Using Automatic Failure Detection for Cognitive Agents in Eclipse

Vincent J. Koeman
Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
v.j.koeman@tudelft.nl

Koen V. Hindriks
Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
k.v.hindriks@tudelft.nl

Catholijn M. Jonker
Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
c.m.jonker@tudelft.nl

1. INTRODUCTION

In order to reduce debugging effort and enable automated failure detection, we proposed an *automated testing framework for cognitive agent programs* that provides support for detecting frequently occurring failure types in [14]. Automated testing yields a reduction in the effort needed to detect a failure and is more effective than manual code inspection methods [16].

A *failure* is an event in which a system does not perform a required function within specified limits [10]. Failures thus are manifestations of undesired behaviour. They are caused by a *fault*, an incorrect step, process, or data definition in a program [10] or mistake in a program [17]. Upon detecting a failure, a programmer needs to locate and correct the fault that causes the failure.

We introduced a test language based on two basic temporal operators, and use this language to specify test templates for detecting failure types. These test templates refine a failure taxonomy introduced previously in [17]. A test approach has also been specified that explains how to instantiate test templates and derive test conditions for specific failure types. The main steps of this approach are (i) to define success in terms of functional requirements, (ii) to test cognitive state updating, and (iii) to classify failures that concern actions and goals.

2. MAIN PURPOSE: THE AUTOMATED TESTING OF COGNITIVE AGENTS

In general, different techniques for detecting failures of program code are available, ranging from *inspection* of source code and logs to automated *testing* tools [16]. The need for debugging techniques and test approaches for agent-oriented programming has been broadly recognized [2, 4, 5]. Techniques for agent-oriented programming need to be based on the underlying agent paradigm [15, 18]. However, this is a significant challenge, as they should for example take into account that agents execute a specific decision cycle and operate in non-deterministic environments [1, 3, 9].

The developed testing framework provides a *systematic approach for detecting failures in cognitive agent programs* by using test templates that target specific types of failures in the taxonomy, and a systematic method for using these templates for testing. Table 1 lists information resources that are particularly useful for testing.

Source	Type of Information
Agent program (comments)	Clues for reasons & design
Agent trace (screen, logs)	Observable behaviour
Agent design & specification	Functional requirements
Environment (documentation)	Percepts, actions available

Table 1: Information sources for testing

3. DEMONSTRATION: TESTING GOAL AGENTS IN THE ECLIPSE IDE

As we have implemented the automated testing framework for GOAL [8], we will use the GOAL agent programming plug-in[1] for Eclipse in our demonstration. This plug-in provides a full-fledged development environment for agent programmers, integrating all agent and agent-environment development tools in a single well-established setting [13]. The Eclipse platform is based on an open architecture that allows for building on top of well-known existing frameworks [6]. By using Eclipse and the DLTK framework [7], for example, a state-of-the-art editor for GOAL has been created, which forms a solid foundation for further tools. The GOAL language itself has been recently updated to use a more modular approach, i.e., better facilitating re-use. In addition, a source-level debugger for agents has been fully implemented in the plug-in based on the work in [12].

The GOAL plug-in for Eclipse contains a fully implemented version of the automated testing framework for GOAL agents. This implementation has been integrated into the source-level debugger, facilitating for example the inspection of an agent's state as soon as a test condition has failed, as illustrated in Figure 1. The new modular approach of the GOAL agent programming language also better facilitates the testing of individual, separate pieces of functionality.

Tests are programs that we write in a *test language*. The test language is built on top of the GOAL programming language and re-uses parts of that language. The test language provides support for two main tasks: *setting up a test* and *specifying* which *test conditions* should be evaluated.

In a test we can execute only part of an agent and even make the agent do things it would not otherwise do. The *testaction*s that are specified in the test program are performed

[1]See `http://goalhub.github.io/eclipse` for a demonstration(video) of the testing framework implementation and instructions on how to install GOAL in Eclipse.

