Citation:
Zhang, Z, Thangarajah, J and Padgham, L 2008, 'Automated unit testing intelligent agents in PDT', in Proceedings of the 7th International Conference on Autonomous Agents and Multi Agent Systems (AAMAS-08), Estoril, Portugal, 12-16 May 2008.

Automated Unit Testing Intelligent Agents in PDT (Demo Paper)

Zhiyong Zhang
RMIT University
Melbourne, Australia
z.zhang@student.rmit.edu.au

John Thangarajah
RMIT University
Melbourne, Australia
johnt@rmit.edu.au

Lin Padgham
RMIT University
Melbourne, Australia
lin.padgham@rmit.edu.au

ABSTRACT

The Prometheus Design Tool (PDT) is an agent development tool that supports the Prometheus design methodology and includes features like automated code generation. We enhance this tool by adding a feature that allows the automated unit testing of agents that are built from within PDT.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent agents*

Keywords

Agent Oriented Software Engineering, Agent platforms and development environments

1. PROMETHEUS AND PDT

Prometheus [7] is an agent development methodology for building agent systems. It defines a detailed development process that consists of System specification, High-level design and Detailed design.

The Prometheus Design Tool (PDT)[1] is a freely available tool that supports the Prometheus methodology. PDT provides a graphical interface for the specification and design phases of the methodology, allowing the designer to enter and edit diagrams and descriptors for entities. PDT also includes features such as consistency checking, report generation, protocol specification and code generation.

Code generation is a key feature of PDT that supports the implementation phase, maintaining consistency between design and code. Skeleton code can be generated from the detailed design of agents in PDT. Currently the code generated is in the JACK agent-oriented programming language [3].

We have extended PDT to incorporate a unit testing framework which we have developed [11] that uses both the agent system design and the implementation of it.

2. THE TESTING FRAMEWORK

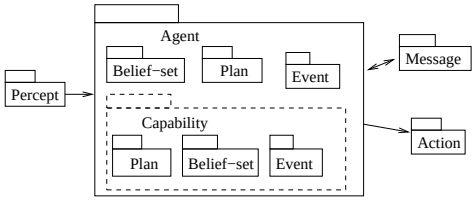

Figure 1: Agent Component Hierarchy

The lack of proper testing mechanisms for agent systems is still a hindrance to the uptake of agent technology. Although some principles can be generalized from testing of object oriented systems [2], there are also aspects which are clearly different and that require knowledge of the underlying agent paradigm. For example, in many agent system development paradigms there is a concept of an *event* which triggers selection of one of some number of identified plans, depending on the situation. If one of these plans is actually never used, then this is likely to indicate an error.

We have developed a testing framework [11], which automatically generates and executes unit test cases for an agent system based on its design model (developed in PDT). The testing framework is based on the notion of model based testing [1] which proposes that testing be in some way based on design models of the system. The Prometheus methodology has well developed structured models that are suitable as a basis for model based testing. The design model provides information against which the implemented system can be tested, and also provide an indication of the kind of faults that one might discover as part of a testing process.

Figure 1 outlines the components of an agent within the Prometheus design [2]. An agent may consist of plans, events and belief-sets, some of which may be encapsulated into capabilities. Percepts and incoming messages are inputs to the agent, while actions and outgoing messages are outputs from the agent. We identify plans, events and belief-sets as the units subject to testing. In our current implementation we do not test belief-sets, which is left for future work.

When we consider a plan as a single unit we test if the plan is ever used, does it complete? and does it post the events as indicated in its design? Some plans may form a cyclic structure if plan A posts an event that is handled by Plan B, and plan B posts an event handled by plan A. For such plans we test if the cycle exist at run-time, and whether the cycle is finite. In the case of an event, we test if the event is handled by some plan (coverage) and if it is handled by

[1]http://www.cs.rmit.edu.au/agents/pdt

Cite as: Automated Unit Testing Intelligent Agents in PDT (Demo Paper), Zhiyong Zhang, John Thangarajah and Lin Padgham, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16.,2008,Estoril,Portugal,pp. 1673-1674.

[2]Other agent oriented methodologies use similar constructs.

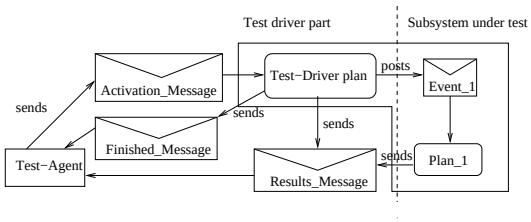

Figure 2: Testing framework: testing a Plan

more than one plan (overlap).

