Model-based Test Oracle Generation for
Automated Unit Testing of Agent Systems

Lin Padgham1, Zhiyong Zhang1, John Thangarajah1 and Tim Miller2
1. RMIT University, Melbourne, Australia.
2. The University of Melbourne, Melbourne, Australia
{lin.padgham, zhiyong.zhang, john.thangarajah}@rmit.edu.au, tmiller@unimelb.edu.au

Abstract—Software testing remains the most widely used approach to verification in industry today, consuming between 30-50% of the entire development cost. Test input selection for intelligent agents presents a problem due to the very fact that the agents are intended to operate robustly under conditions in which developers did not consider, and would therefore be unlikely to test. Using methods to automatically generate and execute tests is one way to provide coverage of many conditions without significantly increasing cost. However, one problem using automatic generation and execution of tests is the oracle problem: how can we automatically decide if observed program behaviour is correct with respect to its specification.

In this paper, we present a model-based oracle generation method for unit testing belief-desire-intention (BDI) agents. We develop a fault model based on the features of the core units to capture the types of faults that may be encountered, and define how to automatically generate a partial, passive oracle from the agent design models. We evaluate both the fault model and the oracle generation by testing 14 agent systems. Over 400 issues were raised, and these were analysed to ascertain whether they represented genuine faults or were false positives. We found that over 70% of issues raised were indicative of problems in either the design or the code. Of the 19 checks performed by our oracle, faults were found by all but 5 of these checks. We also found that 8 out the 11 fault types identified in our fault model exhibited at least one fault. The evaluation indicates that the fault model is a productive conceptualisation of the problems to be expected in agent unit testing, and that the oracle is able to find a substantial number of such faults with relatively small overhead in terms of false positives.

Index Terms—test oracles, unit testing, BDI agents

1 INTRODUCTION

There is increasing use of agent technology for building complex systems that operate in dynamic environments. Recent studies show average productivity gains of over 300% in developing such systems using Belief Desire Intention (BDI) agents [1], a particular paradigm of agent systems. This is generally attributed to the agent paradigm, and associated execution engine, being more natural and intuitive than other paradigms, such as object orientation, for capturing, the kind of goal oriented behaviour desired.

A key feature of agent systems is that they provide flexible solutions to complex problems that demand it. Padgham and Winikoff [20, page 16], show that there may be over 2 million ways to achieve a goal even from a very modest plan and goal structure. Further, agent systems are often required to be robust under unpredictable situations. The very nature of such systems implies that such unpredictable situations will not explicitly considered to be tested by system developers. This flexibility, non-determinism, and robustness means that manual test input selection is unlikely to yield sufficiently good results with regards to finding faults.

It is well accepted that testing software systems is a critical aspect of software development. The National Institute of Standards and Technology claims that inadequate software testing costs the U.S.A. up to $59.5 billion dollars each year, representing just under 1% of the country’s Gross Domestic Product [26]. Testing is also time consuming and estimates are that in general, testing consumes between 30-50% of the software development life-cycle.

Automated test case generation is one way to provide comprehensive testing of software systems. In Section 2, we describe our previous work on automatically generating tests for BDI agent units. However, executing large suites of test inputs suffers from the oracle problem. That is, for each test, how do we determine whether the behaviour shown by the system satisfies its specification? Most complete and sound solutions involve implementing (simplified) an alternative version of the system under test – which can be expensive, and also prone to errors itself.

While certain principles can be generalised and adopted from model-based testing of object-oriented and component-based systems [2], there are aspects that are particular to agent systems that require an understanding of their structure and semantics. In Section 3, we present a fault model that defines the types of faults that may be identified with respect to particular units.

In Section 4, we present an approach for model-
based test oracle generation for unit testing in BDI agent systems, which aims to find the faults identified in the fault model. The oracle performs a series of checks on the unit under test, comparing the observed behaviour to design models. The models are partial models of behaviour, and therefore, our oracles are themselves partial.

In Section 5, we evaluate our approach by using our framework to test an exemplar program developed for demonstration of the Prometheus methodology and design tool, plus 13 programs developed by final year undergraduate students in an agent-oriented design and programming course. Although these systems were all manually tested by their authors, and the student systems had been marked by staff, a substantial number of new faults were discovered. We found that most of the types of faults defined in our fault model were also found in practice. We also found that the model-based oracles were highly effective, both discovering a high number of faults, and producing under 30% false positive notifications.

2 TEST FRAMEWORK

In this section, we provide an overview of the test framework (described in more detail in [31]), in which our oracle resides. We note that once the design documents have had testing descriptors added to capture those aspects of implementation necessary for test input generation and appropriate code augmentation, the testing process is automatic, supporting the generation, execution and reporting of comprehensive test suites without involvement from the test engineer.

The system model that we use as the oracle consists of the artifacts produced during the detailed design phase of the Prometheus methodology. These are the overview diagrams of agents and their capabilities, along with the detailed descriptors of the internal components. It is of course the case that errors found may be the result of errors in the design documents, which are corrected in the implementation. However, in the interests of maintaining up to date and correct documentation of a system, it is as important to correct these as to correct implementation errors.

2.1 Architectural Overview

The test framework has four different components as shown in Figure 1. The design documents of the system under test (SUT) are used to extract information for building test harnesses, for determining the testing order of units and for generation test oracles. The system code is augmented to collect the necessary information for test case analysis. A test harness is automatically produced for each unit, executes all test inputs, and collects the relevant information. The information collected for a unit is then analysed to produce a report that identifies any errors detected for that unit. The report provides both a summary, and details. Figure 2 shows the process of testing the SUT as described above. Each test harness includes a test agent that initiates each test case, collects the results, and analyses the results for the report.

Figure 3 shows the basic process conducted by a test harness for testing one particular unit, in this case a plan. The process begins with the test agent generating an activation message, which then triggers the particular test driver, which has been (automatically) constructed for testing this unit. It sends a triggering event to the unit under test (the plan and its substructure). The plan, which has had its code automatically augmented by the test framework’s related tools, then sends back observations to the test agent. When the execution is finished the test agent logs all observations for later analysis.

2.2 Test input generation

Figure 4 shows the process of generating test inputs for the unit under test. Inputs are generated for each unit by extracting the relevant variables from the design documents and generating variable value combinations.

When testing a plan, the relevant variables are those referenced in the context condition or body of the
Variables are specified as a tuple \([\text{scope}, \text{name}, \text{type}, \text{info}]\) where:

- **scope**: one of \(\text{agent-var}, \text{event-var}, \text{belief-var}, \text{capability-var}\) or \(\text{system-var}\), indicating the scope of the variable.
- **name**: a string variable identifier.
- **type**: the type of the variable. Our framework handles base types \(\text{integer}, \text{float}, \text{string}, \text{enumerated}\), and \(\text{array}\).
- **info**: for all types other than an array, this is the valid input domain of the variable. For an array, this is a tuple \(\langle\text{size}, \text{element-type}, \text{domain}\rangle\), which specifies the size of the array, the type of the elements, and the valid input domain of each element respectively.

