

This is a repository copy of *Mutation Testing for Jason Agents*.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/78890/>

Version: Accepted Version

---

**Proceedings Paper:**

Huang, Zhan, Alexander, Rob [orcid.org/0000-0003-3818-0310](https://orcid.org/0000-0003-3818-0310) and Clark, John Andrew [orcid.org/0000-0002-9230-9739](https://orcid.org/0000-0002-9230-9739) (2014) Mutation Testing for Jason Agents. In: Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems. 13th International Conference on Autonomous Agents and Multiagent Systems, 05-09 May 2014 IFAAMAS , FRA

---

**Reuse**

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing [eprints@whiterose.ac.uk](mailto:eprints@whiterose.ac.uk) including the URL of the record and the reason for the withdrawal request.

# 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 are 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 faulty versions of the program following mutation operators then checks if the tests can distinguish the original program from those versions. In this paper, we propose a set of mutation operators for the Jason agent-oriented programming language, and then introduce a mutation testing system for individual Jason agents that implements some of our proposed mutation operators. We use our implemented mutation operators to assess the tests for a small Jason system, and show that the tests that meet a combination of existing coverage criteria do not kill all mutants.

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

## 1 Introduction

Multi-agent systems (MAS) are getting increasing attention in academics and industry as an emerging paradigm for engineering autonomous and distributed systems. In MAS engineering, testing is a challenging activity 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 the tests, however, test coverage criteria or simulation of real faults can be used to evaluate it.

For coverage based test evaluation, the executions of the tests are measured against some coverage criteria based on some model of the SUT; if these executions traverse 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].

Simulation of real faults offers a more direct way to assess the fault detection ability of the tests than test coverage criteria: faults can be hand-seeded or seeded by *mutation* [12], which is a systematic and automatic way of generating modified versions of the SUT ("mutants") following some rules ("mutation operators"). After seeding faults, each test is executed against first the original SUT then each faulty version of the SUT. 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 faults 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 generated faulty versions: higher the ratio is, more effective the tests are.

Mutation is more commonly used to seed faults than the hand-seeded way because it has solid theoretical foundation, and empirical studies suggest that it provides an efficient way to seed faults that are more representative of real faults than hand-seeded ones [13]. However, the mutation operators used to guide mutant generation may lead to a huge 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 which are not actually faulty, and thus which must be excluded from test evaluation. Generally, excluding equivalent mutants is a laborious manual process.

This process of using mutation to assess tests is called mutation testing. The key to success is to design an effective set of mutation operators that can simulate an adequate set of realistic faults in a reasonable (computationally tractable) number of generated 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). In contrast to standard operators for existing languages, it is likely that MAS-model-specific mutation operators will better simulate MAS-specific faults. In this vein, 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. Similarly, Savarimuthu and Winikoff [7, 8] systematically derive a set of mutation operators for the AgentSpeak BDI agent language and another set for GOAL agent language. None of the above papers on MAS-specific mutation operators, however, actually implement and evaluate their operators except [8].

We aim to explore the use of mutation testing for MAS because mutation testing is widely thought to be a more rigorous test evaluation technique than coverage-based approaches [11], with the intention that our work can be used to assess and enhance the tests derived from the existing test generation/evaluation techniques (that are based on some coverage/mutation criteria) for MAS. This paper presents our preliminary work. In Section 2 we propose a set of mutation operators for Jason [14], which is a practical implementation of the AgentSpeak language; in Section 3 we introduce our mutation testing system for individual Jason agents that implements some of our

proposed mutation operators; in Section 4 we show the use of our implemented mutation operators in assessing and enhancing the tests for a Jason project; we end with a summary of our work, a discussion of the relationships to previous related work and some suggestions for where this work could go in the future.

## 2 Mutation Operators for Jason

Mutation operators are rules to guide mutant generation. 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 source code level, a set of mutation operators is specific to a given programming language (C, Java, etc.). To design mutation operators for a programming language, it is common to start by proposing a large set on the syntax and features of the language, and then to refine an effective set through evaluation.

