-list a sequence of variable length and a {}-list a set.

Capitals are used for variables, lowercase or lower camel case (i.e. starting with a lowercase character and starting subsequent words with a capital, but no space; e.g. lowerCamelCase) for functions and predicates and small caps for procedures defined in one of the algorithms below. Occasionally bold face capitals are used to denote a sequence of which the elements will be explicitly dereferenced later.

The '**for all** $N \in \mathbf{N}$' construct, where $\mathbf{N}$ is a set or a sequence, is assumed to visit all elements in $\mathbf{N}$ exactly once. In the case that $\mathbf{N}$ is a sequence, the elements will be visited in order. Equality '$=$' and inequality '$\neq$' are defined as usual and $V \Leftarrow X$ where V is a variable name and X is an expression denotes assignment and should be interpreted as 'from now on, the name V will refer to the thing X refers to now'.

Algorithm 5.1 TABLEAU0(F)

Require: Formula $F \in \mathcal{L}_n(\Phi)$
Ensure: *Closed* if the pre-tableau for F is closed, *Open* otherwise
  return TABLEAU($\langle \emptyset, \emptyset, [], \mathit{Unknown} \rangle, \langle \emptyset.\alpha_\emptyset, F \rangle$)

Algorithm 5.2 TABLEAU(S, N)

Require: State S, Node N
Ensure: *Closed* if this sub-tableau is closed, *Open* otherwise
1: $S' \Leftarrow$ HANDLENODE(S, N)
2: **if** $r(S') \neq \mathit{Unknown}$ **then**
3: **return** $r(S')$
4: **end if**
5: **while** $\neg$ empty($q(S')$) **do**
6: $M \Leftarrow$ first($q(S')$)
7: $S' \Leftarrow \langle b(S'), n(S'), \text{rmFirst}(q(S')), r(S') \rangle$
8: **if** type(M) = *Split* **then**
9: $S' \Leftarrow$ HANDLESPLIT(S', nodes(M))
10: **else if** type(M) = *Linear* **then**
11: $S' \Leftarrow$ HANDLELINEAR(S', nodes(M))
12: **else if** type(M) = *Create* **then**
13: $S' \Leftarrow$ HANDLECREATE(S', nodes(M))
14: **else if** type(M) = *Access* **then**
15: $S' \Leftarrow$ HANDLELINEAR(S', nodes(M))
16: **end if**
17: **if** $r(S') \neq \mathit{Unknown}$ **then**
18: **return** $r(S')$
19: **end if**
20: **end while**
21: **return** *Open*

Algorithm 5.3 HANDLESPLIT($S, \mathbf{N}$)

Require: State S, Nodes $\mathbf{N} = [N_0, \ldots, N_k]$
Ensure: State S' that results from splitting the current sub-tableau into k new sub-tableaux, according to $\mathbf{N}$
1: **for all** $N \in \mathbf{N}$ **do**
2: $T \Leftarrow$ TABLEAU(newState(S), N)
3: **if** $T = \mathit{Open}$ **then**
4: **return** $\langle b(S), n(S), q(S), \mathit{Open} \rangle$
5: **end if**
6: **end for**
7: **return** $\langle b(S), n(S), q(S), \mathit{Closed} \rangle$

Formula	Structure representing a formula in $\mathcal{L}_n^+(\Phi)$ (definition 3.10)
Label	Structure representing a label (definition 3.12)
Rule	Represents a rule (as in table 2 and 3 of the soundness and completeness proof)
Node: $\langle L, F \rangle$	where Label L, Formula F
Match: $\langle R, \mathbf{N} \rangle$	where Rule R and $\mathbf{N} = [N_0, \dots N_k]$ are Nodes
Branch: $\{N_1, \dots N_k\}$	where $N_1, \dots N_k$ are Nodes
Necessities: $\{N_1, \dots N_k\}$	where $N_1, \dots N_k$ are Nodes with a label that contains free variables
Queue: $[M_1, \dots M_k]$	where $M_0, \dots M_k$ are Matches
Result	$\mathit{Open} \vert \mathit{Closed} \vert \mathit{Unknown}$
State: $\langle B, N, Q, R \rangle$	Branch B, Necessities N, Queue Q and Result R

Table 3: Definitions: data types

Algorithm 5.4 HANDLELINEAR$(S, \mathbf{N})$

Require: State S, Nodes $\mathbf{N} = [N_0, \dots, N_k]$
Ensure: State S' after adding all $N \in \mathbf{N}$ to the current sub-tableau

$S' \Leftarrow S$
for all $N \in \mathbf{N}$ **do**
  if $\neg$ contains$(b(S'), N)$ **then**
    $S' \Leftarrow$ HANDLENODE(S', N)
    if $r(S') = \mathit{Closed}$ **then**
      return S'
    end if
  end if
end for
return S'

- newState($\langle B, N, Q, R\rangle$) = $\langle B, N, Q, \mathit{Unknown}\rangle$
- $b(\langle B, N, Q, R\rangle) = B$
- $n(\langle B, N, Q, R\rangle) = N$
- $q(\langle B, N, Q, R\rangle) = Q$
- $r(\langle B, N, Q, R\rangle) = R$
- $l(\langle L, F\rangle) = L$
- $f(\langle L, F\rangle) = F$
- opposite(F) = G if $F = \neg G$, $\neg F$ otherwise
- rule($\langle R, \mathbf{N}\rangle$) = R
- type($\langle R, \mathbf{N}\rangle$) = typeOf($R$)
- nodes($\langle R, \mathbf{N}\rangle$) = $\mathbf{N}$
- empty(Q) = **true** $\Leftrightarrow Q = []$
- first($[M_1, \ldots, M_k]$) = M_1
- rmFirst($[M_1, M_2, \ldots, M_k]$) = $[M_2, \ldots, M_k]$
- add(X, Y) = $X \cup \{Y\}$
- addAll(X, Y) = $X \cup Y$
- matches(N) = $[M_0, \ldots, M_k]$ from matching all rules against Node N (see table 2 and 3 from the soundness and completeness proof)
- apply(N, L) = $[M_0, \ldots, M_k]$ from matching all Necessities N against Label L
- apply(M, B) = $[M_0, \ldots, M_k]$ from matching the incomplete Match M against the Labels on Branch B
- typeOf(R) for Rule R is *Linear* if R is a propositional conjunctive or negation rule, *Split* if R is a propositional disjunctive rule, *Create* if R is a modal possibility rule and *Access* if R is a modal necessity rule.

