Test Coverage Criteria for Agent Interaction Testing

Tim Miller[1], Lin Padgham[2], and John Thangarajah[2]

[1] Department of Computer Science and Software Engineering, University of Melbourne, Australia
[2] Department of Computer Science, RMIT University, Melbourne, Australia

Abstract. By the very definition of complex systems, complex behaviour emerges from the interactions between the individual parts. This emergent behaviour may be difficult or impossible to predict by analysing the parts. As a result, systematic and thorough testing of the interactions of complex systems, including multi-agent systems, is an important part of the verification and validation process. This paper defines two sets of test coverage criteria for multi-agent interaction testing. The first uses only the protocol specification, while the second considers also the plans that generate and receive the messages in the protocol. We describe how an existing debugging agent can be used as a test oracle for assessing correctness of a test, and how the Petri Net representation of the debugging agent can be annotated to support test coverage measurements. This work both specifies, and shows how to measure, the degree of thoroughness of a set of test cases. It also provides a basis for the future specification of test case input, designed to provide good coverage.

1 Introduction

Like other types of complex systems, the overall behaviour of multi-agent systems emerges from the interaction of their parts. Often, this emergent behaviour is difficult or even impossible to identify without running the system. This increased complexity makes verification and validation of these systems a non-trivial task. Furthermore, the fact that the behaviour cannot be accurately predicted implies that manual test case generation is unlikely to test the more complex behaviour. Automated test generation offers one solution to help with this problem.

Previous work on testing multi-agent systems [1, 2, 8, 10, 15, 16] has contributed to testing frameworks and automated test case generation. However, none have explicitly focused on testing interactions, the source of complexity in many systems. In many multi-agent methodologies, such as Prometheus, Tropos and OMaSE [3], interactions are captured via interaction protocols in design diagrams.

Our focus in this paper is on using protocol specifications, as well as information about how the interacting agents use these specifications, to define and measure systematic interaction testing. We also describe how correctness can be determined using the debugging agent of Poutakidis et al. [13]. Section 2 defines two sets of *test coverage criteria* for interaction testing, the first using only the protocol specification, and the second including information about the plans involved in receiving and sending messages for a particular protocol. Section 3 describes the use of Poutakidis et al.'s "debugging agent" as a test oracle for determining whether a set of interacting agents is correctly following a valid protocol. Modifications to this debugging agent are made to automatically measure how well a test set achieves the coverage criteria. We finish with a discussion of relationships to previous work and a comment on future work.

2 Test Coverage Criteria

To measure the quality of a set of test cases, a criterion is necessary. Standard control-flow and data-flow criteria [9] that are defined for imperative programming languages are based on program statements and predicates, so are not directly applicable to agent interaction. However, many of the underlying ideas are valid. In this section, we define two sets of criteria based on the control-flow of interactions. This control-flow is extracted from the design models. The first set is based on the ordering of messages, which we obtain from protocol specifications. We refer to these as *protocol-based* criteria. The second set also considers the plans that send and receive the messages in protocols. We refer to these as *plan-based* criteria. We describe and compare each of these.

2.1 Protocol-based Coverage Criteria

Based on protocols specified in a standard protocol language such as AUML2 interaction diagrams, it is possible to construct a *protocol graph* that shows all possible orderings of messages[3]. Figure 1 shows the protocol graph corresponding to the FIPA Query Interaction Protocol [5].

The *conversation IDs* annotated to each message identify six conversations that have happened using this protocol, in which a conversation is a possible chaining of messages.

Criterion Definition Our coverage criteria are based on graph traversal of the protocol graph. For protocol coverage, we define three criteria:

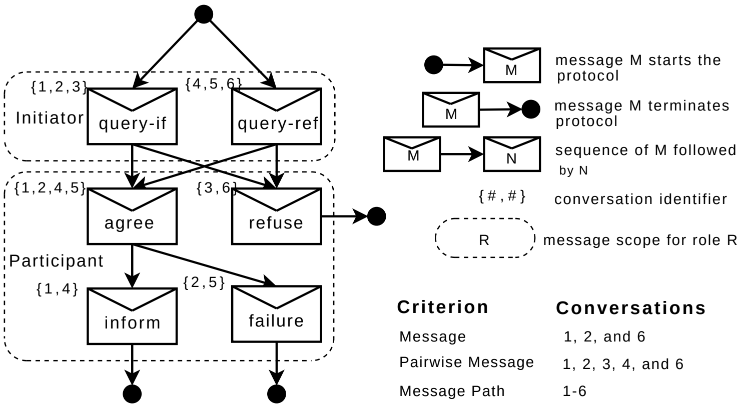
Message coverage Every message in the protocol must be sent at least once.