The descriptors can also contain statements indicating relationships between variables that are expected to hold, to allow generation of both valid and invalid test case input. For full details regarding the representation and use of variable information in generating test case input, refer to [30].

We use the standard notions of *equivalence class partitioning* and *boundary-value analysis* [5, p.67] to generate a limited set of values for each relevant variable. Once test values have been chosen for each variable, these are combined to produce test inputs. We use *combinatorial design* [9] to generate a set of value combinations that cover all \(n\)-wise \((n \geq 2)\) interactions among the test parameters and their values. We use the *CTS* (Combinatorial Testing Service) software library of Hartman and Raskin\(^1\), which implements this approach. However, by default we apply this reduction only to test inputs involving invalid values. Where possible we support the generation of all combinations of valid values, on the assumption that unpredictable behaviour results from interactions between valid variable values. However, as this can produce an unmanageably large number of test inputs, we allow the test engineer to specify different levels of thoroughness based on the number of values for each variable. We can also use combinatorial reduction to reduce the numbers of combinations for valid variables, but attempt first to reduce the number of test values. We also allow the test engineer to specify additional test inputs using domain and design knowledge.

Figure 5 is an example of the *Test Case Input* window for a plan that has three variables, \(\text{BookID}, \text{NumberOrdered}\) and \(\text{Urgent}\). The test engineer can edit the values in the text box next to the variables and add or remove test inputs as needed.

### 2.3 Test input execution

To execute tests, several setup and managerial tasks may be required. Our framework allows the test engineer to implement and register hooks to support automated execution. These are managed in an automated manner given the testing specific information specified by the test engineer in the *unit test descriptor* of the relevant unit. 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. Our framework supports the following hooks:

\(^1\) http://www.alphaworks.ibm.com/tech/cts (obtained July 2006, technology now retired)
1) **Initialisation**: These include setup tasks such as setting up connections to external servers, populating databases, or initialising global variables.

2) **Variable assignment**: These inform the test framework how to assign variables using a unit’s mutator methods to set up tests. That is, they specify a fragment of code to assign values to a unit as part of a test.

   A simple variable is a public or private variable that is set/get via public mutator methods. Our framework handles these automatically, but can be overridden manually. A complex variable is one that is part of a nested structure, such as an attribute of an object. These require custom code to refine the abstract test inputs into concrete test inputs.

3) **External interaction**: These allow the test engineer to isolate the unit under test by providing test stubs or mock agents to represent the entities external to the unit; for example, external systems or other agents within the system.

   If the interaction is with another agent in the same system, then the form of interaction is known via protocols in the design documents (see Figure 6), making it possible to simulate and control this interaction.

   Our framework automatically generates mock agents to work at the interaction level, but does not realise their internal logic. The test engineer can implement these to provide specific logic if desired.

## 3 Test Coverage and Fault Model

### 3.1 Testable units

The class of agent systems we have focussed on are the popular BDI agent systems, for which there are a number of implementation platforms [3], [22], [28], as well as a number of design tools [4], [20], [10]. The implementation platform we have used is JACK² [28], while the design tool and related artefacts are those produced by the Prometheus methodology [20] and the related Prometheus Design Tool³, which generates JACK skeleton code from design models. However, the approach is general and could be applied to any of the well known agent implementation platforms using the typical agent entities of agents, events and/or goals, plans and beliefs (such as Jason [3] or Jadex [22]), and any of the agent design tools (e.g. OMASE [10] or Tropos [4]), which produce similar models to PDT. Prometheus is a popular agent oriented software development methodology that has been developed over the last 14 years and is used extensively in academia and to some extent by industry. JACK is a well established commercial agent development platform, which has been used for large complex agent systems, including the high-level control for unmanned aerial vehicles and large logistics applications.

The key components of BDI-style agents are: (i) plans, which are predefined recipes that specify executable steps for achieving particular goals; (ii) events, which can represent messages, goals, subgoals, or percepts from the environment; and (iii) beliefs, which capture the agent’s knowledge about the state of the world. The agent may also have capabilities that encapsulate subsets of plans, beliefs and events, for modularity. Figure 7 shows this basic view of an agent and its components.

Plans (or rather plan templates) are stored in a plan library and consist of the trigger, which is the event type for which this plan is relevant, the context condition, which describes situations where this plan is a viable choice, and a body, which describes the subgoals that may be posted, the messages that may be sent, and any actions to be performed. For example,

---


Event and plan libraries can be visualized as an and/or tree where each event has as its children some number of plans, one of which must be executed in order to respond to that event; and each plan has as its children some number of events which are the subgoals or messages which will/may be posted when executing that plan. These event-plan trees are part of the detailed design specification of the internals of an agent. An example is shown in Figure 8.

Agent beliefs, in addition to determining the applicability of plans may also automatically generate events when modified, thereby invoking plan execution. For example, if the agent sells the last book in stock and the corresponding stock level beliefs are updated, this may cause a goal event to reorder-stock to be posted.

The testable features specify a set of characteristics which we can observe during testing. If we find that a feature specified in the design is not exhibited by the implementation, or that a feature that has been exhibited by the implemented is not specified in the design, then we say that there is a fault. This fault can be in either the design or the implementation.

We define the following fault model for BDI agents, based on the testable units identified in Section 3.1. In total, we specify 11 different fault types: three for events, and four each for beliefs and plans. These fault types represent an enumeration of the possible ways in which single units in BDI programs can be faulty with respect to their design model. Because the design models are partial, certain fault types are omitted, such as agents incorrectly calculating output values.

### 3.2 Test coverage

It is possible that some parts of a unit will not be tested by a test suite. Test coverage criteria (or adequacy criteria) have been proposed in the literature [17] to address this for structured programs. BDI agent units are somewhat different to normal structured programs, so the notion of test coverage is different.

In this work, we consider that a testable unit is covered by a set of test inputs if and only if:

1. For each plan, at least one test input causes its triggering event to occur.
2. For each event, at least one test input causes each handling plan of that event to be executed.
3. For a context condition, at least one test input causes this context condition to evaluate to TRUE, and at least one test input causes it to evaluate to FALSE.
4. For each event specified as part of a plan, at least one test input causes this event to be posted.

### 3.3 Fault Model

The testable features specify a set of characteristics which we can observe during testing. If we find that a feature specified in the design is not exhibited by the implementation, or that a feature that has been exhibited by the implemented is not specified in the design, then we say that there is a fault. This fault can be in either the design or the implementation.

We define the following fault model for BDI agents, based on the testable units identified in Section 3.1. In total, we specify 11 different fault types: three for events, and four each for beliefs and plans. These fault types represent an enumeration of the possible ways in which single units in BDI programs can be faulty with respect to their design model. Because the design models are partial, certain fault types are omitted, such as agents incorrectly calculating output values.