For sequences (notably, the Queue), the add and addAll operations do not necessarily preserve the order in the original sequence (in fact, the Queue is implemented as a priority queue). The rmFirst operation does preserve the order of the elements.

Table 4: Definitions: functions and predicates

Algorithm 5.5 HANDLECREATE$(S, \mathbf{N})$

Require: State S, Nodes $\mathbf{N} = [N_0, \ldots, N_k]$
Ensure: State S' after adding all $N \in \mathbf{N}$ to the current sub-tableau

$S' \Leftarrow S$
for all $N \in \mathbf{N}$ **do**
  if $\neg$ contains$(b(S'), N)$ **then**
    $S' \Leftarrow$ HANDLENODE(S', N)
    if $r(S') = \mathit{Closed}$ **then**
      return S'
    end if
    $Q \Leftarrow$ addAll$(q(S'),$ apply$(n(S'), l(N)))$
    $S' \Leftarrow \langle b(S'), n(S'), Q, r(S') \rangle$
  end if
end for
return S'

Algorithm 5.6 HANDLENODE(S, N)

Require: State S, Node $N \notin b(S)$
Ensure: State S' after adding node N to the current sub-tableau

if contains$(b(S),$ opposite$(N))$ **then**
  return $\langle$add$(b(S), N), n(S), q(S), \mathit{Closed}\rangle$
else
  return MATCHPUT(S, N)
end if

Algorithm 5.7 MATCHPUT(S, N)

Require: State S, Node $N \notin b(S)$, opposite$(N) \notin b(S)$
Ensure: State S' after adding node N to the current sub-tableau

$\mathbf{M} \Leftarrow$ matches(N)
for all $M \in \mathbf{M}$ **do**
  if type$(M) = \mathit{Access}$ **then**
    $Nec \Leftarrow$ addAll$(n(S),$ nodes$(M))$
    $Q \Leftarrow$ addAll$(q(S),$ apply$(M, b(S)))$
  else
    $Nec \Leftarrow n(S)$
    $Q \Leftarrow$ add$(q(S), M)$
  end if
  return $\langle$add$(b(S), N), Nec, Q, r(S)\rangle$
end for

5.3 Parser

The parser for `OOPS` was generated by SableCC[2] (Gagnon, 1998), a parser generator for Java that keeps a clean separation between machine-generated and user-written code. The grammar uses SableCC's ability to transform a Concrete Syntax Tree into an Abstract Syntax Tree to very easily generate the desired Syntax Tree.

This tree is then traversed once, in depth first fashion, to build a tree of the type `Formula` from the `.formula` package. Tables of propositions, agents, formula variables and agent variables are constructed at the same time. This also ensures that the same propositions and agents are represented by identical objects (in the sense of Java's `Object.equals`). These tables are maintained in a `Context`, which is may be passed along to other objects (e.g. the rule factories require a context).

The current parser is very minimal, in the sense that it only supports the language $\mathcal{L}_n^+(\Phi)$; there is no way to input multiple formulas, to assign values to variables or to use 'programming' constructs of any kind.

5.4 Visualization

The pre-tableaux generated by `OOPS` (and tableaux in general) have a convenient graphical notation that may aid the user in understanding how `OOPS` arrives at its conclusions and to verify that this is indeed correct. See figure 1 for an example and De Boer (2006) for further examples. Furthermore, it may help students to get acquainted with the workings of $S5_n$ and tableau methods.

Therefore, during the current project, `OOPS` was extended to include a module for the visualization of tableaux. First, several design considerations are discussed. Then the current implementation is described.

5.4.1 Considerations

The style of visualisation should resemble De Boer (2006), where a branch is made up of a series of lines, each consisting of a label, a formula, a justification of why the formula has been added to the branch and a line number. Line numbers are unique across the pre-tableau, so when a branch splits in two, both branches will not contain the same line numbers. Closure is indicated by a line containing no label or line number, $=$ instead of a formula and $(m \times n)$ as a justifications, where m and n are the line numbers where nodes demonstrating a blatant inconsistency may be found. An open and fully expanded branch is indicated by an $\uparrow$.

The following observations can be made regarding the rendering task:

[2] http://sablecc.org

1. Obviously, rendering of labels, formulas, justifications and line numbers resembles rendering of text and mathematical formulas.

2. A branch section that is not split has a structure that resembles a table.

3. The full pre-tableau can be viewed as a tree made up of a number of tables.

Furthermore, several possible rendering targets may be identified:

1. Direct to screen

2. Export to a file-format intended for rendering on screen

3. Direct to print

4. Export to a file-format intended for printing

In the case of OOPS, the primary goal is (1), while (2) is of secondary importance. Support for (3) or (4) does not need to be much different than (1) and (2) for simple cases, but for more elaborate trees issues such as paper size become a problem, and the tree may have to be simplified, or the user may want to 'configure' the tree to show only the relevant portions. These features are currently out of the scope of the project.

