ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/318830008

Omniscient Debugging for GOAL Agents in Eclipse (Demonstration)

Conference Paper · August 2017

DOI: 10.24963/ijcai.2017/774

CITATIONS
0

READS
29

3 authors:

Vincent Koeman
Delft University of Technology

12 PUBLICATIONS **33** CITATIONS

SEE PROFILE

Koen V. Hindriks
Vrije Universiteit Amsterdam

229 PUBLICATIONS **3,048** CITATIONS

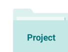
SEE PROFILE

Catholijn M. Jonker
Delft University of Technology

541 PUBLICATIONS **6,312** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Architectures for Virtual Characters for Real-Time interaction View project

negotiation support systems View project

All content following this page was uploaded by Vincent Koeman on 08 August 2017.

The user has requested enhancement of the downloaded file.

Omniscient Debugging for GOAL Agents in Eclipse
(Demonstration)

Vincent J. Koeman, Koen V. Hindriks and **Catholijn M. Jonker**
Delft University of Technology, Mekelweg 4, 2628CD, Delft, The Netherlands
{v.j.koeman, k.v.hindriks, c.m.jonker}@tudelft.nl

Abstract

The main goal of our demonstration is to show how omniscient debugging can be applied in practice to cognitive agents. A concrete implementation of the mechanisms proposed in [Koeman *et al.*, 2017] has been created for the GOAL agent programming language in the Eclipse environment, integrated with the source-level debugger of [Koeman *et al.*, 2016], thus fully implementing the proposal within a state-of-the-art setting. The implementation will be used together with typical agent programs to demonstrate its practical use.

1 Introduction

Running the same agent system again more often than not results in a different program run or trace, which complicates the iterative process of debugging [Bracha, 2012]. Therefore, in [Koeman *et al.*, 2017], we propose a tracing mechanism that supports *omniscient debugging for cognitive agents*. This mechanism facilitates debugging by allowing a developer to move backwards in time through a program's execution. Such a 'time travelling debugger' is regarded as one of the most powerful debugging tools [Zeller, 2009; Bracha, 2012]. We show in [Koeman *et al.*, 2017] that the proposed tracing mechanism for cognitive agent programs is efficient and does not affect the runs of agent programs in the sense that the same failures can be reproduced when the mechanism is turned on and off. This means that the mechanism is *fast enough to be used in practice for debugging failures in cognitive agents*, opposed to existing implementations (e.g., for Java) that have a significant performance impact.

In [Koeman *et al.*, 2017] we also introduce a trace visualization method tailored to cognitive agents based on a *space-time view of the execution history* [Azadmanesh and Hauswirth, 2015]. A developer can navigate this view, evaluate queries on a trace, and apply filters to it to obtain views of only the relevant parts of a trace. The approach is integrated with a source-level debugger and traces source code locations, which enables a developer to *single-step through a program's execution history*, facilitating fault localization.

Prototypes of the proposed mechanisms have been implemented for the GOAL agent programming language [Hindriks, 2009], embedded in the GOAL plug-in for Eclipse [Koeman and Hindriks, 2015].

In this demonstration proposal, we will describe the omniscient debugging mechanisms that have been implemented for the GOAL agent programming plug-in in Eclipse. A typical example will be used in the demonstration[1] to provide an insight into the practical use of these features, integrated with the source-level debugger [Koeman *et al.*, 2016].

2 Demonstration

In [Koeman *et al.*, 2017], we discuss how to efficiently facilitate reverting an agent to any previous state by recording its execution (i.e., tracing). This recording can be toggled on or off in the development environment. For efficient fault localization, however, it also needs to be easy for a developer to identify states in a program's execution that are related to the failure under investigation. Moreover, a developer should not get lost in navigating between these states, but always have a sense what point in the execution s/he is evaluating and how the current state affected the execution.

Therefore, in [Koeman *et al.*, 2017], we adapt the concept of a *space-time view* first developed in [Azadmanesh and Hauswirth, 2015] in the context of Java programming to cognitive agent programming. A space-time view is a table that is structured along space and time dimensions, where the rows in the table correspond to the space dimension, which is composed of the different elements in a state that are traced. The columns entail each step in the agent's execution history.