### Plans

The structure of a plan can be described by its triggering event, its context condition and its plan body. The triggering event activates the plan for consideration, the context condition determines applicability depending on beliefs, and the plan body indicates what is done, including particular subtasks generated, or messages sent (both of which are represented as events). The set of faults that can occur related to plans are:
1.1 **Incorrect triggering event**
Each plan is triggered by exactly one event type. If the event specified as the trigger in the design and in the implementation are different types, then the plan may be evaluated for events that it should not be, or may not be evaluated for events that it should be; both of which we consider a fault.

1.2 **Incorrect context condition**
The context condition, if present, specifies when a plan should be considered applicable. If it is not present (or is simply TRUE) then the plan should always be applicable. If the context condition is implemented incorrectly with respect to the design, then the plan may become applicable under conditions that it should not be, or may not be applicable under conditions that it should be; both of which we consider a fault.

1.3 **Incorrect event posted**
Agent designs specify which events will be posted. A plan fails to satisfy its design if an unspecified event is posted, or a specified event is not posted.

1.4 **Plan failure**
The plan fails to complete.

**Events**
In PDT, events can be annotated in the design models with the following two properties:

1) If an event is annotated as complete, then whenever it occurs, at least one plan is applicable.
2) If an event is annotated as overlapping, then it is possible that, when it occurs, more than one plan is applicable.

Unintended overlap or incompleteness are common causes of faults in agent systems, due to incorrect specification/implementation of context conditions [21, page 22].

The set of faults that can occur related to events are:

2.1 **Unintended incompleteness**
Events are specified as complete/incomplete. For a complete event, if there is a situation where this event occurs and no plan is applicable, this is an error. For an incomplete event, the lack of applicable plan may not be an error, but it is not possible to ascertain whether the situation is that in which incompleteness was intended.

2.2 **Unintended overlap**
Events are specified as overlapping/not overlapping. For a non-overlapping event, if there is a situation where this event occurs and more than one plan is applicable, this is an error. For an overlapping event, multiple applicable plans may not be an error, but it is not possible to ascertain whether the situation is that in which overlapping was intended.

2.3 **Redundant plan**
An event is specified as a triggering condition for that plan, but the plan will never be executed. That is, there is always another plan that will be executed for this plan’s triggering event and context condition. This fault type is included under the assumption that such a plan is redundant and may indicate a larger problem.

**Beliefs**
Beliefs are implemented somewhat differently in different agent platforms, and are also handled less uniformly than plans and events in design tools and methodologies. In JACK, beliefs are represented as a relational database table where each field has a name, a type, and a flag to indicate whether or not it is part of the key. As PDT includes generation to JACK skeleton code, it allows specification of this information in the form of a set of tuples as shown in Figure 9.

**Fig. 9. Data fields of the Book DB Beliefset in PDT**

In a number of agent platforms, including JACK, it is also possible to specify that certain changes to a belief will generate a particular event (similar in many ways to active databases). PDT allows for design specification of a set of action rules for any beliefset of the form \(<operation, event>\), where operation is one of insert, delete, update and modify\(^5\), and event is the event type posted when the specified operation succeeds. The event posting behavior is implemented in different ways in different platforms, such as the use of listener methods in Jadex, and the use of callback methods in JACK, where the method is automatically invoked when the relevant change occurs.

JACK provides six different callback methods, which, if defined, are automatically invoked when relevant belief operations are applied. However, two of these callback methods (addfact() and delfact()) are activated when the relevant operation is attempted,

\(^4\) More exactly, the plan will never be executed as an agent’s first choice. It could be executed in a situation where the preferred plan had inexplicably failed, even though its context condition remained true.

\(^5\) Modify is a generalization of insert, delete, and update.
even if it does not succeed. (We call these attempt-triggered methods.) Use of an attempt triggered callback method can lead to errors of events being posted without the relevant operation being successful.

With respect to beliefs, there are four fault types:

3.1 Incorrect belief structure
The implemented structure of the belief is inconsistent with the design.

3.2 Event posted when operation failed
Upon an operation being attempted but failing, the event specified to be posted on success, is still posted.

3.3 Event not posted when operation succeeded
Upon an operation being successfully applied to a belief, the event specified is not posted.

3.4 Incorrect event posted
Upon an operation being applied to a belief, an incorrect event is posted.

4 TEST ORACLE GENERATION

In this section, we present our approach for automated oracle generation. Due to the partial design models and the nature of agent systems, the fault types identified in Section 3.3 are different to fault types seen in structured programs. For example, a redundant plan cannot be detected by executing a single test, and in fact, would require exhaustive testing to conclude that the plan is redundant.

To detect such cases, our oracle collates information over all test inputs, rather than single tests; although some faults can be discovered with single tests. In the example above, if none of the executed tests trigger a specific plan, our oracle reports a warning rather than error, due to the fact that it may not be a fault. Instead, it could be that such a test exists, but was not executed. However, in this case, and several other fault types, if none of the tests trigger the plan, we know that the reason must be because either: (a) the plan is redundant; or (b) the test coverage is insufficient. As a result, a side effect of this check is that it may detect test coverage problems. While we use warning messages in these uncertain cases, some of these warnings are true positives, but the oracle cannot determine whether the problem is an inconsistency between the design and implementation, or whether it is due to insufficient test coverage. The warning messages produced in such cases identify both possibilities. These cases are explicitly identified in this section.

Two of the checks made by the oracle on belief units are performed statically; that is, not using test inputs. These static checks are straightforward to automate. All other checks are done dynamically, using a range of test cases, although in principle some of them could also be done statically.

4.1 Oracle data
To make observations that can then be checked with the test oracle, the program code is augmented to send certain information to the test driver for later analysis using the oracle.

When unit testing a plan, the code of that plan is augmented to:
- observe that the plan is triggered, and which event it is triggered by;
- observe the value of the context condition (TRUE or FALSE); and
- observe the events posted by the plan.

In addition, the handling plans of any events the plan is supposed to post are augmented to observe their triggering (as evidence of event posting).

When unit testing an event, the code of all handling plans are augmented to:
- report the value of the context condition to the test agent; and
- report the start of execution of the selected plan to the test agent.

When unit testing a belief:
- the test driver queries the value of the belief before and after executing each specified action rule;
- the code of the belief method is augmented to report events posted; and
- the code of all plans handling events posted from the belief are augmented to report triggering of a plan by an event.

As an example of the code augmentation performed, consider the case of sending the value of the context condition for a plan to the test agent. Context condition values are not directly observable. They also potentially contain logical variables which are bound during evaluation. To obtain the value of the context condition, we augment the JACK context() method, which evaluates the context condition, to send the value of the context condition to the test agent. An example augmentation is shown in Figure 10. In this figure, the context condition expression, CC_Expression is replaced with a conditional expression that passes the value of the context condition (TRUE or FALSE) as a parameter to the BIT_log_CC method. This method sends the value to the test agent, and returns the value again, preserving the semantics of the original context() method.

