OOPS: An $S5_n$ Prover for Educational Settings

Gert van Valkenhoef Elske van der Vaart Rineke Verbrugge

12 October, 2009

university of groningen

What is OOPS?

- OOPS: Object Oriented Prover for $S5_n$
- Proven Sound & Complete, Correct
- Implemented in Java:
  - cross-platform
  - widely understood
  - long-lived
- Click & Run
- Open Source (GPL):
  - Get the source: http://github.com/gertvv/oops
- Aimed at students learning $S5_n$

university of groningen

Why OOPS?

- Started as a Multi-Agent Systems (MAS) student project
- Out of frustration with the LWB
  - Binary blob with archaic dependencies
  - Hard to get working
  - No $S5_n$ support
- Because we want those with frustrations about OOPS to have the power to do something about it!

university of groningen

What OOPS is *not*

- Our proof method is not:
  - Highly innovative
  - Super efficient
  - Very complicated
- A competitor to the Tableaux Workbench
  - Our aim is not to support research in new logics

university of groningen

The Prover

- Implementation of **ELtap** (by Matthijs de Boer)
- Uses labeled tableaux
- Was proven (by me) to be Sound & Complete
- Complexity class: EXPSPACE
  - Problem for large theories
  - Algorithm is easy to replace (little coupling)
- Details in the paper (and references)

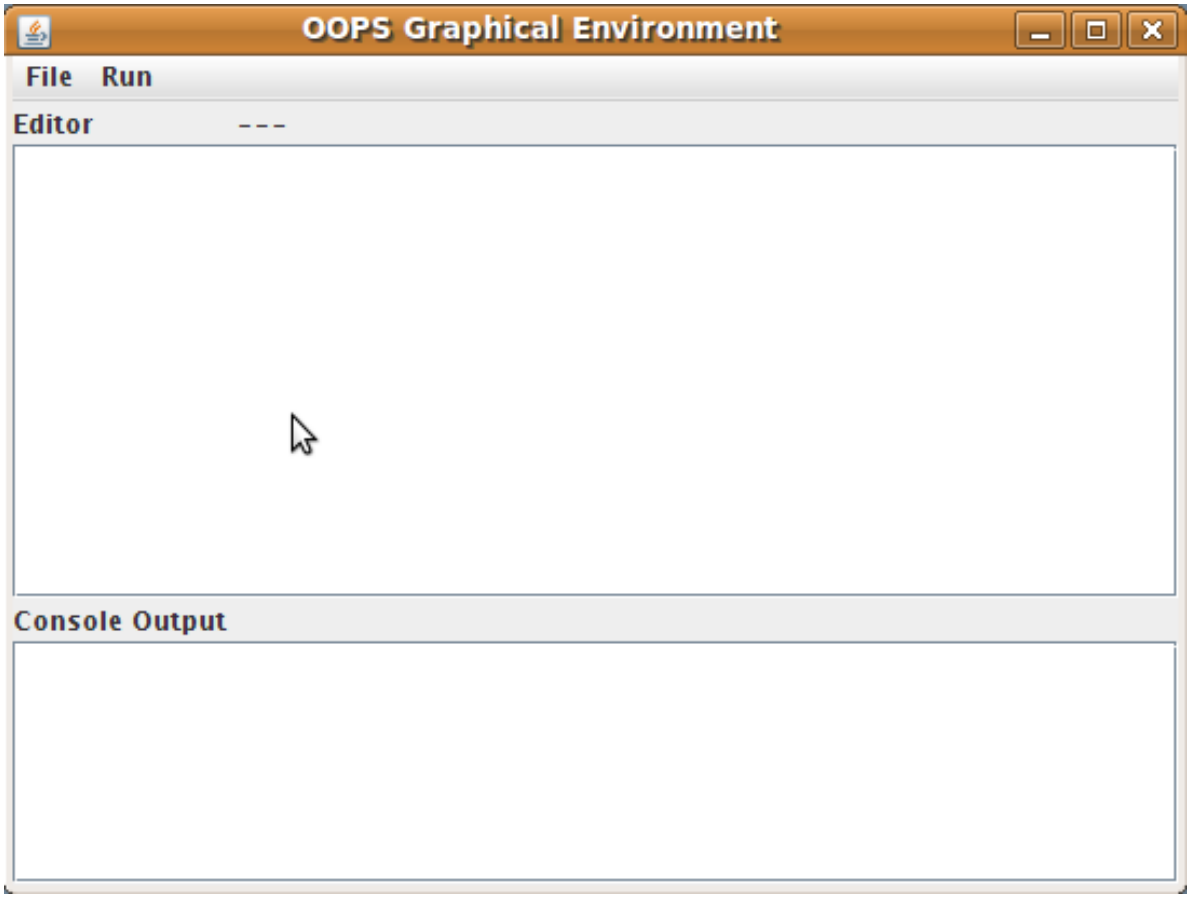
university of groningen

Features (overview)

