Model based Testing for Agent Systems

(Extended Abstract)

Zhiyong Zhang, John Thangarajah, Lin Padgham
RMIT University
Melbourne, Australia
{Zhiyong.Zhang,John.Thangarajah,Lin.Padgham}@rmit.edu.au

Keywords

Agent Oriented Software Engineering, Testing

Introduction

The use of agent technology for building complex systems is increasing, and there are compelling reasons to use this technology. Benfield [1] showed a productivity gain of over 300% using a BDI (Belief Desire Intention) agent approach, while other work calculated that a very modest plan and goal structure provides well over a million ways to achieve a given goal, providing enormous flexibility in a modular manner.

However the complexity of the systems that can be built using this technology, does create concerns about how to verify and validate their correctness. In this paper we describe briefly an approach and tool to assist in comprehensive automated unit testing within a BDI agent system. While this approach can never *guarantee* program correctness, comprehensive testing certainly *increases confidence* that there are no major problems. The fact that we automate both test case generation, as well as execution, greatly increases the likelihood that the testing will be done in a comprehensive manner.

Given the enormous number of possible executions of even a single goal, it is virtually impossible to attempt to test all program traces. Once interleaved goals within an agent, or interactions between agents are considered, comprehensive testing of all executing becomes clearly impossible. Instead, we focus on testing of the basic units of the agent program - the beliefs, plans and events (or messages). Our approach is to ascertain that no matter what the input variables to an entity, or the environment conditions which the entity may rely on, the entity behaves "as expected" (obtained from design artefacts, produced as part of an agent design methodology).

We build on previous work [6] which described a basic architecture and approach. In this work we address some of the details of setting up the environment that is necessary to effectively realise that approach. More specifically, mechanisms to specify the *initialization procedures* for a given unit, *variable assignment* to execute test cases, and managing any *interaction with external entities*. The testing tool and approach described has been implemented within PDT[1], relying on the implemented agent system being in JACK[2].

The testing process as described in [6] is as follows:

1. Determine the order in which the units are to be tested.
2. Develop test cases with suitable input value combinations. Equivalence Classes Partitioning and Boundary Value Analysis (or Boundary Conditions) [4, p.52] [5, p.67] are used to obtain values which cover all cases for a given variable.
3. Augment the source code of the system under test with special testing code to provide appropriate information to the testing system.
4. Execute the test cases, collect and analyse results and produce a detailed report.

All of the above steps are automated and can be performed on a partial implementation of the system if desired.

Initialization Procedures

The testing process tests each individual unit. Prior to executing the test case for that unit however, it may be necessary to perform some initialization routines such as setting up connections to external servers, populating databases, initializing global variables and so on.

In the testing framework we allow these routines to be specified as initialization procedures for each unit in the *Unit Test Descriptor*. The *Unit Test Descriptor* captures testing specific properties of the unit which is in addition to the usual design descriptor that specifies the properties of the unit.

We allow multiple procedures to be specified where each are in the following format:

<order, owner_object, is_static, function_call, comment> .

Figure 1, is an example of such a specification. In this example, the unit requires the $initBookDB$ function of the $StockAgent$ to be called as the first initialization procedure. The method is static, hence can be invoked directly from the $StockAgent$ class. When complete the next method $initConnAmazon$ is executed.

order	owner object	is static	function call	comment
1	Stock Agent	yes	initBookDB()	method to populate the books database
2	BuyPlan	no	initConnAmazon()	sets up connection to Amazon.com

Figure 1: Example specification of initialization procedures

Variable Assignment

Test cases are defined by generating values for each specified variable of the unit under test [6]. To execute the test case these values must be assigned to the variables. The technique for assigning a value to a variable may vary depending on how the variable is

[1]http://www.cs.rmit.edu.au/agents/pdt
[2]http://www.aosgrp.com/products/jack/

Cite as: Model based Testing for Agent Systems, (Extended Abstract), Zhiyong Zhang, John Thangarajah, Lin Padgham, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009),* Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 1333–1334

coded in the implementation of the system. For example, a variable that is private to an object[3] needs to be set via the object's mutator functions. Because the testing process is fully automated, it is necessary that there be some specification of an *assignment relation* to allow appropriate assignment of variable values to the implementation of the variable in the code.