The observations obtained during testing are then checked with the oracle either for consistency with the design, or for behaviour consistent with a well behaving BDI agent program, and faults identified are categorised and reported.

4.2 Fault categorisation
Our oracle categorises fault types into three different levels according to severity:
public plan Plan_Under_Test extends Plan {
    ...
    context() {
        /* the CC is encapsulated as a parameter */
        (CCExpression)?BIT_log_CC(true):BIT_log_CC(false);
    }
    boolean BIT_log_CC(boolean CC_result) {
        /* report the CC value to the test agent */
        inform_testagent_CC(CC_result);
        return CC_result;
    }
    ...
}

Fig. 10. Test Code for Reporting on The CC Value of The Plan under Test

1) Level 1 is an exception that is thrown by the system. These are not based on the design, but must be caught and information provided to the tester. Our oracle does not implement system-specific functionality to check these.

2) Level 2 is an error — an inconsistency between design and implementation.

3) Level 3 is a warning, indicating something that may be caused by a fault, and should be reviewed.

Many level 3 (warning) fault types are indicative of either a fault or inadequate coverage by the test suite. That is, if we could run exhaustive testing on a unit, then some level 3 warnings could be considered as level 2 errors. As such, the oracle failure notifications need to identify the nature of the possible fault, as well as the nature of the possible coverage issue. Such cases are identified clearly in this section.

Other level 3 warning types that do not result from inadequate test coverage can be “switched off”, allowing the user to indicate that it is not a problem, and the warning will no longer be generated for the specified unit and set of tests inputs. In cases of inadequate test coverage, a test case can be manually added.

4.3 Fault detection

In this section, we present the checks that are performed by the oracle to attempt to detect the fault types identified in Section 3.3.

Plans

Table 1 presents an overview of the checks that are made by the oracle for the plan-related fault types.

For fault type 1.1 (incorrect triggering event), the oracle checks that, if the triggering event specified in the design occurs, the plan is triggered (that is, the plan’s context condition is evaluated), and that it is triggered by the specified event. It can occur that some other plan responds to the specified triggering event, and in turn this plan posts an event that triggers the plan being tested. Thus, it is important to check not only that the plan is triggered, but that it is triggered by the specified event.

For fault type 1.2 (incorrect context condition), three checks are performed. For the first check, if the context condition is not specified (equivalent to TRUE) then the plan should always be applicable. If it is not applicable for at least one test case, an error is raised.

In Prometheus design models, context conditions are collections of variables and types, with a textual description of the condition, rather than logical expressions that can be evaluated. Consequently it is not possible to detect if the evaluation of the context condition by the implementation is correct. However, if a context condition exists, then we should expect that it will evaluate to TRUE/FALSE in at least some cases. The latter two checks each determine whether a context condition is true for every test or false for every test respectively. In such cases, either the context condition always evaluates to true/false, which is considered a fault, or the test set is inadequate, as one of the true/false cases has not been covered by the tests.

For fault type 1.3 (incorrect event posted), two checks are performed: (1) If the plan never posts a specified event, a warning is raised, indicating a failure to post the event, or that the test set is inadequate. (2) If an unspecified event is posted, a warning is raised. Although an unspecified event indicates a definite deviation from the design, our experience indicates that developers often use extra events for debugging or logging, which would not be shown in design. Therefore, categorising this as a warning allows developers to ignore these to prevent warning fatigue.

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Check Description</th>
<th>Failure level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The plan is not triggered by its triggering event as specified</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>The plan has no context condition (equivalent to true), but is not applicable (context condition evaluates to false) at least once when its specified triggering event occurs</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1</td>
<td>The plan has a context condition, but is not applicable (context condition evaluates to false) every time its specified triggering event occurs</td>
<td>3*</td>
</tr>
<tr>
<td>1.2.2</td>
<td>The plan has a context condition, but is applicable (CC evaluates to true) every time its specified triggering event occurs</td>
<td>3*</td>
</tr>
<tr>
<td>1.3</td>
<td>A specified outgoing event is never posted</td>
<td>3</td>
</tr>
<tr>
<td>1.3.1</td>
<td>A specified outgoing event is never posted</td>
<td>3*</td>
</tr>
<tr>
<td>1.3.2</td>
<td>A message posted at runtime by the plan is not specified as an outgoing event in the design</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>The plan fails to complete for at least one test case</td>
<td>2</td>
</tr>
</tbody>
</table>

* Due to possibility of inadequate test case coverage.

TABLE 1

Oracle checks for plans
For fault type 1.4 (plan failure), the oracle checks that if a plan commences execution, it terminates successfully. If not, an error is raised.

**Events**

The checks performed by the oracle on events are outlined in Table 2:

<table>
<thead>
<tr>
<th>Fault type</th>
<th>Check Description</th>
<th>Failure Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>2.1.1 An incomplete event occurs, but no plan is applicable</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.1.2 A complete event occurs, but no plan is applicable</td>
<td>2</td>
</tr>
<tr>
<td>2.2</td>
<td>2.2.1 An overlapping event occurs, but more than one plan is applicable</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.2.2 A non-overlapping event occurs, but more than one plan is applicable</td>
<td>2</td>
</tr>
<tr>
<td>2.3</td>
<td>2.3.1 A plan that handles the event is never executed, despite the event occurring</td>
<td>3*</td>
</tr>
</tbody>
</table>

* Due to possibility of inadequate test case coverage.

**Beliefs**

Table 3 outlines the checks performed by the oracle for beliefs.

For fault type 3.1 (incorrect belief structure), three checks are made: that all specified data fields are implemented; that only the specified data fields are implemented, and that their types are correct. These checks are all performed statically, and all three raise errors if a discrepancy is found.

For fault type 3.2 (event posted when operation failed), two checks are made. The first (3.2.1) checks that there is no attempt-triggered callback method implemented for an action rule, which could allow this to happen. This only raises a warning as the developer may have implemented a check for success within the method. The second (3.2.2) checks whether, after a failed attempt to modify a belief, the specified event is still posted. If it is, an error is raised. 3.2.1 is checked statically, while 3.2.2 is checked dynamically.

For fault type 3.3 (event not posted when operation succeeds), two checks are made. Check 3.3.1 statically verifies whether a callback method is implemented for each action rule. If not, an error is raised. Check 3.3.2 dynamically verifies whether a specified event is posted when the operation succeeds. If it is not, this is treated as a warning as it is possible that the event posting is intended to be (and is) conditional, which is not captured in the design.

For fault type 3.4 (incorrect event posted), the check ascertains whether the type of the event posted following a successful operation is of the type specified in the design. If it is not, then an error is raised.

