Mutation Operators for the GOAL Agent Language

Sharmila Savarimuthu and Michael Winikoff

University of Otago, New Zealand
{sharmila.savarimuthu,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. In order to develop an adequate test suite, it is clearly important to be able to *assess* the adequacy of a given test suite. A well-established technique for assessing this is the use of *mutation testing*, where mutation operators are used to generate variants ("mutants") of a given program, and a test suite is then assessed in terms of its ability to detect ("kill") the mutants. However, work on mutation testing has focussed largely on the mutation of procedural and object-oriented languages. This paper is the first to propose a set of mutation operators for a cognitive agent-oriented programming language, specifically GOAL. Our mutation operators are systematically derived, and are also guided by an exploration of the bugs found in a collection of undergraduate programming assignments written in GOAL. In fact, in exploring these programs we also provide an additional contribution: evidence of the extent to which the two foundational hypotheses of mutation testing hold for GOAL programs.

Keywords: Mutation Testing, Agent-Oriented Programming Languages, GOAL.

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. For example, Munroe *et al.* note that "*However, the task* [validation] *proved challenging for several reasons. First, agent-based systems explore realms of behaviour outside people's expectations and often yield surprises* ... " [12, Section 3.7.2]. Similarly, Pěchouček and Mařík [16, Page 413] note that[1]: "*Although the agent system performed very well in all the tests, to release the system for production would require testing all the steel recipes with all possible configurations of cooling boxes*".

There has been work on testing of multi-agent systems, especially in the last 4-5 years. Most of this work has focussed on tool support for executing (manually defined) tests (e.g. [2,3]). However, some work has investigated test generation based on design models [21], ontologies [13], or using evolutionary techniques [14]. Space precludes

[1] On the other hand, for another application they note that: "*Even though this negotiation process has not been theoretically proved for cycles' avoidance* [sic], *practical experiments have validated its operation*" [16, Page 407].

a detailed review of testing, and we refer the reader to [20, Section 8.1] for a review of work on testing and debugging MAS. Overall, the conclusion of this review was that "*testing of agent based systems is an area where there is a need for substantial additional work*" [20, Section 8.1].

Given a collection of tests (a "test suite"), a key question when testing an agent system is to what extent is the test suite *adequate*? A test suite is adequate to the extent that it is able to distinguish between a correct and an incorrect program. In developing an adequate test suite, it is obviously useful to be able to *assess* the adequacy of the test suite. This assessment can assist a tester in detecting when a test suite is not sufficient and needs to be refined or extended. It can also guide a tester in knowing when to stop adding test cases.

So far, work on assessing the adequacy of test cases (e.g. [9,11,19]) has only considered the use of various *coverage metrics* to assess test suite adequacy. However, although coverage is necessary, it is 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 these parts of the program were tested in a way that allows errors in the program to be detected, i.e. to distinguish between correct and incorrect programs.

An alternative, well-established, technique for assessing test suite adequacy is *mutation testing* [6] (see Section 2.1). 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*" [10, Page 503].

Most work on mutation testing of programs has focussed on programs in procedural and object-oriented languages [6, Figure 5]. There has been a (very) small amount of work on applying mutation to agents [18,1]. However, this work has not considered mutating agent programs written in a cognitive agent-oriented programming language.

In this paper we propose a set of mutation operators for the cognitive agent-oriented programming language GOAL (see Section 2.2 for a brief introduction to the language). Although we derive mutation operators for a specific language, the process by which the operators are derived is generic, and can easily be applied to other agent-oriented programming languages (see Section 6).

In deriving our mutation operators we are guided by an exploration of actual bugs in GOAL programs. We want mutation operators to generate "realistic" bugs, and we assess this by considering a collection of GOAL programs (written by undergraduate students at another university). Section 4 compares the bugs that exist in these programs against the sorts of bugs that our mutation operators generate, and uses the results to guide the selection of mutation operators. In fact, the results of this assessment of bugs also forms an additional contribution, in that it provides evidence of the extent to which the two foundational hypotheses of mutation testing hold for GOAL programs.

The remainder of this paper is structured as follows. We begin by briefly reviewing mutation testing (Section 2.1) and introducing the GOAL programming language (Section 2.2). We then present our mutation operators in Section 3. Section 4 looks at

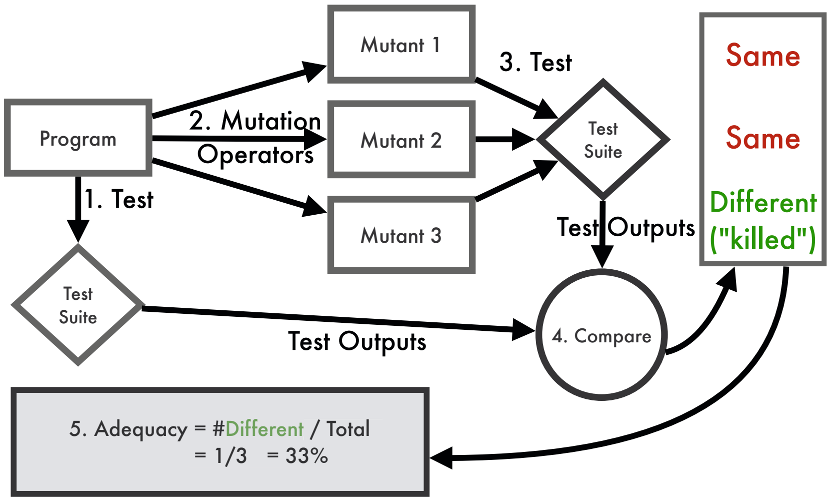

Fig. 1. Mutation Testing Process

a collection of GOAL programs and considers what bugs they contain. We then (Section 5) describe an implementation of the mutation operators, and report the number of mutants generated by different operators for a number of example GOAL programs. Finally, we conclude with a discussion, including future work (Section 6).

2 Background

2.1 Mutation Testing

We now very briefly introduce the key ideas of mutation testing, which is a long established field, going back to the 70s. For a detailed introduction to mutation testing see Chapter 7 of [10], and for a recent review of the field see Jia & Harman [6]. 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"). Given a test suite, a program P written in a programming language, and a set of mutation operators for that programming language, the process of mutation testing is as follows (see Figure 1):

1. Execute the program P against all tests in the test suite, recording the results;
2. Use the mutation operators to generate a set of mutant programs $P_1 \ldots P_n$ from P (where each P_i is the result of applying a single mutation);
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"[2];
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 for the programming language used. A set of mutation operators is good to the extent that it (1) generates errors that are realistic; and (2) does so without generating a huge number of mutants.