For cognitive agents in GOAL, the elements in a space-time view (i.e., that are traced) are the agent's events, beliefs, goals, actions, and modules. We use the corresponding *signatures* as the rows in the space dimension. For example, the signature `print/1` in Fig. 1 represents a `print` action with one parameter. Each point in a trace represents a step (column) in the time dimension. Multiple space elements (signatures) can be used in a single step, e.g., evaluating a query may require accessing several beliefs and goals. The cells in our space-time view contain information about how an element was used at a particular step, which differs per type of element (e.g., a belief can be modified or inspected, an action or plan can be called and performed, a module can be

[1] A short demonstration video of the tool is available at: `https://www.youtube.com/watch?v=qAhtnGtczVM`

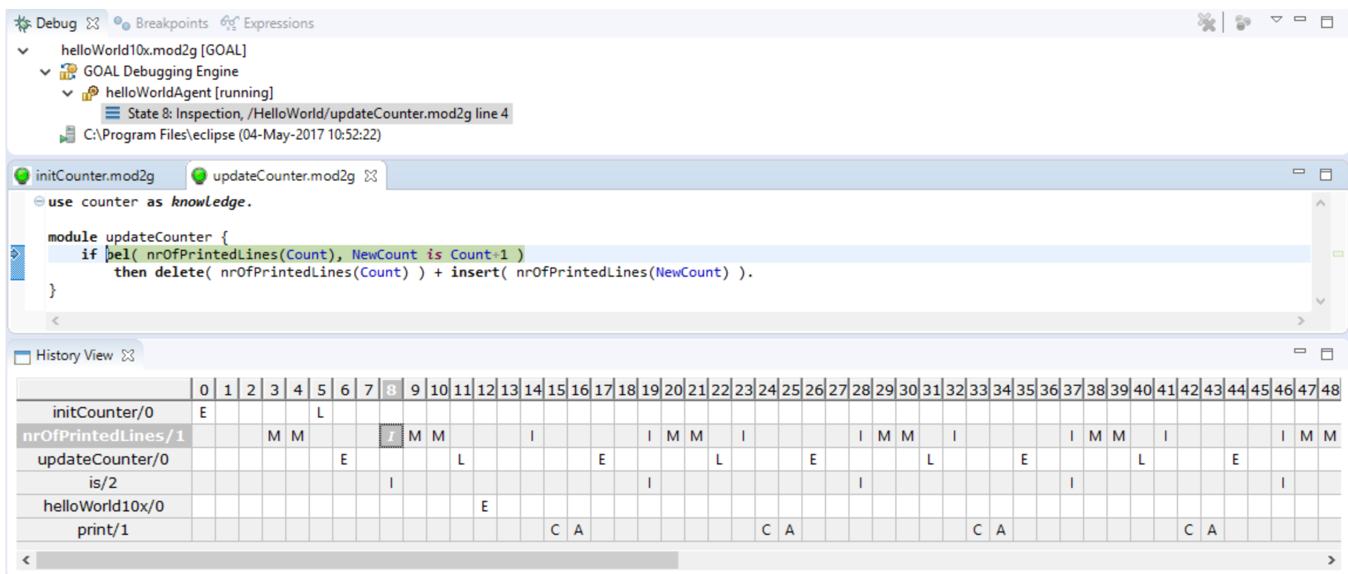


Figure 1: An example of the space-time view of an execution history integrated in the debugger for GOAL in Eclipse.

entered or exited). Empty cells indicate the element was not used. To prevent cluttering the table, events (i.e., percepts or messages) are not given a row in the table, whilst for example receiving a new message does entail a separate step and thus an empty column in the table. Moreover, a single letter is used in each cell to ensure more information fits in a limited space: **E**ntered a module, **L**eft a module, **M**odified of a belief or goal, **I**nspected of a belief or goal, **C**alled an action, or **A**ction executed. We note that calling an action is different from executing an action, as the action's precondition might fail after calling it (after which the action is not executed). An example of such a space-time view is shown in Fig. 1.

A developer can use and manipulate the space-time view in several ways. The signatures listed in the space-time view's rows can be ordered based on type (e.g., beliefs next to beliefs), alphabetically, or by order of occurrence (the default). A developer can also apply a filter to a trace by for example selecting the signatures of interest (i.e., hiding rows) or entering a query that should hold in each state of interest (i.e., hiding columns). By clicking on any cell in the table, the debugger will reverse the agent to the state matching the column, allowing a developer to use all debugging tools (e.g., inspecting an agent's beliefs and goals) in that specific historic state. This feature is also highlighted in Fig. 1.