**4.4 Discussion**

In this section, we discuss some properties of our oracle.

Staats et al. [24] define several properties of oracles, including soundness and completeness. They state that an oracle is *sound* if and only if, when the oracle reports that a test passes, the program is correct with respect to its specification for *that test*. An oracle is *complete* if and only if, when the result of running a test on the program is correct with respect to its specification, the oracle reports that the test passes.

Our oracle is *unsound*, because the design models on which we base the oracles only specify partial behaviour. For example, as discussed in Section 4.3, the design does not specify context conditions as logical expressions that can be evaluated. As a result, it is not possible to determine, from a single test input, whether the implementation has evaluated a context condition consistent with the design. However, for the
properties that are checked, our oracle is sound. That is, if an error or warning is not raised for a particular test suite, then this particular property holds for that test suite.

Our oracle is complete for errors, but not warnings. That is, if an error is raised, this indicates an inconsistency between the design and implementation. Warnings are raised for some properties when the oracle cannot determine correctness, and this incompleteness is why we use warnings. In our initial design, our experimental evaluation revealed that our oracle was not complete for errors, due to a level 2 fault category (error) that turned out to be a false positive. This is discussed further in Section 6.

We also note that our oracle is passive. That is, it checks the implementation is correct against the design for a given input, rather than generating the expected behaviour and comparing this against the implementation’s behaviour. This is necessary due to the inherent non-determinism of the design models.

5 Evaluation

In this section, we present an evaluation of the fault model and automated test oracle. The goals of the evaluation are:

1) to assess whether the fault model accurately captures the types of faults found in BDI agent units;
2) to assess the effectiveness of our oracles’ ability to detect faults; and
3) to measure the number of true and false positive failures raised by the oracle.

5.1 Objects of analysis

5.1.1 CLIMA agents

The first set of objects of analysis were 13 different designs and implementations developed by students as an assignment in an agent design and programming class over two consecutive years. All students developed designs using PDT and implementations using JACK. The assignment was to develop an agent system for the (then) CLIMA agent programming competition, where a team of agents had to collect gold in a grid world and deposit it in a store, competing against a rival team. The player agents communicated with the CLIMA server to specify actions and obtain updates on perceived world state (the neighbouring cells). Students were encouraged to develop cooperation between their agents, and to include additional agents to do certain tasks (such as coordination) as well as the mandatory player agents. Although there were certain similarities between the systems, due to the underlying structure of the problem, there was also significant variability in the systems, with the number of units available for testing ranging from 29 to 85 across systems. Table 4 shows the numbers of units (that is, the combined total number of plans, events, and beliefs that were tested), number of classes, number of test cases within each of the systems, and logical lines of code (LLOC) – a line of code metric that does not count white space or comments, and is independent of coding style.

<table>
<thead>
<tr>
<th>system</th>
<th>units</th>
<th>classes</th>
<th>LLOC</th>
<th>test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIMA01</td>
<td>62</td>
<td>74</td>
<td>2781</td>
<td>16629</td>
</tr>
<tr>
<td>CLIMA02</td>
<td>52</td>
<td>83</td>
<td>2693</td>
<td>20209</td>
</tr>
<tr>
<td>CLIMA03</td>
<td>29</td>
<td>61</td>
<td>1869</td>
<td>11346</td>
</tr>
<tr>
<td>CLIMA04</td>
<td>81</td>
<td>91</td>
<td>3611</td>
<td>19350</td>
</tr>
<tr>
<td>CLIMA05</td>
<td>69</td>
<td>88</td>
<td>2979</td>
<td>14864</td>
</tr>
<tr>
<td>CLIMA06</td>
<td>72</td>
<td>91</td>
<td>3454</td>
<td>32763</td>
</tr>
<tr>
<td>CLIMA07</td>
<td>75</td>
<td>96</td>
<td>3760</td>
<td>21068</td>
</tr>
<tr>
<td>CLIMA08</td>
<td>85</td>
<td>105</td>
<td>3445</td>
<td>15320</td>
</tr>
<tr>
<td>CLIMA09</td>
<td>44</td>
<td>49</td>
<td>1369</td>
<td>428</td>
</tr>
<tr>
<td>CLIMA10</td>
<td>60</td>
<td>113</td>
<td>2890</td>
<td>19130</td>
</tr>
<tr>
<td>CLIMA11</td>
<td>70</td>
<td>79</td>
<td>2969</td>
<td>4135</td>
</tr>
<tr>
<td>CLIMA12</td>
<td>71</td>
<td>84</td>
<td>3065</td>
<td>15532</td>
</tr>
<tr>
<td>CLIMA13</td>
<td>65</td>
<td>73</td>
<td>2426</td>
<td>13258</td>
</tr>
<tr>
<td>Total</td>
<td>835</td>
<td>1087</td>
<td>37148</td>
<td>204032</td>
</tr>
<tr>
<td>Average</td>
<td>64</td>
<td>84</td>
<td>37148</td>
<td>15695</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>units</th>
<th>classes</th>
<th>LLOC</th>
<th>test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weather Alert</td>
<td>30</td>
<td>38</td>
<td>1106</td>
<td>354</td>
</tr>
</tbody>
</table>

TABLE 4 Metrics for each system

5.1.2 Weather alerting system

The second object of analysis is a system that has been developed by a research assistant as an exemplar for use in tutorials and workshops, to illustrate agent-oriented design using Prometheus. This is a prototype of a weather alerting system developed in collaboration with the Bureau of Meteorology in Australia. Due to its use as an exemplar, it had been extensively tested and reviewed, and was considered stable. We report separately on the results from testing this system.

5.2 Experimental Process

The experimental process for testing each system consists of the three following steps:

1) Setup: The system is annotated with the additional testing specific information required within the testing descriptor. This includes information such as the mapping between design variables and their implementation, as well as any initialisation procedures required in order to test a particular unit.

2) Execution: Each system is tested by the testing framework, using the automatically generated oracle and test harness for each unit. Test generation and execution was automated using the framework. Reported failures were logged and a report was generated by the test framework’s supporting tool.

3) **Fault investigation:** All failures were investigated to understand the underlying fault leading to them by reviewing the design descriptor and the implemented code of the associated unit. The underlying fault corresponding to each failure was categorised according to the following five options:

a) **incomplete/incorrect design:** a mismatch between design and code, but the design, rather than code, was incorrect;

b) **incomplete implementation:** an aspect of the design did not exist in the code;

c) **incorrect implementation:** the internal logic of a unit is faulty with respect to its design;

d) **unclassified mismatch:** there is inconsistency between design and implementation but it is not clear which is incorrect;

e) **false positives:** there is no fault, despite the failure (e.g. where an attempt-triggered callback is used, but in fact the code has checked for success); and

f) **redundant:** a failure has been recorded, but the underlying cause is the same as a previous failure that has already been analysed.