Mutation testing rests on two foundational hypotheses [6]. The first is the *competent programmer hypothesis*, which states that programmers tend to develop programs that are close to being correct. This hypothesis is important in that a consequent of it is that a simple syntactic mutation is a good approximation of the faults created by competent programmers. In other words, the competent programmer hypothesis is what justifies the use of simple syntactical mutations as proxies for real bugs. The second foundational hypothesis is the *coupling effect hypothesis*. This 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 [15]. This hypothesis justifies the generation of mutants by the application of a single mutation operator instance, rather than having to consider the application of multiple mutation operators to generate a mutant.

In Section 4 we consider a collection of GOAL programs and assess to what extent these two foundational hypotheses hold. Although there is empirical evidence to support both these hypotheses for procedural programs, this paper is the first to consider evidence for these hypotheses in the context of agent systems.

2.2 GOAL

This section briefly introduces GOAL (**G**oal **O**riented **A**gent **L**anguage); for further details we refer the reader to the existing literature [5,4]. A Multiagent System in GOAL is defined using a configuration file that specifies the environment, configuration options, and a number of GOAL agents, each with a GOAL program. A GOAL agent program consists of five components: domain knowledge (e.g. Prolog rules), initial beliefs, initial goals, action definitions, and a program. Note that both the domain knowledge and the beliefs are specified using a knowledge representation language which can be varied (the GOAL implementation uses SWI-Prolog). In this paper we focus on the program component, both because it corresponds most closely to other agent-oriented programming languages (AOPLs), and because that is usually where the complexity of the agent is [17], and where errors are made (see Section 4).

[2] Some mutants may be equivalent in behaviour to the original program ("equivalent mutants"), and, since program equivalence is undecidable, identifying and removing these mutants is a manual and partial process. This is a standard problem in the field of mutation testing but there is evidence that most equivalent mutants are actually fairly easy to detect. A related issue is where a mutant may be non-equivalent, but may still be correct. Mutation testing is not concerned with whether a mutant is correct, but with whether it is *different*, and whether this difference can be detected by a test suite.

GOAL programs are built out of actions and mental state conditions. Actions in GOAL are either user-defined (pre and post condition), or are one of the five built-in actions that insert or delete beliefs, adopt or drop goals, or send a message. GOAL also defines an achievement goal **a-goal**$(\phi) \equiv$ **goal**$(\phi) \wedge \neg$**bel**(ϕ) and an achieved goal **goal-a**$(\phi) \equiv$ **goal**$(\phi) \wedge$ **bel**(ϕ). A mental state condition (MSC) in GOAL is built out of conditions over the agent's beliefs and its goals. A GOAL program definition then, in essence, consists of a sequence of rules of the form[3] "**if** MSC **then** action$_1 + \ldots +$ action$_n$" where the actions are performed in order[4], and there can be at most one user-defined action.

These rules are actually placed within modules. However, in this paper we do not consider the mutation of modules, since the module construct is unique to GOAL, and was not used in the programs we considered (see Section 4). The grammar in Figure 2 summarises the subset of the language that we focus on. It thus differs from the original grammar given by Hindriks [5] in that it omits modules. It also differs in a couple of places where it has been changed to match what the implementation supports (specifically for **drop**(ϕ) the ϕ must not contain negations; and in fact in mentalatoms belief conditions can actually contain disjunctions).

Semantics: A rule "**if** condition **then** action" is *applicable* if the condition holds, and is *enabled* if the actions' preconditions are met. Applicable and enabled rules are *options*. The execution cycle consists of the following steps:

1. Clear previous cycle's percepts.
2. Update percepts by executing *all* options (i.e. enabled rules) in the distinguished `event` module.
3. Focus on the `main` module: compute the options, select one (by default rules are evaluated in linear order and the first option is selected), and perform its actions.
4. Update goals by dropping goals that are believed to hold.

Compared with other cognitive agent programming languages, GOAL's most distinctive (relevant) features are: (a) The limitation that an action rule can only result in a sequence of actions, rather than a mixture of actions and subgoals; and (b) The lack of a trigger condition. This makes GOAL action rules more general, in that a rule doesn't require a particular trigger. However, it also means that a rule can be applied repeatedly: in, say, Jason a rule of the form $+!$*goal* : *context* $\leftarrow$ *planBody* can be applied (if the *context* is true) to deal with the creation of a *goal*. However, the rule will not be applied subsequently unless the goal is re-posted. By contrast, in GOAL a rule of the form "if goal(*goal*) then *actions*" can be applied repeatedly, as long as *goal* remains a goal of the agent.

[3] There is also a form "**forall** MSC **do** actions" used in the percept processing module.

[4] The GOAL documentation states that "The actions that are part of a composed action may be put in any order in the action part of a rule. However, the order the actions are put in is taken into account when executing the composed action: **The actions are executed in the order they appear**" (emphasis added).

```
          program ::= actionrule+
       actionrule ::= if mentalstatecond then actioncombo
                    | forall mentalstatecond do actioncombo

 mentalstatecond ::= mentalliteral { , mentalliteral }*
    mentalliteral ::= true | mentalatom | not( mentalatom )
       mentalatom ::= bel( litconj ) | goal( litconj )

      actioncombo ::= action { + action }*
           action ::= user-def-action | built-in-action | communication
  user-def-action ::= id[parameters]
  built-in-action ::= insert( litconj ) | delete( litconj )
                    | adopt( poslitconj ) | drop( poslitconj )
    communication ::= send( id , poslitconj )

       poslitconj ::= atom { , atom }* .
          litdisj ::= litconj { ; litconj }* .
          litconj ::= literal { , literal }* .
          literal ::= atom | not( atom )

             atom ::= predicate[parameters] | ( litdisj )
       parameters ::= ( term { , term }* )
```

Fig. 2. GOAL Agent Program syntax (adapted from [5]): *term* is a legal term and *id* is an identifier

3 Deriving GOAL Mutation Operators

"*the design of mutation operators is as much of an art as it is science.*" [10, Page 530].

In deriving a set of mutation operators for GOAL we follow the approach of Kim *et al.* [8] 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.* 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. Figure 3 shows their interpretation of the HAZOP guide words (note that some of the guide words, such as those to do with scope, or quantitative changes, are not applicable to GOAL, and so have been left out of the figure).

