Semantic Mutation Testing for Multi-Agent Systems

Zhan Huang and Rob Alexander

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

Abstract. This paper introduces semantic mutation testing (SMT) into multi-agent systems. SMT is a test assessment technique that makes changes to the interpretation of a program and then examines whether a given test set has the ability to detect each change to the original interpretation. These changes represent possible misunderstandings of how the program is interpreted. SMT can also be used to assess robustness to and reliability of semantic changes. This paper applies SMT to three rule-based agent programming languages, namely Jason, GOAL and 2APL, provides several contexts in which SMT for these languages is useful, and proposes three sets of semantic mutation operators (i.e., rules to make semantic changes) for these languages respectively, and a systematic approach to derivation of semantic mutation operators for rule-based agent languages. This paper then shows, through preliminary evaluation of our semantic mutation operators for Jason, that SMT has some potential to assess tests, robustness to and reliability of semantic changes.

Keywords: Semantic Mutation Testing, Agent Programming Languages, Cognitive Agents

1 Introduction

Testing multi-agent systems (MASs) is difficult because MASs may have some properties such as autonomy and non-determinism, and they may be based on models such as BDI which are quite different to ordinary imperative programming. There are many test techniques for MASs, most of which attempt to address these difficulties by adapting existing test techniques to the properties and models of MASs [9, 15]. For instance, SUnit is a unit-testing framework for MASs that extends JUnit [19].

Some test techniques for MASs introduce traditional mutation testing, which is a powerful technique for assessing the adequacy of test sets. In a nutshell, traditional mutation testing makes small changes to a program and then examines whether a given test set has the ability to detect each change to the original program. These changes represent potential small slips. Work on traditional mutation testing for MASs includes [1, 10, 16–18].

In this paper, we apply an alternative approach to mutation testing, namely semantic mutation testing (SMT) [5], to MASs. Rather than changing the program, SMT changes the semantics of the language in which the program is written. In other words, it makes changes to the interpretation of the program. These changes represent

possible misunderstandings of how the program is interpreted. Therefore, SMT assesses a test set by examining whether it has the ability to detect each change to the original interpretation of the program.

SMT can be used not only to assess tests, but also to assess robustness to and reliability of semantic changes: Given a program, if a change to its interpretation cannot be detected by a trusted test set, the program is considered to be robust to this change, in other words, this change is considered to be reliable for the program.

This paper makes several contributions. First, it applies SMT to three rule-based agent programming languages, namely Jason, GOAL and 2APL. Second, it provides several contexts (scenarios) in which SMT for these languages is useful. Third, it proposes three sets of semantic mutation operators (i.e., rules to make semantic changes) for these languages respectively, and a systematic approach to derivation of semantic mutation operators for rule-based agent languages. Finally, it presents a preliminary evaluation of the semantic mutation operators for Jason, which shows some potential of SMT to assess tests, robustness to and reliability of semantic changes.

The remainder of this paper is structured as follows: Section 2 describes two types of mutation testing, namely traditional mutation testing and semantic mutation testing. Section 3 describes SMT for Jason, GOAL and 2APL by showing several contexts in which it is useful and the source of semantic changes required to apply SMT in each context. Section 4 proposes sets of semantic mutation operators for these languages and an approach to derivation of semantic mutation operators for rule-based agent languages. Section 5 evaluates the semantic mutation operators for Jason. Section 6 compares our approach to related work, summarizes our work and suggests where this work could go in the future.

2 Mutation Testing

2.1 Traditional Mutation Testing

Traditional mutation testing is a test assessment technique that generates modified versions of a program and then examines whether a given test set has the ability to detect the modifications to the original program. Each modified program is called a *mutant*, which represents a potential small slip. Mutant generation is guided by a set of rules called *mutation operators*. For instance, Figure 1(a) shows a piece of a program and Figure 1(b) – 1(f) show five mutants generated as the result of the application of a single mutation operator called *Relational Operator Replacement*, which replaces one of the relational operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) by one of the others.