This assignment relation is specified in the Unit Test Descriptor of the unit and takes the following form for each variable: <*variable-name, type, assignment*>, where the *type* is classified as *simple, complex, belief* or *function* based on how the variable is implemented in the source code of the system under test. The *assignment* relation depends on these types and we describe them below: A *simple* variable is implemented as a public variable or a private variable that is set via a public mutator function. A *complex* variable is one that is part of a nested structure, such as an attribute of an object, which may in turn be part of another object and so on. *Belief* variables are variables that are fields of a particular belief-set. They do not need an assignment relation as the technique for assigning variables would be the same for any field of the belief. That is, create and insert a record with the values generated for the variable in concern. Although the automated test cases contain randomly generated values for the belief fields that are not specified as unit test variables, as with all the test units, the user may manually specify additional test cases (value combinations).

There may be instances where a variable in the design is realized by a function in the implementation. We term these variables as *function* variables. For example, the variable *total_order_cost* which requires some calculation. It is not possible to set the value of these variables as it depends on the value returned by the function. This value may depend on a number of other variables, some local to the function others outside. The current testing framework ignores such variables when generating test cases, and is left for future work.

In general, if no assignment relation is specified for a variable, the assumed default is a simple variable that is publicly accessible.

Interaction with external entities

Ideally the test cases should be run in a controlled environment such that any errors may be isolated to the unit under test. As a plan of an agent executes, it may interact with other entities, external to the agent containing the units being tested. We deal with two types of such external interactions: interactions with (i) external systems (e.g. external database server or another agent system) or with (ii) external agents that are part of the same system.

With respect to external systems, the interface to the agent under test may take various different forms, hence it is not straightforward to simulate this interaction in a controlled manner. Also, under the assumption that the external system is "fixed" (i.e. it is not under development, or able to be influenced), it is important that the unit under test respond appropriately to any interaction that happens with regard to such a system. Therefore the user is expected to ensure that such systems are accessible and functioning correctly prior to start of testing. User alerts/notifications to check this are generated based on design documentation, but can be turned off.

If the interaction is with another agent of the same system, the form of interaction is known, making it possible to simulate and control this interaction. This is important as some of these *interactee agents* (i.e. the external agents that the plan interacts with) may not have been fully implemented or tested. The technique employed is to replace the interactee agent with a test stub [3, p.148], [2, p.963]. We call such a test stub as a *Mock Agent*.

[3]In terms of Object Oriented programming.

Functionality of a Mock Agent : A mock agent simulates the message send-reply logic of the interactee agent that it replaces. When a plan is executed during its testing process, any message from the plan to an interactee agent will be received by the replacement mock agent. When the mock agent receives a message from the plan-under-test, it will have one of two possible responses according to the specification about this message in design: (a) If the design does not specify a reply to the message received, the mock agent will just log the message received, and do nothing else. (b) If the design specifies a reply message to the message received, the mock agent will log the message received, generate a reply message and send it to the plan-under-test. The data contained in the reply message in the current implementation is randomly assigned as the mock agent only simulates the interactee agent at the interaction level, and does not realise its internal logic [4].

Implementing a Mock Agent : The mock agents are automatically created when the testing framework builds the testing code for a plan. The process for generating the code of mock agents for a plan is as follows. First, all outgoing messaged are extracted from the plan under test. For each message, the interactee agent type that receives the message is identified. For each of the identified interactee agent type, the testing framework generates the code of a mock agent type that replaces this interactee agent type, following the rules below:

- The mock agent type shares the same type name as its replaced interactee agent type. Furthermore, if the interactee agent type has been implemented, the mock agent type also implements the same constructors as the constructors of it .
- For each message received by the interactee agent type, one plan is defined that handles this message in the mock agent. If this message has a reply specified in the design, this newly defined plan will create an object of the reply message and send it back to the agent-under-test. Else, the plan simply logs the message received.
- The code of this mock agent type is embedded into the test implementation to replace the code of its respective interactee agent type. Any message that is sent to the interactee agent will be received by this mock agent.

Acknowledgments
We acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant LP0882234.

1. REFERENCES

[1] S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *AAMAS '06*, pages 10–15, NY, USA, 2006. ACM.

[2] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] I. Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[4] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing, Second Edition*. Wiley, June 2004.

[5] R. Patton. *Software Testing (Second Edition)*. Sams, Indianapolis, IN, USA, 2005.

[6] Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing for agent systems. In *ENASE-07*, pages 10–18, Spain, July 2007.

[4]We are in the process of allowing users to define specific responses associated with certain test cases, in a similar manner to other user defined test case information.