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. distinguishing 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 discriminating 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).

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 \( P_1 \ldots P_n \) 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”; (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, the plan body, PART OF suggests removing some of the steps in the plan body, 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 the plan body, and other (appropriate) constants.

4. REFERENCES