Guide Words	Interpretation
NO/NONE	No part of the intention is achieved. No use of syntactic components.
AS WELL AS	Specific design intent is achieved but with additional results
PART OF	Only some of the intention is achieved, incomplete
REVERSE	Reverse flow - flow of information in wrong direction . . . negation of condition
OTHER THAN	A result other than the original intention is achieved, complete but incorrect

Fig. 3. HAZOP guide words and their interpretation for software (copied from [8])

In deriving mutation operators for GOAL we actually go through two stages. We firstly apply HAZOP to abstract syntactical classes in order to develop generic mutation schemas (Figure 4). These are generic in that they are applicable to a wide range of programming languages. We then apply these schemas to the GOAL syntax to generate specific mutation rules for GOAL (Figure 5). The advantage of doing the derivation in two stages is that we can then more easily derive mutation operators for other AOPLs by applying the generic schemas.

In deriving our generic schemas we consider three generic syntactical types: a sequence of elements, a (binary) operator that has two sub-elements, and a unary operator. For each of these generic syntactical types we consider what mutations are suggested by the HAZOP guide words, which gives a set of generic mutation schemas for that syntactical type. In addition, there is also a generic schema, suggested by the NO/NONE HAZOP guide word, that can be applied to delete ("drop") *any* syntactical type. Formally we write drop:$x \leadsto \epsilon$ to capture this: the "drop:" is a label, x is a variable for a syntactical element, the arrow "$\leadsto$" indicates a mutation, and ϵ denotes the empty syntactical construct of the appropriate form (e.g. empty sequence, "True" Boolean value).

Consider now a sequence of elements ($x_1 \ldots x_n$). The HAZOP guide word PART OF suggests that we consider removing an element in the sequence ("drop1" in Figure 4). The OTHER THAN guide word suggests changing part of the sequence, specifically, we select an element, and replace it with a variant (derived using other appropriate mutation operators; "mut1"). The REVERSE suggests changing the order of the sequence. However, in general reversing a sequence of syntactic elements doesn't make much sense, and instead we propose a rule to swap two adjacent elements in the sequence ("seqswap"): note that we choose to only swap adjacent elements in order to avoid generating a large number of possible mutants (but see the discussion of program:seqtop and seqbot later in this section). The AS WELL AS guide word suggests that we add an item to the sequence. However, we prefer to avoid adding things because this raises the issue of what to add? If we add, say, a new rule to a GOAL program, what rule should we add? It is possible to define a way of creating a new rule from existing fragments in the program, but this tends to result in a very large number of possible mutations. The NO/NONE guide word has already been handled by a rule that applies to all syntactical types, including sequences.

Consider now a binary operator (notation: $x \oplus y$ or $x \otimes y$, where we assume that $\oplus$ and $\otimes$ are different). Using similar reasoning, we are inspired by PART OF to consider dropping either the left or right element ("dropL", "dropR"); by REVERSE to swap the elements ("swap2"); and by OTHER THAN to change either the operator ("op2") or to mutate one of the elements ("mutL", "mutR").

Finally, consider a unary operator (notation: $\Box x$ or $\Diamond x$, assuming $\Box \neq \Diamond$). Using similar reasoning, we are inspired by the PART OF guide word to delete the operator ("delop"); by OTHER THAN to change the operator ("op1") or mutate the component ("mut"). We also can add an operator ("addop") which can be seen as inspired by the "negation of condition" interpretation of the REVERSE guide word (e.g. $F \rightsquigarrow \neg F$).

Figure 4 shows the resulting generic mutation schemas. Recall that each rule is of the form "keyword : $x \rightsquigarrow y$". We also employ a convention that where we have $p(x) \rightsquigarrow p(x')$, there is an implied "if $x \rightsquigarrow x'$". In other words, the mut1 rule, for instance, is really shorthand for "mut1: $x_1 \dots x_j \dots x_n \rightsquigarrow x_1 \dots x'_j \dots x_n$ if $x_j \rightsquigarrow x'_j$".

drop: $x \rightsquigarrow \epsilon$

seqswap: $x_1 \dots x_j\ x_{j+1} \dots x_n \rightsquigarrow x_1 \dots x_{j+1}\ x_j \dots x_n$
mut1: $x_1 \dots x_j \dots x_n \rightsquigarrow x_1 \dots x'_j \dots x_n$
drop1: $x_1 \dots x_j\ x_{j+1} \dots x_n \rightsquigarrow x_1 \dots x_{j+1} \dots x_n$

dropL: $x \oplus y \rightsquigarrow y$	dropR: $x \oplus y \rightsquigarrow x$	swap2: $x \oplus y \rightsquigarrow y \oplus x$	
op2: $x \oplus y \rightsquigarrow x \otimes y$	mutL: $x \oplus y \rightsquigarrow x' \oplus y$	mutR: $x \oplus y \rightsquigarrow x \oplus y'$	
addop: $x \rightsquigarrow \Diamond x$	delop: $\Diamond x \rightsquigarrow x$	op1: $\Diamond x \rightsquigarrow \Box x$	mut: $\Diamond x \rightsquigarrow \Diamond x'$

Fig. 4. Generic Mutation Schemas

The second step is to apply these generic mutation schemas to the GOAL syntax in order to derive a set of mutation operators specific to GOAL. In doing so, we sometimes leave out rules that don't make sense. For example, when a binary operator is commutative, it doesn't make sense to mutate by swapping its arguments ("swap2"). We now proceed to consider in turn each syntactical element type in GOAL and consider how the generic mutation schemas apply to it.

We begin by considering a GOAL program. This is a *sequence* of action rules, and therefore the relevant generic mutation schemas are those for a sequence (seqswap, mut1, drop1), as well as the universal "drop" schema. In this case, it doesn't make sense to drop the whole program, so we have three rules (labelled in Figure 5 "program:seqswap", "program:mut1" and "program:drop1"). For example, given a program that consists of the rules $r_1 r_2 r_3$ we could apply the mutation operator program:seqswap to swap any two rules, for example swapping r_1 and r_2 to obtain the mutated program $r_2 r_1 r_3$. We could alternatively apply program:drop1 to remove a single rule, for instance dropping r_1 yielding the mutated program $r_2 r_3$. A final option is to select a rule, for instance r_2, and mutate it using a rule mutation operator, yielding the mutated rule r'_2, and the overall mutated program $r_1 r'_2 r_3$.

