Ontology-based Test Generation for MultiAgent Systems (Short Paper)

Cu D. Nguyen, Anna Perini and Paolo Tonella
Fondazione Bruno Kessler
Via Sommarive, 18
38050 Trento, Italy
{cunduy, perini, tonella}@fbk.eu

ABSTRACT

This paper investigates software agents testing, and in particular how to automate test generation. We propose a novel approach, which takes advantage of agent interaction ontologies that define content semantic of agent interactions to: *(i)* generate test inputs; *(ii)* guide the exploration of the input space during generation; and, *(iii)* verify messages exchanged among agents with respect to the defined interaction ontology. We integrated the proposed approach into a testing framework, called *eCAT*, which can generate and evolve test cases automatically, and run them continuously.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Ontology-based test generation

1. INTRODUCTION

Testing of Multi-Agent Systems (MAS) is a challenging task because these systems are distributed, they are often programed to be autonomous and deliberative, and they operate in an open world, which requires context awareness. MAS face issues concerning communication and semantic interoperability, as well as coordination with peers. All these features are known to be hard not only to design and to program [1], but also to test.

A few studies focused on MAS verification [2], debugging [3, 10], and, more recently on MAS testing [4, 13]. Tiryaki et al. [13] and Coelho et al. [4] have proposed two testing frameworks, working on different agent platforms, both based on JUnit and sharing the idea of using mock agents that interact with the agents under test by sending messages to them, according to a given interaction protocol. The replies of the agents under test are then evaluated against expected behaviors.

Cite as: Ontology-based Test Generation for MultiAgent Systems (Short Paper), Cu D. Nguyen, Anna Perini and Paolo Tonella, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008,Estoril,Portugal,pp.1315-1318.

Although existing approaches [4, 13] represent an important step toward the automation of MAS testing, they mainly address test case execution. Issues related to test cases generation, that concern how to effectively generate valid and invalid input data to thoroughly exercise the agents' behavior, are still largely unexplored in MAS testing.

Agent behaviors are often influenced by messages received. Hence, at the core of test case generation is the ability to build sequences of messages that exercise the agent under test so as to cover most of the possible running conditions. Often, failures appear only after long execution periods and under specific conditions and environmental contexts. A key feature of agent testing is the possibility to carry on long interactions with the agents under test. This demands for automated approaches to test case production, so as to overcome the necessarily limited number of cases considered in a manually defined test suite.

Automated test case generation requires the ability to fill-in message templates with input data that are both meaningful and diverse enough to stimulate the alternative reactions of the agent under test. Manual and purely random input generation are deemed to cover a limited portion of the input space for different reasons: manual input data are necessarily a small set; random input data are generated without taking the semantics of the messages into account. For example, a string can be easily generated randomly, by picking up a random sequence of characters, but such an automatically generated input is very unlikely to match, e.g., a valid airline name, which might be necessary to construct a valid message processed by the agent under test.

We have developed a MAS testing framework, called *eCAT*, which supports continuous testing and automated test case generation [8]. The algorithms used for test case generation are based on a fitness function for the selection of those test cases that are most likely to reveal faults. However, no guidance is provided for the selection of meaningful input values.

In this paper, we address the problem of meaningful input value generation, in the context of continuous, automated MAS testing. We take advantage of agent interaction ontologies, which define the semantics of agent interactions, in order to automatically generate both valid and invalid test inputs, to provide guidance in the exploration of the input space, and to obtain a test oracle against which to validate the test outputs.

We implemented the ontology-based test case generation approach within our framework *eCAT*, so that test cases are generated and executed continuously, resulting in a fully

automated testing process, which can run unattended for long periods of time. On the one hand, continuous testing enables extensive exploration of the space of MAS behaviors, on the other hand, ontology-based test generation guides this exploration toward the most interesting regions of the space of input data that appear in meaningful messages.

In this paper, we describe our tool-supported test generation framework. Results of its evaluation on BDI case studies can be found in [9].

2. ONTOLOGY-BASED TEST GENERATION

2.1 Agent interaction ontology

In order for a pair of agents to understand each other, a basic requirement is that they speak the same language and talk about the same things. This is usually achieved by means of an ontology, namely an *interaction ontology*. Popular multi-agent platforms like JADE [12], JADEX [11], widely support the use of ontologies. They provide tools for generating code from ontology documents, thus, reducing the development effort, and for runtime binding of the message contents with concepts defined in an ontology.

