Mutation Operators for Cognitive Agent Programs

(Extended Abstract)

Sharmila Savarimuthu
University of Otago
Dunedin, New Zealand
sharmila.savarimuthu@otago.ac.nz

Michael Winikoff
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

ABSTRACT

Testing multi-agent systems is a challenge, since by definition such systems are distributed, and are able to exhibit autonomous and flexible behaviour. One specific challenge in testing agent programs is developing a collection of tests (a "test suite") that is *adequate* for testing a given agent program. This requires a way of assessing the adequacy of a test suite. A well-established technique for assessing test suite adequacy is the use of *mutation testing*, where a test suite is assessed in terms of its ability to distinguish a program from its variants ("mutants"). However, work in this area has focussed largely on the mutation of procedural and object-oriented languages. This paper proposes a set of (systematically derived) mutation operators for AgentSpeak.

Categories and Subject Descriptors

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

Keywords

Mutation Testing; Agent Programming

1. INTRODUCTION

Testing multi-agent systems (MAS) is a challenge, since by definition such systems are distributed, and are able to exhibit autonomous and flexible behaviour. Most work on testing MAS has focussed on tool support for executing (manually defined) tests, although some work has investigated test generation (e.g. [7]).

Given a collection of tests (a "test suite"), a key question is whether the test suite is adequate or not? So far work on this question (e.g. [5]) has only considered the use of *coverage metrics* to assess test suite adequacy. However, coverage is necessary but not sufficient. Knowing that a test suite covers a certain portion of a program simply indicates that parts of the program were executed by the tests. It doesn't allow us to draw conclusions about whether executing the test suite is likely to be able to actually detect errors in the program being tested, i.e. distinguish between correct and incorrect programs.

An alternative, well-established, technique for assessing test suite adequacy is *mutation testing* [2] (see Section 2). Mutation testing directly assesses the ability of a test suite to distinguish between different programs, and is considered a more powerful and discerning metric than coverage, for instance Mathur notes that "*If your tests are adequate with respect to some other adequacy criteria ... then chances are that these are not adequate with respect to most criteria offered by program mutation*" [4, Page 503].

Most work on mutation testing of programs has focussed on programs in procedural and object-oriented languages [2, Figure 5]. Although there has been a little work on applying mutation to agents, this work has only considered mutating messages in agent systems [1], or mutation of JADE programs [6]. In other words, programs written in a cognitive agent-oriented programming language have not been considered.

This paper proposes a set of mutation operators for cognitive agent programs, that is, agents that are written using constructs such as beliefs, plans and goals. We use AgentSpeak as a representative for a whole class of such languages (e.g. Jason, 2APL, 3APL, GOAL, JACK, Jadex, Brahms).

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013),* Ito, Jonker, Gini, and Shehory (eds.), May 6–10, 2013, Saint Paul, Minnesota, USA.

2. MUTATION TESTING

In a nutshell, mutation testing assesses the adequacy of a test suite by generating variants ("mutants") of the program being tested, and assessing to what extent the test suite is able to distinguish the original program from its mutants (termed "killing the mutant"). For a detailed introduction to mutation testing see Chapter 7 of [4], and for a review of the field see Jia & Harman [2].

The process of mutation testing is as follows: (1) Execute the original program P against all tests in the test suite, recording the results; (2) Use *mutation operators* (see below) to generate a set of mutant programs $P_1 \dots P_n$ from P; (3) Test each mutant P_i against the tests in the test suite; (4) Each mutant that behaves differently to the original program is flagged as having been "killed"[1]; (5) The adequacy of the test suite is D/n where D is the number of killed mutants and n is the number of mutants. A quality score of 1 (highest) is good, and 0 is bad.

The mutants are generated using *mutation operators*: rules that take a program and modify it, yielding a syntactically valid variant. The key challenge in developing a mutation testing scheme is the definition of a good set of mutation operators that generate errors that are realistic, without generating a huge number of mutants.