In fact, in our exploration of bugs in example GOAL programs, we also found that the limitation to only swap adjacent rules in a program was too strong: there were a number of cases where bugs corresponded to other sorts of changes to the order of rules in a program. Although we do not want to introduce a mutation operator to allow arbitrary reorderings of the rules in a program, we do propose a compromise that allows many of the bugs seen to be generated by our mutation rules, whilst not increasing the number of possible mutants too much. This compromise is to add mutation operators that allow a single rule in the program to be moved to the start ("program:seqtop") or end ("program:seqbot") of the program. Applying these rules to the program $r_1r_2r_3r_4r_5$ with rule r_3 being moved yield respectively the mutated programs $r_3r_1r_2r_4r_5$ and $r_1r_2r_4r_5r_3$.

Next we consider a GOAL action rule (abbreviated AR). An action rule is effectively a binary connective with two sub-components, and hence the generic schemas for binary connectives apply (i.e. dropL, dropR, swap2, op2, mutL, mutR, as well as drop). However, for an AR the components cannot be deleted, since a rule must have both a condition and actions (although an MSC could be replaced with "true"), and they cannot be swapped, so we only have rules for op2, mutL, and mutR. For op2 we consider replacing "if *MSC* then *AC*" with "forall *MSC* do *AC*" (and vice versa), but only in the context of the percept processing module. Note that op2 has two instances, and that it is fairly specific to GOAL: other AOPLs don't deal with percepts in the same way. The universal "drop" rule isn't needed for actionrules, since actionrules only occur within a sequence, and we already have a rule to delete an element in the sequence. Thus, for an action rule in the main module (i.e. not in the percept module), we can only mutate it by selecting either its mental state condition or its action combo, and using an appropriate mutation operator to mutate the selected element.

A mentalstatecond (MSC) is also a sequence. Here it *does* make sense to also consider the overall drop rule, dropping the whole MSC, as well as the usual mut1 and drop1 rules. However, in fact the result of dropping an MSC completely is rarely a valid GOAL program: GOAL requires that variables appearing in the actions of a rule also appear in that rule's condition. Since this requirement only holds for a "true" condition when the action list has no variables, the MSC:drop rule is unlikely to ever be applicable (see Section 5). Note that we only consider mutation by dropping an MSC if it has more than one element (otherwise the same effect is achieved by the ML:drop rule). Finally, we did not initially define mutation operators to change the order of conditions in an MSC, since in many cases the order doesn't matter, for instance where each condition has disjoint variables. However, in other cases the order may matter, and we could consider extending our set of mutation operators with rules to change the order as future work.

A mentalliteral (ML), as defined in the GOAL syntax, is an optional unary operator ("not") that is applied to a mental atom (which itself is either a **bel** or a **goal** operator applied to a litconj). We can mutate a mentalliteral by dropping it completely. We can also add or remove a "not" ("addop", "delop"), or we can mutate the mental atom. Mutating a mental atom (MA) can be done by either changing a **bel** to a **goal** (or vice versa), or by mutating the literal conjunction. Note that we cannot mutate the "not" into another operator, since there is no alternative operator.

An actioncombo (AC) is a sequence of actions. It cannot be entirely deleted. However, we can mutate an individual action or drop one. Note that the swap rule doesn't really make sense: although GOAL specifies that actions in an actioncombo are executed sequentially, it would in fact be non-idiomatic to have a sequence of actions that is order dependent.

An action (A) is either user defined ("id[parameters]") or is one of the five built-in actions: insert, delete, adopt, drop, send. Dropping an action completely is already covered by the rule AC:drop1, so we only consider mutating the parameters of the action, or the action type. In mutating the action type we exclude changing a belief operation to a goal operation and vice versa, since this doesn't make sense, and is unlikely to yield a sensible mutant. When mutating a message, we can mutate the message content, or the recipient. Finally, we can mutate a user-defined action by mutating either the id (by replacing it with a different user defined action or by mutating the parameters. In replacing a user-defined action with a different user-defined action we need to ensure that the two actions have the same number of parameters (which we term being "compatible"). This condition also applies when mutating atoms by changing their predicate.

We did observe that some programs had "typos" (e.g having "at" instead of "atBlock") in predicates. However, we did not introduce a mutation operator to create such typos for the simple reason that this operator would be redundant. Consider, for example, replacing the action "delete(p)" with "delete(typo)". This is actually equivalent to just deleting the action. Similarly, replacing "bel(p)" with "bel(typo)" in an MSC is equivalent to replacing it with "false", i.e. with deleting the rule; and having "adopt(typo)" is equivalent to a null action since the goal won't have rules that handle it, so won't have any effect.

A poslitconj (PLC) and a litconj (LC) are both sequences (respectively of Atoms "At" or Literals "Lit"), so we can remove an element of the sequence or mutate an element of the sequence. As for MSCs, we did not define swapping operations, but these could be considered as future work.

A Literal is an optional unary connective ("not") applied to an Atom, and hence can be mutated by adding or removing a negation, or by mutating the atom. Mutating an atom can be done by mutating the predicate (replacing it with another predicate found in the program), or by mutating the parameters. Parameters are a sequence of terms (we abbreviate $term$ to t), but we do not want to change the length of the sequence, hence can only mutate individual terms. However, swapping terms is a reasonable mutation.

Mutating a list (of the form $[t_1|t_2]$) is done similarly to any other binary connective, yielding the following rules (for space reasons, these are not shown in Figure 5):

termlist:drop1	$[A\|As] \rightsquigarrow A$	$[A\|As] \rightsquigarrow As$
termlist:seqswap	$[A\|As] \rightsquigarrow [As\|A]$	
termlist:mut	$[A\|As] \rightsquigarrow [A'\|As]$	$[A\|As] \rightsquigarrow [A\|As']$

Finally, we consider the mutation of terms (excluding lists). A term is of the form $f(t_1, \ldots, t_n)$ and, viewed as a sequence of arguments, can be mutated by dropping a sub-term ("term:drop1"), mutating a sub-term, or swapping adjacent sub-terms. There is one special case: equality. For the term $t_1 = t_2$ it does not make sense to drop either sub-term, nor to swap the sub-terms.