Several different output modes can be distinguished:

1. The full tree (all nodes expanded by `OOPS`) is displayed.

2. 'Redundant' nodes may be pruned (those nodes that have been expanded, but are not required for the closure of any branches), although in this case it is unclear what to do when a branch is open.

3. Display the full tree, but use a different color or font to highlight the 'critical nodes', those that figure directly or indirectly in the closure of the tree.

4. Only show an open branch.

5. Allow manual pruning of certain nodes or even whole sub-branches.

During the current project, (1) is the most relevant.

As a non-functional requirement, the visualisation of the pre-tableau may not aversely affect the space requirement for the proof search. That is: when the pre-tableau is not being visualised, the proof system should still be able to discard any branches that are closed as it does currently.

5.4.2 Current implementation

To support the visualization of the generated pre-tableau, yet adhere to the principles of separation of concerns and encapsulation, the `Tableau` class (section 5.2) was extended to support the Observer pattern (Gamma et al., 1994). Any class may subscribe to the `Tableau` in order to be notified of any changes to it. Several classes from the `.render` package make use of this facility to visualize the pre-tableau. Conveniently, the use of the Observer pattern allows the space requirements to remain the same when no listener is attached (that is, any additional space requirements are within the scope of the listener).

It turns out that Java has very little to offer in the way of useful classes that facilitate the rendering of tableaux. There is no standard way to render logical formulas, there is no easy way to draw tables onto a canvas (there are table layout elements, but these require a top-level window in order to render, so export to an image is not feasible) and there is no suitable tree layout algorithm available (in the standard library, or in the many open source libraries I've evaluated). The trouble with the trees required by tableau visualization is that the tables (nodes) may vary in size considerably. Most algorithms assume the nodes to be of approximately the same size. The current implementation addresses most of these problems, but not all, and not all of them uniformly.

The rendering of formulas is addressed in two ways. The first is the 'simple' way, through a `JLabel` component, using an HTML representation of the formula. Although this is all-right on screen, it does not allow flawless export to an image. The second way is quite involved and requires custom rendering using Java's font rendering capabilities. Although this second option is also implemented, it is not integrated with the actual rendering yet.

The layout of branches is now done using a simple box-layout container from Swing (the Java GUI library). This again is a problem for export to an image. No replacement is implemented at the moment, but it should be rather straightforward.

The layout of the tableau tree is done using a custom component and a custom layout algorithm, which is an implementation of the algorithm by (Miyadera et al., 1998). This implementation, however, will also need to be adjusted to allow flawless SVG export.

Since many fonts lack the characters used in the rendering of labels and formulas, the DejaVu Sans[3] font is included with `OOPS`.

For SVG export, the Apache Batik[4] open source class library is used. It allows for the unmodified use of existing rendering code to generate SVG images. However, there are some restrictions due to the architecture of

[3] http://dejavu.sourceforge.net/

[4] http://xmlgraphics.apache.org/batik/

Swing that complicates the rendering of Swing components directly to SVG. Therefore, some modifications are necessary (as mentioned above).

So although a working implementation of the visualization of `OOPS` pre-tableaux is available, it still leaves much to be desired. Especially export to SVG is currently crippled.

5.5 Counter-models

The generation of counter-models from the pre-tableau is supported through the Observer pattern, as is the visualization of pre-tableaux (section 5.4).

The `ModelConstructingObserver` in package `.model` listens to events from the `Tableau` class. It uses this information to keep track of the labels that are on the active branch. This not only entails adding new labels to the list, but also deleting labels when the tableau algorithm backtracks. It also derives the values assigned to primitive propositions per label (the valuations) by assigning 'true' to any proposition that occurs by itself on the branch or 'false' to any proposition whose negation occurs by itself.

When it receives an event that indicates the active branch is open (that is, exhaustively expanded and open), a model can be constructed from the stored list of labels and valuations. This is done by first creating a world for the root label and assigning the correct valuation to it. Then all direct sub-labels are added as worlds to the model, with an arrow for the correct agent from the root world. The algorithm goes on in that fashion until a world is created for every label. Then for every agent relation (set of arrows), the reflexive, symmetric, transitive closure is calculated. The constructed model is an $S5_n$ model of the formula at the root of the tableau (definition 2.7, theorem 2.14).

A preliminary implementation of the visualization of models is available, based on JGraphT[5] and the freely available version of JGraph[6]. Unfortunately, the open source version of JGraph does not include any graph layout algorithms, so automatic layout of the models is not possible. Valuations are also not displayed at the moment, but should be comparably easy.

5.6 Coding formulas as numbers

In section 3.1, a 1-1 encoding function $\ulcorner \cdot \urcorner : \mathcal{L}_n(\Phi) \mapsto \mathcal{N}$ is assumed to exist and used in the definition of rules. Such a function exists (and is often called a Gödel function) and is defined below. But in the implementation its behavior may also be mimicked in several ways.

One of these ways is not to calculate integers at all, but to keep the formulas themselves as part of the label. This is what was done in the initial implementation of `OOPS`.

[5] http://jgrapht.sourceforge.net/
[6] http://www.jgraph.com/

5.6.1 Coding formulas

In coding formulas into the natural numbers, I consider the language $\mathcal{L}_n^+(\Phi)$ (definition 3.10) which extends $\mathcal{L}_n(\Phi)$ in several ways that correspond to how `OOPS` represents formulas.