Mathur notes that "*the design of mutation operators is as much of an art as it is science.*" [4, Page 530]. However, although the design of mutation operators is not a science, it is guided by two foundational hypotheses [2]: The *competent programmer hypothesis* states that programmers tend to develop programs close to being correct, and that therefore a simple syntactic mutation is a good approximation of the faults created by competent programmers. The

[1]Some mutants may be equivalent in behaviour to the original program ("equivalent mutants"). Identifying and removing equivalent mutants is a standard issue in mutation testing, since it is a manual process.

coupling effect hypothesis proposes that a test suite that can find the simple faults in a program, will also find a high proportion of the program's complex faults. There is empirical evidence to support both these hypotheses for procedural programs (but not for agent systems ...).

3. AGENTSPEAK MUTATION OPERATORS

In deriving a set of mutation operators for AgentSpeak we follow the approach of Kim *et al.* [3] and derive mutation operators based on HAZOP and the *syntax* of the language. HAZOP (Hazard and Operability Study) is a technique for identifying hazards in systems by considering each element of the system and applying "guide words" such as NONE, MORE, LESS, PART OF, AS WELL AS, or OTHER THAN. For example, in a chemical processing system, engineers might consider what hazard exists if a certain pipe carries MORE chemical than it should, or if there is a contaminant ("AS WELL AS").

Kim *et al.* [3] applied this idea to generating mutation operators by applying these guide words to the *syntax* of Java. For example, when considering a method invocation, the guide word OTHER THAN suggests that the designer consider the possibility that a different method to the intended one is invoked. This then leads directly to the definition of a mutation operator that rewrites a method invocation by changing the method name.

We apply this to AgentSpeak by considering the syntax of the language, and the guide words. For example, a plan in AgentSpeak includes a sequence of steps (such as updating the belief base, checking a belief, posting a sub-goal, or performing an action). The guide word NONE suggests a mutation operator that deletes the plan body, PART OF suggests removing some of the steps in the plan body, AS WELL AS suggests adding steps, and OTHER THAN suggests modifying a step.

Space limitations preclude a detailed presentation of the derivation or of the resulting rules. Figure 1 contains a (rather compact and summarised) rendition of the mutation operators that we have derived. The top part (above the first line) actually shows *generic schemas* rather than mutation operators. These schemas are instantiated to form AgentSpeak mutation operators. Each rule is presented in the format $x \rightsquigarrow y$ indicating that a syntactical element x can be written to y, yielding a mutated program. The first schema allows an element to be deleted ($x \rightsquigarrow \epsilon$, where ϵ denotes the empty syntactic element of the appropriate type, e.g. "true" for Boolean conditions). The second and third schemas allow a syntactical element to be replaced with PART OF it. The fourth schema allows the order of sub-elements (e.g. plans, steps in a plan body) to be swapped. The next two schemas specify that a combining form $\oplus$ can be replaced with a different (appropriate for the context) combining form $\otimes$, for example, in a trigger, we could replace a prefix such as $+!$ with another (different) prefix (e.g. $+, -, +!, -!, +?, -?$). The next three schemas capture compositionality: for example, $x \oplus y$ can be mutated to $x' \oplus y$ by applying a mutation rule to mutate x to x'. We then have two schemas that permit unary operators (e.g. "not") to be added or removed, and finally a schema that allows constants (e.g. numbers, illocutionary forces) to be replaced with other (appropriate) constants.

The second part of Figure 1 shows example rules that illustrate how these schemas can be used to derive AgentSpeak mutation operators. The first mutation operator is instantiated from the 8th schema and states that a plan ($trigger : context \leftarrow planBody$) can be mutated by mutating the context condition. The next two mutation operators state that a context condition can be deleted by replacing it with $true$ (which is an instantiation of the first schema), and that a conjunction can be replaced with one of its conjuncts (an instance of the second and third schemas). The next mutation operator, which is an instance of the fifth schema, shows the prefix of a trigger being mutated (as discussed above).