Appears in: *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016), J. Thangarajah, K. Tuyls, C. Jonker, S. Marsella (eds.), May 9–13, 2016, Singapore.*

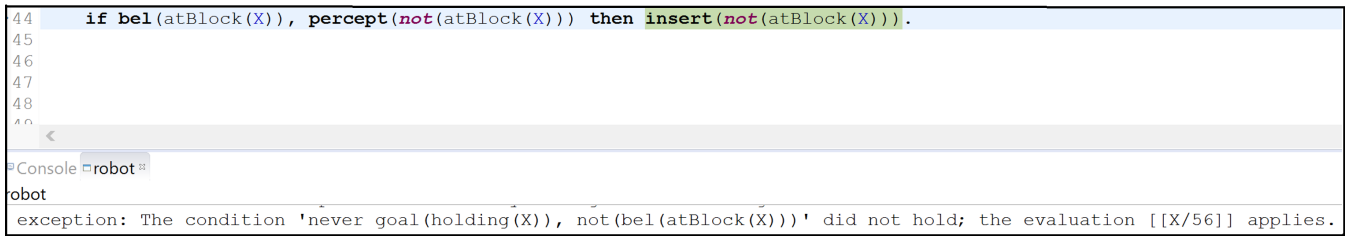

Figure 1: A partial illustration of how test failures are displayed in the source-level agent debugger.

when the test is run. Test actions can be preparatory actions for, e.g., initializing an agent's state, but they can can also be instructions to execute a module. Test conditions are built on top of the cognitive state queries that are used in program rules, like querying the beliefs or goals of an agent. In addition to this, a condition `done`(*action*) can be used to test whether some action has just been performed. A *test condition* is a temporal condition that expresses that something (i.e. a state query) should happen always, never, eventually, or when some other condition has been true before. Test conditions are associated with a module, and evaluated whilst this module is executed. It is possible to associate a pre-condition, a post-condition, and an in-condition with a module test. The *pre-condition* of a module is a state condition that should hold when a module is entered. Similarly, a *post-condition* is a state condition that should hold when a module is exited. An *in-condition* is a set of temporal test conditions that specify which behaviour is expected of a module while it is executed.

The examples we will use to demonstrate this test framework are some of the educational environments[2] that are developed alongside the GOAL language, like the Blocks World for Teams (BW4T) [11]. In our demonstration, we will show that something we want to happen eventually actually never happens (i.e. detect a failure). For example, an agent should move a block in order to achieve its goal, but it does not do that. We thus want our test to fail. But to show that something will never happen takes a long time. We can instead use a *time out* to ensure termination of the test after a specified time. A time out is global and specifies how much time (in seconds) is allowed to pass before the entire test should be completed. If a time out happens, the test is aborted.

In this example, we want to check whether our agent will move a specific block at some point in time during the execution of its main module (*stackBuilder*). More precisely, we want to know whether at some point in time the agent will perform the action *move(b8,table)*. We can use the `eventually` operator for this. As temporal conditions are specified as in-conditions, and we want to evaluate the *stackBuilder* module, we get the following module test example:

Note that tests should be repeated sufficiently often as states generated will differ per run, if only because environments are more often than not non-deterministic. When a failure is detected, i.e., when running a test it fails at some point, the corresponding fault in the agent program must be located. The program location where the agent is at when the test failed is indicated by the testing framework (when using the debugger). Although it is often the case, it is not always true that this location also is the fault location, i.e., the place of the actual error in the code. If the fault is not located immediately additional debugging is needed. In particular, faults related to actions that are performed but should not have been performed are usually more difficult to locate, as we will show as well.

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

```
use BlocksWorld as mas.
use stackBuilder as module.
timeout = 1.

test stackBuilder with
   in{ eventually done(move(b8,table)). }

stackBuilderAgent {
   do stackBuilder.
}
```

Figure 2: An example of a test that determines whether the *stackBuilderAgent* moves *b8* to the *table* at least once in the *stackBuilder* module.