We illustrate the use of these features for analysing a failure of the following example test condition associated with an agent program for the Blocks World for Teams (BW4T) [Johnson *et al.*, 2009], which is one of the environments used for educational purposes[2]: "`goal(holding(B)), bel(atBlock(B))` *leadsto* `done(pickUp(B))`". This condition expresses that if the agent has the goal to hold block `B`, and believes it is at the block, that it should (eventually) pick up `B`. A failure to do so will lead to failure of the test condition (i.e., when the agent is terminated). Without an omniscient debugger, a developer would need to restart the agent, navigate to a point where the goal-belief query holds, and continue by manually stepping to try to understand why the action is not performed. With an omniscient debugger, we do *not* need to restart the agent, and can use the clues provided by the test condition itself to navigate to the last time that `holding/1` and `atBlock/1` were modified in the space-time view. We can do so either by double-clicking the corresponding cell, or, even faster, by using the filter query `goal(holding(B)), bel(atBlock(B))`.

The goal is to facilitate locating bugs in an efficient manner, especially when dealing with very dynamic environments. An example of such an environment that we will also use in our demonstration is StarCraft [Griffioen and Plenge, 2016], i.e., we will use GOAL agent programs for StarCraft to illustrate how history-based source-level debugging facilitates fault localization for highly dynamic agent programs.

3 Conclusion

A concrete implementation of the mechanisms proposed in [Koeman *et al.*, 2017] has been created for the GOAL agent programming language in the Eclipse environment, integrated with the source-level debugger of [Koeman *et al.*, 2016], thus fully implementing the proposal within a state-of-the-art setting. The implementation will be used together with typical agent programs to demonstrate its practical use. This implementation and its source are publicly available[3], thus together with this demonstration providing a valuable example for the adaptation of omniscient debugging into other agent programming languages.

[2] All (educational) agent environments are freely available at `https://github.com/eishub`. Most of these projects include an assignment for (novice) agent programmers.

[3] See `https://goalapl.atlassian.net/wiki` for more information. Note, however, that the omniscient debugging features are still under development, and thus not released in the latest stable version(s) of the GOAL plug-in at the time of writing.

References

[Azadmanesh and Hauswirth, 2015] Mohammad R. Azadmanesh and Matthias Hauswirth. Space-time views for back-in-time debugging. Technical Report 2015/02, University of Lugano, 2015.

[Bracha, 2012] Gilad Bracha. Debug mode is the only mode. `https://gbracha.blogspot.nl/2012/11/debug-mode-is-only-mode.html`, November 2012. Accessed: 2017-02-19.

[Griffioen and Plenge, 2016] H.J. Griffioen and D. Plenge. Multi-agent systems in StarCraft. Bachelor thesis, Delft University of Technology, 2016.

[Hindriks, 2009] Koen V. Hindriks. Programming rational agents in GOAL. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

[Johnson *et al.*, 2009] Matthew Johnson, Catholijn Jonker, Birna van Riemsdijk, Paul J. Feltovich, and Jeffrey M. Bradshaw. Joint activity testbed: Blocks World for Teams (BW4T). In Huib Aldewereld, Virginia Dignum, and Gauthier Picard, editors, *Engineering Societies in the Agents World X: 10th International Workshop, ESAW 2009, Utrecht, The Netherlands, November 18-20, 2009. Proceedings*, pages 254–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Koeman and Hindriks, 2015] Vincent J. Koeman and Koen V. Hindriks. A fully integrated development environment for agent-oriented programming. In Yves Demazeau, Keith S. Decker, Javier Bajo Pérez, and Fernando de la Prieta, editors, *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*, volume 9086 of *LNCS*, pages 288–291. Springer International Publishing, 2015.

[Koeman *et al.*, 2016] Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Designing a source-level debugger for cognitive agent programs. *Autonomous Agents and Multi-Agent Systems*, 2016.

[Koeman *et al.*, 2017] Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Omniscient debugging for cognitive agent programs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, IJCAI '17, Palo Alto, CA, USA, 2017. AAAI Press.

[Zeller, 2009] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

View publication stats