Savarimuthu and Winikoff [7] applied the guidewords of HAZOP (Hazard and Operability Study) into the syntax of AgentSpeak, to systematically derive a set of mutation operators for AgentSpeak. Now we build on their work: we propose mutation operators for an implementation of AgentSpeak called Jason [14], which implements AgentSpeak’s operational semantics and extends AgentSpeak with various features useful for practical agent implementation. In contrast to their systematic method that may produce a large amount of mutation operators, we have used our judgment and borrowed the ideas of existing mutation operators to refine our operator set so as to preferentially implement and evaluate it, in the hope of avoiding implementing some ineffective operators. It can be seen that our mutation operators contain some shared with Savarimuthu and Winikoff for the core AgentSpeak language and others for the Jason specific features.

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 that excludes some advanced features of Jason such as the use of *directives* and allowing conditional/loop statements in the plan body. These could, of course, be considered in further work. 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 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:	<code>agent -&gt; (belief)* (init_goal)* (plan)*</code>
2:	<code>belief -&gt; literal [“:-” log_expr] “.”</code>
3:	<code>init_goal -&gt; “!” literal “.”</code>
4:	<code>plan -&gt; [label] triggering_event [“:” context] [“&lt;” body] “.”</code>
5:	<code>label -&gt; “@” atomic_formula</code>
6:	<code>triggering_event -&gt; (“+”   “-”) [“!”   “?”] literal</code>
7:	<code>context -&gt; log_expr   true</code>
8:	<code>body -&gt; body_formula (“;” body_formula)*   true</code>
9:	<code>body_formula -&gt; (“!”   “!!”   “?”   “+”   “-”   “-+”) literal   [“.”] atomic_formula   rel_expr</code>
10*:	<code>action_for_comm -&gt; “.” ( “send(“ receiver “,” illocutionary_force “,” message_content [“,” reply] [“,” timeout] “)”   “broadcast(“illocutionary_force “,” message_content “)” )</code>
11*:	<code>receiver -&gt; agent_id   “[” agent_id (“,” agent_id)* “]”</code>
12*:	<code>illocutionary_force -&gt; tell   untell   achieve   unachieve   askOne   askAll   tellHow   untellHow   askHow</code>
13*:	<code>message_content -&gt; propositional_content   “[” propositional_content (“,” propositional_content)* “]”</code>
14*:	<code>propositional_content -&gt; belief   triggering_event   plan   label</code>

**Fig. 1.** High-level production rules in Jason’s EBNF (Rule 1–9 are adapted from [14], 10–14 are we add 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):** *Each belief in the agent is deleted.*
- **Initial Goal Deletion (IGD):** *Each initial goal in the agent is deleted.*
- **Plan Deletion (PD):** *Each 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* [16]. From this production rule we derive the following mutation operator:

- **Rule Condition Deletion (RCD):** *The condition part of each 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 each plan is replaced by another operator.*

Production rule 7 states that the context of a plan can be a logical expression or set *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 each plan is deleted if it is non-empty or 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 each plan is deleted if it is non-empty or not set true.*
- **Formula Deletion (FD):** *Each formula in the body of each non-empty plan is deleted.*
- **Formulae Order Swap (FOS):** *The order of any two adjacent formulae in the body of each plan that contains more than one formula is swapped.*

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 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) 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 Triggering Event Operator Replacement (TEOR) operator, from this production rule we derive the following mutation operator:

- **Formula Operator Replacement (FOR):** *The operator of each achievement goal formula (! or !!) is replaced by another operator, so is that of each 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.

Production rules 10–14 (marked with asterisks) 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 each .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 each action for sending messages is replaced by another illocutionary force.*
- **Propositional Content Deletion (PCD2):** *Each propositional content in the message content is deleted.*

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 the existing ones for conventional programs.