Figure 5 shows the collected mutation operators for GOAL. Recall that by convention where we have $p(x) \rightsquigarrow p(x')$, there is an implied "if $x \rightsquigarrow x'$". We also assume that there are implicit checks for syntactic elements being of the correct type. For instance, in the rule ML:addop, the element MA must be a MentalAtom, and hence cannot be "true" or "not(MA)". There is also a constraint: for those rules that involve an element $j+1$ (i.e. the swap and drop1 rules) we have $1 \leq j$ and $j+1 \leq n$, and hence that $n \geq 2$ (so we don't drop the last element in a list). For other rules we have $1 \leq j \leq n$. Finally, the program:mut rule has an additional condition, discussed above, that all variables appearing in the actions also appear in the rule's condition.

program:seqtop $AR_1 \ldots AR_j\ AR_{j+1} \ldots AR_n \rightsquigarrow AR_j\ AR_1 \ldots AR_{j+1} \ldots AR_n$
program:seqbot $AR_1 \ldots AR_j\ AR_{j+1} \ldots AR_n \rightsquigarrow AR_1 \ldots AR_{j+1} \ldots AR_n\ AR_j$
program:seqswap $AR_1 \ldots AR_j\ AR_{j+1} \ldots AR_n \rightsquigarrow AR_1 \ldots AR_{j+1}\ AR_j \ldots AR_n$
program:mut1 $AR_1 \ldots AR_j \ldots AR_n \rightsquigarrow AR_1 \ldots AR'_j \ldots AR_n$ (see text for condition)
program:drop1 $AR_1 \ldots AR_j\ AR_{j+1} \ldots AR_n \rightsquigarrow AR_1 \ldots AR_{j+1} \ldots AR_n$

AR:op2 **if** *msc* **then** *actioncombo* $\rightsquigarrow$ **forall** *msc* **do** *actioncombo* (only percept rules)
AR:op2 **forall** *msc* **do** *actioncombo* $\rightsquigarrow$ **if** *msc* **then** *actioncombo* (only percept rules)
AR:mutL **if** *msc* **then** *actioncombo* $\rightsquigarrow$ **if** *msc*$'$ **then** *actioncombo*
AR:mutR **if** *msc* **then** *actioncombo* $\rightsquigarrow$ **if** *msc* **then** *actioncombo*$'$
AR:mutL **forall** *msc* **do** *actioncombo* $\rightsquigarrow$ **forall** *msc*$'$ **do** *actioncombo*
AR:mutR **forall** *msc* **do** *actioncombo* $\rightsquigarrow$ **forall** *msc* **do** *actioncombo*$'$

MSC:drop $ML_1 \ldots ML_j\ ML_{j+1} \ldots ML_n \rightsquigarrow$ true
MSC:mut1 $ML_1 \ldots ML_j \ldots ML_n \rightsquigarrow ML_1 \ldots ML'_j \ldots ML_n$

ML:drop $ML \rightsquigarrow$ true (if $ML \neq$ true)
ML:addop $MA \rightsquigarrow$ **not(** MA **)**   ML:delop **not(** MA **)** $\rightsquigarrow MA$
ML:mut $MA \rightsquigarrow MA'$ **not(** MA **)** $\rightsquigarrow$ **not(** MA' **)**
MA:op1 **bel(** *litconj* **)** $\rightsquigarrow$ **goal(** *litconj* **)**   **goal(** *litconj* **)** $\rightsquigarrow$ **bel(** *litconj* **)**
MA:mut $\Diamond$(*litconj*) $\rightsquigarrow$ $\Diamond$(*litconj*$'$) where $\Diamond \in \{$**bel**, **goal**$\}$

AC:mut1 $A_1 \ldots A_j \ldots A_n \rightsquigarrow A_1 \ldots A'_j \ldots A_n$
AC:drop1 $A_1 \ldots A_j\ A_{j+1} \ldots A_n \rightsquigarrow A_1 \ldots A_{j+1} \ldots A_n$

A:op1 **insert(** *litconj* **)** $\rightsquigarrow$ **delete(** *litconj* **)**   **delete(** *litconj* **)** $\rightsquigarrow$ **insert(** *litconj* **)**
A:op1 **adopt(** *poslitconj* **)** $\rightsquigarrow$ **drop(** *poslitconj* **)**   **drop(** *poslitconj* **)** $\rightsquigarrow$ **adopt(** *poslitconj* **)**
A:mut $\Diamond$(*litconj*) $\rightsquigarrow$ $\Diamond$(*litconj*$'$) where $\Diamond \in \{$**insert**, **delete**$\}$
A:mut $\Diamond$(*poslitconj*) $\rightsquigarrow$ $\Diamond$(*poslitconj*$'$) where $\Diamond \in \{$**adopt**, **drop**$\}$
A:mut **send(** *id* **,** *poslitconj* **)** $\rightsquigarrow$ **send(** *id* **,** *poslitconj*$'$ **)**
A:mut **send(** *id* **,** *poslitconj* **)** $\rightsquigarrow$ **send(** *id*$'$ **,** *poslitconj* **)**
A:mut(*) *id*$[parameters] \rightsquigarrow$ *id*$'[parameters]$
A:mut *id*$[parameters] \rightsquigarrow$ *id*$[parameters']$

PLC:mut1 $At_1 \ldots At_j \ldots At_n \rightsquigarrow At_1 \ldots At'_j \ldots At_n$
PLC:drop1 $At_1 \ldots At_j\ At_{j+1} \ldots At_n \rightsquigarrow At_1 \ldots At_{j+1} \ldots At_n$
LC:mut1 $Lit_1 \ldots Lit_j \ldots Lit_n \rightsquigarrow Lit_1 \ldots Lit'_j \ldots Lit_n$
LC:drop1 $Lit_1 \ldots Lit_j\ Lit_{j+1} \ldots Lit_n \rightsquigarrow Lit_1 \ldots Lit_{j+1} \ldots Lit_n$

Lit:addop $At \rightsquigarrow$ **not(** At **)**   Lit:delop **not(** At **)** $\rightsquigarrow At$
Lit:mut $At \rightsquigarrow At'$ **not(** At **)** $\rightsquigarrow$ **not(** At' **)**
At:mut(*) $predicate[parameters] \rightsquigarrow predicate'[parameters]$
At:mut $predicate[parameters] \rightsquigarrow predicate[parameters']$
At:mut $lit_1; \ldots; lit_j; \ldots; lit_n \rightsquigarrow lit_1; \ldots; lit'_j; \ldots lit_n$
parameters:seqswap $t_1 \ldots t_j\ t_{j+1} \ldots t_n \rightsquigarrow t_1 \ldots t_{j+1}\ t_j \ldots t_n$
parameters:mut $t_1 \ldots t_j \ldots t_n \rightsquigarrow t_1 \ldots t'_j \ldots t_n$
term:drop1 $f(t_1 \ldots t_j, t_{j+1} \ldots t_n) \rightsquigarrow f(t_1 \ldots t_{j+1} \ldots t_n)$
term:mut1 $f(t_1 \ldots t_j, t_{j+1} \ldots t_n) \rightsquigarrow f(t_1 \ldots t_j, t', t_{j+1} \ldots t_n)$
term:seqswap $f(t_1 \ldots t_j, t_{j+1} \ldots t_n) \rightsquigarrow f(t_1 \ldots t_{j+1}, t_j \ldots t_n)$
term:mut $X = Y \rightsquigarrow X' = Y$ $X = Y \rightsquigarrow X = Y'$

Constraints: $1 \leq j$ and $j+1 \leq n$ for constraints that have $j+1$, otherwise $1 \leq j \leq n$
(*) = compatibility constraint (see text)

Fig. 5. Mutation Operators for GOAL

These rules have the property that when applied to a valid GOAL program, they result in another valid program, since they always replace a syntactical element of a certain type (e.g. MSC) with another valid syntactical element of the same type (see also Section 5). Figure 6 shows an example GOAL rule and its mutations (generated by the implementation described in Section 5).

