Automated Testing for Intelligent Agent Systems

Zhiyong Zhang, John Thangarajah, and Lin Padgham

RMIT University, Melbourne, Australia
pdt@cs.rmit.edu.au

Abstract. This paper describes an approach to unit testing of plan based agent systems, with a focus on automated generation and execution of test cases. Design artefacts, supplemented with some additional data, provide the basis for specification of a comprehensive suite of test cases. Correctness of execution is evaluated against a design model, and a comprehensive report of errors and warnings is provided to the user. Given that it is impossible to design test suites which execute all possible traces of an agent program, it is extremely important to thoroughly test all units in as wide a variety of situations as possible to ensure acceptable behaviour. We provide details of the information required in design models or related data to enable the automated generation and execution of test cases. We also briefly describe the implemented tool which realises this approach.

1 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 Padgham and Winikoff calculated that a very modest plan and goal structure provides well over a million ways to achieve a given goal [2, p.16], 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 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". Our notion of expectations is obtained from design artefacts, produced as part of an agent design methodology. We focus in this paper on the details of how we determine variable values for test cases to provide comprehensive coverage, and the representations and mechanisms we use to allow us to automate the process (though also providing mechanisms for user specified test cases where desired.) We build on our previous work in [3] which described a

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 66–79, 2011.
© Springer-Verlag Berlin Heidelberg 2011

basic architecture and approach. However we address in this paper some of the details necessary to effectively realize that approach.

There are two specific aspects of automated test generation and execution which we focus on in this paper. The first is how to specify values relevant to a particular unit to be tested, and how to generate appropriate test cases that adequately cover the space of value combinations for all relevant variables. Doing this appropriately is critical in order to be able to have confidence that the testing process was thorough. The second important aspect is the setting up of the environment such that the unit to be tested can be executed appropriately. One aspect of this is to ensure that units which are depended on are tested first, and then included in the environment for the unit being tested. However in many real applications there may be interaction with an external program within a plan unit (e.g. a query to an external database of a web service). There may also be interaction with another agent within a testing unit. In these cases we must ensure that access to the external program is set up prior to testing, and that there is some mechanism for dealing with necessary interaction with other agents. We note that most interaction with other agents would not be *within* a testing unit. Typically an agent would generate a message in one plan, and handle it another, in which case there is no need to model the other agent in any way.

In the following sections we first provide an overview of the testing process of our previous work [3], the framework on which this work is based. In section 3, we describe the automated process for test case generation. Section 4 then covers the details needed to provide the environment where the units can be executed, including how to automatically generate *mock agents* to simulate interaction if necessary. We provide in section 5 a brief discussion of the implemented system and some results from its use, concluding in section 6 with a brief comparison to other work and directions for ongoing development.

2 Testing Process Overview

In [3], we present a model based framework for unit testing in agent systems, where the models used are provided by the design diagrams of the Prometheus methodology, a well established agent development methodology [2]. The Prometheus methodology, and associated Prometheus Design Tool (PDT) produces graphical models capturing relationships between basic entities such as goals, plans and beliefs. It also produces *descriptors* which collect together a range of design information for each design entity. The detailed design entities map closely to implementation entities and are utilized for generation of skeleton code. They can also be used for generation of test cases and analysis of testing results.

Figure 1 outlines the components of an agent that is developed using the Prometheus methodology. These components are typical for most systems that follow the Belief Desire Intention (BDI) model of agency [4]. An agent contains *plans*, *events* and *beliefs* and can have *capabilities* that encapsulate them for modularity. Percepts (external input to the system), messages (from other agents) and internal events are all considered as events in agent development tools such as JACK [5] and we use this same generalization. In brief, the agent uses its plans to handle events as they occur and may query and

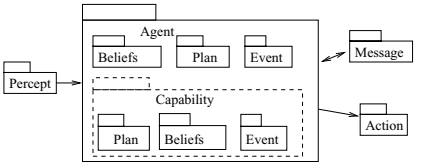

Fig. 1. Agent Component Hierarchy in Prometheus

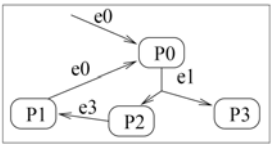

Fig. 2. Plan Dependencies

update its beliefs as needed during this process. Belief changes may automatically generate events. Given the above components, the units to be tested are the *events*, *plans* and *beliefs*. This extends the work in [3] which covered only events and plans.