Definition 5.1. *For each of the sets A, Φ, V_a, V_f (see definition 3.10) define a function that assigns a unique number to each of it's elements: e_A, e_Φ, e_a and e_v respectively. Each of these functions is assumed to assign numbers densely, that is if $\exists_x e(x) = y$, and $y > 0$, then $\exists_x e(x) = y - 1$. Furthermore, e_o assigns a number to every operator as follows:*

- $e_o(\neg) := 0$
- $e_o(\wedge) := 1$
- $e_o(\vee) := 2$
- $e_o(\rightarrow) := 3$
- $e_o(\leftrightarrow) := 4$
- $e_o(\Box_i) := 5 + 4e_A(i)$, *where* $i \in A$
- $e_o(\Diamond_i) := 6 + 4e_A(i)$, *where* $i \in A$
- $e_o(\Box_a) := 7 + 4e_a(a)$, *where* $a \in V_a$
- $e_o(\Diamond_a) := 8 + 4e_a(a)$, *where* $a \in V_a$

Definition 5.2. *Define the code $e(x)$ of each element of the language as follows:*

- $e(p) = 1 + 3e_\Phi(p)$, *where* $p \in \Phi$
- $e(v) = 2 + 3e_v(v)$, *where* $v \in V_f$
- $e(o) = 3 + 3e_o(o)$, *where o is an operator.*

Definition 5.3. Sequences $a_1, \ldots, a_k$ *are coded (Buss, 1986, p. 7) by writing the a_i in binary expansion, separated by commas. Then write the string in reverse order and replace each* 0 *by* 10, *each* 1 *by* 11 *and each comma by* 00.

Note that if the a_i are are bounded from above by some constant (i.e. $\exists_c \forall_{a_i} c \geq a_i$) then the length of the sequence code is linear in the number of elements in the sequence.

Definition 5.4. *The encoding function $\ulcorner \cdot \urcorner : \mathcal{L}_n^+(\Phi) \mapsto \mathcal{N}$ is defined as follows. Write the formula φ in Polish notation, so it may be viewed as a sequence of language elements. Then use the scheme from definition 5.3 and the code of definition 5.2 to encode this sequence.*

Note that no formula is assigned the code 0. Therefore 0 may be used when a code for a formula would be expected, but is not available (e.g. α in definition 3.11).

Theorem 5.5. *The number of bits* $\log^{\ulcorner}\varphi^{\urcorner}$ *required to encode a formula* φ *is linear in* $|\varphi|$, $\log|A|$, $\log|\Phi|$, $\log|V_a|$ *and* $\log|V_f|$.

Proof. The codes defined in definitions 5.1 and 5.2 are clearly linear in the size of the sets concerned. Hence, the length of these codes is linear in the log of the size of the sets.

Furthermore, the encoding (definition 5.4) is simply a sequence of length $|\varphi|$, hence the length of the encoding as a binary string is linear in the length of the formula. ⊠

Corollary 5.6. *When the sets* A, Φ, V_a *and* V_f *are the least sets such that* $\varphi \in \mathcal{L}_n^+(\Phi)$, *then* $\log^{\ulcorner}\varphi^{\urcorner}$ *is polynomial in* $|\varphi|$. *In fact, it is* $\mathcal{O}(|\varphi|\log|\varphi|)$.

Proof. If the sets are minimal, then their size is bounded by $|\varphi|$, hence the length of the codes of language elements is $\mathcal{O}(\log|\varphi|)$. The code for φ is a sequence of length $|\varphi|$ of these elements and the code for a sequence is linear in it's length. Hence the code is of length in $\mathcal{O}(|\varphi|\log|\varphi|)$. ⊠

5.6.2 Coding formulas in `OOPS`

Although according to corollary 5.6 the code numbers assigned by $\ulcorner\cdot\urcorner$ are 'small', they rapidly grow larger than the 32 or 64 bit integers available on modern computers. Fortunately, the Java standard class library provides the `java.math.BigInteger` class that was designed to work with (very) large integers. This class is used to implement $\ulcorner\cdot\urcorner$ in `OOPS`.

In implementing the encoding of formulas, the main difficulty lies in the assignment of numbers to the primitives (the elements of Φ, A, V_a and V_f). If the objects representing the elements of these classes can each determine their own number, implementation of the code generation is straight forward.

If codes are to be a reliable means of comparison between formulas, it is not enough that the parser assigns a unique number to each instance of a primitive formula. This assignment of numbers must also be consistent across formulas so, e.g. the rule instantiations also use the same enumeration functions. For this purpose, the `Context` (section 5.3) allows these objects to use a consistent assignment of numbers to primitives.

The Visitor pattern (Gamma et al., 1994) is used to implement the class `CodingVisitor` in package `.formula`. Unfortunately, it currently implements a code based on the standard pairing function (Hájek and Pudlák, 1993, definition 1.18), which produces codes of exponential length. It just goes to show how important a good complexity analysis really is.

It should, however, be a trivial matter to implement the small codes described in the previous section. They may be constructed using a simple

depth first traversal of the syntax tree, hence their implementation in a Visitor is feasible.

6 Correctness

The aim of this section is to show that the implementation of the `OOPS` tableau algorithm is correct. That is, the `OOPS` implementation actually generates a pre-tableau as described in section 3.1 and the conclusion it draws corresponds to the closure rules as given there.

The proof of correctness will proceed by making more precise the demands that are made of the algorithm. Then it will be shown that the implementation of `OOPS` actually conforms to these definitions.

For definitions of some of the terms used in this section, see section 3, especially definitions 3.7, 3.8, 3.18 and 3.19. The notations from these definitions and notations introduced for the sake of the pseudo-code (section 5.2) are sometimes used interchangeably in what follows, hopefully without causing too much confusion.

There are two issues to consider when concerned with the modal rules:

- More than one rule may apply to the same node;
- The label resulting from a match may itself match several labels already on the branch, or that will appear on the branch in the future (in the case of necessity rules).

In the soundness and completeness proof, it is assumed that all rules are eventually applied exhaustively, so there the above issues are not problematic.