Original:	if not(goal(target(A, B))), bel(holding(C)) then adopt(target(C, table)).
ml:drop	if **true**, bel(holding(A)) then adopt(target(A, table)).
ml:delop	if **goal**(target(A, B)), bel(holding(C)) then adopt(target(C, table)).
ma:op1	if not(**bel**(target(A, B))), bel(holding(C)) then adopt(target(C, table)).
ml:addop	if not(goal(target(A, B))), **not**(bel(holding(C))) then adopt(target(C, table)).
ma:op1	if not(goal(target(A, B))), **goal**(holding(C)) then adopt(target(C, table)).
lit:addop	if not(goal(target(A, B))), bel(**not**(holding(C))) then adopt(target(C, table)).
at:mut	if not(goal(target(A, B))), bel(**block**(C)) then adopt(target(C, table)).
a:op1	if not(goal(target(A, B))), bel(holding(C)) then **drop**(target(C, table)).
parameters:seqswap	if not(goal(target(A, B))), bel(holding(C)) then adopt(target(**table, C**)).

Fig. 6. GOAL clause and example mutations, with the changes **highlighted**

4 An Empirical Evaluation of Programs

We now turn to an empirical evaluation by examining a collection of GOAL programs. The aim of this examination is primarily to assess how well the mutation operators are able to generate realistic bugs. However, we also briefly consider what our empirical evaluation tells us about the two foundational hypotheses of mutation testing (Section 4.1).

Methodology: We obtained a collection of 55 GOAL programs, written as an assignment by first year undergraduate students at Delft university. These programs each implement a solution to "Blocks World for Teams" (BW4T) [7]: a single[5] agent that moves around an environment with a number of rooms (see Figure 7), collecting blocks of various colours, and delivering them to the "dropzone" in a specified order (e.g. a red block, then a blue block). The environment (which runs in a separate process) provides the agent with percepts (e.g. in(Room), color(BlockID, Color), holding(BlockID)), and four actions (goTo(Location), goToBlock(BlockID), pickUp, and putDown).

We are interested in assessing how well the mutation operators are able to generate realistic bugs. We therefore consider the collection of GOAL programs as being a source of realistic buggy programs, and consider whether each of the buggy programs could have been generated from a correct program by applying our mutation operators. We therefore proceeded by testing each program to locate bugs, and then fixing the bugs. In fixing bugs we were careful to only make changes that were necessary, and to consider what alternative changes might be used to fix the bug. Once a bug was fixed we retested the program to confirm that the fix was correct. When testing the programs we

[5] We only considered the initial version of the assignment with one agent.

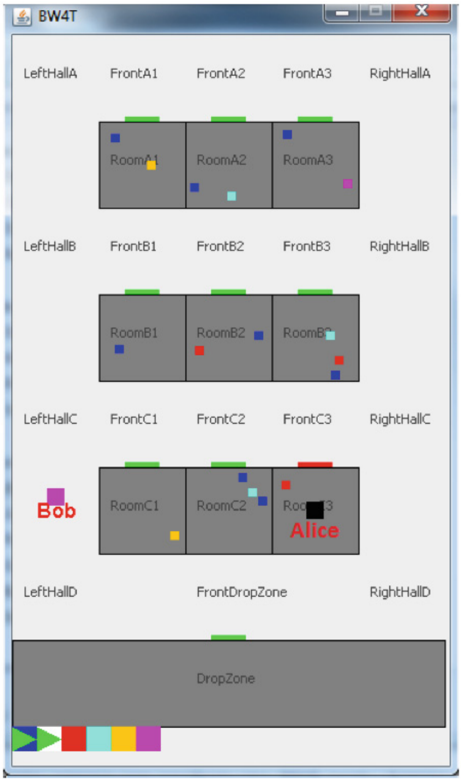

Fig. 7. Blocks World for Teams

used a number of test suites: the two example scenarios that were used in the original assignment, a set of ten randomly generated test cases, and a generated enumeration of all possible starting configurations within a limited scope (for scope sizes 1 and 2). Overall, we considered a program to be correct if it managed to deliver the desired blocks in all runs. Note that in some cases programs were successful in delivering all blocks, but had other issues, for example, a program might deliver incorrect blocks along the way, or may require a large number of additional actions. However, as long as it ended up delivering the desired blocks, we considered it to be "correct". Of the 55 programs, 4 were excluded, since they did not run at all (e.g. syntax errors), and 15 further programs were excluded since they did not have any (detected) bugs. This left a total of 36 buggy programs.

Before we consider how well the mutation operators are able to generate realistic bugs (as represented by the 36 buggy programs), we need to consider the assumptions that were made in developing the mutation operators. To what extent do these assumptions hold?

Recall that GOAL programs have a number of components (e.g. domain knowledge, action definitions), and that we have focussed on the program rules, i.e. assumed that errors only occur in the program rules. Is this a valid assumption? Out of the 36 programs with bugs, only 9 programs involved errors that related to non-supported GOAL features. One of these 9 programs involved incorrect usage of nested rules, and the remaining 8 programs had problems in the definition of actions, mostly incorrect definitions of their pre/post conditions[6]. The somewhat surprising number of programs with issues in defining actions may be due to a feature of the BW4T environment: the environment is a separate process, and there is a delay between performing an action, and the action actually taking place. This means that when defining an action, such as pickUp, the action's post-condition should be "true", rather than, say, "holding(Block)", because the environment will, in due course, perform the action and inform the agent of the action's success (or failure) by sending suitable percepts (such as "holding(Block)"). Having pickUp make holding(Block) true is a problem because in reality (i.e. in the environment) the agent may fail to pick up the block, or may take a while to succeed. If holding(Block) is asserted immediately (by an incorrect post condition), then the agent may then proceed to move to the dropzone, based on the false belief that it has already picked up the block.