Some faults were initially detected due to incompletely or incorrectly annotated test descriptors. For example, an event variable accessed by a plan was not specified as an input variable, leading to an exception when the plan was tested as the variable was not assigned values. In such a case, the information in the test descriptor was improved and the associated unit was tested again until there were no further faults due to incomplete or incorrect annotation. Each system required between two and four rounds of annotation to eliminate these problems. More effort was required in the earlier systems and this decreased as the annotator gained experience. In most cases, test descriptor annotation was done by a researcher not involved with the development of the agent systems, and therefore was arguably more prone to missing information than if it had been provided by the developer. In four cases, students volunteered to provide testing descriptors for their own code. However, as they were inexperienced in doing this, these systems also required some updating before successful testing could occur.

An advantage of automated testing is that a much larger number of test cases can be run than if one was manually testing. Table 4 shows for each system, the number of units (i.e. beliefs + plans + events) tested, and the total number of test cases for that system.

### RESULTS

In this section, we present the results of the evaluation.

#### 6.1 Fault model

Table 5 shows, for each of the fault types in our fault model from Section 3, how many failures were recorded, and how many of these were true and false positives respectively. Of the 11 types of faults in our fault model, 9 have failures associated with them, and 8 have actual faults associated with them. This indicates that the fault model is a productive conceptualisation of the problems to be expected in BDI agent units.

<table>
<thead>
<tr>
<th>Item</th>
<th>Fault type</th>
<th># True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Plan</td>
<td>Incorrect triggering event</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Incorrect context condition</td>
<td>124</td>
<td>121</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Incorrect event posted</td>
<td>117</td>
<td>93</td>
<td>24</td>
</tr>
<tr>
<td>1.4 Plan failure</td>
<td>62</td>
<td>13</td>
<td>49</td>
</tr>
<tr>
<td>2 Event</td>
<td>Unintended incompleteness</td>
<td>33</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Unintended overlap</td>
<td>23</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>2.3 Redundant plan</td>
<td>13</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>3 Belief</td>
<td>Incorrect belief structure</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3.2 Event posted when operation failed</td>
<td>17</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Event not posted when operation succeeded</td>
<td>17</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>3.4 Incorrect event posted</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4 *</td>
<td>Exception</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**TABLE 5**

Number of failures per fault type in the fault model

#### 6.2 True positives vs. false positives

Table 6 presents a comparison of true positive failures vs. false positive failures. One can see that the majority of failures are true positives, varying from 50% to 92%. The largest number of faults discovered in a system was 39 (CLIMA02), with the least number being 6 (CLIMA09). The total number of failures across the 13 CLIMA systems was 419. Of these, 304 were true positives, with 294 unique faults (10 redundant), and 115 were false positives (no underlying fault).

In the meteorological alerting system the number of failures was 13, of which 10 reflected true positives, 2 were false positives, and 1 was redundant.

Figure 11 shows the numbers of faults in each of the six fault categories: incorrect implementation, incomplete implementation, incomplete design, unclassified mismatch between design and implementation, redundant failures, and false positives. Errors (as opposed to warnings) were intended to be cases in which there is definitely an issue to be addressed. To our surprise, when we analysed all failures, we found that 49 “errors”, were in fact intended, and arguably correct, behaviour; that is they were false positive failures. As a result, we show these false positives generating errors separately from those generating warnings.
Table 6

<table>
<thead>
<tr>
<th>System</th>
<th>Failures</th>
<th>True Positives</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIMA01</td>
<td>22</td>
<td>13 (59.09%)</td>
<td>9 (40.91%)</td>
</tr>
<tr>
<td>CLIMA02</td>
<td>45</td>
<td>39 (86.67%)</td>
<td>6 (13.33%)</td>
</tr>
<tr>
<td>CLIMA03</td>
<td>29</td>
<td>26 (89.66%)</td>
<td>3 (10.34%)</td>
</tr>
<tr>
<td>CLIMA04</td>
<td>33</td>
<td>22 (66.67%)</td>
<td>11 (33.33%)</td>
</tr>
<tr>
<td>CLIMA05</td>
<td>36</td>
<td>25 (69.44%)</td>
<td>11 (30.56%)</td>
</tr>
<tr>
<td>CLIMA06</td>
<td>29</td>
<td>23 (65.71%)</td>
<td>6 (34.29%)</td>
</tr>
<tr>
<td>CLIMA07</td>
<td>35</td>
<td>33 (94.29%)</td>
<td>2 (5.71%)</td>
</tr>
<tr>
<td>CLIMA08</td>
<td>37</td>
<td>34 (91.89%)</td>
<td>3 (8.11%)</td>
</tr>
<tr>
<td>CLIMA09</td>
<td>12</td>
<td>6 (50.00%)</td>
<td>6 (50.00%)</td>
</tr>
<tr>
<td>CLIMA10</td>
<td>39</td>
<td>36 (92.31%)</td>
<td>3 (7.69%)</td>
</tr>
<tr>
<td>CLIMA11</td>
<td>36</td>
<td>23 (63.89%)</td>
<td>13 (36.11%)</td>
</tr>
<tr>
<td>CLIMA12</td>
<td>29</td>
<td>19 (65.52%)</td>
<td>10 (34.48%)</td>
</tr>
<tr>
<td>CLIMA13</td>
<td>39</td>
<td>36 (92.31%)</td>
<td>3 (7.69%)</td>
</tr>
<tr>
<td>Total</td>
<td>419</td>
<td>304 (72.55%)</td>
<td>115 (27.45%)</td>
</tr>
<tr>
<td>Average</td>
<td>32</td>
<td>23 (72.5%)</td>
<td>9 (27.5%)</td>
</tr>
</tbody>
</table>

Weather Alert 13 Programs

<table>
<thead>
<tr>
<th>System</th>
<th>Failures</th>
<th>True Positives</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>UM</td>
<td>84</td>
<td>106 (25.3%)</td>
<td>102 (74.7%)</td>
</tr>
<tr>
<td>FPE</td>
<td>49</td>
<td>66 (15.75%)</td>
<td>7 (8.25%)</td>
</tr>
<tr>
<td>FPW</td>
<td>66</td>
<td>75 (17.9%)</td>
<td>3 (82.1%)</td>
</tr>
<tr>
<td>ID</td>
<td>102</td>
<td>106 (25.3%)</td>
<td>102 (74.7%)</td>
</tr>
<tr>
<td>II1</td>
<td>29</td>
<td>29 (6.92%)</td>
<td>20 (93.08%)</td>
</tr>
<tr>
<td>II2</td>
<td>75</td>
<td>75 (17.9%)</td>
<td>75 (82.1%)</td>
</tr>
<tr>
<td>RF</td>
<td>102</td>
<td>102 (25.3%)</td>
<td>102 (74.7%)</td>
</tr>
</tbody>
</table>

Table 7 shows the number of units in each system, and how many of these units have faults, failures (errors + warnings), or no failures.