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

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 ( $\sim I$ ). 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 each 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, e.g., *source* is an important annotation that is appended to some atomic formulae automatically by Jason is used to represent where the atomic formulae (or its represented component) 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):** *Each annotation of each atomic formula is deleted, if one exists.*
- **Source Replacement (SR):** *The source of each atomic formula is replaced by another source, if it exists.*

Production rule 4 and 5 define logical expressions. From these rules we derive the following three mutation operators:

- **Logical Operator Replacement (LOR):** *Each logical operator (& or |) is replaced by the other operator.*
- **Negation Operator Insertion (NOI):** *The negation operator (“not”) is inserted before each (sub) logical expression.*
- **Logical Expression Deletion (LED):** *Each sub logical expression is deleted.*

Production rule 6 and 7 define relational expressions. From these rules we derive the following two mutation operators:

- **Relational Operator Replacement (ROR):** *Each relational operator (“<”, “<=”, “>”, “>=”, “==”, “\==”, “=”, “=..”) is replaced by another operator.*
- **Relational Term Deletion (RTD):** *Each relational term in each relational expression is deleted.*

Production rule 8 and 9 define arithmetic expressions. From these rules we derive the following three mutation operators:

- **Arithmetic Operator Replacement (AOR):** *Each arithmetic operator (“+”, “-”, “\*”, “\*\*”, “/”, “div”, “mod”) is replaced by another operator.*
- **Arithmetic Term Deletion (ATD):** *Each arithmetic term in each arithmetic expression is deleted.*
- **Minus Insertion (MI):** *A minus (-) is inserted before each of the arithmetic term.*



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

We have developed a mutation testing system for individual Jason agents called muJason<sup>1</sup>, 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. Next we will introduce muJason from the perspective of the users.

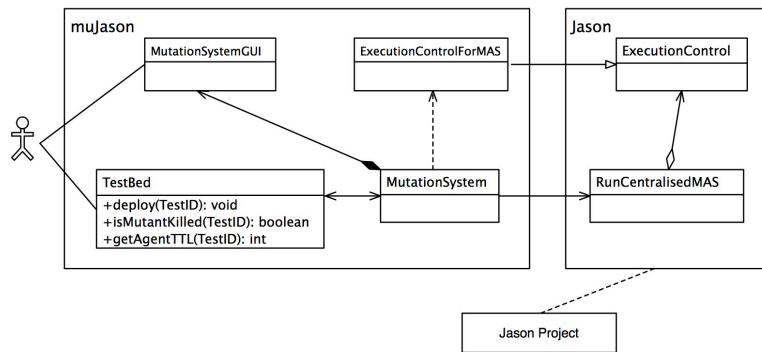


Fig. 3. The class diagram of muJason

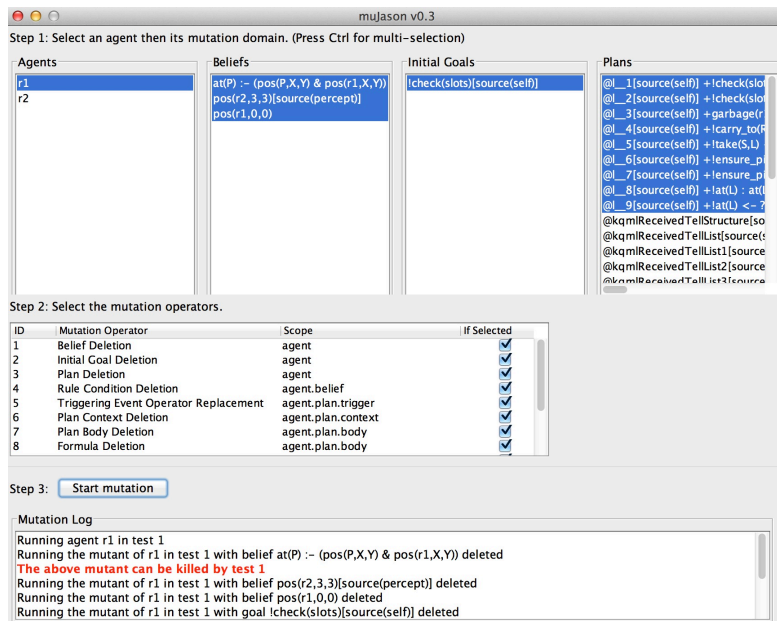


Fig. 4. The user interface of muJason

<sup>1</sup> <http://mujason.wordpress.com>

A user can launch muJason 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 the user can configure, start and observe a mutation testing process.