The second assumption that we made in developing the mutation operators was to ignore certain features specific to GOAL, namely modules, nested rules, and macros. This assumption was clearly reasonable: of the 36 buggy programs, only 6 programs used nested rules (2 of these 6 also used macros). However, only one of these 6 programs had a bug that was related to the use of nested rules.

Having considered, and evaluated, the assumptions, we now consider to what extent the bugs that we observed could be seen as the result of one or more applications of the defined mutation operators. As noted earlier, 27 out of the 36 buggy programs had bugs that solely related to supported GOAL features. Of these 27 programs, 16 programs had errors that did not require additional mutation operators. The remaining 11 programs had errors that corresponded to the application of a number of mutation operator instances, where at least one of the operators was additional to the ones that we had defined. The additional mutation operators were: (i) addition of elements (either actions or literals) [7 programs]; (ii) changes to the order of rules in a program other than the cases defined[7] [3 programs]; (iii) mutating a variable to another (legal) variable name [3 programs]; and (iv) replacing an "insert" with a "drop" [1 program]. As discussed in Section 3, mutation operators that add to the program are problematic; however, the other three types of rules could be easily added.

Finally, we consider *which* of the mutation operators are used to generate the observed bugs. Figure 8 contains a summary of the number of times that each rule was used in deriving buggy programs, summed up over the 36 programs. Note that since some programs had bugs that corresponded to the application of multiple mutation operators, the sum of the number of rule applications (final row) is greater than the number of buggy programs. As can be seen in Figure 8, many of the rules that we defined actually do *not* correspond to the sorts of errors that we found in real buggy programs. Indeed, as often appears to be the case in mutation testing, only a few rules account for most of the bug types. For example, the four most commonly used rules (program:seqswap,

[6] Of these 8, one also had an error in the domain knowledge where a "$>$" should have been "$\geq$", and two had incorrectly defined actions using e.g. pickUp(Block) instead of pickUp.

[7] These could, in principle, be regarded as repeated application of program:seqswap.

Rule	Observed Bugs
a:mut	0
a:op1	0
ac:drop1	**14**
ar:op2	6
at:mut	4
lc:drop1	**18**
lit:addop	0
lit:delop	0
ma:op1	0
ml:addop	1
ml:delop	0
ml:drop	13
msc:drop	0
parameters:seqswap	0
plc:drop1	0
program:drop1	**29**
program:seqbot	9
program:seqswap	**31**
program:seqtop	5
term:drop1	0
term:seqswap	0
TOTAL	130

Fig. 8. Summary of observed bugs and the rules involved

program:drop1, LC:drop1, and AC:drop1, which are bolded in Figure 8) correspond to 71% of the mutation operator applications.

Overall, we conclude that: 75% (27 out of 36) of the programs had bugs that did not involve excluded GOAL features, such as modules, action definitions, or nested rules; 59% of these (16 out of 27) had bugs that were able to be generated by the mutation operators that we defined; and many (71%) of the mutation operator applications were instances of only four rules.

One question that might be asked is whether we could derive the mutation operators based on the buggy programs, rather than using the syntax-based approach discussed in Section 3. The advantage of the approach that we used is that it is based on the programming language itself, rather than on a given set of programs. In this case, since we had programs that all solved the same problem, if we had derived the operators based on the programs, there would be a danger that the operators would be biased to this specific problem.

4.1 Evidence for the Foundational Hypotheses

Recall that the field of mutation testing rests on two foundational hypotheses. The *competent programmer hypothesis* states that programmers write programs that are "almost correct", i.e. programs that are "a few mutants away from a correct program" [10, Page

# Mutation Operator Applications	# Programs
1	7
2	7
3	2
4	3
5	3
6	5
7	0
8	2
9	1
10	2
>10	4

Fig. 9. Number of programs that are N mutants away from being correct, for different values of N (see also Figure 10)

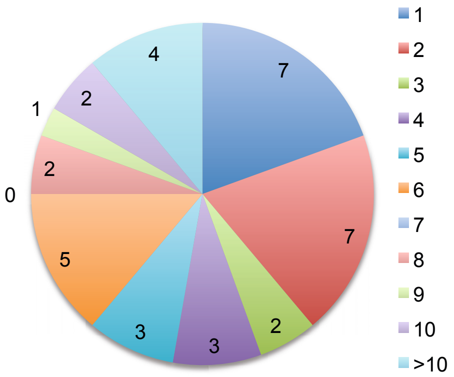

Fig. 10. Number of programs that are N mutants away from being correct, for different values of N (see also Figure 9)

531]. Of the 36 buggy programs, we found that 27 programs (75%) were indeed a few mutants away from a correct program (defining "a few" to be "6 or fewer"). Thus we conclude that the there is evidence that the competent programmer hypothesis holds for GOAL programs, even when they are written by first-year students. Figures 9 and 10 show how many programs were N mutation operators away from being correct. For instance, 7 of the 36 buggy programs corresponded to the application of a single mutation operator (first row of Figure 9), and 7 programs had 2 mutations (second row). The last row indicates that there were 4 programs that required more than 10 mutation operator applications: these required 11, 16, 20, and 22 applications respectively.

The *coupling effect hypothesis* states that a test set that is adequate with respect to single mutations is also adequate for multiple mutations. Since it concerns test sets and adequacy, this hypothesis is not easy to assess, and a full assessment is beyond the scope of this paper. However, we can provide some initial evidence: if, in fact, most of the observed bugs are generated by the application of a single mutation, then this would be evidence for the coupling effect hypothesis. Note that the converse is not true: even if the observed bugs mostly involve the application of multiple mutations, this does not mean that the coupling effect hypothesis fails to hold. There could be other mutants that can be used to assess whether the test suite is adequate with respect to the given bug. Considering the programs we found that of the 36 programs, 7 (19%) were generated by a single mutation operator application.

5 Implementation

The mutation operators defined in Figure 5 have been implemented. The implementation reads in a GOAL program and generates a collection of mutated programs, each of which is the result of applying a single mutation. The implementation considers mutations in both the main module, and in the percept processing module. It changes a single GOAL program rule, and then reassembles the complete program, including generating a modified mas2g file to run the mutated program.