Therefore, in order to show correctness of the algorithm, three things need to be shown:

- Any branch $B' = (B, \gamma)$ that is generated is justified by the application of a rule, or $B' = B_0$ (the root);
- If a branch B is marked as *Closed*, then the branch is closed according to definition 3.18;
- If a branch B is marked as *Open*, then all rules have been exhaustively applied and the branch is not closed according to definition 3.18.

Note that if these conditions are shown to hold, then Tableau0 trivially implements definition 3.19, since then Tableau generates all branches that may be generated by the valid application of rules, starting from only the root. It returns *Open* if and only if one of these branches is *Open*.

Theorem 6.1. *Any branch $B' = (B, \gamma)$ that is generated is justified by the application of a rule, or $B' = B_0$ (the root).*

Proof. As can be seen in algorithm 5.1, the construction of any tableau starts out with an empty branch $\emptyset$. After entering TABLEAU (algorithm 5.2), the branch is expanded to $(\emptyset, \emptyset.\alpha_0\ \varphi) = B_0$. Afterwards, any node that is added to the branch is the result of a match that has been placed on the queue. Therefore, it must be shown that any match that enters the queue is justified by a rule (remember that a match is just a sequence of nodes, together with some information on how to create new branches with them, therefore if a match is justified, so are the nodes it contains).

There are two procedures that add nodes to the queue. The first is HANDLECREATE (algorithm 5.5), the other MATCHPUT (algorithm 5.7).

HANDLECREATE applies all necessities to a label that has just entered the branch (a match is generated if the label of the necessity matches the new label, this match will contain one node, consisting of the new label and the formula of the necessity). So, if these necessities are the result of the sound application of rules then so is the new match.

MATCHPUT adds matches to the queue in two ways. First of all, it applies all rules to a certain node γ, adding the resulting matches to the queue. Therefore, assuming that the matching process is correct, these matches are justified. In addition, if a match has the type *Access*, it attempts to apply all nodes (in essence, these are necessities) of the match to all labels currently on the branch. If the matching process is correct, the resulting matches will contain only nodes that can be generated by directly applying one of the necessity rules to the given node and substituting for the variables that remain free in the resultant label. So, these matches are justified also.

It must be shown that all necessities are justified by the sound application of rules. Necessities are only added by MATCHPUT. As is argued above, the nodes that are added to the set of necessities are the result of the application of necessity rules and applying them to any label will result in matches that could have been the result of a direct application of the given rule.

Now, it only remains to be shown that from matches, new branches are generated in the appropriate way. For linear propositional rules (algorithm 5.4), modal possibility rules (algorithm 5.5) and modal necessity rules (algorithm 5.4), nodes are added to the branch in a linear fashion as is dictated by the rule definitions. For the disjunctive propositional rules (algorithm 5.3), for every node γ in a match a new branch (B, γ) is created, also in accordance with the rule definition. For each of these generated branches, TABLEAU (algorithm 5.2) is applied with a local copy of the entire state. Because of this, matches generated on an (extension of) branch (B, γ) will not enter the queue for another branch (B, γ'). ⊠

Theorem 6.2. *If a branch B is marked as* Closed*, then the branch is closed according to definition 3.18*

Proof. The state may be set to *Closed* in two places, namely, HANDLESPLIT

(algorithm 5.3) and HANDLENODE (algorithm 5.6).

HANDLENODE sets the state to *Closed* if the branch B contains a node N and opposite(N). From the definitions (table 4), it must be the case that for some formula F and label L, $\langle L, F\rangle \in B$ and $\langle L, \neg F\rangle \in B$. This conforms to the first item in definition 3.18.

HANDLESPLIT generates a number of branches $B' = (B, \gamma)$ and if one of these is *Open*, then it sets the state B to *Open*, otherwise it sets it to *Closed*. So, if all branches of the form $B' = (B, \gamma)$ are closed, so is B. This corresponds directly to the second item of definition 3.18. ⊠

Theorem 6.3. *If a branch B is marked as* Open*, then all rules have been exhaustively applied and the branch is not closed according to definition 3.18.*

Proof. Here the term 'open branch' will be used liberally to refer to any branch that is not yet marked as *Closed* or *Open*. If a branch that is *marked* as *Open* is meant, this will be said explicitly.

First of all, whenever a node is added to an open branch (this is only done by MATCHPUT, algorithm 5.7), all rules are applied to it and the resulting matches are added to the queue and (when applicable) to the set of necessities.

Secondly, whenever a node is added to the list of necessities (again in MATCHPUT), it is immediately applied to all labels on the branch and the resulting matches are placed on the queue.

Third, whenever a new label is introduced on the branch (in HANDLECREATE, algorithm 5.5), all currently known necessities are applied to it and the resulting matches are added to the queue.

Together, the above three points ensure that for any node, all applicable rules are matched and that, if the resulting matches may match to more than one label (that may not yet have been introduced), they will be matched to all those labels through the use of the set of necessities. It remains to be shown that all matches on the queue are eventually executed for all subbranches of the current branch.

It has already been noted that in HANDLESPLIT (algorithm 5.3), the entire local state is copied over for each subbranch, so the entire queue and all necessities are also copied for all subbranches. If one of these $B' = (B, \gamma)$ is marked as *Open*, so is the current branch B. Since B' is marked as open, there must be some fully expanded subbranch of B' that is not closed (here I assume that TABLEAU is correct for B' to show that by induction it follows that the implementation of HANDLESPLIT is correct).

