The Short User's Guide to OOPS

Elske van der Vaart Gert van Valkenhoef

February 25, 2008

1 Introduction

OOPS is an *O*bject *O*riented *P*rover for $S\mathbf{5}_{(n)}$, which can automatically prove or disprove formulas of epistemic logic, using the $\mathbf{S5}_{(n)}$ axioms. It can be run from the commandline, and will correctly return `true` or `false` in response to an $\mathbf{S5}_{(n)}$ input formula.

OOPS was developed as a final project for the course Multi-Agent Systems at the University of Groningen, in May 2007. The resulting report, *OOPS: An Automated Proof Tool for* $S\mathbf{5}_{(n)}$, details OOPS' technical specifications, as well as its underlying proof method.[1]

This paper, by contrast, presents a Short User's Guide to OOPS, to introduce new users to its operation. Section 2 documents how to install and run OOPS, while Section 3 describes its input options. Finally, Section 4 demonstrates a brief example of OOPS at work.

2 Installing and Running OOPS

OOPS is distributed as `oops.jar`[1], and will work for users who have the Sun Java Runtime Environment (JRE)[2] installed, version 1.5 or higher. OOPS can then be run by typing the following command, where <formula> is the epistemic expression to be evaluated:

```
$ java -jar oops.jar '<formula>'
```

If this command is not run from the folder where `oops.jar` is located, the `oops.jar` argument should be preceded by a correct path specification, such as `~/oops/oops.jar`. OOPS should then return `true` or `false` as appropriate, or an error message in response to an incorrectly formatted formula.

[1] At time of writing, available from http://www.ai.rug.nl/~valkenhoef/oops.
[2] Available from http://java.sun.com.

3 Inputting Formulas to OOPS

OOPS should be able to handle any $S\mathbf{5}_{(n)}$ formula as input, provided certain formatting restrictions are upheld. First, every logical operator has an OOPS equivalent, to facilitate easy access by computer keyboard. These are shown in Table 3.1.

Table 3.1 *Logical Operators in* OOPS

Logical Operator	OOPS Symbol
$\neg$	~
$\wedge$	&
$\vee$	|
$\rightarrow$	>
$\leftrightarrow$	=
K_i	#_i
M_i	%_i

OOPS also expects propositions and agents to be of a specific form. Propositions may be represented by any combination of letters and numbers, but the first symbol must be a lowercase letter. Agents i must be denoted by a single natural number, which may take any reasonable value from 0 upwards.

Operator precedences are specified in Table 3.2, but may be circumvented using parentheses. As an illustration: `#_i p & q` would be evaluated as $(K_i p) \wedge q$, while `#_i (p & q)` would be interpreted as $K_i(p \wedge q)$. Three example runs of OOPS are presented in Example 3.1.

Table 3.2 OOPS *Input Details*

Precedence	Logical Operator(s)
1	$\neg$, K_i, M_i
2	$\wedge$
3	$\vee$
4	$\rightarrow$, $\leftrightarrow$

Example 3.1 *Example Runs of* OOPS.

A run demonstrating a test of $\neg M_1(\neg a \wedge a)$:

```
$ java -jar oops.jar '~%_1(~a & a)'
       true
```

A run demonstrating a test of $K_1K_2(\neg propNr1 \wedge propNr2)$:

```
$ java -jar oops.jar '#_1 #_2(~propNr1 & propNr2)'
       false
```

A run demonstrating a test of $K_{Alice}(P \rightarrow Q)$:

```
$ java -jar oops.jar '#_Alice (P > Q)'
       nl.rug.ai.mas.oops.TableauErrorException:  Could not parse formula
```

4 A Detailed Example of OOPS at Work

One possible application of OOPS is to model small logic games. Consider an extremely simplified version of Cluedo, which goes as follows[3]. In total, there are three cards of different colors, and two players are each given one. The third card is placed face down on the table.

Object of the game is to establish the color of the face down card, which can be done by asking the other player a question. Let's say that the cards are red, yellow and black, and that this is common knowledge. Then, if player 1 has the red card, she can ask player 2 if he has the yellow card.

Should player 2 say "yes", then player 1 knows that the black card must be face down on the table, as she herself holds the red card; should player 2 say "no", then player 1 knows that the yellow card must be face down on the table, as player 2 must hold the black card.

Although this game is not particularly challenging - it should always be over after the first round - it makes for an interesting demonstration of OOPS. Given the right background knowledge, OOPS can derive what each of the players knows about the current game state.

First, the relevant propositions must be defined. Let `r1` mean that 'the red card is held by player 1', `r2` that 'the red card is held by player 2', and `rt` that 'the red card is face down on the table'. Then let `y1`, `y2` and `yt`, and `b1`, `b2` and `bt`, mean the same for the yellow and black cards, respectively.