Pairwise message coverage For every message, start node, and end node in the protocol, all directly proceeding messages/nodes must be executed after the first message/node at least once; that is, we must test every case in which one message can be followed by another.

Message path coverage Every possible interaction sequence permitted by the protocol must be executed at least once.

These three criteria correspond to node, arc, and path coverage of a graph. Figure 1 contains a minimal set of conversations that, if fully executed, achieve these criteria on the protocol graph.

[3] Our coverage criteria are then based on these orderings. We are not concerned with the content of messages, nor the time at which they are sent, only the relative ordering.



Criterion	Conversations
Message	1, 2, and 6
Pairwise Message	1, 2, 3, 4, and 6
Message Path	1-6

Fig. 1. A protocol graph for the FIPA Query interaction protocol specified, and the conversations required to achieve coverage criteria.

Achieving path coverage is sometimes not possible as a protocol may be defined as an infinitely iterative or recursive structure, leading to an infinite number of paths. Workarounds include achieving only non-cyclical path coverage, or using heuristics such as the 0-1-*many* rule, which specifies that we test only three of these paths: paths containing 0 loops, 1 loop, and more than one loop.

Coverage Measures Spillner [14] defines *coverage measures* for integration testing criteria. Coverage measures are defined as "*the ratio between the test cases [inputs] required for satisfying the criteria and those of these which have been executed*". These measures can be applied to test sets to determine how complete they are for a particular program.

The interaction coverage measures (IC) for our three protocol-based criteria are defined as follows:

$$IC_{protocol\_message} = \frac{\#\text{messages sent at least once}}{\#\text{totalmessages in protocol}}$$

$$IC_{pairwise\_message} = \frac{\#\text{arcs executed}}{\#\text{total arcs in protocol}}$$

$$IC_{message\_path} = \frac{\#\text{paths executed}}{\#\text{total message paths in protocol}}$$

As an example, in Figure 1 the set of conversations 1, 2, and 6 achieves 100% for protocol message coverage (6 messages that are all executed), 82% for pairwise message coverage (11 arcs, 9 arcs covered), and 50% for message path coverage (6 different paths, 3 paths covered).

Protocol-based coverage criteria are intuitively useful for interaction testing because they are strictly related to the interactions that can occur between the agents. However, purely message-based criteria do not consider the internal structure of the agents. For example, an agent may be able to send or receive the same message in many different plans. Consequently, we develop an additional set of coverage criteria that take into account the plans of the agents, and the relationship of messages to these plans.

2.2 Plan-based Coverage Criteria

We extract from the design artifacts, the information to build a *plan graph* for each protocol, of the kind shown in Figure 2. This graph represents the relationship between plans and messages for a particular protocol.

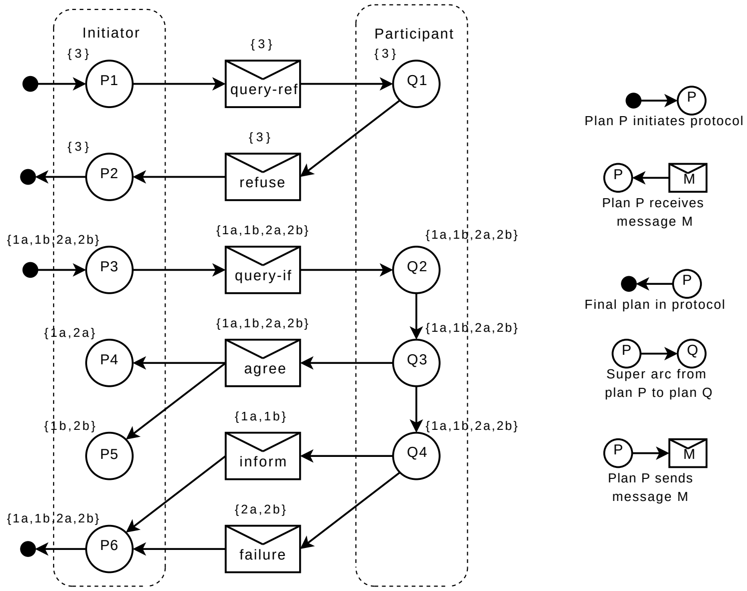
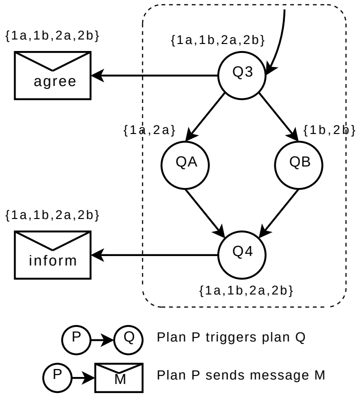