6.3 Oracle effectiveness

Table 8 shows a detailed breakdown of each of the fault categories in Figure 11, including which oracle checks produced these failures. There are several interesting points to note. First, a large number of faults (almost 25% of all failures and around 34% (99) of faults) are due to context conditions not being specified in design. Many faults of type 1.2 may disappear.
if context condition specifications were no longer required in design. However, it should be noted that this information does provide valuable data for generating classes of test inputs, and specification of the context condition at some level is really an important aspect of design. Checks 1.2.2 and 1.2.3, which relate to context conditions not discriminating, if they are present in design, do seem to be important for identifying faults as 8 cases of incorrect implementation have been found based on check 1.2.3.

We note that false positive warnings 2.1.1 and 2.2.1 about completeness and overlap could have been avoided had we not chosen to warn of these situations even if the lack of completeness or existence of overlap had been specified. Further investigation is needed to determine whether there is any value in such warnings, as there was not evidence of such here.

The most common causes of incomplete implementation have to do with non-posting of specified events, either from plans (check 1.3.1) or from beliefs (check 3.3.1). This would seem to be that the design specified more than was implemented. Such a case would seem highly likely during implementation, and would lead to many unnecessary failures. To avoid these, a simple flag in the descriptor of units, indicating that they were not implemented, would allow recognition of this situation.

The most common implementation faults are plans failing to complete (1.4.1), plans never being executed (2.3.1), and specified context conditions being constantly invalid (1.2.3).

Of the 19 oracle checks specified as part of the oracle design in Section 4, all but 5 of them produced failure notifications. The 5 that did not were 3.1.1 (all data fields implemented), 3.1.2 (all implemented data fields exist in the design), 3.1.3 (key and type of data field correctly implemented), 3.3.1 (appropriate callback method specified), and 3.2.2 (belief does not post event in failure cases). All of these are related to belief units, which may indicate that these aspects are less error-prone than events and plans. Further investigation on a wider range of systems is needed before any such conclusions could be drawn.

Three of the checks produced only false positives: check 3.2.1 (no attempt-triggered callback methods), 2.1.1 (incomplete event has applicable plan), and 2.2.1 (overlapping events have only one applicable plan).

Most faults observed originated from plans and events rather than beliefs. Further analysis is required to ascertain whether this reflects the relative numbers of each type of unit, whether it is due to the more specialised nature of belief units, or some other reason. However, we note that 5 of the 8 belief checks are done statically, so are inexpensive to check.

### 6.4 Threats to validity

There are three main threats to external validity in our experiment. First, while 14 systems are tested, 13 of these implement the same (loosely defined) specification; therefore only two general systems are tested. Second, the scale of systems ranges up to only a few thousand logical lines of code, which is not particularly large in scale compared to systems deployed for industry use. Third, 13 of the systems evaluated were written by students, so the types of faults may not be representative of those made by more experienced software engineers. Further studies on additional systems are required to provide more generalisable results.

A threat to internal validity is the measure used: the number of faults detected. While we located and investigated faults corresponding to failures produced by the oracle, there is no record of the number of faults that remain undetected. However, we again note that all implementations had been tested by their authors, and the weather alerting system had been extensively reviewed and tested by several people. Despite this, our approach detected many new faults, indicating that is has value when complemented with systematic manual testing.

### 7 Related Work

There have been other approaches to test agent systems based on design models. Nguyen et al. describe eCAT [18]—a testing tool that supports testing agents developed following the TROPOS [4] methodology. eCAT supports testing agents as the base units (opposed to testing the internals) using TROPOS design models. It also supports testing interaction between agents using interaction-onontologies, and automatically generates test cases using ontologies to create meaningful, but varied, message content. This ontology is then used as a partial oracle to ensure that the agent under test responds appropriately. eCAT also
<table>
<thead>
<tr>
<th>Fault category</th>
<th>#</th>
<th>Oracle check used to detect fault</th>
<th>#</th>
</tr>
</thead>
</table>
| incomplete implementation         | 75  | 1.2.2 The plan has a context condition, but is not applicable (context condition evaluates to false) every time its specified triggering event occurs  
1.3.1 A specified outgoing event is never posted  
1.3.2 A message posted at runtime by the plan is not specified as an outgoing event in the design  
3.3.2 A belief fails to post the specified event after the operation is successfully applied | 5   |
| incorrect implementation          | 29  | 1.1.1 The plan is not triggered by its triggering event as specified  
2.3.1 The plan has a context condition, but is applicable (CC evaluates to true) every time its specified triggering event occurs  
2.4.1 The plan fails to complete for at least one test case  
2.1.2 A complete event occurs, but no plan is applicable  
2.1.3 A plan that handles the event is never executed, despite the event occurring | 1   |
| incomplete design                 | 106 | 1.2.1 The plan has no context condition (equivalent to true), but is not applicable (context condition evaluates to false) at least once when its specified triggering event occurs  
2.1.1 An incomplete event occurs, but no plan is applicable  
2.2.1 An overlapping event occurs, but more than one plan is applicable  
2.2.2 A non-overlapping event occurs, but more than one plan is applicable | 99  |
| unclassified mismatch             | 84  | 1.2.2 The plan has a context condition, but is not applicable (context condition evaluates to false) every time its specified triggering event occurs  
1.3.1 A specified outgoing event is never posted  
1.3.2 A message posted at runtime by the plan is not specified as an outgoing event in the design  
1.4.1 The plan fails to complete for at least one test case | 1   |
| false positives                   | 49  | 1.4.1 The plan fails to complete for at least one test case | 49  |
| false positives (warnings)        | 66  | 1.2.2 The plan has a context condition, but is not applicable (context condition evaluates to false) every time its specified triggering event occurs  
2.3.1 The plan has a context condition, but is applicable (CC evaluates to true) every time its specified triggering event occurs  
2.1.1 A complete event occurs, but no plan is applicable  
2.2.1 An overlapping event occurs, but more than one plan is applicable  
2.3.1 A plan that handles the event is never executed, despite the event occurring  
3.2.1 For an action rule, an attempt-triggered callback method is implemented | 1   |
| redundant faults                  | 10  | -- An exception occurs when the plan under test is executed (level 1 fault)  
1.3.2 A message posted at runtime by the plan is not specified as an outgoing event in the design  
2.1.2 A complete event occurs, but no plan is applicable  
2.3.1 A plan that handles the event is never executed, despite the event occurring | 6   |

**TABLE 8**

Categories of faults detected

supports generating skeleton test cases to test individual agents based on the **goal-models**, which the developer manually completes and eCAT automatically executes. eCAT does not however provide a detailed means of testing the internal units of an agent, which is our focus.