Events are tested with respect to *coverage* and *overlap*. Coverage refers to the expectation that an event will always be handled by at least one plan in any situation. *Overlap* refers to the possibility that there may be more than one plan that is applicable for handling a given event, in some situation. While it is not *necessarily* an error to have an event without full coverage, or with overlap, these are common sources of error in agent programs [6]. Consequently PDT prompts developers to consider and to specify expected coverage and overlap as part of design of events. Test case results are then compared against the design specification.

A plan definition includes the type of event which triggers it, a context condition that determines its applicability in a given situation, and a plan body. The body of a plan may contain actions (considered atomic) and/or post sub-events (sub-tasks). Plans are tested for the following:

- Does the plan get triggered by the event that it is supposed to handle? If it does not, then either some other plan always handles it or there is an inconsistency between design and implementation.
- Is the context condition valid in some situations? If no context condition is specified then the plan is always applicable. However, if a context condition is specified then there must be at least some situation where it is satisfied for the plan to be applicable. If it is satisfied in any situation that could also indicate an error as it that is the equivalent of an empty context condition.
- Does the plan post the events that it should? Events are posted from a plan to initiate sub-tasks (or messages sent to other agents). If some expected events are never posted, we need to identify them as this may be an error. Here we can only check if the design matches the code, and can not check, for example, if the internal logic of the plan is correct/sensible or if the event/message contains the correct data.
- Does the plan complete? It is possible to determine whether the plan executed to completion. However, we cannot determine whether the plan completed successfully as the information required to determine the success of the plan is not currently contained in an extractable format in the design specification.

If a plan A posts a sub-task (sub-event) that is handled by plan B then the success of plan A is dependent on plan B, which should in general be tested before A. There may however be cyclic dependencies. For example, in Figure 2 $P0, P2$ and $P1$ are what is termed a *Plan Cycle* [3].

Plan cycles are a special case of testing plans, in that, in addition to testing for the aspects specific to plans, each plan cycle is considered as a single unit and tested for the existence and the termination of the cycle at run-time. If the cycle doesn't terminate after a maximum number of iterations this generates a warning to the user. If the cycle never occurs a warning is given as the cycle may have been a deliberate design decision.

While in the general case, events are generated from external input or from within a plan body, events may also be generated automatically when certain beliefs (conditions) become true. For example, if a robot agent believes the energy level to be less than 10% it may generate an event to re-charge. In agent systems like JACK [5] such events are implemented as database call-backs, other systems may implement them in the form of active databases, for example [7]. For every belief that generates such events, we test if the appropriate events get posted in the situations as specified in the design. This aspect of testing beliefs is an addition to the framework presented in [3].

The testing process consists of a number of steps which we outline briefly below.

1. Determine the order in which the units (events, plans, plan cycles and belief-sets) are to be tested. If a plan contains sub-tasks the success of the plan is partially dependent on the plan(s) that satisfy the sub-tasks. For example, in Figure 2 the success of plan $P0$ depends on the success of $P2$ or $P3$. The order of testing is therefore bottom-up where a unit is tested before the units that depend on it.
2. Develop test cases with suitable input value combinations as described in section 3.
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 analyze results and produce a detailed report. The report contains both errors and warnings, where the warnings are things which are likely to be errors, but which could be intended, or be due to lack of an appropriate test case (e.g. failure to have a context condition become true in any test, may be that a new specialized test needs to be added).