Fig. 2. A plan graph for the FIPA Query interaction protocol specified.

Plan graphs are built by extracting as nodes, those plans that send or receive any message in the protocol, and the messages themselves. In addition to the obvious send/receive links between plans and messages, we add a link between any two plans in the graph, which are connected by a chain (or multiple chains) of triggering links. We will call such links between plans *super-arcs* as they represent an entire plan structure. Figure 3 shows the internals of the super-arc between plan nodes $Q3$ and $Q4$ in Figure 2.

From Figure 2, one can see that the participant always agrees to a *query-if* request and always refuses a *query-ref*; so while the agents may follow the protocol, they do not use all parts of it. We also note that plans can send more than one message or receive more than one message, for example, plan $Q4$ sends both *inform* and *failure*.

Unlike other branches in the graph, the branch at $Q3$ is not a choice. Instead $Q3$ sends the message *agree*, and then triggers the plan $Q4$. For the purpose of test criteria, it is not necessary to model whether this is a choice or the ability to do more than one action because we need only measure whether the message was sent.



Criterion	**Conversations**
Message	1(a or b), 2(a or b), and 3
Plan	1a, 1b, and 3
Plan arc	1a, 2b, and 3
Pairwise internal path	1a, 1b, 2a, 2b, and 3
Plan path	1a, 1b, 2a, 2b, and 3

Fig. 3. Internal plan structure between plans $Q3$ and $Q4$ for the graph from Figure 2, and the conversations required to achieve coverage criteria for the entire protocol.

Criterion Definition We define a set of coverage criteria using plan graphs, in a similar way to those we defined on the protocol graph. We note how these correspond to criteria in standard (non-agent-oriented) integration testing [14].

Message coverage Every message in the plan graph is sent at least once. The analogous case in standard integration testing is ensuring that each method/function in the target component's interface is executed at least once.

Plan coverage Every plan that sends or receives a message in the protocol is executed at least once. The analogous case in integration testing is ensuring that each method in the program that calls a method in the target component's interface is executed.

Plan arc coverage Every occurrence of a message being sent by a plan and every occurrence of a plan being triggered (by a message, an event (a start node), or another plan) is executed at least once. This is different from plan coverage because a plan may be able to send more than one message (e.g. plan $Q4$ sending *inform* and *failure* in Figure 2). The analogous case in integration testing is ensuring that every call made to every method in the target component's interface is tested.

Pairwise internal path coverage Every possible path, including paths in super arcs, between two pairwise messages, or between a first/last message in a protocol and its corresponding start/end node is executed at least once. This ensures that all paths that could be used to produce a particular message in the protocol, are tested. The analogous case in integration testing is ensuring that every path between two method calls from the target component's interface is executed. Note that pairwise messages cannot be determined from the plan graph, but must be determined from the protocol specification or protocol graph. For example, in Figure 2, one cannot determine that *agree* is sent directly before *inform.*

Plan path coverage Every possible path through the structure induced by expanding super-arcs within the plan graph is executed at least once. The analogous case in integration testing is ensuring that every possible sequence of calls to every method in the target component's interface is tested. Even without the expansion of super-arcs this differs from message path coverage defined on the protocol graph, in that it addresses the case where the same message may be sent from, or received by, two different plans. (e.g. plan $P4$ and $P5$ receiving *agree* in Figure 2).

Again, some of the above criteria correspond to graph coverage criteria. Message and plan coverage combined correspond to node coverage. Plan arc coverage, and plan path coverage correspond to arc, and path coverage respectively. Pairwise internal path coverage corresponds to path coverage between plan nodes within a super arc, combined with arc coverage on the other arcs of the graph.

To illustrate, Figure 3 contains a minimal test sets that, when fully executed, achieve each criteria, using the plan graph from Figures 2 and 3.