After mutant generation, the original program and each mutant are executed against all tests in the test set. For a mutant, if its resultant behaviour differs from the behaviour of the original program on some test, the mutant will be marked as *killed*, which indicates that the corresponding modification can be detected by the test set. Therefore, the fault detection ability of the test set can be assessed by the *mutant kill rate* – the ratio of the killed mutants to all generated mutants: the higher the ratio is, the more adequate the test set is. In the example shown in Figure 1, a test set consist-

ing of a single test in which the input is *x=3, y=5* cannot kill the mutants shown in Figure 1(b) and 1(f) because on that test these two *live* mutants result in the same behaviour as the original program (i.e., *return a*). Therefore, the mutant kill rate is 3/5. According to this result we can enhance the test set by adding a test in which the input is *x=4, y=4* and another test in which the input is *x=4, y=3* in order to kill these two live mutants respectively and get a higher mutant kill rate (the highest kill rate is 1, as this example shows).

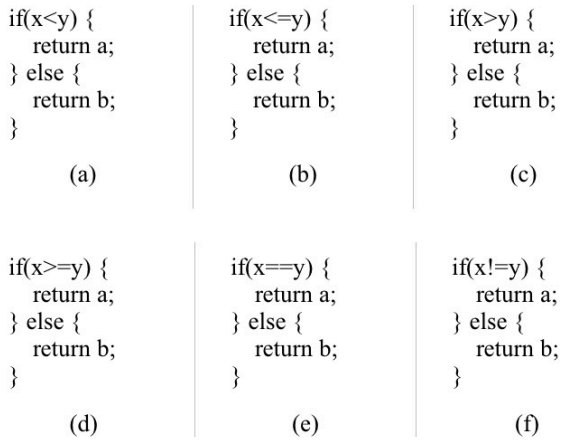

Fig. 1. An example of traditional mutation testing

Many studies provide evidence that traditional mutation testing is a very rigorous test assessment technique, so it is often used to assess other test techniques [2, 14]. However, the mutation operators used to guide mutant generation may lead to a large number of mutants because a single mutation operator has to be applied to each relevant point in the program and a single mutant only contains a modification to a single relevant point (as shown in Figure 1). This makes comparing the behaviour of the original program with that of each mutant on each test is computationally expensive.

Another problem is that traditional mutation testing unpredictably produces equivalent mutants – alternatives to the original program that are not representative of faulty versions, in that their behaviour is no different from the original in any way that matters for the correctness of the program. Thus, no reasonable test set can detect the modifications they contain. Equivalent mutants must therefore be excluded from test assessment (i.e., the calculation of the mutant kill rate). The exclusion of equivalent mutants requires much manual work although this process may be partially automated.

2.2 Semantic Mutation Testing

Clark et al. [5] propose semantic mutation testing (SMT) and extend the definition of mutation testing as follows: suppose N represents a program and L represents the semantics of the language in which the program is written (so L determines how N is

interpreted), the pair (N, L) determines the program's behaviour. Traditional mutation testing generates modified versions of the program namely N → (N_1, N_2, …, N_k) while SMT generates different interpretations of the same program namely L → (L_1, L_2, …, L_k). For SMT, L_1, L_2, …, L_k represent *semantic mutants*, their generation is guided by a set of rules called *semantic mutation operators*. For instance, Figure 2 shows a piece of a program, a semantic mutant (i.e., a different interpretation of this program) is generated by the application of a single semantic mutation operator that causes the *if* keyword to be used for mutual exclusion (i.e., when an *if* is directly followed by another *if*, the second *if* statement is interpreted the same as an *else-if* statement).

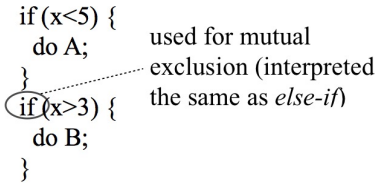

Fig. 2. An example of semantic mutation testing

SMT assesses a test set in a similar way as traditional mutation testing – comparing the behaviour under each semantic mutant with that under the original interpretation, in order to detect the killed mutants. In the example shown in Figure 2, a test set consisting of a single test in which the input is *x=2* cannot kill the semantic mutant because on that test the mutant results in the same behavior as the original interpretation (i.e., only *do A*). Therefore, the mutant kill rate is 0/1 = 0. We can enhance this test set by adding another test in which the input is *x=4* in order to kill the live mutant.