A common structure of an interaction ontology involves two main concepts (also known as *Class*es): `Concept` and `AgentAction`. Sub-classes of `AgentAction` define actions that can be performed by some agents (e.g., `Propose`), while sub-classes of `Concept` define common concepts understandable by agents that interact (e.g., `Book`). These sub-classes can have multiple properties of different type. For each class, the user can define a number of individuals (or instances) of the class. For example, `Book` can have *title* as a property, and a particular book having the title "Testing Agents" may be an instance of `Book`.

A specific agent action can now be built, based on the shared understanding of the concept `Book`. For example, an agent *Buyer* could send an ACL message of type `REQUEST` to agent *Seller*, with the following content:

(Propose (Book :title "Testing Agents") :price 135.7)

The message is understood by both agents thanks to the shared interaction ontology. Let us consider a book-trading multi-agent system in which *Seller* and *Buyer* agents negotiate in order to sell and buy books.

Rules can be added to the ontology properties in order to restrict admitted values. For example, the *price* property of the Book (above) may be constrained to be within 0 and 2000. The related OWL rule is the following:

```
<owl:Restriction>
  ã <owl:onProperty rdf:resource="#price"/>
ã ã <owl:hasValue ...>min 0 and max 2000</owl:hasValue>
</owl:Restriction>
```

Testing the *Buyer* and *Seller* agents accounts for generating instances of the messages that each agent is supposed to be able to process and let the *Tester Agent* send them to the proper agents. The *Tester Agent* continues the interaction in accordance with the selected protocol and generates new messages whenever needed. The *Monitoring Agents* observe and record any deviation from the expected behavior of the agents under test. Hence, the problem for the *Tester Agent* is how to generate meaningful messages in the course of the interaction. We take advantage of the interaction ontology for this purpose.

2.2 Domain ontology and ontology alignment

By domain ontology we mean ontologies that exist in a specific domain of interest, not necessarily being interaction ontologies. For instance, there are ontologies that describe books and all related information such as title, author, category, year of publication and the like. A number of domain ontologies are available on the Internet (a useful ontology search service is available at: `http://swoogle.umbc.edu`).

Ontology alignment [5, 7] refers to techniques aimed at finding relationships between elements of two ontologies. It can be used to map classes, properties, rules etc. of one ontology onto another one, and eventually to compare or transfer data from one to the other. Several tools are available that support ontology mapping (e.g., *Prompt*, available from: `http://protege.stanford.edu/plugins/prompt/prompt.html`). For more details on ontology mapping, the interested reader can refer to the paper by Euznat et al. [5].

In order to generate meaningful testing data, i.e., data that represent real instances of ontology classes in the domain of interest, we use ontology alignment, so as to augment the agent interaction ontology with instances. This is achieved by mapping an existing domain ontology onto the MAS interaction ontology. Since domain ontology may come with a large amount of associated data, by ontology alignment we can augment the interaction ontology with a rich set of diverse data, that can be used for test case generation.

2.3 Ontology-based test generator

We develop an ontology-based input generator. It is integrated with our testing framework *eCAT* to provide inputs. Moreover, the generator takes care of input space exploration.

Valid inputs.

The task of the ontology-based test generator consists of completing the content part of the message the *Tester Agent* is going to send to the agent under test. For each concept to be instantiated in the message, the generator either picks up an existing or creates a new instance of the required concept. No input value is generated by the test generator if the interaction protocol prescribes that a value from a previously exchanged message must remain the same.

Then, the selected instance is encoded according to the proper content codec (for the message content) and is made available to the *Tester Agent*. As an example, the following excerpt shows an XML-encoded content of a message that contains information about a proposal for a book, including the `Propose` action:

```
<root ... xmlns="jadex.examples.booktrading.ontology"/>
   <Book n:id="2" title="Introduction to MultiAgent Systems"
                  author="Michael Wooldridge"/>
   <Propose n:id="1" price="47.50" r:book="2"/>
</root>
```

When new instances are generated, the test generator selects one from those available in the ontology based on the number of usages of each instance, so as to increase diversity and explore the input space more extensively.

In the case when no ontology instances are available, valid test inputs can be still generated taking into account information, such as rules and property datatypes, specified in the interaction ontology. For example, based on the rule about the *price*, the generator can generate any value in the range from 0 to 2000 as a valid input value to be processed by the *Seller* or *Buyer* agents.