All of the above steps are automated and can be performed on a partial implementation of the system if desired. Figure 3, provides an abstract view of the testing framework for a plan unit. There are two distinct components: the *Test driver* and the *Subsystem under test*. The former contains the *Test Agent*, that generates the test cases and executes the test cases by sending an *Activation Message* to the *Test-Driver Plan*. This plan is embedded into the Subsystem under test during the code augmentation phase, and is

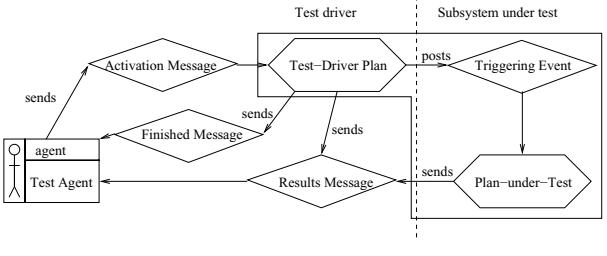

Fig. 3. Abstract Testing Framework for a Plan

responsible for setting up the test input and activating the *Subsystem under test* via a *Triggering Event*.

The subsystem contains the units to be tested and other parts of the system required to activate them. These units under test are able to send *Results Message*s back to the *Test Agent* via testing specific code inserted into them. After all testing is complete the *Test Agent* generates the test report.

3 Test Case Generation

Test cases are automatically generated for each unit, by extracting relevant variables from the design documents and generating variable value combinations. It is important to ensure that there is comprehensive coverage of the different possible situations the unit may execute in, so the choice of variable values is very important. At the same time, it is impossible to test all values, so choices must be made carefully. We also need to know which variables (in the environment, in the agent's beliefs, or in the unit itself) affect the particular unit. This is specified in the design documentation. These specifications enable automatic extraction of the variables.

Some examples are:

[event-var, book.bookName, string, !=null]
[belief-var, (StockDB, numberInStock), int, >0]

Table 1. Example of equivalence classes

name	index	domain	valid	minimal	normal	comprehensive
bookID	EC-1	$(-\infty, 0)$	no	N/A	-1	-1
	EC-2	$[0, +\infty)$	yes	0	0, 1	0, 1
stock	EC-1	$(-\infty, 0)$	no	N/A	-1	-1
	EC-2	[0, 200]	yes	0, 200	0, 1, 199, 200	0, 1, 100, 199, 200
	EC-3	$(200, +\infty)$	no	N/A	201	201
price	EC-1	$(-\infty, 0.0]$	no	0.0	-0.1, 0.0	-0.1, 0.0
	EC-2	(0.0, 90.0]	yes	90	0.1, 89.9, 90	0.1, 45.0, 89.9, 90.0
	EC-3	$(90.0, +\infty)$	no	90.1	90.1	90.1

Having automatically extracted the variables from the design for a particular unit, the next step in the testing process is to generate values for these variables which will test the unit. In many cases it is not possible to test all possible values as the domain may be infinite or very large. We use the standard notions of *Equivalence Class Partitioning* and *Boundary Value Analysis* [8, p.67] to generate a limited set of values for each relevant variable. An Equivalence Class (EC) is a range of values such that if any value in that range is processed correctly (or incorrectly) then it can be assumed that all other values in the range will be processed correctly (or incorrectly). Boundary values mark the (non-infinity) ends of an EC, and are often values which cause errors. Some approaches to testing use only valid ECs [9, p.98], while others improve robustness by also using invalid ECs [9, p.99] [10, p.39] [8, p.67] .

In our approach we allow the user to select from three different levels for choice of values for variables:

1. Minimal level is restricted to valid values, and uses boundary values for the valid ECs.
2. Normal level uses both valid and invalid ECs, selecting boundary values, and values close to the boundary, for each EC.
3. Comprehensive level also adds a nominal value in the mid-range of each EC.

To illustrate the above let us consider the following variables:

[agent-var, bookID, int, ≥ 0]
[agent-var, stock, int, ≥ 0, ≤ 200]
[agent-var, price, float, >0.0, $<= 90.0$]

Table 1 gives their ECs, and the choice of values from the three levels. Having chosen test values for each relevant variable these must be combined in various ways to produce test cases. Some approaches to testing simply ensure that for each variable, there is at least one test case with each of the values chosen [9, p.98] [8, p.71] [11, p.55]. A more thorough approach recognizes that many errors are a result of interactions between variable values, and takes the cross product of all values for each variable. The problem is that the latter quickly gives a very large number of test cases. The first approach caps the number of test cases at the largest number of values for any variable. The second is the cartesian product of the number of values for each variable, which quickly explodes. For example if we have five variables, each with five values, it gives over 3,000 test cases.