Coverage Measures We define coverage measures for these criteria in the same way as the protocol-based criteria: the ratio of executed nodes/arc/paths to the total number of nodes/arc/paths.

For example, in Figures 2 and 4, the set of conversations 1a, 2b, and 3 achieves 100% coverage for message coverage (6/6), plan coverage (10/10), and plan arc coverage (19/19), 85% for pairwise internal path coverage (11/13), and 55.5% for plan path coverage (5/7).

2.3 Comparison of Coverage Criteria

To compare these criteria, we are interested in any subsumption relationships between them. Test criterion A *subsumes* test criterion B if and only if any test set that achieves 100% coverage on criterion A also achieves 100% coverage on criterion B.

Figure 4 shows the subsumption relationship between our criteria. We know from graph theory that path coverage subsumes arc coverage, and arc coverage subsumes node coverage. This directly gives us the subsumption relation between the different protocol graph criteria (message path subsumes pairwise message, and pairwise message subsumes message).

In the plan graph, plan path subsumes plan arc, and plan arc subsumes message/plan coverage, directly from graph theory. Although neither message nor plan coverage are equivalent to node coverage, both are subsumed by it. Plan coverage does not subsume message coverage, because plans can send and receive multiple messages. For example, plan coverage of Figure 2 can be achieved by executing plans $Q4$ and $P6$ once each, which means either *inform* or *failure* will not be sent. There is also an additional coverage metric, pairwise internal path coverage, which sits between plan path coverage and plan arc coverage. The argument for this is straightforward: by definition it subsumes arc coverage, and if every path, including every path internal to a super arc, is executed, then every arc plus all super-arc paths must also be executed.

To compare the criteria for the two different types of graph, we make the assumption that all criteria are feasible. For example, in plan message coverage, we assume that the participating agents are programmed such that every message in a protocol is able to be sent by these agents. Otherwise, 100% coverage is not achievable. It is not uncommon for this assumption to be false, particularly when pre-existing protocols are used. For example, a developer using a third-party protocol may choose not to use some messages defined in a protocol.

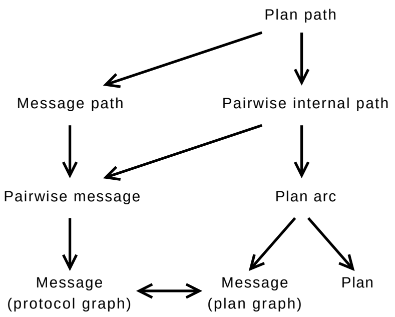


Fig. 4. The *subsumes* relation between the protocol-based and plan-based coverage.

If this assumption is relaxed, the result is simply that there is no subsumption relation between any of the criteria[4]. With this assumption of feasibility we can establish some relationships between the criteria based on the two different graphs. Firstly, we note that the two types of message coverage are equivalent. That is, they both require test cases that send every message in the protocol. The next relation is that pairwise internal path coverage subsumes pairwise message coverage. Pairwise internal path coverage is defined as executing all paths (including super arcs) between all pairwise messages, therefore, it trivially subsumes pairwise message coverage. Finally, we have that plan path coverage subsumes message path coverage. With our assumption of feasibility, this subsumption relation holds because if there is a path defined by the protocol, there must be a path in the plan graph that executes it. If all paths through the plan graph are executed, then this implies all paths in the protocol graph must also be executed.

We argue that the combination of message path coverage and pairwise internal path coverage is a minimum testing level to aim for in rigorous interaction testing. It tests the various plan combinations that may be used in moving from receipt of a message, to the production of the next message in the protocol, and also tests every possible conversation. Although there is some amount of exponential growth, this is likely to be substantially more limited than that required for testing all paths in the plan graph.

3 Measuring Correctness and Coverage Using a Debugging Agent

The model-based measure of correct behaviour of agent interaction is primarily whether the agents follow the specified interaction protocols. While the coverage measures we have defined can tell us how thoroughly a given set of test cases actually exercises the program under test, we require some way of knowing whether the agents interact as specified. To establish this we use the work of

[4] This can be demonstrated by the examples in Figures 1 and 2: the agents are programmed such that the sequence $\langle$*query-ref* $\rightarrow$ *agree*$\rangle$ is infeasible, therefore, pairwise message coverage is not achievable on the protocol graph, but pairwise internal path coverage is achievable on the plan graph.