From TABLEAU (algorithm 5.2) we see that matches are removed from the queue one by one and executed appropriately. Unless HANDLESPLIT has marked the branch as *Open*, or it has been marked as *Closed* at some other point, TABLEAU will continue execution until the queue is entirely empty. In the case that the tableau has been marked as *Open* by HANDLESPLIT,

another recursive invocation of TABLEAU has exhaustively expanded a subbranch of the current branch. In the case that it has been marked as *Closed*, the theorem holds by default. In all other cases, because it is shown above that all possible applications of the rules eventually enter the queue, and all items on the queue have been processed, the tableau must have been exhaustively expanded. And it can't be closed (per definition 3.18), otherwise it would have been marked as such. ⊠

7 Complexity

Satisfiability of $S5_n$ is known to be PSPACE-complete (Halpern and Moses, 1992). Therefore, `OOPS` should also use polynomial space. Besides showing if this is the case, there is an interest in finding as tight as possible upper bounds for the space as well as the time complexity of `OOPS`.

Roughly speaking, if we view the tableau algorithm as performing a depth-first search of the tableau space, the space complexity will be proportional to the depth of the search tree and the time complexity proportional to the size of the entire tree. Since the branching factor of this tree in general is larger than one, even if the space complexity is polynomial in the size of the formula, the time complexity will be exponential.

7.1 Space complexity

I consider the space used by `OOPS` in terms of the number of nodes used. As is known from theorem 4.1, the size of any node will be polynomial in the length of the input formula $|\varphi|$.

Theorem 7.1. *The total space used at any time will be bounded by a polynomial of the space used by the largest branch when it is exhaustively expanded.*

Proof. Call the size of the exhaustively expanded branch x. `OOPS` keeps the entire branch as expanded thus far in memory. The space taken for this is at most x.

All elements on the queue are the result of the application of rules to elements already on the branch. Propositional and possibility rules add only a constant number of elements to the queue, necessity rules add only as many as there are labels on the branch. Therefore, if x is the size of the exhaustively expanded branch, $8x^2 + 22x$ (8 necessity rules, 11 other rules) is a very pessimistic upper bound on the space requirements for the queue.

For the number of necessities, a pessimistic upper bound is $8x$.

For the number of times the tableau may split (on a single branch), a pessimistic estimate is x, thus the queue is copied at most x times, raising the space estimate to $\mathcal{O}(x^3)$, a polynomial in the size of an exhaustively expanded branch. ⊠

Theorem 7.2. *OOPS is in* EXPSPACE

Proof. From theorem 4.1, we have for every label:

- The number of formulas is bound by some polynomial;
- The length of the label is bound by some polynomial;
- The number of descendants is bound by some polynomial

Notice that this describes a tree with a branching factor > 1 and hence an exponential growth. In theorem 4.1 this is bound to $(2|\varphi|)^{\mathrm{dep}(\varphi)+1}$ nodes.

One might hope that the branching of the (label) tree is counter-acted by a steady decrease of the number of formulas on each label. However, that is not the case here. ⊠

This begs the question why OOPS is in EXPSPACE, when e.g. Halpern and Moses (1992) clearly offer an algorithm that is PSPACE. The answer is that Halpern and Moses (1992) apply their rules in such a way that whenever a new label is generated (a state in their terms), the tree branches. By this method, the depth of the tree is polynomial in the size of the formula (hence, only the time complexity suffers from the exponential number of labels).

8 Performance

Much has been said above about the theoretical properties of OOPS. But in order to know what good it is in the 'real world', it is necessary to do an empirical analysis of its performance. Therefore, in this document, three different parameter settings for OOPS are compared to each other and to a second system, MOLtap[7], a prover based on sequent calculus that has been developed during another student project[8].

The experiments performed are meant to be exploratory in nature: they point to important qualitative differences in performance, without giving much statistical analysis and without the claim of statistical significance. Only a few weeks were available and a more thorough analysis would have taken many months of computer time. Therefore, these results are to be taken with a grain of salt, but still provide valuable clues as to the properties of the systems under investigation.

8.1 Method

An implementation of the $\mathrm{CNF}_{\Box_m}$ random formula generator (Patel-Schneider and Sebastiani, 2003) was used to generate a number of test sets containing

[7] Currently available at http://twan.home.fmf.nl/moltap/
[8] Soon to appear on http://www.ai.rug.nl/mas/

	OOPS0	OOPS1	OOPS2
Linear(1)	1	1	1
Linear($n > 1$)	2	2	2
Split	5	3	4
Access	3	4	3
Create	4	5	5

Table 5: Heuristic settings used for OOPS

100 formulas each, with different properties. The random generator allows for fine-grained control of several properties of the generated formulas: depth (d), number of agents (m), number of propositions (N), probability distribution of propositional to modal clauses at each depth (p), the number of clauses at the top level (L) and a distribution of clause size at each depth (C). All parameters, except for p and C, are integer values.

The provers check each of the formulas in the test set for satisfiability, the time they take is recorded, as well as the result. A time limit of 60 seconds per formula was imposed in order to make significant exploration of the parameters possible (Patel-Schneider and Sebastiani (2003) use a time limit of 1000 seconds). Since this clips a (potentially) long runtime to the maximum, the average runtime over a test set is not very meaningful. Therefore, as in Patel-Schneider and Sebastiani (2003), the median and 90th percentile values are calculated. In addition, the number of successfully completed runs is counted.

Most parameter settings were chosen to correspond to those in Patel-Schneider and Sebastiani (2003). However, they set the number of agents to one, as that 'produces the hardest formulae'. For $S5_n$, this is clearly not true (at least in theory), since for $S5_1$ the satisfiability problem is in NP, not PSPACE (Halpern and Moses, 1992). Therefore, several test sets with a different number of agents were created.

The different heuristic settings used for OOPS are listed in table 5.

