# Mutation Testing for Jason Agents

Zhan Huang, Rob Alexander, and John Clark

Department of Computer Science, University of York, York, United Kingdom
{zhan.huang,robert.alexander,john.clark}@cs.york.ac.uk

**Abstract.** Most multi-agent system (MAS) testing techniques lack empirical evidence of their effectiveness. Since finding tests that can reveal a large proportion of possible faults is a key goal in testing, we need techniques to assess the fault detection ability of test sets for MAS. Mutation testing offers a direct and powerful way to do this: it generates modified versions of the program ("mutants") following a set of rules ("mutation operators") then checks if a test set can distinguish the original program from the (functionally non-equivalent) mutants. In this paper, we propose a set of mutation operators for the agent-oriented programming language Jason, and then introduce a mutation testing system for individual Jason agents that implements a subset of our proposed mutation operators. We use this subset to assess a test set for a Jason agent that meets a combination of existing agent-based coverage criteria. The assessment shows that this test set is not adequate to kill all mutants.

**Keywords:** Test Evaluation, Mutation Testing, Agent-Oriented Programming, Jason.

## 1 Introduction

Multi-agent systems (MAS) are a promising paradigm for engineering autonomous and distributed systems. Testing MAS is a challenging activity, however, because of the increased complexity, large amount of data, irreproducibility, non-determinism and other characteristics involved in MAS [9]. Although many techniques have been proposed to address the difficulties in MAS testing, most of them lack empirical evidence of their effectiveness [10].

Effective testing requires tests that are capable of revealing a high proportion of faults in the system under test (SUT). It can be difficult to find real faulty projects to verify the real fault detection ability of a test set; instead, we can use coverage-based and fault-based testing techniques.

For coverage based techniques, the tests or their executions are measured against some coverage criteria based on some model of the SUT (or other relevant model); if they cover all model elements defined in the coverage criteria, the tests are said to be adequate for the coverage criteria – in other words, they examine the involved model elements thoroughly. Existing coverage criteria for MAS testing include Low et al.'s plan and node based coverage criteria for BDI agents [1], Zhang et al.'s plan and event based coverage criteria for Prometheus agents [2], and Miller et al.'s protocol and plan based coverage criteria for agent interaction testing [3].

F. Dalpiaz et al. (Eds.): EMAS 2014, LNAI 8758, pp. 309–327, 2014.

Fault based techniques offer a more direct way to assess the fault detection ability of the tests than coverage based ones: faults are seeded into the SUT by some means, typically by hand or by *mutation* [12]. After seeding faults (i.e. producing faulty versions of the SUT), each test is executed against first the original SUT then each faulty version. For each faulty version, if its behaviour differs from the original SUT in at least one test, it will be marked as "killed" to indicate that the fault(s) seeded in it can be detected by the tests. Therefore, the fault detection ability of the tests can be assessed by the "kill rate" – the ratio of the killed faulty versions to all faulty versions: higher the ratio is, more effective the tests are. Those non-killed faulty versions reveal the weaknesses of the existing tests so that testers can enhance these tests (in order to kill those versions) by improving some of them or adding new ones.

Mutation is a systematic and automatic way of generating modified versions of the SUT ("mutants") following a set of rules ("mutation operators"). The process of using mutation to assess tests is called mutation testing. Mutation is more commonly used to seed faults than the hand-seeded way because many theories and empirical evidences support it; for instance [13] shows that it provides an efficient way to seed faults that are more representative of realistic faults than hand-seeded ones. However, the mutation operators used to guide mutant generation may lead to a large number of mutants so that comparing the behaviour of each mutant with that of the original SUT in each test is computationally costly. Another problem is that mutation unpredictably produces *equivalent mutants* – alternate implementations of the SUT that are not actually faulty (as the result, no tests can differentiate the original SUT from them), and thus which must be excluded from test evaluation. Although the process of detecting equivalent mutants may be partially automated, much manual work is still required.

Many studies show that mutation testing provides a more rigorous test evaluation than coverage-based techniques [11], so it is usually used to evaluate or compare other testing techniques (e.g. that are based on some coverage criteria). The key to successful mutation testing is to select an appropriate set of mutation operators. Here we define "appropriate" in terms of two criteria: effectiveness and efficiency. Effectiveness is the value of the individual operators for assessing tests, it requires *representativeness*, which means a mutation operator should be able to guide seeding faults that are representative of realistic ones, and *power*, which means an operator should be able to guide generating hard-to-kill non-equivalent mutants. Efficiency is concerned with the computational cost due to the operator set, it requires that the operator set generate a reasonable (computationally tractable) number of non-equivalent mutants.

There is some preliminary work on mutation testing for MAS. Nguyen et al. [4] use standard mutation operators for Java to assess tests for JADE agents (which are implemented in Java). As to the work on MAS model/language specific mutation operators, Adra and McMinn [5] propose a set of mutation operator classes for agent-based models. Saifan and Wahsheh [6] propose and classify a set of mutation operators for JADE mobile agents. Savarimuthu and Winikoff [7, 8] systematically derive a set of mutation operators for the AgentSpeak agent language and another set for the GOAL agent language. Most existing work focuses on deriving mutation operators from agent models/languages, a recent paper [8] evaluates the representativeness of the mutation operators for the agent language GOAL by comparison with realistic bugs.

In our work, we aim to explore the use of mutation testing for MAS, with the intention that our work can be used to assess and enhance the tests derived from existing testing techniques (e.g. that are based on some coverage criteria) for MAS. This paper presents our preliminary work – in Section 2 we propose a set of mutation operators for Jason [14], which is an implementation of the AgentSpeak language; in Section 3 we introduce a mutation testing system for individual Jason agents that implements a subset of our proposed mutation operators; in Section 4 we show the use of our implemented mutation operators in assessing and enhancing a test set (for a Jason agent) satisfying some existing agent-based coverage criteria, and the evaluation of the power of these operators by observing which one(s) lead to hard-to-kill non-equivalent mutants; in Section 5 we discuss the relationships between our work and previous related work; in Section 6 we summarise our work and make some suggestions for where this work could go in the future.