We have run the implementation on three example GOAL programs (we selected the three longest examples in the GOAL distribution, excluding an example which uses modules extensively). The results were used in two ways. Firstly, we ran the mutants (for the 1st and 3rd programs) to check that each mutant was indeed a syntactically valid GOAL program (we couldn't do this for the 2nd program, because it could not be run from the command line, due to the way the agent's environment was implemented). Secondly, we observed which of the mutation operators were applicable to each program, and how many mutants were generated by the different rules. Figure 11 shows how many mutants were generated by the application of each of the mutation operators. Note that mutation operators that simply select part of a rule and invoke another mutation operator to make the change are not shown, since they do not actually change the program.

6 Discussion

We have presented rules for generating mutants of programs in a typical cognitive agent-oriented programming language (namely GOAL). We have also presented initial evidence that the rules are able to generate a significant proportion of the realistic bugs encountered in a simple problem, as well as evidence that supported the competent programmer foundational hypothesis of mutation testing in an agent context.

There are a number of issues (threats to validity) that need to be acknowledged. Firstly, we only considered a single problem, and although we considered 55 different programs, all these programs only involved a single agent, and all were written by relatively inexperienced programmers. Clearly, one area for future work is to revisit the empirical evaluation using a wider range of problems, and a wider range of programmers. Note that these limitations are the reason why we have derived the mutation operators systematically based on the syntactical structure of GOAL, rather than by considering what mutation operators correspond to errors in the GOAL programs.

Rule	Tower/ towerbuilder	BW4T2/ robot	Elevator/ elevatoragent
a:mut	0	0	0
a:op1	16	8	8
ac:drop1	2	2	0
ar:op2	6	6	6
at:mut	11	27	40
lc:drop1	20	8	13
lit:addop	39	17	20
lit:delop	8	4	5
ma:op1	26	7	12
ml:addop	27	8	12
ml:delop	4	0	2
ml:drop	19	0	5
msc:drop	0	0	0
parameters:seqswap	29	2	9
plc:drop1	0	0	6
program:drop1	18	8	9
program:seqbot	16	6	7
program:seqswap	16	6	7
program:seqtop	16	6	7
term:drop1	8	0	6
term:seqswap	4	0	3
TOTAL	285	115	177

Fig. 11. Mutants generated by different operators for three example GOAL programs

Another area for future work is assessing the prevalence of equivalent mutants, and whether equivalent mutants are generated by all mutation operators with roughly equal likelihood, or by certain rules. We also intend to apply this approach to define mutation operators for other AOPLs. Indeed, we have already defined mutation operators for AgentSpeak, but space precludes presenting them here. More broadly, the data that we have collected also tells us information on the sorts of mistakes that (novice) GOAL programmers make. Analysing the data from this perspective would be valuable.

Acknowledgements. We would like to thank our colleagues at Delft university for providing access to the student projects, and for answering questions about GOAL, and fixing bugs that we found in the BW4T environment.

References

1. Adra, S.F., McMinn, P.: Mutation operators for agent-based models. In: Proceedings of 5th International Workshop on Mutation Analysis. IEEE Computer Society (2010)
2. Ekinci, E.E., Tiryaki, A.M., Çetin, Ö., Dikenelli, O.: Goal-oriented agent testing revisited. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 173–186. Springer, Heidelberg (2009)

3. Gómez-Sanz, J.J., Botía, J., Serrano, E., Pavón, J.: Testing and debugging of MAS interactions with INGENIAS. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 199–212. Springer, Heidelberg (2009)
4. Hindriks, K.V.: Programming rational agents in GOAL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Tools and Applications, ch. 4, pp. 119–157. Springer (2009)
5. Hindriks, K.V.: Programming rational agents in GOAL (May 2011), `http://mmi.tudelft.nl/trac/goal`
6. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
7. Johnson, M., Jonker, C., van Riemsdijk, B., Feltovich, P.J., Bradshaw, J.M.: Joint Activity Testbed: Blocks World for Teams (BW4T). In: Aldewereld, H., Dignum, V., Picard, G. (eds.) ESAW 2009. LNCS, vol. 5881, pp. 254–256. Springer, Heidelberg (2009)
8. Kim, S., Clark, J.A., McDermid, J.A.: The rigorous generation of Java mutation operators using HAZOP. Technical Report 2/8/99, Department of Computer Science, University of York (1999)
9. Low, C.K., Chen, T.Y., Rönnquist, R.: Automated test case generation for BDI agents. Journal of Autonomous Agents and Multi-Agent Systems 2(4), 311–332 (1999)
10. Mathur, A.P.: Foundations of Software Testing. Pearson (2008) ISBN 978-81-317-1660-1
11. Miller, T., Padgham, L., Thangarajah, J.: Test coverage criteria for agent interaction testing. In: Weyns, D., Gleizes, M.P. (eds.) Proceedings of the 11th International Workshop on Agent Oriented Software Engineering, pp. 1–12 (2010)
12. Munroe, S., Miller, T., Belecheanu, R.A., Pěchouček, M., McBurney, P., Luck, M.: Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. Knowledge Engineering Review 21(4), 345–392 (2006)
13. Nguyen, C.D., Perini, A., Tonella, P.: Experimental evaluation of ontology-based test generation for multi-agent systems. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 187–198. Springer, Heidelberg (2009)
14. Nguyen, C.D., Perini, A., Tonella, P.: Goal-Oriented Testing for MASs. International Journal of Agent-Oriented Software Engineering 4(1), 79–109 (2010)
15. Offutt, A.: Investigations of the software testing coupling effect. ACM Transactions on Software Engineering and Methodology 1(1), 5–20 (1992)
16. Pěchouček, M., Mařík, V.: Industrial deployment of multi-agent technologies: review and selected case studies. Journal of Autonomous Agents and Multi-Agent Systems 17, 397–431 (2008)
17. van Riemsdijk, M.B., Hindriks, K.V., Jonker, C.M.: An empirical study of cognitive agent programs. Multiagent and Grid Systems 8(2), 187–222 (2012)
18. Saifan, A.A., Wahsheh, H.A.: Mutation operators for JADE mobile agent systems. In: Proceedings of the 3rd International Conference on Information and Communication Systems, ICICS (2012)
19. Thangarajah, J., Sardiña, S., Padgham, L.: Measuring plan coverage and overlap for agent reasoning. In: van der Hoek, W., Padgham, L., Conitzer, V., Winikoff, M. (eds.) International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, 3 Volumes, Valencia, Spain, June 4-8, pp. 1049–1056. IFAAMAS (2012)
20. Winikoff, M., Padgham, L.: Agent oriented software engineering. In: Weiss, G. (ed.) Multiagent Systems, ch. 5, 2nd edn. MIT Press (2013)
21. Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing for agent systems. In: 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), pp. 10–18 (2007)