The testing process consists of the following four phases.
Generation of the testing order - The dependency between units in an agent system determines the order in with the units are to be tested. For example, a plan may fail because of the failure of one of its subtask plans. So a plan should be tested after all its subtask plans to avoid the affects from the subtasks.
Development of test cases - The test cases are generated utilizing the variables defined in the Prometheus design model of the system. Existing techniques of variable combinations and pairwise testing are applied. The test cases are generated to catch potential faults that occur at run-time. We have defined the kinds of faults expected for the different units.
Code augmentation - The code of the agent system under test is augmented with special testing code to facilitate the testing process.
Testing and report generation - The test case execution is also an automated process, similar to all the above. During this process a report that contains a summary of the testing and a detailed analysis is generated. Although not currently implemented we hope to incorporate an interactive feature where the developer may input test cases to the test system during this process if necessary.

3. TESTING WITHIN PDT

We have integrated the above testing framework into PDT and is available under the *Tools* menu option within PDT. In order to use the feature, the *detailed design* of the components to be tested should be completed and the code implemented in the JACK agent language (it is recommended that the code generation feature within PDT is used to generate the skeleton code). The system may be partial, but must be executable at runtime as the tests are performed by executing the components with augmented code. For example, if the system design has five agents, and only two of them are to be tested, the user only needs to ensure complete code for these two agents such that they are executable. This completed code of the units to be tested is called the *system under test*(SUT).

When the SUT is tested, the code is copied to a testing directory, augmented to incorporate testing specific code and the *test driver* component which generates and executes the test cases. Figure 2 outlines the runtime testing process for a plan unit of the SUT.

4. DISCUSSION AND RELATED WORK

There has been some research on agent testing. However, they do not support an existing agent development methodology and embedded into a design tool as presented in this work. Some of them only concentrate on testing for path-coverage [6] or behavioral properties [12] of abstract BDI agents. Some look at agents as the base unit, studied the message-based [9] or state-based [10] model of agents, and performed black-box testing. Caire et al.[4] describe an original testing framework for agent-based system, but do not discuss the implementation details.

Most of the above work is based on *conformance testing*, which tests if the system meets the business requirements and are restricted to *black-box* testing. In contrast to these approaches, our work looks at *fault-directed testing* which tests the internal processes of the system and not the business requirements. Our approach is also integrated with the design methodology and supports testing at early stages of development.

In this work we have only addressed unit testing, in future work we will extend this work to include *integration* testing. To this end, we expect to build on existing work such as [8, 5]. The former described a debugger which, similar to this work, used design artefacts of the Prometheus methodology to provide debugging information at run-time. The latter presented a unit testing approach for multi-agent systems based on the use of *Mock-Agents*, where each Mock-Agent tests a single role of an agent under various scenarios.

5. REFERENCES

[1] L. Apfelbaum and J. Doyle. Model Based Testing. In *the 10th International Software Quality Week Conference*, CA, USA, 1997.
[2] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
[3] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents — components for intelligent agents in Java. AgentLink News, Issue 2, 1999.
[4] G. Caire, M. Cossentino, A. Negri, A. Poggi, and P. Turci. Multi-Agent Systems Implementation and Testing. In *the Fourth International Symposium: From Agent Theory to Agent Implementation*, Vienna, April 14-16 2004.
[5] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena. Unit Testing in Multi-Agent Systems using Mock Agents and Aspects. In *Proceedings of the 2006 International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, pages 83–90, 2006.
[6] C. K. Low, T. Y. Chen, and R. Ronnquist. Automated Test Case Generation for BDI agents. *Autonomous Agents and Multi-Agent Systems*, 2(4):311–332, 1999.
[7] L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.
[8] L. Padgham, M. Winikoff, and D. Poutakidis. Adding Debugging Support to the Prometheus Methodology. *Engineering Applications of Artificial Intelligence, special issue on Agent-Oriented Software Development*, 18(2):173–190, March 2005.
[9] C. Rouff. A Test Agent for Testing Agents and their Communities. *Aerospace Conference Proceedings, 2002. IEEE*, 5:2638, 2002.
[10] H.-S. Seo, T. Araragi, and Y. R. Kwon. Modeling and Testing Agent Systems Based on Statecharts. volume 3236, pages 308 – 321, 2004.
[11] Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing for agent systems. In *2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE-07)*, pages 10–18, Spain, July 2007.
[12] M. Zheng and V. S. Alagar. Conformance Testing of BDI Properties in Agent-based Software Systems. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 457–464, Washington, 2005. IEEE Computer Society.