## 2 Mutation Operators for Jason

Mutation operators are rules to guide mutant generation by making changes to the description (syntax) of the program[1]. For instance, a mutation operator for procedural programs called *Relational Operator Replacement (ROR)* requires that *each occurrence of one of the relational operators* ($<, \leq, >, \geq, =, \neq$) *is replaced by each of the other operators* [11]. A mutant usually only contains a simple, unary fault (e.g., in the above example, each generated mutant only replaces a single relational operator by another), because of the two underlying theories [12] in mutation testing: the *Competent Programmer Hypothesis* states that programmers create programs that are close to being correct; the *Coupling Effect* states that tests that can detect a set of simple faults can also find complex faults.

Since mutation is typically performed at program level, a set of mutation operators is specific to a given programming language. To design mutation operators for a programming language, it is common to start by proposing an initial set based on the syntax and features of the language, and then to refine an effective set through evaluation.

The language we chose is Jason, which is a multi-agent system programming language that uses the extended AgentSpeak to specify agents in terms of beliefs, initial goals and plans, uses Java to customize agent architectures, define agent environments and implement various extensions. We chose to mutate the extended AgentSpeak code at first because it directs the behaviour of Jason agents.

Savarimuthu and Winikoff [7] apply the guidewords of HAZOP (Hazard and Operability Study) into the syntax of AgentSpeak to systematically derive a set of mutation operators. In contrast to their work, firstly we explicitly describe each of our derived operators while they do not give and describe their actual full operator set. Secondly we mutate the Jason-extended version of AgentSpeak, so some of our operators are specific to Jason. Finally, we borrow some ideas from other existing mutation opera-

[1] This paper only concerns conventional mutation testing, i.e. syntactic mutation testing, although some recent work applies mutation testing to program semantics.

tors (for both conventional programs and MAS) when deriving ours, in the hope of preliminarily refining our set, e.g., by excluding ones that are not sensible.

We base our work on Jason's Extended Backus–Naur Form (EBNF), where a list of production rules is defined that describe Jason's grammar. The EBNF we use is a simplified version in [14] that does not include some advanced features such as *directives* and conditional/loop statements in the plan body. We divide these production rules into high-level and low-level ones – the high-level production rules specify the main syntactical concepts that are closely related to how Jason agents generally work, while the low-level ones specify the basic logical representations forming the Jason syntactical concepts. Accordingly our mutation operators for Jason can also be described as high- or low-level. In the following two subsections we present these mutation operators according to which production rules they are derived from.

### 2.1 High-Level Mutation Operators for Jason

Fig. 1 shows the high-level production rules in Jason's EBNF; from this, we have derived 13 high-level mutation operators.

```
1:   agent ::= (belief)* (init_goal)* (plan)*
2:   belief ::= literal [":-" log_expr] "."
3:   init_goal ::= "!" literal "."
4:   plan ::= [label] triggering_event [":" context] ["<-" body] "."
5:   label ::= "@" atomic_formula
6:   triggering_event ::= ("+" | "-") ["!" | "?"] literal
7:   context ::= log_expr | true
8:   body ::= body_formula (";" body_formula)* | true
9:   body_formula ::= ("!" | "!!" | "?" | "+" | "-" | "-+") literal |
           action | internal_action | rel_expr
10:  action ::= atomic_formula
11:  internal_action ::= "." (atomic_formula | formula_for_comm)
-------------------------------------------------------------------
12:  formula_for_comm ::=
           "send(" receiver "," illocutionary_force "," message_content
                ["," reply] ["," timeout] ")" |
           "broadcast(" illocutionary_force "," message_content ")"
13:  receiver ::= agent_id | "[" agent_id ("," agent_id)* "]"
14:  illocutionary_force ::= tell | untell | achieve | unachieve | askOne |
           askAll | tellHow | untellHow | askHow
15:  message_content ::= propositional_content |
           "[" propositional_content ("," propositional_content)* "]"
16:  propositional_content ::= belief | triggering_event | plan | label
```

**Fig. 1.** High-level production rules in Jason's EBNF (Rule 1–11 are slightly adapted from [14], 12–16 are the ones we added for specifying Jason agent communication)

Production rule 1 states that an agent is specified in terms of beliefs, initial goals and plans. From this rule we derive the following three mutation operators:

- **Belief Deletion (BD):** *A single belief in the agent is deleted.*
- **Initial Goal Deletion (IGD):** *A single initial goal in the agent is deleted.*
- **Plan Deletion (PD):** *A single plan in the agent is deleted.*

Production rule 2 states that a belief can be a literal representing some fact, or a rule representing some fact will be derived if some conditions get satisfied. The introduction of rules enables Jason to perform *theoretical reasoning* [15]. From this production rule we derive the following mutation operator:

- **Rule Condition Deletion (RCD):** *The condition part of a rule is deleted.*

A rule that RCD is applied to will only have its conclusion part – a literal – left, as a belief held by the agent regardless of whether the (now deleted) conditions get satisfied.

Production rule 6 states that the triggering event of a plan consists of a literal following one of the six types: belief addition (+), belief deletion (−), achievement goal addition (+!), achievement goal deletion (−!), test goal addition (+?) and test goal deletion (−?). It can be seen that an event that can be handled by Jason plans represents a change – addition or deletion (represented using + or − operator respectively) – to the agent's beliefs or goals. From this rule we derive the following mutation operator:

- **Triggering Event Operator Replacement (TEOR):** *The triggering event operator (+ or −) of a plan is replaced by the other operator.*

We don't have an operator that changes the trigger type i.e. one of achievement goal, test goal and belief to another. This is because as learned from [8], this type of change doesn't make sense, and because in the case no events can match the modified trigger it will be equivalent to PD (Plan Deletion) anyway.