Poutakidis et al. [13] on debugging agent interactions. The monitor that is used in that work for detecting bugs, can equally well be used as a *test oracle*.

The IEEE Standard Glossary of Software Engineering Terminology [6] defines a test oracle as: "*any means of determining whether a system or component's behaviour is consistent with its specification.*"

In Poutakidis et al.'s work, the agent platform is modified so that the debugging agent receives copies of all messages sent within the system. This debugging agent then raises an alert if a sent message does not follow one of the specified protocols, or if a protocol does not reach a specified end state. These are the two possible errors that can arise with respect to the agent interactions.

We use the infrastructure of Poutakidis et al. to collect information regarding our protocol graph interaction coverage criteria. This information can be collected by an independent observer. For the plan-graph coverage criteria, information must be known about the inner details of the participants. In current work, we are adapting Zhang et al.'s automated unit test framework [16] to measure plan-graph coverage criteria.

3.1 Petri-Net Representation for Protocols

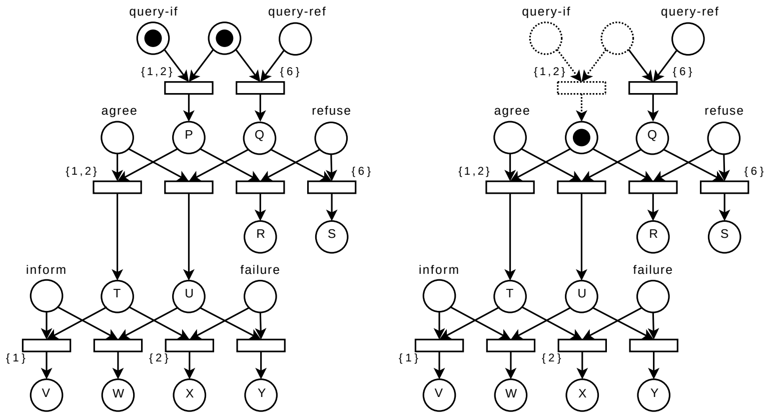
Poutakidis et al. systematically translate AUML protocol specifications into Petri Nets, and executing these as agents interact, are able to ascertain whether the interaction is following a specified protocol.

A Petri Net is a bipartite graph containing two types of nodes: *places* and *transitions*. Places are represented with circles, and transitions are represented with rectangles (see Figure 5). Arcs connect transitions to places. The execution semantics of Petri Nets specifies that *tokens* can be located at places. A transition can be *fired* if all incoming places contain a token and the outgoing place is empty; when the transition fires, a token is placed at all outgoing places.

Poutakidis et al. define Petri Nets with two kinds of places: state places and message places. State places represent the state prior to a given message being received, or end states. When a Petri Net instance is initialised by the debugger, it has a token placed on its relevant message and state places. At each cycle all Petri Net instances are fired to completion, and then retained until the next cycle, when a token is added to the message place in the relevant Petri Nets. Poutakidis et al. define mappings from protocols to Petri Nets to model the possible protocol executions.

Figure 5(a) shows the Petri Net for the FIPA query protocol. When a *query-if* message arrives, this is identified as a start message for this protocol, and a new Petri Net instance is created. A token is placed in the *query-if* message place, and the corresponding state place. The Petri Net is then executed allowing the transition to fire producing a token on the outgoing state-place, P, as in Figure 5(b). The Petri Net is now in a state where, when a token is placed on either the *agree* or *refuse* message place, it can fire the relevant transition, producing a token in either R or T.

The debugging agent contains a copy of every protocol (and its corresponding Petri Net) used in the system. Each conversation held between a set of agents must contain a *conversation ID* to allow placement of a message into the correct



(a) (b)

Fig. 5. An example of a Petri Net transformation.

Petri Net instance as there may be concurrent conversations. This requirement is supported by the FIPA standard for agent communication [4].

Each time the debugging agent receives a copy of a message, it first confirms whether the conversation ID corresponds to a current conversation. If not, it creates a Petri Net for *all* protocols that contain the received message as their initial message, initialises these appropriately, (as in Figure 5(a)). This is necessary because it has no way of knowing which protocol the sending agent is using if more than one protocol has that message as the initial message. The agent maintains a set of Petri Net instances for each conversation, until it becomes clear by a process of elimination, which protocol is being used.