A commonly used approach to reduce the number of combinations is combinatorial design [12]. This approach generates a new set of value combinations that cover all n-wise (n>=2) interactions among the test parameters and their values in order to reduce the size of the input data set. We use the *Combinatorial Testing Service* software library of Hartman and Raskin[1] which implements this approach, in our system. However we apply this reduction only to test cases involving invalid values. We use all combinations of valid values, on the assumption that, for agent systems, it is interactions between valid variable values which will cause different aspects of the code to be activated (most commonly different plans chosen), and that covering all such interactions is necessary in order to adequately evaluate if the system is behaving correctly.

As with choice of values we allow the user to choose between different levels of thoroughness in testing. We provide the following options:

1. Basic level takes the cartesian product of valid values, and then adds one additional case for each invalid value, based on the assumption that invalid values do not require so rigorous testing as valid values.
2. Extended level supplements the results with all valid value cartesian products not in the set obtained using the pairwise reduction of combinatorial design [12].

In addition to the test cases that are auto-generated as described above, the user may wish to specify additional test cases using his domain and design knowledge. The testing framework accommodates this by means of a *Test Case Input* window in PDT for a given unit under test.

[1] `http://www.alphaworks.ibm.com/tech/cts` (obtained July 2006, technology now retired).

4 Test Case Execution

In order to execute a test case we need to first set up the environment so that it is able to run. Depending on the unit this may include such things as setting up sockets for communicating with an external process, setting up a global data structure that will be accessed, and so on. Then we also need to assign the values we have chosen for the relevant test case, to the appropriate variables, which requires knowing how these variables are implemented in the code. In this section we describe the details of the execution process, and the information representation which we require in order for this to be fully automated.

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* (see Figure 5). 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 4, 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

Fig. 4. Example specification of initialization procedures

Variable Assignment
In section 3 test cases were defined by generating values for each specified variable of the unit under test. 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 coded in the implementation of the system. For example, a variable that is private to an object[2] 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.

[2] In terms of Object Oriented programming.

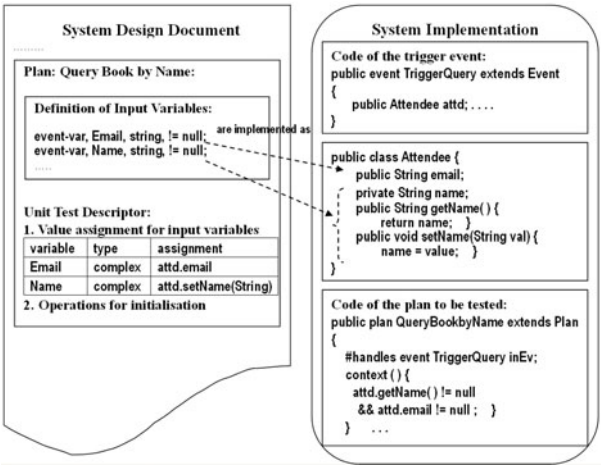

Fig. 5. Illustration of complex variables

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, which could be directly set, or a private variable that is set via a public mutator function. In general, if no assignment relation is specified for a variable, the assumed default is a simple variable that is publicly accessible.

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. For example, in Figure 5, the variables $Email$ and $Name$: are attributes of the $attd$ object of type $Attendee$ in the triggering event of the plan. The assignment relation for the $Email$ variable is $attd.email$ as it is a public attribute of that object and can be set directly. The $Name$: however, has to be set via the $att.setName(String)$ method.