Production rule 7 states that the context of a plan can be a logical expression, or be always true (the latter is equivalent to the context not being specified at all). The plan context defines the condition under which the plan that has been triggered becomes a candidate for commitment to execution. From this production rule we derive the following mutation operator:

- **Plan Context Deletion (PCD)**: *The context of a plan is deleted if it is non-empty and not set true.*

Production rule 8 states that the body of a plan can be a sequence of formulae, each of which will be executed in order, or set *true* (the latter is equivalent to the body not being specified at all). From this rule we derive the following three mutation operators:

- **Plan Body Deletion (PBD)**: *The body of a plan is deleted if it is non-empty or not set true.*
- **Formula Deletion (FD)**: *A single formula in the body of a non-empty plan is deleted.*
- **Formulae Order Swap (FOS)**: *The order of any two adjacent formulae in the body of a plan that contains more than one formula is swapped.*

FOS comes from an idea behind some existing mutation operators that the order of elements in a sequence is changed. Although elements can be arranged in many ways, we choose to only swap two adjacent elements (i.e. formulae) because as suggested in [8], it can avoid generating a large number of mutants.

In many cases, PBD is equivalent to PD (Plan Deletion). However, since the plan context can contain internal actions that may cause changes in the agent's internal state, the plan that PBD is applied to may still have an effect on the agent although its body has been deleted, in this case PBD is not equivalent to PD.

Production rule 9–11 states that a body formula can be one of the six types: achievement goal (!literal or !!literal), test goal (?literal), mental note (+literal, −literal, −+literal), action (atomic_formula), internal action (.atomic_formula or .formula_for_comm[2]) and relational expression. The former three types are involved in generating *internal events* that correspond to changes in achievement goals, test goals and beliefs respectively. Similar to how we derived the Triggering Event Operator Replacement (TEOR) operator, from this production rule we derive the following mutation operator:

- **Formula Operator Replacement (FOR):** *The operator of an achievement goal formula (! or !!) is replaced by the other operator, so is that of a mental note formula (+, −, −+).*

It is worth noting that the achievement goal formula has two types: "!" is used to post a goal that must be achieved before the rest of the plan body can continue execution, "!!" allows the plan containing the goal to run alongside the plan for achieving the goal. In the latter case, the two plans can compete for execution due to the normal intention selection mechanism.

We don't consider changing the formula type i.e. one of achievement goal, test goal and belief to another because as noted in [8], this type of change doesn't make sense; neither do we consider the formula type of action, internal action or relational expression for the similar reason.

Production rules 12–16 are the ones we added for specifying Jason agent communication. It can be seen that two internal actions: *.send* and *.broadcast*, are used by Jason agents to send messages. The main parameters in these actions include the message receiver(s) (only used in *.send* action) that can be a single or a list of agents identified by the agent ID(s), the illocutionary force (*tell*, *untell*, *achieve*, etc.) representing the intention of sending the message and the message content that can be one or a list of propositional contents. From these production rules we derive the following three mutation operators:

- **Message Receiver Replacement (MRR):** *The receiver or the list of receivers in a .send action is replaced by another agent ID (or some subset of all the agent IDs in the MAS). If the action is .broadcast, it will be first converted to its equivalent .send action and then applied this mutation operator.*
- **Illocutionary Force Replacement (IFR):** *The illocutionary force in an action for sending messages is replaced by another illocutionary force.*
- **Propositional Content Deletion (PCD2):** *A single propositional content in the message content is deleted.*

[2] *formula_for_comm* actually belongs to *atomic_formula*. We separate it in order to specify rules for agent communication.

It is worth noting that a propositional content is some component of another type (e.g., belief, plan, etc.). Therefore, the mutation operators for these components can also be applied for mutating agent communication.

### 2.2 Low-Level Mutation Operators for Jason

Fig. 2 shows the low-level production rules in Jason's EBNF; from this, we have derived 11 low-level mutation operators, most of which are borrowed from existing operators for conventional programs.

```
1:  literal ::= ["~"] atomic_formula
2:  atomic_formula ::= (<ATOM> | <VAR>) ["(" term ("," term)* ")"] ["[" term
        ("," term)* "]"]
3:  term ::= literal | list | arithm_expr | <VAR> | <STRING>
4:  log_expr ::= simple_log_expr | "not" log_expr | log_expr "&" log_expr |
        log_expr "|" log_expr | "(" log_expr ")"
5:  simple_log_expr ::= (literal | rel_expr | <VAR>)
6:  rel_expr ::= rel_term [("<" | "<=" | ">" | ">=" | "==" | "\=" | "=" | "=..")
        rel_term]+
7:  rel_term ::= literal | arithm_expr
8:  arithm_expr ::= arithm_term [("+" | "-" | "*" | "**" | "/" | "div" | "mod")
        arithm_term]*
9:  arithm_term ::= <NUMBER> | <VAR> | "-" arithm_term |
        "(" arithm_expr ")"
```

**Fig. 2.** Low-level production rules in Jason's EBNF (Source: [14])

Production rule 1 states that a literal is an atomic formula or its strong negation (*~l*). Strong negation is introduced to overcome the limitation of default negation in logic programming: an agent can explicitly express that something is *false* by using strong negation, or express that it cannot conclude whether something is *true* or *false* using default negation (i.e. by the simple absence of a belief on the matter). From this production rule we derive the following mutation operator:

- **Strong Negation Insertion/Deletion (SNID):** *The form of a literal (affirmative or strong negative) is transformed to the other form.*

Production rule 2 and 3 state that an atomic formula consists of a relation followed by a list of annotations. Annotations can be used to provide further information about the relation. *source* is an important annotation that is appended to some atomic formulae automatically by Jason is used to represent where the atomic formulae (or the component it represents) come from by taking one of the three parameters: *percept*, *self* or an agent ID. For instance, belief *likes(rob, apples)[source(tom)]* implies the information that *rob* likes apples comes from agent *tom*. From these production rules we derive the following two mutation operators:

- **Annotation Deletion (AD):** *A single annotation of an atomic formula is deleted, if one exists.*
- **Source Replacement (SR):** *The source of an atomic formula is replaced by another source, if it exists.*

Production rule 4 and 5 define logical expressions; rule 6 and 7 define relational expressions; rule 8 and 9 define arithmetic expressions. Since some mutation operators for conventional programs have been designed for these concepts [11], we can just slightly adapt so as to use them in the context of Jason:

- **Logical Operator Replacement (LOR):** *A single logical operator (& or |) is replaced by the other operator.*
- **Negation Operator Insertion (NOI):** *The negation operator ("not") is inserted before a (sub) logical expression.*
- **Logical Expression Deletion (LED):** *A single sub logical expression is deleted.*
- **Relational Operator Replacement (ROR):** *A single relational operator ("<", "<=", ">", ">=", "==", "\==", "=", "=..") is replaced by another operator.*
- **Relational Term Deletion (RTD):** *A single relational term in a relational expression is deleted.*
- **Arithmetic Operator Replacement (AOR):** *A single arithmetic operator ("+", "–", "*", "**", "/", "div", "mod") is replaced by another operator.*
- **Arithmetic Term Deletion (ATD):** *A single arithmetic term in an arithmetic expression is deleted.*
- **Minus Insertion (MI):** *A minus (–) is inserted before an arithmetic term.*

## 3 muJason: A Mutation Testing System for Jason Agents

We have developed a mutation testing system for individual Jason agents called muJason[3], where we have implemented the 13 high-level mutation operators via Jason APIs and Java reflection, both of which can be used to access and modify the architectural components of the agents and the state of the MAS at runtime. The class diagram and the user interface of muJason are shown in Fig. 3 and Fig. 4 respectively.

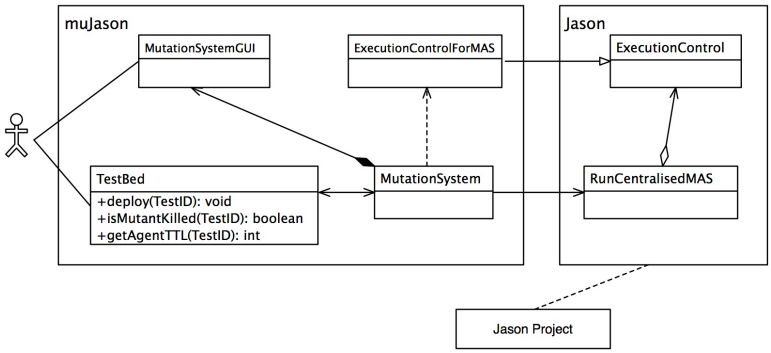


**Fig. 3.** The class diagram of muJason

[3] http://mujason.wordpress.com

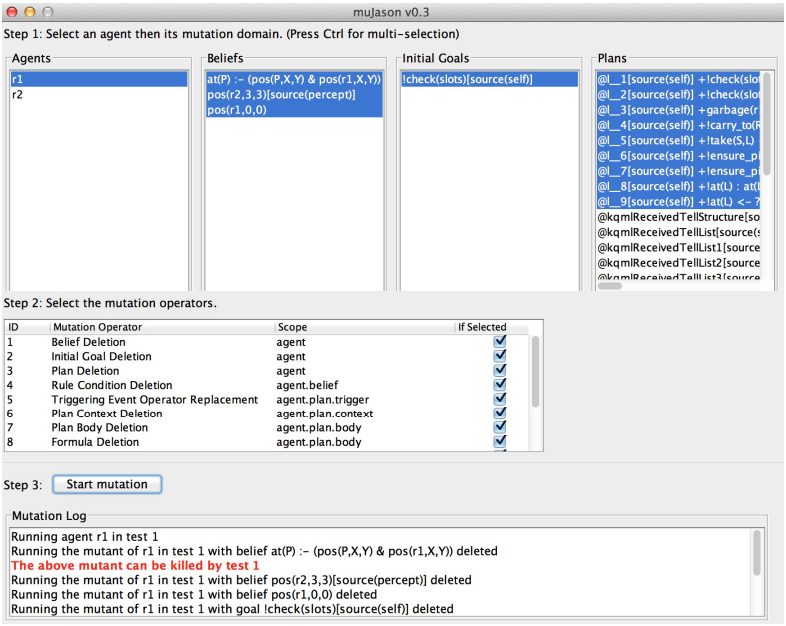


**Fig. 4.** The user interface of muJason

muJason can be launched by running the *MutationSystem* class and passing the name of the Jason project configuration file (postfixed with ".mas2j") as the parameter. Then muJason will load the Jason project and display the mutation testing control panel (as shown in Fig. 4), where users can configure, start and observe mutation testing processes.

Before initiating a mutation testing process, users need to specify the input of each test, the killing mutant criterion (or the oracle) for each test and the TTL (Time to Live) of the original/mutated agent under each test in the *deploy(testID)*, *isMutantKilled(testID)* and *getAgentTTL(testID)* methods provided by the *TestBed* class (as shown in Fig. 3), respectively. Each of these methods is described as follows:

- *deploy(testID)*: this method sets up the initial configuration of the Jason system prior to each test run. It is called each time by taking an ID identifying one of the tests, so users can write code to specify the starting configuration as the input of each test.
- *isMutantKilled(testID)*: this method is used to determine whether a mutant under some test is killed (as indicated by the Boolean return value). It is called as soon as each mutant terminates, and is passed the ID of the current test. Therefore, in this method users can write code to check whether the mutated agent has been killed by each individual test, in other words, in there users can implement the oracle for each test.
- *getAgentTTL(TestID)*: this method is used to specify the lifetime of the original/mutated agent (as the return value) under each test. Since agents usually run indefinitely, an original/mutated version of the agent can only be allowed to run for a

certain period of time so that the next one can run. The whole Jason project will restart as soon as one version terminates, so that the next version can be observed from (and mutated at) the same starting point of the MAS. The lifetime or TTL of an agent is measured by the number of cycles the agent can perform; it must be enough for the agent to expose all the behaviour involved in the process of killing mutants. The TTL for a test is actually part of the killing mutant criterion/oracle for that test. Although there may be ways to automatically terminate the mutant once it is observed being killed, for simplicity in the beginning, the TTL for a test is fixed and manually set depending on the users' experience.