If the message corresponds to an existing conversation, the debugging agent places the message in the appropriate message-place of all the Petri Nets in the set for the conversation. Those that do not have an appropriate place, or where the message place does not enable a transition, are removed from the set, because it is evident the conversation is not following this protocol. If the set becomes empty, then this indicates a fault: the agents are not following any known protocol. For example, if the Petri Net is in the state shown in Figure 5(b), and the debugging agent received the message *agree*, then this transition could fire. Alternatively, if the message was *refuse*, the transition could not fire because there would be no token at Q. This indicates a fault.

The process continues until the conversation is deemed to have terminated. If there are tokens remaining in any non-terminal places, this indicates a fault, because the conversation did not follow the protocol to completion.

3.2 Measuring Coverage Using Petri Nets

We adapt Poutakidis et al.'s debugging agent to measure the coverage corresponding to the protocol-based criteria defined in Section 2. First, we modify the oracle such that, when a transition is fired, the transition is annotated with the conversation ID, such as in Figure 5(a).This records all of the conversations that take place using a particular protocol. In this example, conversations 1, 2, and 6 from Figure 1 have been executed, and the appropriate transitions have been annotated.

To measure message coverage, we analyse each message place in the Petri Net and determine if at least one of its outgoing transitions has been fired (is annotated). If so, the message has been sent. We can then use the coverage measure definition from Section 2.1 to measure coverage.

To measure pairwise message coverage, we analyse each place in the Petri Net that represents an intermediate state; that is, all non-message-places between two transitions. If the incoming transition and at least one outgoing transition share at least one conversation ID, then this pair was executed. For example, the place P contains one incoming transition and two outgoing transitions. The incoming arc and the left outgoing transition both contain the conversation IDs 1 and 2, so this pair was executed in sequence. To show this is valid, we note that the unfolding rules specified by Poutakidis et al. [13] result in a graph such that any two places are linked by at most one path. As a result, each intermediate place in a Petri Net must have at least one input and one output transition, and all pairwise messages in a protocol are connected by exactly one such message place in the Petri Net representation. If the incoming and outgoing message place share a conversation ID, then the pair of messages must have been executed.

Finally, to measure message path coverage, we take each terminal place, and determine if the incoming transition to that place was fired; that is, contains at least one conversation ID. To demonstrate validity of this, we again note that Poutakidis et al.'s unfolding rules result in a graph such that any two places are linked by at most one path. Therefore, the final transition is unique to a path, so if this transition has been fired, the entire path must have been executed.

3.3 Measuring Coverage for Concurrent Conversations

To monitor multiple conversations over a single protocol, Poutakidis et al. create multiple instances of the same Petri Net. Using a single instance is not suitable because, upon testing to see if a message is valid, the Petri Net may be in a configuration such that the message is valid for another conversation, but not the current one. Creating multiple instances of a Petri Net is suitable for monitoring interactions, however, to measure coverage, the coverage information is spread over multiple Petri Nets for a single protocol.

One solution is to collate all information from all Petri Nets after a test suite has been executed. However, this is somewhat inefficient and cumbersome.

A more elegant solution is to adapt Poutakidis et al.'s solution to use *coloured Petri Nets* [7]. Coloured Petri Nets extend Petri Nets by (among other things) allowing tokens to carry a *value*. A transition can be fired only if all incoming places contain a token with the same shared value.

To handle concurrent conversations, each protocol corresponds to a single Petri Net in the test oracle. Rather than creating multiple copies for multiple conversations, tokens are given values corresponding to the conversation ID. Using this, the test oracle receives a message containing a conversation ID, and can determine at which place the correct token resides. From here, it can determine whether the message is valid.

Taking this approach, tokens remain at the terminal places after conversations have terminated. Therefore, measuring path coverage is as simple as counting the number of terminal places that contain at least one token, and dividing this by the total number of terminal places.

4 Discussion and Conclusion

Due to the complex emergent behaviour that results from agents interacting with each other, testing these interactions is an important part of the verification and validation process. The fact that emergent behaviour in complex systems is often difficult or impossible to identify without running these systems implies that using human test engineers to generate test cases manually is not sufficient, and automated test case generation techniques are required.