Belief variables are fields of a particular belief-set. For example:

[belief-var, (StockDB, bookID), int, $==1$]
[belief-var, (StockDB, inStock), int, >0 , <5]

They do not require an assignment function 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 belief variables in concern and random values for the rest of the fields for that record. For example, in Figure 6 the first test case is setup by creating and inserting a record for the $StockDB$ beliefset with fields $bookID$ and $inStock$ set to 1

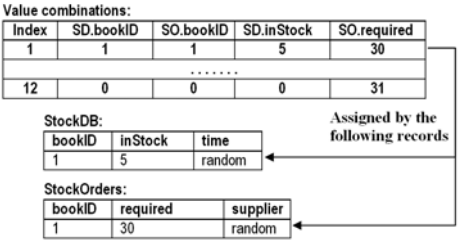

Fig. 6. Illustration of belief variables

and 5 respectively, and creating and inserting a record for the $StockOrders$ beliefset with fields $bookID$ and $required$ set to set to 1 and 30 respectively[3].

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 specify additional test cases (value combinations) prior to executing the tests.

A special case of belief variables is when historical information is required. For example, if the context condition of the plan contains the following:

StockDB.getStockAt(t_1) $>$ *StockDB.getStockAt(t_2)*
the design document will contain:
[belief-var, StockDB, numberInStock, int, >0].

If we follow the approach above, value combinations will be generated for $numberInStock$ but only one record will be inserted per test case, which would be insufficient to evaluate that particular context. In the general case, this situation is where the belief-set is to be populated with multiple rows. There are two alternatives for testing such situations in our testing framework.

The first is for the user to specify a test case manually for this belief. When specifying a test case manually for a belief, the user is given the option to add as many rows as desired. This approach however could be tedious if many rows are to be inserted. The second approach is to provide a method for populating the database as an initialization procedure of the unit under test.

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.

One way in which function variables could be tested is if the user specified all non local variables within the function that determine the return value in the design document and value combinations are generated for these variables. Another approach would be to augment the system source code to replace the call to the function by another variable whose values can be set for testing purposes.

[3] "SD" and "SO" are the abbreviated names of "StockDB" and "StockOrders" respectively, and *random* indicates that the value is a generated randomly.

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 desired.

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 particularly 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 [8, p.148], [13, p.963]. We call such a test stub as a *Mock Agent*.

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 dependent on the design specification: (i) 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; (ii) 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.

[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.

- 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.

5 Implementation

The testing tool and approach described has been implemented within PDT, relying on the implemented agent system being in JACK. The implementation which does code augmentation for testing also relies on the code having a particular structure, and so requires that code skeletons are generated via PDT. The design descriptors used are those that are part of the normal design process. However, additional testing descriptors have been added, in order to map design variables to implementation variables, and to specify details of initialization that is necessary for execution.

An initial validation of the testing tool has been done using the case study of the *Electronic Bookstore* system as described in [14]. The *Stock Manager* agent was implemented according to the design specified, and then deliberately seeded with a range of faults, such as failing to post an expected message. For example, the *Stock Manager* agent is designed with the plan *Out of stock response* which posts the subtask *Decide supplier*, if the default supplier is out of stock and the number of books ordered is not greater than 100. The code was modified so that the condition check never returned true, resulting in the *Decide supplier* subtask never being posted. Examples of all faults which the testing system would be expected to recognize, were introduced into the *Stock Manager* agent. The testing framework generated 252 test cases using the "Normal" level of value generation and "Extended" combinations of values, as described in section 3. As could be expected, all seeded faults were found by the tool.

We are currently in the process of doing a more thorough evaluation, using a number of programs with design models, developed as part of a class on Agent Programming and Design. Once this is completed we will be able to comment on the kind of faults identified and the relationship of these to errors identified during marking the assignments. We have also evaluated the testing system on a demonstration program developed for teaching purposes. In this program 22 units were tested with 208 automatically generated test cases. This uncovered 2 errors where messages were not posted as specified in the design (apparently due to unfinished coding), and one error where invalid data caused an exception to be thrown. This program was developed as an exemplar and had been tested by its developer. Consequently the results of this testing process do indicate that the generation and execution of automated test cases, based on the design models, was effective in uncovering bugs in the system.