After specifying the input, the killing mutant criterion (oracle) and the TTL for each test, users can configure and start a mutation testing process in the mutation testing control panel through the following steps (as shown in Fig. 4):

1. *Select an agent and its mutation domain*. Since muJason aims at individual agents, users need to select one from the MAS, and then they can choose which belief(s), initial goals(s) and plan(s) of the selected agent the mutation operators will be applied into. They can ignore the agents/components unnecessary for testing, e.g., the GUI agents and the built-in plans for enabling agent communication.
2. *Select the mutation operators*. After specifying the mutation domain of an agent, users can select the mutation operators that will be applied into the mutation domain.
3. *Start the mutation testing process*. After the above steps, users can start the mutation testing, observe its process in the mutation testing control panel and wait for its result. The mutation testing process can be described using the following pseudo-code:

```
1:  For each test identified by a testID:
2:      Set up the starting config as the input of the
3:        test
4:      Get the specified TTL for the test
5:      Run the original Jason project for the TTL
6:      Restart the Jason project
7:      Create a mutant generator taking the selected
          agent, mutation domain and mutation operators
8:      While the generator can generate another mutant:
9:           Generate the next mutant
10:          Run the modified Jason project for the TTL
11:          Check if the mutant is killed under the
               current test, if so mark it "killed"
12:          Restart the Jason project
```

## 4 Evaluation

To perform a preliminary evaluation of the power of our implemented mutation operators, we use them to guide generating mutants of an agent in a Jason project, then

examine whether a test set designed using a combination of existing agent-based coverage criteria can kill all the non-equivalent mutants. We think the operators that can guide generating the hard-to-kill non-equivalent mutants are powerful to reveal the weaknesses of this test set.

### 4.1 Experimental Setup

The Jason project we chose is available on the Jason website[4], and is called *Cleaning Robots*. It involves a cleaner agent, an agent besides an incinerator (we call it incinerator agent later for convenience) and several pieces of garbage located in a gridded area as shown in Fig. 5 (*R1* represents the cleaner agent, *R2* represents the incinerator agent, *G* represents the garbage). When this project is launched, the cleaner agent will move along a fixed path that covers all grid squares (move from the leftmost square to the rightmost one in the first row, then "jump" to the leftmost square in the second row and move to the rightmost one in the same row, and so on). If it perceives that the square it is in contains garbage, it will pick it up, carry it and then move to the square where the incinerator agent is along a shortest path (diagonal movement is allowed). The cleaner agent will drop the garbage after arriving so that the incinerator agent can take it to burn. After dropping garbage the cleaner agent will return to the square where it just found the garbage along a shortest path (diagonal movement allowed), and then continue moving along the fixed path until it reaches the last square.

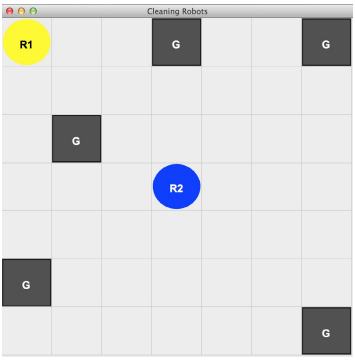


**Fig. 5.** The *Cleaning Robots* example

In order to test the cleaner agent, we specify test inputs that each describe a different environment in which the agent is located. We design test inputs according to the test coverage criteria proposed by Low et al. [1]. Their criteria are based on plans and nodes (formulae) in BDI agents, so they are suitable for the Jason agent paradigm. Fig. 6 shows the subsumption hierarchy of their criteria – the criterion at the starting point of an arrow subsumes the one at the end of the arrow, e.g., for any agent, a test set satisfying node path coverage criterion also satisfies node coverage criterion. This Jason project is simple and doesn't concern plan and node failure, so we ignore the

[4] `http://jason.sourceforge.net/wp/examples/`

related criteria, i.e. node with success and failure coverage criterion and plan with success and failure coverage criterion (that is to say, for this project they are equivalent to node coverage criterion and plan coverage criterion respectively). After manual analysis of the AgentSpeak program of the cleaner agent we design ten test inputs (different environments) that collectively meet node path coverage criterion, plan context coverage criterion and plan path coverage criterion, and for the involved cyclic paths we apply the *0-1-many* rule. We think this combination forms the most rigorous one among Low et al.'s criteria (as can be seen in Fig. 6) and is viable for testing the cleaner agent.

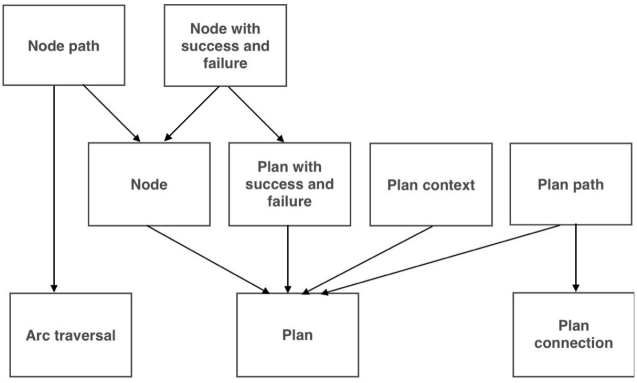


**Fig. 6.** The subsumption hierarchy of the coverage criteria proposed by Low et al. (Redrawn from [1])

These test inputs (environments) differ in at least one of the three variables – the location of the incinerator agent, the amount (and locations) of garbage and the probability the cleaner agent has to pick up each piece of garbage successfully when it attempts to. Since the agent environment is hard-coded into a java file, we use text replacement and class reload techniques in the *deploy(testID)* method to modify the values of these three variables in order to specify each test input. We consider a mutant to be killed if, at the end of any test, there is any garbage uncollected (in contrast, the non-mutated version always collects all the garbage). To implement this criterion/oracle, we use Jason APIs and Java reflection in the *ifMutantKilled(testID)* method to check whether all the squares in the environment are empty except the two taken by the cleaner agent and the incinerator agent respectively. In the *getAgentTTL(testID)* method, for each individual test, we set the lifetime of the original/mutated agent to a value that is enough to collect all garbage. This value equals the exact time taken by the original agent to finish its work (we observe this by giving the original agent a normal run under that test) plus a modest tolerance value.