8.2 Results

The results are divided into two sections. The first gives data that was obtained in order to compare several heuristics in their performance on different test sets. The second aims to observe how well OOPS responds to an increasing number of agents.

8.2.1 Comparison of heuristics

Table 6 shows the results of an experiment that was chosen, somewhat arbitrarily, to represent a 'typical' input to OOPS, exploiting all features of multi-agent formulas (nesting of modal operators, mutliple agents, multiple

	Set	MOLtap	OOPS0	OOPS1	OOPS2
Median values:	00	0.09	0.30	0.34	0.34
90th percentile values:	00	0.12	0.54	0.36	0.37
Number completed:	00	100	99	100	100

Table 6: $d = 2$, $N = 5$, $m = 3$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 3, 3, 0]]]$, $L = 10$

	Set	L	MOLtap	OOPS0	OOPS1	OOPS2
Median values:	01	3	0.06	0.24	0.24	0.24
	02	15	0.14	60.00	60.00	60.00
90th percentile values:	01	3	0.08	0.29	0.29	0.27
	02	15	60.00	60.00	60.00	60.00
Number completed:	01	3	100	100	100	100
	02	15	71	36	43	45

Table 7: $d = 2$, $N = 3$, $m = 1$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 3, 3, 0]]]$

propositions). The results, however, are not very conclusive.

The other experiments presented in this section were chosen to correspond to parameter settings used by Patel-Schneider and Sebastiani (2003) for the generation of their figures. Table 7 corresponds to figure 6, table 8 to figure 8, table 9 to figure 10, table 10 to figure 2 and table 11 to figure 3.

8.2.2 Increasing the number of agents

In this section, results are presented of several experiments that investigate wether increasing the number of agents has an influence on the difficulty of showing their satisfiability. Tables 12, 13 and 14 show the numbers.

8.3 Conclusion

In all cases, OOPSvariant 0 performs worse that the other two variants, as was expected, since it will create labels sooner than it will create branches due to propositional rules (the exponential increase in the number of labels is the cause of OOPS being in EXPSPACE, hence this choice should be suboptimal. Tables 10 and 11 are the only experiments to show a clear difference between variants 1 and 2, with 1 being the winner. Most other tables show a very

	Set	MOLtap	OOPS0	OOPS1	OOPS2
Median values:	06	0.11	60.00	0.40	0.40
90th percentile values:	06	60.00	60.00	60.00	60.00
Number completed:	06	81	37	71	69

Table 8: $d = 2$, $N = 3$, $m = 1$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 1, 4, 0]]]$, $L = 15$

	Set	L	MOLtap	OOPS0	OOPS1	OOPS2
Median values:	08	9	0.10	0.33	0.31	0.31
	07	15	60.00	60.00	60.00	60.00
90th percentile values:	08	9	14.40	60.00	60.00	60.00
	07	15	60.00	60.00	60.00	60.00
Number completed:	08	9	91	78	82	82
	07	15	46	9	36	34

Table 9: $d = 2$, $N = 3$, $m = 1$, $C = [[0, 2, 1]]$, $p = [[[], [0, 3, 0], [0, 3, 3, 0]]]$

	Set	L	MOLtap	OOPS0	OOPS1	OOPS2
Median values:	09	15	0.11	0.88	0.37	0.40
	11	30	60.00	60.00	60.00	60.00
90th percentile values:	09	15	1.27	60.00	2.47	7.05
	11	30	60.00	60.00	60.00	60.00
Number completed:	09	15	99	84	98	98
	11	30	48	9	44	30

Table 10: $d = 1$, $N = 3$, $m = 1$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 3, 3, 0]]]$

	Set	L	MOLtap	OOPS0	OOPS1	OOPS2
Median values:	10	15	0.13	1.59	0.41	0.42
	12	30	60.00	60.00	60.00	60.00
90th percentile values:	10	15	28.17	60.00	2.43	16.70
	12	30	60.00	60.00	60.00	60.00
Number completed:	10	15	93	70	100	95
	12	30	23	7	34	21

Table 11: $d = 1$, $N = 3$, $m = 1$, $C = [[0, 0, 1]]$, $p = [[[], [], [1, 0, 0, 0]]]$

	Set	m	MOLtap	OOPS1	OOPS2
Median values:	01	1	0.06	0.24	0.24
	17	2	0.07	0.25	0.26
	18	3	0.07	0.26	0.25
	19	5	0.07	0.27	0.26
	20	10	0.07	0.26	0.25
90th percentile values:	01	1	0.08	0.29	0.27
	17	2	0.10	0.30	0.30
	18	3	0.09	0.28	0.28
	19	5	0.09	0.30	0.30
	20	10	0.10	0.30	0.30
Number completed:	01	1	100	100	100
	17	2	100	100	100
	18	3	100	100	100
	19	5	100	100	100
	20	10	100	100	100

Table 12: $d = 2$, $N = 3$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 3, 3, 0]]]$, $L = 3$

	Set	m	MOLtap	OOPS1	OOPS2
Median values:	21	1	0.10	0.32	0.33
	22	2	0.10	0.31	0.33
	23	3	0.10	0.32	0.33
	24	5	0.10	0.31	0.33
	25	10	0.10	0.31	0.33
90th percentile values:	21	1	0.13	43.77	29.07
	22	2	0.13	0.38	0.47
	23	3	0.14	0.36	0.46
	24	5	0.13	0.35	0.38
	25	10	0.13	0.35	0.40
Number completed:	21	1	100	90	91
	22	2	100	96	95
	23	3	100	99	99
	24	5	100	100	100
	25	10	100	100	100

Table 13: $d = 2$, $N = 3$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 1, 4, 0]]]$, $L = 9$

	Set	m	MOLtap	OOPS1	OOPS2
Median values:	09	1	0.11	0.37	0.40
	26	2	0.12	0.36	0.40
	27	3	0.11	0.35	0.38
	28	5	0.12	0.38	0.40
	29	10	0.13	0.36	0.40
90th percentile values:	09	1	1.27	2.47	7.05
	26	2	0.15	1.42	19.00
	27	3	0.15	2.77	3.71
	28	5	0.15	33.04	15.17
	29	10	0.16	4.98	12.80
Number completed:	09	1	99	98	98
	26	2	100	97	94
	27	3	100	96	98
	28	5	100	94	97
	29	10	100	96	96

Table 14: $d = 1$, $N = 3$, $C = [[0, 0, 1]]$, $p = [[[], [], [0, 3, 3, 0]]]$, $L = 15$

slight advantage for variant 2.