$$
\begin{array}{c}
x \rightsquigarrow \epsilon \qquad x \oplus y \rightsquigarrow x \qquad x \oplus y \rightsquigarrow y \qquad x \oplus y \rightsquigarrow y \oplus x \\
\oplus x \rightsquigarrow \otimes x \qquad x \oplus y \rightsquigarrow x \otimes y \qquad \oplus\, x \rightsquigarrow \oplus x' \text{ if } x \rightsquigarrow x' \\
x \oplus y \rightsquigarrow x' \oplus y \text{ if } x \rightsquigarrow x' \qquad x \oplus y \rightsquigarrow x \oplus y' \text{ if } y \rightsquigarrow y' \\
\oplus x \rightsquigarrow x \qquad x \rightsquigarrow \oplus x \qquad const \rightsquigarrow const' \\
\hline
T{:}C{\leftarrow}B \rightsquigarrow T{:}C'{\leftarrow}B \text{ if } C \rightsquigarrow C' \\
C \rightsquigarrow true \qquad C_1 \& C_2 \rightsquigarrow C_i \\
\oplus Term \rightsquigarrow \otimes Term \text{ where } \oplus, \otimes \in \{+, -, +!, -!, +?, -?\} \\
\hline
f(\ldots t_{j-1}, t_j, t_{j+1} \ldots) \rightsquigarrow f(\ldots t_{j-1}, t_{j+1} \ldots) \\
f(\ldots t_j, t_{j+1} \ldots) \rightsquigarrow f(\ldots t', t_{j+1} \ldots) \text{ if } t_j \rightsquigarrow t' \\
f(\ldots t_j, t_{j+1} \ldots) \rightsquigarrow f(\ldots t_{j+1}, t_j \ldots) \\
\text{.send}(aid, ilf, msg) \rightsquigarrow \text{.send}(aid', ilf, msg) \text{ if } \ldots \\
aid' \subseteq aid \text{ (low), or } aid' = new \text{ (wrong), or} \\
aid' = aid \cup new \text{ (high), where } new \subset allAgents \setminus aid
\end{array}
$$

Figure 1: Mutation Operators for AgentSpeak

The final part of Figure 1 shows specific rules for AgentSpeak that are in addition to the rules resulting from instantiating the generic schemas. The first three mutation operators allow a term to be mutated by deleting, mutating, or swapping arguments. Finally, we have a mutation operator for messages (.send(aid, ilf, msg), where aid is the recipient agent ID(s), ilf is the illocutionary force (e.g. tell, ask), and msg is the message content (a term)). The earlier schemas, when instantiated for messages, already allow the message content and the illocutionary force to be mutated, but we need a rule to capture the mutation of the recipient(s). Following [1] we allow four options for the *scope* of the recipient: none (do not send the message - already covered by the first generic rule), low (send to fewer agents than intended), high (send to more agents than intended), and wrong (send to entirely wrong agent(s)).

We have presented rules for generating mutants of programs in a typical cognitive agent-oriented programming language (namely AgentSpeak, augmented with Jason-style message sending). These rules are applied to the programs of individual agents in a multi-agent system, and include a rule for mutating messages. The next step is to assess empirically which subset of the rules is most useful.

4. REFERENCES

[1] S. F. Adra and P. McMinn. Mutation operators for agent-based models. In *Workshop on Mutation Analysis*. IEEE, 2010.

[2] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[3] S. Kim, J. A. Clark, and J. A. McDermid. The rigorous generation of Java mutation operators using HAZOP. Technical Report 2/8/99, Department of Computer Science, University of York, 1999.

[4] A. P. Mathur. *Foundations of Software Testing*. Pearson, 2008. ISBN 978-81-317-1660-1.

[5] T. Miller, L. Padgham, and J. Thangarajah. Test coverage criteria for agent interaction testing. In *AOSE*, pages 1–12, 2010.

[6] A. A. Saifan and H. A. Wahsheh. Mutation operators for JADE mobile agent systems. In *International Conference on Information and Communication Systems (ICICS)*, 2012.

[7] Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing for agent systems. In *Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 10–18, 2007.