Next we configure a mutation testing process for the cleaner agent as shown in Fig. 4: first we choose *r1* which is the name of the cleaner agent, and then all of its three beliefs, one initial goal and nine plans excluding those built-in ones for enabling agent communication. Next we check all the implemented operators. After these we start and observe the mutation testing itself.

### 4.2 Results

After the mutation testing, muJason displays the results as shown in the first three columns of the table in Fig. 7: the first column lists the mutation operators we selected, the second column lists the total number of mutants generated by each selected operator and the third column lists the number of the killed mutants that corresponds to each selected operator. From the displayed results we can see that the three operators for agent communication – Message Receiver Replacement (MRR), Illocutionary Force Replacement (IFR) and Propositional Content Deletion (PCD2) – are not useful because this Jason project doesn't involve agent communication. We also observe that our implemented operators (excepts the ones for agent communication) have resulted in a manageable number of mutants, i.e. 70 mutants, among which 60 have been killed while 10 not killed. We track these non-killed mutants in the log of the mutation testing process and analyse their corresponding changes in the code. We present our analysis results in the last two columns, and discuss each non-killed mutant below.

| Mutation Operator | No. of Generated Mutants | No. of Killed Mutants | No. of Non-Killed Equivalent Mutants | No. of Non-Killed Non-Equivalent Mutants |
|---|---|---|---|---|
| BD | 3 | 1 | 2 | 0 |
| IGD | 1 | 1 | 0 | 0 |
| PD | 9 | 8 | 1 | 0 |
| RCD | 1 | 1 | 0 | 0 |
| TEOR | 9 | 8 | 1 | 0 |
| PCD | 4 | 4 | 0 | 0 |
| PBD | 6 | 6 | 0 | 0 |
| FD | 16 | 15 | 0 | 1 |
| FOS | 10 | 9 | 1 | 0 |
| FOR | 11 | 7 | 3 | 1 |
| MRR | 0 | 0 | 0 | 0 |
| IFR | 0 | 0 | 0 | 0 |
| PCD2 | 0 | 0 | 0 | 0 |
| Total | 70 | 60 | 8 | 2 |

**Fig. 7.** The results of the mutation testing

**Equivalent Mutants**

The Belief Deletion (BD) operator generates three mutants, in each of which an initial belief in the belief base of the agent is deleted. Two mutants are equivalent, however, they should not have been generated. Recall that muJason provides access to and makes changes to the initial state of the MAS rather than the agent code. This implementation is equivalent to mutating the code directly because the code will be interpreted to the initial state that subsequently affects the MAS behaviour. However, two

of the three beliefs we choose – *pos(r2, 3, 3)* and *pos(r1, 0, 0)* representing the initial positions of the incinerator agent and the cleaner agent respectively, are not defined in the agent (AgentSpeak) code – they are from the environment (Java) code, which is not our mutation target. Like beliefs from the agent code, they have been automatically added into the belief base by the Jason engine before the initial state of the MAS becomes accessible, so they appear as mutation options, which have been selected by us. Also, deleting them before the MAS runs will not change the agent behaviour because they will be automatically added again soon due to the mechanism of how Jason handle beliefs from environments.

The Plan Deletion (PD) operator generates one equivalent mutant, in which a plan that has empty context and empty body is deleted. This plan only exists in the first place to prevent a certain source of spurious runtime errors; when it is deleted, the agent will throw error messages at runtime, but there is no other effect on the agent behaviour. It could be suggested that this is in fact a non-equivalent mutant, but the runtime of the system is variable and the difference here is tiny. This mutant is not killed because our killing mutant criteria or test oracles don't check for this source of errors.

The Triggering Event Operator Replacement (TEOR) operator generates one equivalent mutant, in which the triggering event of the empty plan (discussed above for the PD operator) is changed from addition to deletion of some goal. This will just prevent the error messages discussed above from being thrown, so there is no change at all to the agent behaviour.

The Formula Operator Swap (FOS) operator generates one equivalent mutant, in which two formulae whose executions are completely independent (their order doesn't matter) are swapped. Specifically, the original order is first to remember the location where the agent just picked up the garbage, and then to move to the incinerator agent; reversing the order makes no difference to the agent behavior because this location will be used only after both formulae completes (more precisely, after the agent drops the garbage), although the original order seems more rational.

The Formula Operator Replacement (FOR) operator generates three equivalent mutants, in each of which a goal formula type "!" is replaced by "!!" or vice versa. As discussed in Section 2, a "!" goal pursuit stops the current plan until completed, while a "!!" goal pursuit can carry on in parallel with the rest of the plan. It is not difficult to see that in some cases they can be replaced by each other with no changes in the agent behaviour (only with semantic difference).

**Non-equivalent Mutants**

The Formula Deletion (FD) operator generates one non-equivalent mutant, in which the formula that is used to drop the carried garbage is deleted. It is not killed because our killing mutant criteria or test oracles are incomplete: they don't check whether the cleaner agent drops the carried garbage – it can pick up all the garbage without dropping any and still pass the tests.