Then, there are four pieces of information that the players may consider self-evident, but that OOPS must be explicitly told. A first important fact is that (1) every card is somewhere. This means that *either* the red card is in player 1's hand *or* it is in player 2's hand *or* it is face down on the table.

This, of course, applies to the yellow and black cards as well. In OOPS, *'every card is somewhere'* can be modelled as follows:

```
(rt | r1 | r2) & (yt | y1 | y2) & (bt | b1 | b2)
```

Second, (2) every card is in only one place. *If* the red card is on the table, *then* in cannot be in player 1's hand, nor can it be in player 2's hand. This holds for all other positions and colors as well. In OOPS, *'every card is in only one place'* can be modelled as follows:

[3]This Cluedo derivative was first formulated by Hans van Ditmarsch, for use with the Logics Workbench (http://www.lwb.unibe.ch), a different automatic proof tool.

```
(rt > ∼r1 & ∼r2) & (r1 > ∼r2 & ∼rt) & (r2 > ∼r1 & ∼rt) &
(yt > ∼y1 & ∼y2) & (y1 > ∼y2 & ∼yt) & (y2 > ∼y1 & ∼yt) &
(bt > ∼b1 & ∼b2) & (b1 > ∼b2 & ∼bt) & (b2 > ∼b1 & ∼bt)
```

By joining these four facts with conjunctions, OOPS can be supplied with the same background knowledge that the players have, after which it can deduce what they do and don't know. Let's assume that player 1's card is red, but that she has not yet asked the other player about his card.

Then, OOPS will correctly derive that she does not yet know what color the card on the table is, or, in other words, that K_1 r1 $\rightarrow$ (K_1 yt $\vee$ K_1 bt) is false. Although she knows that her own card is red, she does not yet know whether the face down card is yellow or black. In OOPS:

```
$ java -jar oops.jar '#_1(
((rt | r1 | r2) & (yt | y1 | y2) & (bt | b1 | b2)) &
((rt > ∼r1 & ∼r2) & (r1 > ∼r2 & ∼rt) & (r2 > ∼r1 & ∼rt) &
(yt > ∼y1 & ∼y2) & (y1 > ∼y2 & ∼yt) & (y2 > ∼y1 & ∼yt) &
(bt > ∼b1 & ∼b2) & (b1 > ∼b2 & ∼bt) & (b2 > ∼b1 & ∼bt)) &
((rt | yt | bt) & (r1 | y1 | b1) & (r2 | y2 | b2)) &
((rt > ∼yt & ∼bt) & (r1 > ∼y1 & ∼b1) & (r2 > ∼y2 & ∼b2) &
(yt > ∼rt & ∼bt) & (y1 > ∼r1 & ∼b1) & (y2 > ∼r2 & ∼b2) &
(bt > ∼yt & ∼rt) & (b1 > ∼y1 & ∼r1) & (b2 > ∼y2 & ∼r2))) >
(#_1 r1 > (#_1 yt | #_1 bt))'
         false
```

Note that it is important to inform OOPS that player 1 has all the background knowledge previously specified, or OOPS will draw the wrong conclusions. As another example, it can, for instance, confirm that player 1 *does* know that the card face down on the table is either yellow or black. In OOPS:

```
$ java -jar oops.jar ‘#_1(
((rt | r1 | r2) & (yt | y1 | y2) & (bt | b1 | b2)) &
((rt > ∼r1 & ∼r2) & (r1 > ∼r2 & ∼rt) & (r2 > ∼r1 & ∼rt) &
(yt > ∼y1 & ∼y2) & (y1 > ∼y2 & ∼yt) & (y2 > ∼y1 & ∼yt) &
(bt > ∼b1 & ∼b2) & (b1 > ∼b2 & ∼bt) & (b2 > ∼b1 & ∼bt)) &
((rt | yt | bt) & (r1 | y1 | b1) & (r2 | y2 | b2)) &
((rt > ∼yt & ∼bt) & (r1 > ∼y1 & ∼b1) & (r2 > ∼y2 & ∼b2) &
(yt > ∼rt & ∼bt) & (y1 > ∼r1 & ∼b1) & (y2 > ∼r2 & ∼b2) &
(bt > ∼yt & ∼rt) & (b1 > ∼y1 & ∼r1) & (b2 > ∼y2 & ∼r2))) >
(#_1 r1 > #_1 (yt | bt))’
         true
```

Of course, there are many other formulas that OOPS can similarly prove in this situation, but these two examples should suffice to convey the general idea. Although a complex derivation may take a few minutes to complete, OOPS should be able to provide the correct answer eventually.