Before initiating a mutation testing process, the user needs to specify the tests that need evaluation, the killing mutant criterion for each test and the TTL (Time to Live) of the original agent and each mutant for 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. The method is called each time by taking an ID identifying one of the tests, and the user can write code to set up the tests corresponding to the passed test IDs.
- *isMutantKilled(testID)*: this method is used to determine whether a mutant under some test is killed. It is called after each mutant terminates, and is passed the ID of the current test. Therefore, in this method the user can write code to check whether the mutant has been killed by each individual test. An alternate approach would have been to compare all the behaviour of the original agent and that of the mutant, but that would have been computationally expensive and prone to declaring mutants “killed” when the behaviour variation was of no consequence. With the approach taken here, the user can just specify the important aspects that need observation and comparison, via Jason APIs or Java reflection that can access the state of the MAS, or other techniques.
- *getAgentTTL(TestID)*: this method is used to specify the lifetime of the original agent and its mutants as the return value for each test. Since agents usually run indefinitely, the original agent or each mutant 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 the original or mutated agent terminates, so that next time the agent can be observed from (and mutated at) the (same) initial point of the MAS. The lifetime or TTL of an agent is measured by the number of the reasoning 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 for that test. Although there may be ways to automatically evaluate the TTL or to automatically terminate the mutant once it is observed being killed, for simplicity in the beginning, the TTL for each test is fixed and manually set depending on the user’s experience.

After specifying the tests, the killing mutant criteria and the TTL, the user 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, the user needs to select one from the MAS, and then he can choose which belief(s), initial goals(s) and plan(s) of the selected agent the mutation operators will be ap-

plied into. He can ignore the agents/components unnecessary for testing, e.g., the GUI agent and the plans pre-defined by Jason for enabling agent communication, etc.