The Formula Operator Replacement (FOR) operator produces one non-equivalent mutant, in which the formula *−+pos(last,X,Y)* in plan *+!carry_to(R)* is replaced by

*+pos(last,X,Y)*. The former formula is used to update the belief that keeps last location where garbage was found, so that the agent can retrieve then return to this location after it drops garbage at the incinerator agent, so as to continue checking the remaining squares along the fixed path. However, when the formula is changed to the new version, each time the cleaner agent finds garbage, it will add a new belief representing the location of this garbage into the belief base rather than replacing the old one.

The above mutation introduces a fault, because it means that the agent will end up with several versions of "last location at which I found garbage" stored in its memory. In many cases, this is not a problem. When the cleaner agent has finished at the incinerator agent, it will try to take a shortest route back to last location where it found garbage. To do this, it queries for its belief about the last location, and it will always retrieve the correct one because Jason's default belief selection mechanism will always select the matching one that is added to the belief base most recently.

After each movement step, however, the agent will query "does my current location correspond to the last location I found garbage" i.e. should it cease its fast movement and go back to its slow side-to-side sweep of the map? If the agent is at any location where it previously found garbage, Jason's belief query mechanism will cause the answer to that question to be "yes" – all of the "last garbage location" beliefs will be checked for a match. At that point, it will go back into its slow sweep, even though (in this simple world) there's no chance of finding new garbage before it reaches the actual last garbage location. As a consequence, the whole collection process will take longer and the agent may not collect all the garbage within its specified time-to-live.

This fault cannot be detected by any of our tests designed for the cleaner agent, because in our tests (by chance) it never passes through a previous garbage location when returning to last collected garbage location (Fig. 5 shows an example where it would happen). In order to detect this fault, we add a test input that satisfies the following three conditions:

- A piece of garbage, *G1*, is located in a shortest path between the incinerator agent and another piece of garbage *G2*.
- *G1* is found prior to *G2*. This requires that *G1* and *G2* be located after where the incinerator agent is along the fixed side-to-side path that the agent uses to check all the squares.
- *G1 and G2* are not in the same row. This enables us to observe that the agent does indeed return to where *G1* was found after dropping either garbage for burning.

Fig. 8 shows a test that will detect this fault and thus kill this mutant. Under this test, the cleaner agent (*R1*) will always return to the location where *G1* was found after dropping either *G1* or *G2* at the incinerator agent (*R2*). It will then continue moving along the fixed side-to-side path from this location, and the second time it does this, this wastes time (there is guaranteed to be no further garbage on the way to the G2 location, since it's already swept that area). The additional time it spends doing this takes it over its time-to-live so the agent fails the test.

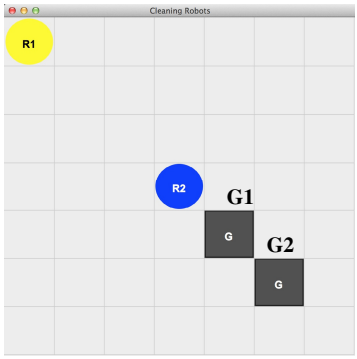


**Fig. 8.** A test that can detect the fault of multiple last locations

### 4.3 Discussion

In order to evaluate the power of our implemented high-level mutation operators, we used them to assess a test set for a simple Jason agent. We designed the test inputs according to what we thought as the most rigorous and viable criterion (a combination of three criteria) among those proposed by Low et al. Then we analysed the generated mutants that were not killed by the test set. To find out the hard-to-kill non-equivalent ones from these mutants, we took the following steps:

Firstly, we excluded two sources of mutants – those that should not have been generated and those non-equivalent but non-killed because of the incomplete test oracles (see 4.2 for the details). Those two sources are due to weaknesses in our implementation so they can be avoided.

Secondly, we excluded the equivalent mutants. Although they are of little interest to our current work, the details of them may be of some interest for future studies on how to reduce the relevant equivalent mutants. For instance, in some cases changing the formula operator "!!" to "!" only affects the agent behaviour in efficiency, so this rule can be ignored if this source of difference is not considered when killing mutants. Similarly, the Triggering Event Operator Replacement (TEOR) operator should not be applied to the empty plans that are only used to prevent spurious runtime messages.

Finally, we found a non-equivalent mutant that was not killed (regardless of how the test oracles are specified). From this, we deduce that Formula Operator Replacement (FOR) that generates this mutant is probably a powerful operator, more precisely, the rule of changing the formula operator "–+" to "+" is powerful.

## 5 Comparison with Related Work

Savarimuthu and Winikoff [7] systematically derive a set of mutation operators from the syntax of the AgentSpeak language (except that they derive those for agent communication from the Jason-style code). In contrast to their work, firstly we explicitly describe each of our derived operators while they do not give and describe the actual full operator set. Secondly we mutate the Jason-extended version of AgentSpeak, so

some of our operators are specific to Jason while others are also applicable to AgentSpeak.

Jason-specific examples are the Rule Condition Deletion (RCD) operator that involves rules (introduced by Jason to enable theoretical reasoning), the Formula Operator Replacement (FOR) operator that involves some Jason specific operators (!! and −+), the Strong Negation Insertion/Deletion (SNID) operator that involves strong negation (introduced by Jason to increase the expressive power), and the Annotation Deletion (AD) and Source Replacement (SR) operator that involves annotations (specific to Jason).

Finally, we borrow some ideas from other existing mutation operators (for both conventional programs and MAS) when deriving ours, in the hope of preliminarily refining our set. For instance, changing a belief type to a goal type or vice versa doesn't make much sense (as learned from [8]), so it is not considered; there have been some well-defined mutation operators for some traditional concepts that are also used in Jason grammar (e.g., arithmetic expression), so we can directly borrow them after small adjustments.

Savarimuthu and Winikoff [8] systematically derive another set of mutation operators for the GOAL agent language (like AgentSpeak, GOAL is another language for programming cognitive agents), and then evaluate the representativeness of their set by comparison with some realistic bugs. In contrast, we evaluate the power of our set by comparison with some existing coverage criteria. Since an effective operator set requires ones that are both representative of realistic faults and powerful to guide generation of hard-to-kill mutants, our evaluation approach is complementary to theirs.