Most tables show that MOLtap outperforms OOPS(variants 1 and 2) by about a factor 4. The exception in this case is table 11, where OOPS's 90th percentile times are much lower than for MOLtap.

For MOLtap, changing the number of agents does not seem to significantly impact the runtime. However, for OOPS, the results are less conclusive. Table 12 seems to suggest there is no difference, table 13 clearly shows a decrease in runtime for more than one agent compared to one agent. Table 14 shows that the runtime may also increase when the number of agents increases from one to more than one.

Although clearly OOPS is not optimal (as is to be expected given the complexity results of section 7) and MOLtap clearly outperforms OOPS in most cases, there does not seem to be a real qualitative difference in performance between the two systems, based on these tests.

9 Discussion

OOPS has been successful in many of its aims. It is a complete Java implementation of a tableau proof system for $S5_n$. It is available under an open source license and only uses components that are likewise available under an open source license. A Java implementation supports the goals of portability, as a runtime is available on many systems and of being extendible during future projects, as Java is a widely known language. The open license also supports the latter goal.

Also, the current project addresses several challenges for the design of OOPS, including the tableau visualisation and the generation of counter models. Furthermore, soundness and completeness of the tableau method have been proved and the algorithm has been shown to be a correct implementation of the tableau method. Thus, OOPS may reasonably be trusted to provide accurate answers.

Furthermore, the system documentation has been extended, both in this document and in the code documentation (which conforms to the widely adopted Javadoc standard for Java code documentation).

However, as section 7 shows, OOPS is in EXPSPACE and thus in EXPTIME (Papadimitriou, 1994). Halpern and Moses (1992) show that satisfiability for $S5_n$ is in PSPACE and so the algorithm for OOPS is problematic. My hope is that this problem may still be remedied without sacrificing too much of the appeal of the tableau system used by OOPS.

10 Further work

As a proof tool, OOPS' use is limited, currently. Although it has the facilities needed to prove and disprove logical formulas, there is no convenient way to work with systems of formulas or to build theories, as, for example, the LWB provides.

As further work, implementing an algorithm or set of rules that brings the complexity of OOPS back into PSPACE seems very important. Then, at least, OOPS should not run out of memory as quickly as it does currently and so provide a useful answer on a larger set of problems, at least eventually.

Then, there are still a number of open issues with the implementation. This includes several proposed optimizations (that may or may not be relevant depending on whether the first point above is addressed) and work on the visualization module. Additionally, the code described in section 5.6.1 should supplant the current coding implementation.

Last, but not least, perhaps even more important than reducing the complexity class, is the implementation of a useful user interface for OOPS. OOPS should help the user both understand and work with $S5_n$, instead of just providing 'yes' and 'no' answers to queries that are very inconveniently formulated.

References

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

Buss, S. R. (1986). *Bounded Arithmetic.* Studies in proof theory, Lecture notes. Bibliopolis.

de Boer, M. S. (2006). Praktische bewijzen in public announcement logica. Master's thesis, University of Groningen.

Doets, K. (1994). *From logic to logic programming.* Foundations of computing. MIT Press.

Fitting, M. and Mendelsohn, R. (1999). *First-Order Modal Logic*, volume 277 of *Synthese Library.* Kluwer Academic Publishers.

Gagnon, É. (1998). Sablecc, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional.

Gosling, J., Joy, B., Steele, G. L., and Bracha, G. (2005). *The Java Language Specification.* The Java Series. Prentice Hall PTR, third edition.

Hájek, P. and Pudlák, P. (1993). *Metamathematics of first-order arithmetic.* Perspectives in mathematical logic. Springer-Verlag.

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

Heuerding, A., Jäger, G., Schwendimann, S., and Seyfried, M. (1996). The logics workbench LWB: A snapshot. *Euromath Bulletin*, 2(1):177–186.

Horstmann, C. S. and Cornell, G. (2004). *Core Java(TM) 2, Volume I – Fundamentals.* Core Series. Prentice Hall PTR, 7th edition.

Horstmann, C. S. and Cornell, G. (2007). *Core Java(TM) 2, Volume I – Fundamentals.* Object Technology. Prentice Hall PTR, 8th edition.

Miyadera, Y., Anzai, K., Unno, H., and Yaku, T. (1998). Depth-first layout algorithm for trees. *Information Processing Letters*, 66:187–194.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison Wesley, 1st edition edition.

Patel-Schneider, P. F. and Sebastiani, R. (2003). New general method to generate random modal formulae for testing decision procedures. *J. Artificial Intelligence Research*, 18:351–389.

van der Hoek, W. and Meyer, J.-J. C. (2004). *Epistemic Logic for AI and Computer Science*. Cambridge University Press.

van der Vaart, E. and van Valkenhoef, G. (2007). OOPS: An automated proof tool for $\mathbf{S5}_{(n)}$. http://www.ai.rug.nl/~valkenhoef/oops/report.pdf.