In more recent work, Nguyen et al. [19] present a method that uses evolutionary algorithms to generate demanding test inputs. The **soft goals** (non-functional goals) specified as part of the requirements are formalised and are used as test oracles. As a result, these oracles do not measure correctness of the agent, as our oracles do. Instead, they measure the ability of agents to achieve their soft goals. Further, Nguyen et al.’s approach uses soft goals to generate increasingly demanding test inputs using the evolutionary algorithms. A test that causes an agent to perform poorly relative to its soft goals is more likely to be used as part of the reproduction phase of the evolutionary algorithm, leading to more demanding test cases.

Knublauch [14] presents a framework for testing agents built following the GAIA methodology [29]. GAIA design models were used to develop a set of APIs that are extended from the JUnit testing framework [25]. Test cases are manually developed following these APIs and the system automates the execution of them. Similar API-based testing approaches were developed for an extension to the PASSI [6] methodology in [7] and for the SEAGENT [12] model of agent systems [27] that presents the SUnit test
framework. SUnit is also an extension to JUnit, however, unlike other approaches that consider the agent as the basic unit of test, SUnit allows for verifying plan hierarchies and the actions performed by the plans.

Coelho et al. [8] present the JAT testing framework, which includes a fault model that captures fault types relevant to general agent features. They consider faults in message ordering, in message content, in beliefs and procedures in terms of object-oriented programming faults, and faults that delays messages. In contrast, the fault model we present in this article is concerned with the internal units of an agent. As future work, we plan to build on this work to investigate fault models for agent interaction and requirements testing.

Low et al. [15] consider test coverage criteria for BDI agents. They derive two types of control-flow graphs: one with nodes representing plans and arcs representing messages or other events that trigger plans; and one with nodes representing statements within plans and arcs representing control-flow between statements (a standard control-flow graph). Several coverage criteria are defined, based on node, arc, and path coverage, as well as some based on the success or failure of executing statements and plans. In this paper, we do not use coverage criteria to select test inputs, however, we do define what it means for events, beliefs, context conditions, and plans to be covered by tests.

Kissoum and Sahnoun [13] use Petri-Nets for testing agent interactions specified in AUML. In their approach, an AUMUL interaction diagram is converted into a Petri Net, and all paths in the Petri Net are extracted and used as test cases. Kissoum and Sahnoun do not discuss how the sequences are used to determine the necessary input or how they can be used as an oracle. Instead, they provide a high-level overview of the framework. In contrast, our work we aim to test the basic units of agents, rather than their interactions.

Poutakidis et al. [23] use agent interaction design models to generate automated test oracles for multi-agents systems. Poutakidis et al. systematically translate AUML protocol specifications into Petri Nets, and ascertain whether the interaction is following the specified protocol by executing the Petri Net as observed messages are sent.

In other work [16], we have investigated model-based test coverage criteria for interactions between agents, and how to couple these with the oracles defined by Poutakidis et al. In contrast, this paper is based on units within agents, rather than integrating multiple agents.

8 CONCLUSION

In this paper, we defined a fault model consisting of 11 fault types, based on the features associated with the basic units of BDI agents: plans, events and beliefs. We used this fault model to design a partial test oracle that uses design models to verify behaviour of agent units, and described a test framework that automatically produces such oracles.

We evaluated both the fault model and the test oracle on 14 agent-based systems, categorising the faults found. Our test framework generated and executed almost 204,000 test cases and reported a total of 432 errors and warning notifications, with the 304 true positive cases corresponding to 294 unique faults. As far as we are aware, none of these faults had been previously detected by the developers. The successful detection of faults relies on both the oracle, and on the test inputs run. The authors believe that the automated generation of the test inputs and test oracles were critical to finding these, as they allowed the execution of large numbers of test cases.

Our evaluation found that 8 out of the 11 fault types occurred at least once in the 14 systems, and that 11 of the 19 checks performed by the oracle identified faults in either design or implementation. Our results show that our fault model is a useful conceptualisation of the types of faults found in BDI agent units, that our test oracles are able to detect faults related to these fault types, and are able to do so with a false positive count below 30%.

Based on our observations, we have suggested some possible modifications to the oracle design to avoid false positives that are of limited value. In particular, we noted that failure to specify context conditions in design generated a substantial number of unnecessary warnings. Further work is needed to investigate the consequences of revising the interpretation of no specified context condition in design.

While we have shown that our test framework, including the oracle, can identify a range of faults, our evaluation tells us nothing about problems that may exist, but that are not identified. In future work, we plan to apply the test framework on systems with known faults, by either requesting students to send versions of their assignments with known faults as they are developing their software, or via the use of controlled mutation [11].

REFERENCES


Lin Padgham has degrees in Social Work, Psychology and Computer Science, with a PhD in Computer Science from Linköping University, Sweden. She is a Professor in Artificial Intelligence at RMIT University, Australia, where she has spent more than 15 years doing research in various aspects of intelligent multi-agent systems. Together with colleagues she has developed the Prometheus design methodology for building agent systems, and co-authored the first detailed book (published 2004) on a methodology for building intelligent multi-agent systems. In 2005, the supporting tool for this methodology, the Prometheus Design Tool, won the award for the best demonstration at AAMAS’05. The Prometheus Design Tool is used by a range of academic and industry groups internationally and can be accessed at https://sites.google.com/site/rmitagents/prometheus. Lin’s recent work in Agent Oriented Software Engineering has focussed on automated testing, using design models. Lin has served on the editorial board of Autonomous Agents and Multi-Agent Systems, was Program Co-Chair for AAMAS 2008, and General Co-Chair for AAMAS 2012.

Zhiyong Zhang has over ten years commercial and academic experience on software testing and multi-agent systems. He has worked in commercial software development/testing for over six years, and completed a PhD on automated testing for agent systems at RMIT University. His PhD research explored an approach for automated unit testing of agent systems. The result is a unit-testing framework that allows completely automated unit testing of an agent system, from generation of test cases, through to automated implementation of test cases and production of detailed reports. The approach follows the principle of model based testing: testing units against what has been specified in the system design model using an AOSE methodology. He is a Master of Artificial Intelligence in K.U.Leuven, Belgium, and an ISTQB certified Test Analyst in advance level.
**John Thangarajah** is a Senior Lecturer at the School of Computer Science at the RMIT University. He completed his PhD at RMIT University in 2005 in the area of Artificial Intelligence. His research experience is in Agent Oriented Software Engineering, developing techniques for designing and testing intelligent agent systems, and in goal-based agent reasoning techniques. He has managed the development of an award-winning agent design tool built within the RMIT agents group.

**Tim Miller** is an academic in the Department of Computing and Information Systems at The University of Melbourne. Tim received his PhD in 2005 from the University of Queensland. He spent four years at the University of Liverpool, UK, as a postdoctoral researcher associate in the Agent ART group. In 2008, he moved to Melbourne to take up his current post. Tim’s primary interest lies in the area of software engineering for intelligent systems.