6 Discussion and Conclusion

In this paper we have presented an approach to testing of agent systems, which relies on unit testing with extensive coverage, and on design models as input to the analysis

of correct vs faulty behaviour. This work is based on the testing framework described in our previous work [3], where, we define the units to be tested, the scope of testing (the type of faults) and outline the testing process, providing some detailed mechanisms such as the algorithms for determining the order in which the units are to be tested.

In this work we provide the specific details of how variables are specified, extracted and assigned values to create test cases, and mechanisms for setting up the environment to execute the test cases automatically.

The approach presented is an unit testing approach to goal-plan based agent systems. The approach and mechanisms are implemented in PDT and JACK, and hence some of the examples that we have illustrated are PDT/JACK specific. However, we note that the approach, concepts, algorithms and processes can be adopted to any goal-plan based agent system. The design tool should allow for the information required to be specified and design time and automatically extracted at the testing phase. The code augmentation and monitoring would need to be implemented in the agent programming language as we have done in JACK following the algorithms and processes detailed in this paper and our previous work [3].

Different approaches to model based testing have focused on different kinds of models, which are then used to generate certain kinds of test cases. For example Apfelbaum and Doyle [15] describe model based testing focusing on use case scenarios defined by sequences of actions and paths through the code, which are then used to generate the test cases. This kind of testing is similar to integration testing or acceptance testing, and is something we would eventually hope to add to our system. Others (e.g. [16]) focus on models that specify correct input and output data, but these are not so appropriate for testing of complex behaviour models.

Binder [13, p.111] has summarized the elements of UML diagrams, exploring how these elements can be used for test design and how to develop UML models with sufficient information to produce test cases. He developed a range of testability extensions for each kind of UML diagram where such is needed for test generation. What we have done is somewhat similar to this for the basic elements of agent systems.

Besides Prometheus, various agent system development methodologies such as Tropos [17], MASE [18] and others, also have well developed structured models that could be used in a similar way to the PDT models used in this work. Several agent platforms do already use design models for some level of assistance in generation of test cases.

For example, the eCAT system associated with Tropos [19,20,21,22] is a testing tool that applies a goal oriented testing approach, which is based on the goal model of an agent based system. Such a goal model specifies the goal hierarchy of the system, describing how goals are decomposed to elementary goals. It is supplemented with ontology information for the purpose of automated test input generation [22], Positive test cases and negative test cases are then derived from the goal model to respectively verify the fulfillment (or not) of every elementary goal. However, the eCAT system does not explore internal components within an agent such as events and plans and does not verify if they work as expected.

Knublauch [23] introduces a set of APIs that are extended from the JUNIT testing framework [24], and explores an approach to test agent systems based on the design model of the Gaia methodology using these APIs. Human developers can use these

APIs to develop test cases that are automatically executed. However, test cases are still required to be manually generated. Also, this approach does not explore agents' internal units.

Other agent development systems also have their testing support subsystems. Coelho et.al [25] develop the JAT framework for testing the agents developed by the JADE [26] platform. They have defined a fault model that specifies a set of fault types relevant to general agent features. The testing framework provides skeleton code for human developer to manually develop test cases based on the fault model. A *tester agent* is used in the testing framework to automate the execution of test cases. Tiryaki develops the SUNIT [27] testing framework, which tests the agent systems developed by the Seagent platform [28]. SUNIT, which is also on top of JUNIT, specifies a set of APIs for human developers to manually generate test cases based on the SEAGENT model of an agent system. The SUNIT framework also explores internal features of an agent, such as the plan-level structure hierarchy of an agent and actions performed by plans.

To our knowledge, our testing tool is the only agent testing system which focuses on unit testing internal components of an agent, and also fully automates the generation of test cases as well as the running of them.

We believe that comprehensive automated unit testing is a critical first step in thorough testing of agent systems. However it is limited in what it tests for. Yet, the fact that the tool managed to find errors in a carefully developed and manually tested system, is an indication that the approach followed is already valuable.