2. *Select the mutation operators.* After specifying the mutation domain of an agent, the user can select the mutation operators that will be applied into the mutation domain.
3. *Start the mutation testing process.* After configuration, the user 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 defined testID:
2:     Set up the test identified by the testID
3:     Get the specified TTL for the test
4:     Run the original Jason project for the TTL
5:     Restart the Jason project
6:     Create a mutant generator taking the selected
       agent, mutation domain and mutation operators
7:     While the generator can generate another mutant:
8:         Generate the next mutant
9:         Run the modified Jason project for the TTL
10:        Check if the mutant is killed under the
        current test, if so mark it "killed"
11:        Restart the Jason project
```

## 4 Evaluation

To perform a preliminary evaluation of our implemented mutation operators, we use them to guide the generation of the mutants for an agent in a Jason project, then examine whether a test set designed using some existing agent test coverage criteria can kill all the non-equivalent mutants. If it cannot kill all those mutants, that means this test set cannot reveal the faults simulated by the non-killed non-equivalent mutants, thereby demonstrating that our operators are effective in finding the weaknesses of this test set.

The Jason project we choose is available on the Jason website<sup>2</sup>, and is called *Cleaning Robots*. It involves a cleaner agent, an incinerator agent 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, 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 the shortest path

---

<sup>2</sup> <http://jason.sourceforge.net/wp/examples/>

(diagonal movement is allowed). The cleaner agent will drop the garbage in the incinerator agent for burning, and after that it will return to the square where it just found the garbage along a shortest path (diagonal movement allowed), then continue moving along the fixed path until it reaches the last square.

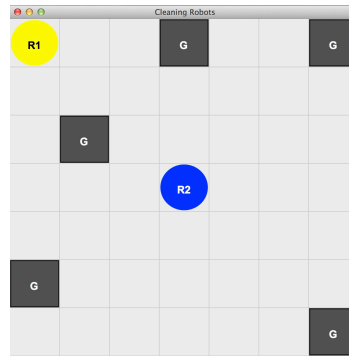


Fig. 5. The *Cleaning Robots* example

In order to test the cleaner agent, we generate tests that each describe a different environment for the cleaner agent. We design the tests according to the test coverage criteria proposed by Low et al. [1]. Their criteria are based on plans and nodes (actions) in BDI agents, which are suitable for Jason agent paradigm. Fig. 6 shows the subsumption hierarchy of their criteria, from which it can be seen the topmost criteria represent the most rigorous ones. Since this Jason project is simple and doesn't concern plan and action failure, after analyzing the AgentSpeak code of the cleaner agent we design ten tests that collectively meet the node path coverage criterion, the plan context coverage criterion and the plan path coverage criterion (we use 0-1-many rule for cyclical path coverage), and accordingly we extract three variables from each test: the location of the incinerator agent, the locations of garbage and the probability the cleaner agent has to pick up each piece of garbage successfully when it attempts to.

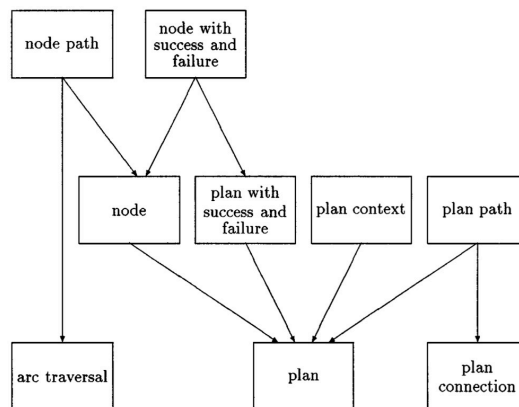
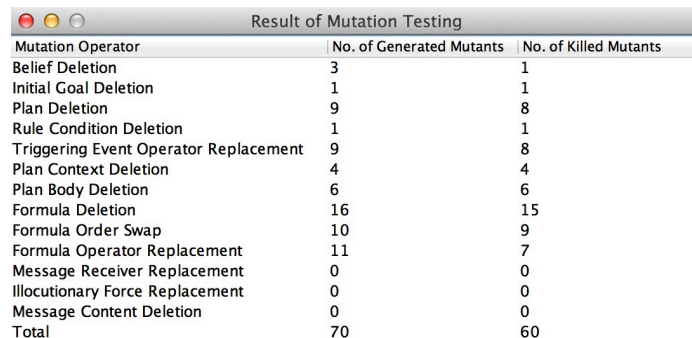


Fig. 6. The subsumption hierarchy of the coverage criteria proposed by Low et al. (Source: [1])

Since the environment is hard-coded into a java file, we use text replacement and class reload techniques in the *deploy(testID)* method to implement and deploy each test. We consider a mutant to be killed by a test if, at the end of the test, there is any garbage uncollected (in contrast, the non-mutated version always collects all the garbage). To implement this, we use Jason APIs and Java reflection in *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 addition, in *getAgentTTL(testID)* method, for each test, we set the lifetime of the original agent and each mutant to an amount that is just enough for the cleaner agent to finish cleaning.

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 non-predefined plans; next we check all the implemented operators that will be applied into the chosen mutation domain. After these we start and observe the mutation testing itself.

After the mutation testing, the result is displayed, as shown in Fig. 7. From the result 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 concern agent communication. We also observe that some mutants are not killed. We track these non-killed mutants in the log of the mutation testing process and analyze their corresponding changes in the code. It appears that many of them are equivalent mutants under the killing mutant criteria and TTL we set.



Mutation Operator	No. of Generated Mutants	No. of Killed Mutants
Belief Deletion	3	1
Initial Goal Deletion	1	1
Plan Deletion	9	8
Rule Condition Deletion	1	1
Triggering Event Operator Replacement	9	8
Plan Context Deletion	4	4
Plan Body Deletion	6	6
Formula Deletion	16	15
Formula Order Swap	10	9
Formula Operator Replacement	11	7
Message Receiver Replacement	0	0
Illocutionary Force Replacement	0	0
Message Content Deletion	0	0
Total	70	60

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

One non-equivalent mutant that is not killed is one that replaces the formula  $+pos(last, X, Y)$  in plan  $+!carry\_to(R)$  by  $+pos(last, X, Y)$ . The former formula is used to update the belief that keeps the last location where garbage was found, so that the agent can retrieve and 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  $+pos(last, X, Y)$ , each time the cleaner agent finds garbage, it will add a new belief representing the location of the garbage into the belief base rather than replacing the old one.