- One-click run (no installation)
- Visualize tableaux
- Generate / visualize counter-models
- Integrated scripting (Lua)
  - Build agents that use OOPS to reason
- Basic graphical user interface

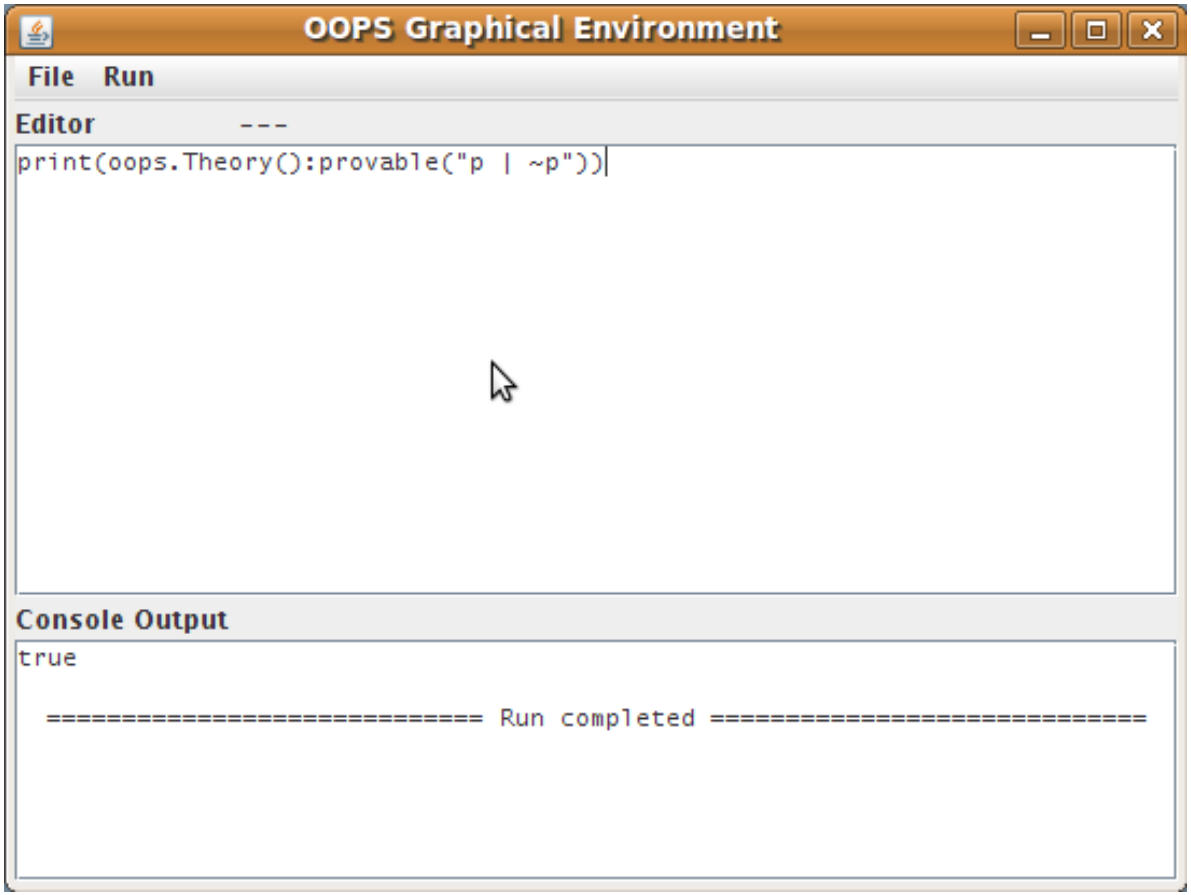
university of groningen

Demonstration



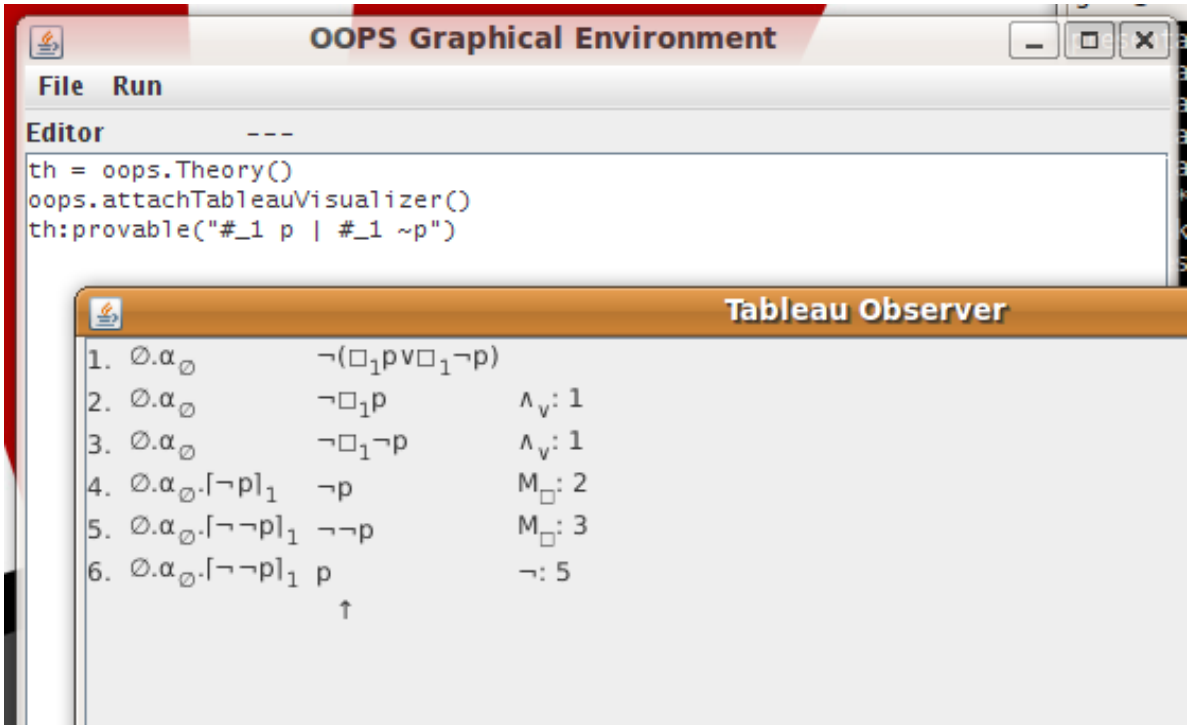
university of groningen

Demonstration



university of groningen

Demonstration



university of groningen

The Puzzle (part 1)

Initial situation:

- There are two wise persons: Abelard (1) and Heloise (2)
- There are 3 hats, two red (r) and one white (w)
- Each wise person wears a hat
- Each wise person can see the other's hat
- Each wise person can *not* see his/her own hat

Can they figure out their own hat's color?

university of groningen

Model: initial situation

```
-- agents[1] = "Abelard", agents[2] = "Heloise"
agents = {"Abelard", "Heloise"}

-- propositions:
-- r1: Abelard wears a red hat
-- w1: Abelard wears the white hat
-- r2: Heloise wears a red hat
-- w2: Heloise wears the white hat
```

university of groningen

Model: initial situation

```
-- each person has only one hat
oneHat = "(r1 = ~w1) & (r2 = ~w2)"
```

Model: initial situation

```
-- each person has only one hat
oneHat = "(r1 = ~w1) & (r2 = ~w2)"

-- there is only one white hat
oneWhite = "(w1 > r2) & (w2 > r1)"
```

university of groningen

Model: initial situation

```
-- each person has only one hat
oneHat = "(r1 = ~w1) & (r2 = ~w2)"

-- there is only one white hat
oneWhite = "(w1 > r2) & (w2 > r1)"

-- Abelard can see Heloise's hat
abelardSees = "#_1 w2 | #_1 r2"
```

university of groningen

Model: initial situation

```
-- each person has only one hat
oneHat = "(r1 = ~w1) & (r2 = ~w2)"

-- there is only one white hat
oneWhite = "(w1 > r2) & (w2 > r1)"

-- Abelard can see Heloise's hat
abelardSees = "#_1 w2 | #_1 r2"

-- Heloise can see Abelard's hat
heloiseSees = "#_2 w1 | #_2 r1"
```