We have attempted to compare our evaluation results with theirs and found that our Formula Operator Replacement (FOR) operator (which is powerful in our experiment) is similar to one of their mutation rules – A:op1 (changing an operator), which doesn't involve any realistic bug observed in their experiment. However, considering our hard-to-kill mutant is generated by changing the mental note formula "−+bel", which is actually a composition of two adjacent formulae "−bel; +bel", to "+bel", while A:op1 doesn't involve any composite operator, our mutant is actually the result of their another rule – AC:drop (dropping an action), which involves the 4$^{th}$ most realistic bugs (i.e. 14 bugs, while the most representative rule involves 31 bugs) observed in their experiment. This comparison is not very convincing since the two studies are based on different agent languages, and both are quite preliminary, but it shows a way to evaluate the effectiveness of mutation operators, i.e. examining both the representativeness and the power of each operator.

Another related work is Adra and McMinn's [5]. Although they use a rather different agent model, some of their ideas are relevant to our work. They propose four mutation operator classes, among which their class for agent communication (Miscommunication, Message Corruption) corresponds to our operators for agent communication (Message Receiver Replacement, Illocutionary Force Replacement, Proposition Content Deletion and other involved high- and low-level operators), and their class for an agent's memory corresponds to our operators for beliefs (Belief Deletion, Rule Deletion and other involved low-level operators). Their mutation

operator class for agent's function execution does not directly correspond to our operators since our agent model adopts the BDI reasoning mechanism, while their model does not. As to their mutation operator class for the environment, it is not relevant to our operators for agents, although environment is an important dimension of MAS.

## 6 Conclusions

In this paper we presented our preliminary work on mutation testing for Jason agents. We proposed a set of mutation operators for the Jason-extended AgentSpeak language and we described a mutation testing system called muJason, which implements the high-level subset of our operators. We then used our implemented operators to assess a test set (for an example agent) that satisfies some coverage criteria proposed by Low et al. [1]. We found a mutation operator – Formula Operator Replacement (FOR) – that guided generation of a non-equivalent mutant that is hard to kill. We are hence able to add a test into the test set for killing this mutant (and, probably, similar mutants or faults).

Our work extends Savarimuthu and Winikoff's work [7, 8] mainly in two respects: first, we extend the mutation operators for AgentSpeak by some operators specific to Jason; second, we propose an approach for assessing the power of operators, which is complementary to their approach for assessing the representativeness of operators.

Our work is preliminary and has a number of weaknesses. In terms of deriving mutation operators, we may miss some that may be of interest to our experiment since we did not consider the complete syntax of Jason and systematic ways to derive them (instead we intended to start with an initial set then implement and evaluate them in an incremental way). As to our evaluation, our results are limited as we only considered a single simple agent and a single source of coverage criteria, and because our finding is specific to Jason (the hard-to-kill mutant we found is the result of changing the Jason-specific formula operator "–+").

Our work also has some scalability issues. Firstly, we did not adopt an appropriate testing technique to specify test inputs and oracles; instead we used inefficient approaches such as manual specification via Java reflection, which are very difficult to use to specify a number of tests that are required by complex systems and tests that are able to detect small differences to the system behavior (mutants often only lead to such small differences). Secondly, in deriving test inputs that satisfies a specific coverage criterion, we did not use any technique for auto-measuring test coverage to guide our derivation work; instead we derived them by manual analysis of the program, which is impractical for complex programs.

Future work will first address the above issues. Before further evaluation of the power of our mutation operators, we will develop a unit testing technique for Jason agents (unit testing provides a flexible way to specify tests that are able to detect small differences in behavior) and techniques for auto-measuring test coverage. We will then be able to apply our approach to more complex Jason systems and to a range of coverage criteria. In the mean time, we will derive more mutation operators for Jason agents, then implement and evaluate them, along with the low-level ones that we proposed in this paper but did not implement in muJason so far.

Other potentially valuable studies include evaluating the representativeness of mutation operators for Jason (or other agent languages), improving the efficiency of mutation testing for multi-agent systems (e.g. by auto-reduction of equivalent mutants) and mutation of other aspects of multi-agent systems (e.g., environments, organizations and semantics).

## References


1. Low, C.K., Chen, T.Y., Rönnquist, R.: Automated test case generation for BDI agents. Autonomous Agents and Multi-Agent Systems 2, 311–332 (1999)
2. 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)
3. 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)
4. Nguyen, C.D., Perini, A., Tonella, P.: Automated continuous testing of multi-agent systems. In: The Fifth European Workshop on Multi-Agent Systems (2007)
5. Adra, S.F., McMinn, P.: Mutation operators for agent-based models. In: Proceedings of 5th International Workshop on Mutation Analysis. IEEE Computer Society (2010)
6. 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)
7. Savarimuthu, S., Winikoff, M.: Mutation operators for cognitive agent programs. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 1137–1138 (2013)
8. Savarimuthu, S., Winikoff, M.: Mutation Operators for the GOAL Agent Language. In: Winikoff, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 255–273. Springer, Heidelberg (2013)
9. Houhamdi, Z.: Multi-agent system testing: A survey. International Journal of Advanced Computer Science and Applications (IJACSA) 2(6), 135–141 (2011)
10. Nguyen, C.D., Perini, A., Bernon, C., Pavón, J., Thangarajah, J.: Testing in multi-agent systems. In: Gleizes, M.-P., Gomez-Sanz, J.J. (eds.) AOSE 2009. LNCS, vol. 6038, pp. 180–190. Springer, Heidelberg (2011)
11. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
12. Mathur, A.P.: Foundations of Software Testing. Pearson (2008)
13. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: International Conference on Software Engineering (2005)
14. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
15. Wooldridge, M.: An Introduction to MultiAgent Systems, 2nd edn. John Wiley & Sons (2009)