In this way, we will demonstrate how in practice our test approach (i.e. of [14]) can be used to automatically detect failures and eventually resolve the faults that cause them. In this way, the implementation details of the automated testing framework for GOAL agents will be highlighted, together with the source-level debugger in which it is integrated.

4. CONCLUSION

The main goal of our demonstration, is to show how an automated testing framework for cognitive agents facilitates the detection of failures and aids in the localization of faults. In [14], we have proposed an automated testing framework for cognitive agents and an associated test approach based on test templates for frequently occurring failure types. By using a concrete implementation of the testing framework for the GOAL agent programming language, an integration with the existing source-level debugger was created within the Eclipse environment, thus fully implementing the design within a state-of-the-art setting. This implementation and its source are publicly available, and used in our demonstration in order to illustrate concrete examples of its use, and provide insight into practical implementation details that may be valuable for the adaptation into other agent programming languages.

REFERENCES

[1] R. Bordini, M. Dastani, and M. Winikoff. Current Issues in Multi-Agent Systems Development. In *Engineering Societies in the Agents World VII*, volume 4457, pages 38–61. 2007.

[2] R. H. Bordini, L. Braubach, J. J. Gomez-sanz, G. O. Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

[3] G. Caire, M. Cossentino, and A. Negri. Multi-agent systems implementation and testing. In *Proc. of the 4th From Agent Theory to Agent Implementation Symposium, AT2AI-4*, 2004.

[4] M. Dastani. Programming multi-agent systems. *The Knowledge Engineering Review*, 30:394–418, 9 2015.

[5] J. Dix, K. V. Hindriks, B. Logan, and W. Wobcke. Engineering Multi-Agent Systems (Dagstuhl Seminar 12342). *Dagstuhl Reports*, 2(8):74–98, 2012.

[6] D. Geer. Eclipse becomes the dominant java ide. *Computer*, 38(7):16–18, July 2005.

[7] S. Gomanyuk. An approach to creating development environments for a wide class of programming languages. *Programming and Computer Software*, 34(4):225–236, 2008.

[8] K. V. Hindriks. Programming rational agents in goal. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

[9] Z. Houhamdi. Multi-Agent System Testing: A Survey. *International Journal of Advanced Computer Science and Applications*, 2(6):135–141, 2011.

[10] ISO. ISO/IEC/IEEE 24765:2010 systems and software engineering - vocabulary. Technical report, Institute of Electrical and Electronics Engineers, Inc., 2010.

[11] M. Johnson, C. Jonker, B. van Riemsdijk, P. J. Feltovich, and J. M. Bradshaw. Joint activity testbed: Blocks world for teams (bw4t). In H. Aldewereld, V. Dignum, and G. Picard, editors, *Engineering Societies in the Agents World X*, volume 5881 of *Lecture Notes in Computer Science*, pages 254–256. Springer Berlin Heidelberg, 2009.

[12] V. J. Koeman and K. V. Hindriks. Designing a source-level debugger for cognitive agent programs. In Q. Chen, P. Torroni, S. Villata, J. Hsu, and A. Omicini, editors, *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, volume 9387 of *Lecture Notes in Computer Science*, pages 335–350. Springer International Publishing, 2015.

[13] V. J. Koeman and K. V. Hindriks. A fully integrated development environment for agent-oriented programming. In Y. Demazeau, K. S. Decker, J. Bajo Pérez, and F. 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.

[14] V. J. Koeman, K. V. Hindriks, and C. M. Jonker. Automating failure detection in cognitive agent programs. In *Proceedings of the 2016 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '16. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[15] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah. Testing in Multi-Agent Systems. In *Agent-Oriented Software Engineering X*, volume 6038, pages 180–190. Springer Berlin Heidelberg, 2011.

[16] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *Software, IEEE*, 23(3):82–90, May 2006.

[17] M. Winikoff. Novice programmers' faults & failures in goal programs. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 301–308, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.

[18] Z. Zhang, J. Thangarajah, and L. Padgham. Model based testing for agent systems. *Software and Data Technologies*, 22:399–413, 2008.