university of groningen

Model: initial situation

```
-- background knowledge:
background = conj(conj(oneHat, oneWhite),
     conj(abelardSees, heloiseSees))

-- add the background and the wise persons'
-- knowledge about it (to depth 2)
th = oops.Theory()
th:add(background)
th:add(knows("1", background))
th:add(knows("2", background))
th:add(knows2("2", "1", background))
th:add(knows2("1", "2", background))
```

university of groningen

Model: initial situation

```
-- print knowledge state of agent
function printKnowledge(th, agent)
    -- ...
end

-- print knowledge state of Abelard
function printAbelard(th)
    printKnowledge(th, 1)
end

-- print knowledge state of Heloise
function printHeloise(th)
    printKnowledge(th, 2)
end
```

university of groningen

Model: initial situation

```
-- make sure we can show the counter model
oops.attachModelConstructor()

print("Initial Situation: ")
printAbelard(th)
printHeloise(th)
```

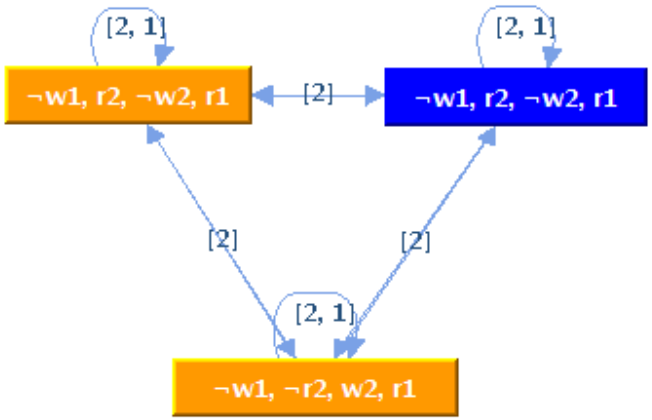
Output

```
Initial Situation:
Abelard doesn't know
Heloise doesn't know
```

university of groningen

Model: initial situation

```
oops.showModel()
```



university of groningen

The Puzzle (part 2)

- Abelard is asked if he knows the color of his hat
- Abelard announces that he doesn't
- Now Heloise knows the color of her hat

What is the color of Heloise's hat?

Model: after Abelard's announcement

```
-- After Abelard announces he doesn't know:
abelardKnows = "#_1 w1 | #_1 r1"
th:add(oops.Formula("#_2 ~ F"):substitute(
    {F = abelardKnows}, {}))
```

university of groningen

Model: after Abelard's announcement

```
print()
print("After Abelard's announcement: ")
printAbelard(th)
oops.showModel()
printHeloise(th)
print("Consistent: " .. tostring(th:consistent()))
```

Model: after Abelard's announcement

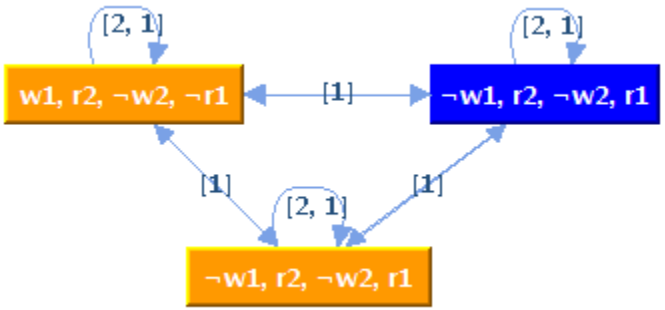
Output

```
After Abelard's announcement:
Abelard doesn't know
Heloise knows his/her hat is red
Consistent: true
```

university of groningen

Model: after Abelard's announcement

Abelard still doesn't know:



university of groningen

Model: after Abelard's announcement

Abelard still doesn't know:

university of groningen

Conclusion

OOPS has some interesting features:

- Cross-platform, easy to run, open source
- GUI and integrated scripting
- Tableau visualization
  - Sometimes useful when 'debugging' simple models
  - Useful when learning tableau methods (check your own answers)
- Counter model visualization
  - Useful when learning about Kripke semantics
  - Useful when 'debugging' less simple models

university of groningen

Future Work

However, much remains to be done:

- Provide nicer Lua interface (implemented in Lua)
- Be able to specify proof rules in Lua
- More efficient proof algorithm
- Formula simplification
- Kripke models:
  - Construct / alter models through scripting
  - Model checking
  - Bisimulation
- DOPS as a Lua library (in addition to vice versa)

university of groningen