Datatype	Rule	Description
Numeric	**RVN1**	New value that has not been used before from ontology instances
	RVN2	Reused value from ontology instances
	RVN3	Randomly generated value respecting rules in ontology
	RVN4	Default or template value defined in ontology
Boolean	**RVB1**	*true*
	RVB2	*false*
String	**RVS1**	New value that has not been used before from ontology instances
	RVS2	Reused value from ontology instances
	RVS3	Randomly generated value respecting rules in ontology
	RVS4	Default or template value defined in ontology

Table 1: Valid input generation rules

More generally, for the properties of *Numeric* datatype, we can exploit the boundaries of the datatype, as well as the rules that limit the values of the specific property, to generate valid input values. For the properties of *string* datatype, we can only exploit the list of allowed values, if available. Most of the times, meaningful values for properties of string datatype are hardly generated without the help of an ontology. The full list of valid input generation rules is provided in Table 1.

Datatype	Rule	Description
Numeric	**RIN1**	Value causing overflow (underflow)
	RIN2	Value violating rules in ontology
	RIN3	Value of different datatype
	RIN4	*null* value
Boolean	**RIB1**	Value of different datatype
	RIB2	*null* value
String	**RIS1**	Value violating rules in ontology
	RIS2	Value of different datatype
	RIS3	*null* value
	RIS4	Empty string
	RIS5	Randomly generated string
	RIS6	Randomly mutated valid string

Table 2: Invalid input generation rules

Invalid input generation.

Invalid input generation is based on rules and datatypes that appear in the interaction ontology. When boundaries are specified for numeric properties, the generator goes beyond them deliberately. For string properties, the generator produces null (or empty) strings as potentially invalid values. Other options available to the generator are to randomly modify a valid input (taken from the available ontology instances) or to randomly generate a new one in order to try to produce an invalid value. Another generation rule available to the test generator involves the creation of an input value of the wrong datatype (e.g., an alphabetic string where a numeric is expected). The full list of invalid input generation rules is provided in Table 2.

The generator aims at producing invalid inputs that are as diverse as possible, in an attempt to test the robustness of the agents under test, making sure that they still behave correctly in most invalid circumstances. According to the book-trading ontology described above, the test generator knows that the property *price* is of datatype *float* and that there is a rule stating that *price* must be between 0 and 2000. The generator may produce the invalid values -1, 2001 to test both sides of the boundaries. Values that are not of type *float* may be also used to exercise the agents under test.

Message	Rule	Description
Valid message	**RVC1**	All values valid
	RVC2	All values valid and from the same instance
Invalid message	**RIC1**	All values invalid
	RIC2	Invalid and valid values interleaved
	RIC3	Just one invalid value
	RIC4	All values valid but from different instances

Table 3: Input combination rules

Message generation.

When generating the full message, the test generator applies the input combination rules described in Table 3. For valid messages, the only possibility is to use only valid input values. For invalid messages, the generator can choose either to have only invalid values, or to have interleaved valid and invalid values, or to have just one invalid value. Rule selection follows the general criteria of maximizing diversity, as explained below.

When a valid message can only be formed with inputs coming from a unique, existing instance, the more restrictive rule **RVC2** must be applied instead of **RVC1**. If input values from different instances can be freely combined, we can use **RVC1**. When **RVC2** must be used, one way to generate invalid inputs is mixing values from different instances, as prescribed by **RIC4**.

Input space exploration.

The generator uses coverage information to decide how to explore the input space. The test generator gives priority to classes and instances never selected before. When instances are reused, if possible the generator selects instances with low reuse frequency. When invalid inputs are produced, the generator chooses the so-far least-used invalid input generation rules.

Ontology as test oracle.

The expected behavior of the agents under test is checked, not only, by a set of OCL constraints, but we can also enrich such constraints with a set of constraints automatically derived from the interaction ontology. In fact, the message content sent by the agents under test is expected to respect the rules and datatypes specified in the ontology for each concept instantiated in the message. Whenever the *Tester Agent* receives a message content that is invalid according to the interaction ontology, a fault is notified to the developer team.

For example, when the *Tester Agent* sends a call for proposal for a book, the *Seller* agent must reply with a message whose content belongs to `Propose` and complies to its rules and datatypes. Otherwise, an error is detected.

3. TOOL

We have integrated the ontology-based input generator into *eCAT*, our continuous agent testing framework, publicly available for download[1]. A testing technique has been implemented for the *Tester Agent*, in which the *Tester Agent*

[1]Download URL not shown to ensure double blind review.

selects a test case template among those specified and invokes the generator to generate appropriate inputs, and then executes the test. This process is repeated continuously until a desired number of test cycles has been executed or the user stops it.