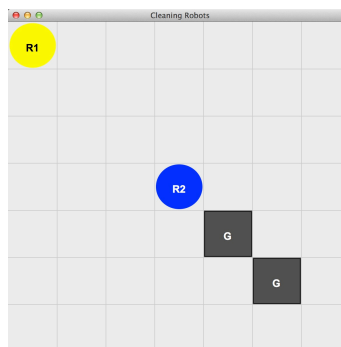
The above mutation is a fault, because it means that the agent will end up with several versions of “last location at which I found garbage” stored in its memory. In many cases, this is not a problem. When the cleaner agent has finished at the incinerator agent, it will try to take the shortest route back to the 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 stop 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 the last collected garbage location (Fig. 5 shows an example where it would happen). In order to detect this fault, we add a test 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  $G1$  and  $G2$  must be located after where the incinerator agent is along the fixed path for the agent 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 case in which the fault of multiple last locations can be detected. In this case, the cleaner robot will always return to the location where the garbage closer to the incinerator agent is after dropping either garbage, it will then continue moving along the fixed path from this location.



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

## 5 Discussion and Conclusion

In this paper we presented our preliminary work on mutation testing for multi-agent systems: we proposed a set of mutation operators for the Jason agent-oriented programming language; we described a mutation testing system called muJason for individual Jason agents, 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 the coverage criteria proposed by Low et al. [1], and found a non-equivalent mutant that was not killed. We were hence able to add a test to the test set for killing this mutant (and, probably, similar mutants or faults).

Our work draws on and expands that of Savarimuthu and Winikoff [7]: we proposed mutation operators for a specific implementation of AgentSpeak, implemented some of them and performed a preliminary empirical assessment. Their method for deriving mutation operators was very systematic, while ours has been more informal: we selected our operators using our judgment so that we can preferentially implement and assess them, in the hope of avoiding implementing some ineffective ones. It can be seen that their operator set contains ours for the core AgentSpeak, but ours contain some to cover Jason-specific features useful for practical agent implementation, such as Rule Condition Deletion (RCD), Annotation Deletion (AD) and Source Replacement (SR).

Another related work is Adra and McMinn's [5]. Although they used a rather different agent model, some of their ideas are relevant to our work. They proposed 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 theirs does not. As to their mutation operator class for the environment, it is not relevant in our operators for agents, although agent environments are an important dimension of MAS that act as the input source of agents, and we plan to mutate environments in future work.

In the future, we will derive mutation operators for Jason's advanced features (e.g., the use of *directives*, etc.), and implement them (along with the low-level ones we proposed in this paper but did not implement in muJason so far). We will also apply our approach to more Jason systems, and generate tests using other existing test evaluation criteria to further evaluate the effectiveness of our proposed mutation operators. There are challenges here – it is difficult to implement the low-level mutation operators because we need to extract the logical representations that these operators are applied into from different agent's architectural components. We have also not yet found any publicly-available Jason projects that are truly complex.

At the same time as the above, we will study the computational cost of our mutation testing and improve muJason in different aspects such as more flexible test setup and killing mutant criteria specification, and automatic measurement on when to kill

the mutant. After that we will expand muJason to support JaCaMo [15], which is a complete MAS programming paradigm that adopts Jason for programming agents, Moise for programming organizations and CArtaGO for programming environments. This will allow us to explore the mutation of organizational and environmental dimensions of MAS.

## References

1. Low, C.K., Chen, T.Y., Rönquist, R.: Automated test case generation for BDI agents. In: *Autonomous Agents and Multi-Agent Systems*, vol. 2, pp. 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-07)*, 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'13)*, pp. 1137–1138 (2013)
8. Savarimuthu, S., Winikoff, M.: Mutation Operators for the Goal Agent Language. In: Cosentino, M., Seghrouchni, A.E.F., Winikoff, M. (eds.) *Engineering Multi-Agent Systems*. Lecture notes in computer science, vol. 8245, pp. 255–273. Springer, Heidelberg (2013)
9. Houhamdi, Z.: Multi-agent system testing: a survey. In: *International Journal of Advanced Computer Science and Applications (IJACSA)*, 2(6), pp. 135–141 (2011)
10. Nguyen, C., Perini, A., Bernon, C., Pavón, J., Thangarajah, J.: Testing in multi-agent systems. In M.P. Gleizes, & J. Gomez-Sanz (eds.) *Agent-oriented software engineering X*. Lecture notes in computer science, 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. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. In: *Science of Computer Programming* (2011)
16. Wooldridge, M.: *An Introduction to MultiAgent Systems* (2nd ed.). John Wiley & Sons (2009)