There has been recent work on automating test case generation such as the Unit test framework of Zhang et al. [16] already mentioned in this work, and the eCAT system associated with Tropos [10–12]. eCAT is a testing tool that automates test case generation and execution. There are 4 test generation techniques employed in eCAT: *goal-oriented*, which is manual test generation using goal diagrams; *ontology-based*, where test cases are derived automatically from the specification of the agent interaction ontology; *random*, where values for test cases are randomly generated; and *evolutionary mutation*, where genetic algorithms generate test cases measured by the quality goals of the system. Our approach to testing correctness, and measuring thoroughness could complement any of these test case generation techniques.

Whether test cases are generated automatically or manually, it is important to have a measure of the quality of the set of test cases. This paper has provided criteria by which to measure this, showing the subsumption relationships between these criteria, and discussing which we would practically aim for. We suggest that testing all paths through the protocol, combined with all plan paths between two messages achieves a high level of coverage, and is likely to be more feasible than plan path coverage, which subsumes both of these criteria. This paper has also shown how to collect these measurements as part of the testing process. We consider that these coverage definitions provide a sound basis for guiding test case generation where test cases are designed to give good coverage.

Low et al. [8] also 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. However, Low et al.'s work builds graphs over the entire program, and thus does not facilitate the modular and focused testing based on specific interaction protocols.

Low et al.'s coverage criteria relate to ours. Their plan graph is similar to our plan graph, except that they consider plans that are not related to interaction. As a result, their coverage criteria subsume ours; for example, their plan path coverage subsumes our plan path coverage. However, their criteria do not consider pairwise messages, as they do not focus on interaction protocols. Low et al. do not define specific coverage measures or how to calculate them, nor do they discuss test oracles.

The work in this paper is one step towards a larger goal: model-based automated testing for multi-agent systems. Future work will define methods for automatically deriving test cases from design artifacts. With respect to interaction testing, we will attempt to automatically generate complete test suites that achieve message path coverage combined with pairwise internal path coverage, using design documents as the models.

References

1. G. Caire, M. Cossentino, A. Negri, A. Poggi, and P. Turci. Multi-Agent Systems Implementation and Testing. In *the Fourth International Symposium: From Agent Theory to Agent Implementation*, Vienna, April 14-16 2004.
2. R. Coelho, U. Kulesza, A. von Staa, and C. Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *Proc. of the 2006 Intl. Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, pages 83–90, 2006.
3. S. DeLoach, L. Padgham, A. Perini, A. Susi, and J. Thangarajah. Using three AOSE toolkits to develop a sample design. *International Journal of Agent-Oriented Software Engineering*, 3(4):416–476, 2009.
4. FIPA. FIPA ACL message structure specification. Standard SC00061G, Foundation for Intelligent Physical Agents, December 2002.
5. FIPA. FIPA query interaction protocol specification. Standard SC00027H, Foundation for Intelligent Physical Agents, December 2003.
6. IEEE. IEEE standard glossary of software engineering terminology. Technical Report 610.12-1990, Institute of Electrical and Electronic Engineers, 1990.
7. K. Jensen. *Coloured Petri Nets*. Springer Verlag, 1997.
8. C. Low, T. Y. Chen, and R. Ronnquist. Automated test case generation for BDI agents. *Autonomous Agents and Multi-Agent Systems*, 2(4):311–332, 1999.
9. Glenford J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
10. C. Nguyen, A. Perini, and P. Tonella. Automated continuous testing of multi-agent systems. In *Fifth European Workshop on Multi-Agent Systems*, Hammamet, Tunisia, December 2007.
11. C. Nguyen, A. Perini, and P. Tonella. eCAT: a tool for automating test case generation and execution in testing multi-agent systems (demo paper). In *Proceedings of AAMAS-08*, pages 1669–1670, Estoril, Portugal, 2008.
12. C. Nguyen, A. Perini, and P. Tonella. Ontology-based test generation for multi-agent systems. In *Proceedings of AAMAS-08*, pages 1315–1320, 2008.
13. D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of AAMAS-02*, pages 960–967, 2002.
14. A. Spillner. Test criteria and coverage measures for software integration testing. *Software Quality Journal*, 4(4):275–286, 1995.
15. A. Tiryaki, S. Öztuna, O. Dikenelli, and R. Cenk Erdur. SUNIT: A unit testing framework for test driven development of multi-agent systems. In *Agent-Oriented Software Engineering VII*, volume 4405 of *LNCS*, pages 156–173, 2006.
16. Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing for agent systems. In *2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering*, pages 10–18, Spain, July 2007.