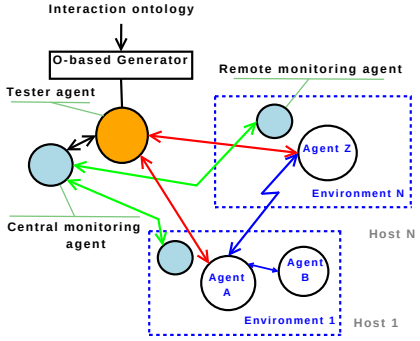

Figure 1: *eCAT* architecture, including ontology-based input generator

The architecture of the tool is depicted in Figure 1. The *Tester Agent* uses the generator (*O-based Generator*) to continuously generate test cases and run them against the agents under test while the *Monitoring Agents* observe their behaviors and informs the *Tester Agent* of faults, if any occur. Moreover, the *Tester Agent* verifies whether the messages received from the agents under test conform to the ontology or not. In the latter case, the *Tester Agent* notifies the developer team of the revealed fault.

A test case template specifies a test scenario, which could follow a standard protocol like FIPA interaction protocols [6] or a user-defined sequence of interactions. Test case templates are encoded in XML according to an XML schema we defined for them. *eCAT* has also a plug-in which allows developers to model test cases graphically.

eCAT has been implemented as an Eclipse[2] plug-in. It supports testing agents implemented in JADE [12] and JADEX [11], and the input ontology formats are those supported by Protégé[3] like OWL.

4. EXPERIMENTAL RESULTS AND CONCLUSIONS

In this paper, we presented a novel approach for automated test case generation using ontologies. The agent interaction ontology is combined with domain ontologies by means of ontology alignment techniques, so that domain ontology instances can be used to populate the agent interaction ontology with instances. Our test generator takes advantage of such instances to produce valid and invalid input messages that can be used to exercise the agents under test continuously.

We applied this approach to two, different size, BDI agent systems to evaluate its performance and capability to reveal faults. Experimental results, described in detail elsewhere [9], show that whenever the interaction ontology has non trivial size, the proposed method achieves a higher coverage of the ontology classes than manual test case derivation. It also overcomes manual derivation in terms of revealed faults, as well as portion of input space explored during testing. The level of automation achieved by our tool *eCAT* allows for test case generation at negligible extra costs.

[2] `http://www.eclipse.org`
[3] Available at `http://protege.stanford.edu`

5. REFERENCES

[1] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook*. Springer, 2004.

[2] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[3] J. A. Botía, J. J. Gómez-Sanz, and J. Pavón. Intelligent data analysis for the verification of multi-agent systems interactions. In *Intelligent Data Engineering and Automated Learning - IDEAL 2006, 7th International Conference, Burgos, Spain, September 20-23, 2006, Proceedings*, pages 1207–1214, 2006.

[4] R. Coelho, E. Cirilo, U. Kulesza, A. von Staa, A. Rashid, and C. Lucena. Jat: A test automation framework for multi-agent systems. In *23rd IEEE International Conference on Software Maintenance*, 2007.

[5] J. Euzenat, T. L. Bach, J. Barrasa, P. Bouquet, J. D. Bo, R. Dieng, M. Ehrig, M. Hauswirth, M. Jarrar, R. Lara, D. Maynard, A. Napoli, G. Stamou, H. Stuckenschmidt, P. Shvaiko, S. Tessaris, S. V. Acker, and I. Zaihrayeu. State of the art on ontology alignment. Knowledge Web Deliverable 2.2.3, August 2004.

[6] FIPA. Interaction protocols specifications. http://www.fipa.org/repository/ips.php3, 2000-2002.

[7] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18(1):1–31, 2003.

[8] C. D. Nguyen, A. Perini, and P. Tonella. Automated continuous testing of multi-agent systems. In *The fifth European Workshop on Multi-Agent Systems*, December 2007.

[9] C. D. Nguyen, A. Perini, and P. Tonella. Ontology-based test generation for multi agent systems. definition and evaluation. Technical Report FBK-IRST0108, FBK, 2008.

[10] L. Padgham, M. Winikoff, and D. Poutakidis. Adding debugging support to the Prometheus methodology. *Engineering Applications of Artificial Intelligence*, 18(2):173–190, March 2005.

[11] A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter Multi-Agent Programming. Kluwer Book, 2005.

[12] TILAB. Java agent development framework. http://jade.tilab.com/.

[13] A. M. Tiryaki, S. Öztuna, O. Dikenelli, and R. C. Erdur. Sunit: A unit testing framework for test driven development of multi-agent systems. In *Agent-Oriented Software Engineering VII, 7th International Workshop, AOSE 2006*, 2006.