In future work we expect to add testing related to the satisfactory following of scenarios as specified during requirements, as well as testing of agent interactions. We would expect to be able to adapt the debugging work of [6] using protocol specifications for the agent interaction testing and also refer to other research on interactions in agent systems (such as [29]).

References

1. Benfield, S.S., Hendrickson, J., Galanti, D.: Making a strong business case for multiagent technology. In: Proceedings of AAMAS 2006, pp. 10–15. ACM, New York (2006)
2. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, Chichester (2004)
3. Zhang, Z., Thangarajah, J., Padgham, L.: Automated Unit Testing For Agent Systems. In: 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Spain, pp. 10–18 (July 2007)
4. Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: Lesser, V. (ed.) The First International Conference on Multi-Agent Systems, San Francisco, pp. 312–319 (1995)
5. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents — Components for Intelligent Agents in Java. AgentLink News (2) (1999)
6. Poutakidis, D., Padgham, L., Winikoff, M.: An Exploration of Bugs and Debugging in Multi-agent Systems. In: Proceedings of AAMAS 2003, pp. 1100–1101. ACM, NY (2003)
7. Paton, N.W., Díaz, O.: Active Database Systems. ACM Comput. Surv. 31(1), 63–103 (1999)
8. Burnstein, I.: Practical Software Testing. Springer, New York (2002)
9. Jorgensen, P.C.: Software Testing: A Craftsman's Approach, 2nd edn. CRC Press, Boca Raton (2002)

10. Copeland, L.: A Practitioner's Guide to Software Test Design. Artech House, Inc., Norwood (2003)
11. Myers, G.J., Sandler, C., Badgett, T., Thomas, T.M.: The Art of Software Testing, 2nd edn. Wiley, Chichester (2004)
12. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An Approach to Testing Based on Combinatiorial Design. Software Engineering 23(7), 437–444 (1997)
13. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
14. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A practical guide. John Wiley and Sons, Chichester (2004)
15. Apfelbaum, L., Doyle, J.: Model Based Testing. In: The 10th International Software Quality Week Conference, CA, USA (1997)
16. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Bellcore, B.: Model-Based Testing in Practice. In: International Conference on Software Engineering (1999)
17. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
18. DeLoach, S.A.: Analysis and Design using MaSE and agentTool. In: Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference, Oxford, Ohio (2001)
19. Nguyen, C.D., Perini, A., Tonella, P.: A goal-oriented software testing methodology. Technical report, ITC-irst (2006), `http://sra.itc.it/images/sepapers/gost-techreport.pdf`
20. Nguyen, C.D., Perini, A., Tonella, P.: Automated continuous testing of multi-agent systems (December 2007)
21. Nguyen, C.D., Perini, A., Tonella, P.: Experimental evaluation of ontology-based test generation for multi-agent systems, pp. 187–198 (2009)
22. Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based Test Generation for Multi-agent Systems. In: Proceedings of AAMAS 2008, pp. 1315–1320 (2008)
23. Knublauch, H.: Extreme programming of multi-agent systems (2002)
24. Husted, T., Massol, V.: JUnit in Action. Manning Publications Co. (2003)
25. Coelho, R., Kulesza, U., von Staa, A., Lucena, C.: 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, pp. 83–90 (2006)
26. Bellifemine, F., Poggi, A., Rimassa, G.: JADE: a FIPA2000 Compliant Agent Development Environment. In: Proceedings of Agents Fifth International Conference on Autonomous Agents, pp. 216–217 (2001)
27. Tiryaki, A.M., Öztuna, S., Dikenelli, O., Erdur, R.C.: SUNIT: A unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, vol. 4405, pp. 156–173. Springer, Heidelberg (2007)
28. Dikenelli, O.: SEAGENT MAS Platform Development Environment. In: Proceedings of AAMAS 2008, pp. 1671–1672 (2008)
29. EI Fallah-Seghrouchni, A., Haddad, S., Mazouzi, H.: A formal study of interactions in multi-agent systems. In: Garijo, F.J., Boman, M. (eds.) MAAMAW 1999. LNCS, vol. 1647. Springer, Heidelberg (1999)