Compared with traditional mutation testing, SMT aims to simulate a different class of faults, namely possible misunderstandings of how the program is interpreted. Although many semantic misunderstandings can also be simulated by mutation of the program, a single semantic change may require multiple changes to the program rather than a single, small change made by traditional mutation testing. In addition, some semantic misunderstandings may lead to complex faults that simple program changes are hard to represent, and these complex faults may be harder to detect than small slips, e.g., [5] shows that SMT has potential to capture some faults that cannot be captured by traditional mutation testing.

SMT has another difference to traditional mutation testing: it generates far fewer mutants because a single semantic mutation operator only leads to a single semantic mutant[1], namely a different interpretation of the same program (as shown in Figure 2), while a single traditional mutation operator may lead to many mutants each of which contains a modification to a single relevant point in the program (as shown in Figure 1). This makes SMT much less computationally costly.

[1] This rule can be relaxed, namely mutating the semantics of only parts of the program instead of mutating the semantics of the whole program. This is useful, e.g., when the program is developed by several people.

SMT can be used not only to assess tests, but also to assess robustness to and reliability of semantic changes. Given a semantic mutant, if it cannot be killed by a trusted test set[2], it will be considered to be "equivalent"[3], which indicates that the program is robust to the corresponding semantic change or the semantic change is reliable for the program, otherwise the program may need to be improved to resist this change, or this change has to be discarded. In the example shown in Figure 2, if the program is required to be robust to the semantic change, it can be modified to ensure that at most one branch is executed in any case.

We know that SMT makes semantic changes for assessing tests, robustness to and reliability of semantic changes. For a particular language, which semantic changes should be made by SMT are context-dependent. For instance, to assess tests for a program written by a novice programmer, semantic changes to be made can be derived from common novices' misunderstandings of the semantics. To assess the portability of a program between different versions of the interpreter, semantic changes to be made can be derived from semantic differences between these versions.

3 Semantic Mutation Testing for Jason, GOAL and 2APL

We investigate semantic mutation testing for MASs by applying it to three rule-based programming languages for cognitive agents, namely Jason, GOAL and 2APL. These languages have generally similar semantics – an agent deliberates in a cyclic process in which it selects and executes rules according to and affecting its mental states. They also have similar constructs to implement such agents such as beliefs, goals and rules. The details of these languages can be found in [4, 6, 8] and are not provided here.

From Section 2.2 we know that for a particular language, the semantic changes that can most usefully be made by SMT is context-dependent. In the remainder of this section we provide several contexts in which SMT for the chosen agent languages is useful – use of a new language, evolution of languages, common misunderstandings, ambiguity of informal semantics and customization of the interpreter. We also show the source of semantic changes required to apply SMT in each context.

3.1 Use of a new language

When a programmer starts to write a program in a new (to him or her) language, he or she may have misunderstandings that come from the semantic differences between the new language and the old one(s) he or she has ever used. Therefore, in order for SMT

[2] A trusted test set is the one that is considered to be "good enough" for the requirement. It doesn't need to be the full test set that is usually impractical; instead it can choose not to cover some aspects or to tolerate some errors.

[3] Here the term "equivalent" is different to the one used in the context of test assessment, in which a mutant is equivalent only if there exist no tests that can kill the mutant. In the context of robustness/reliability assessment, a mutant is equivalent if only the trusted test set cannot kill it.

to simulate such misunderstandings, we should first find out their source, namely the semantic differences, by comparison between the new and the old languages. We use Jason, GOAL and 2APL as an example: a programmer who has ever used one of these languages may start to use one of the others. Since these languages each have large semantic size and distinctive features, we use the following strategies to guide the derivation of the semantic differences between them.

- Dividing the semantics of each of these languages into five aspects, as shown in Table 1. We do this because first of all, it provides a focus on examining four aspects of the semantics, namely *deliberation step order*, *rule selection*, *rule execution*, and *mental state query and update*, all of which are important and common to rule-based agent languages, while including *other* aspects that will be generally examined in order for completeness. Second, it is reasonable that common aspects of the semantics are more likely to cause misunderstandings than distinctive aspects in the context of using a new language, because distinctive aspects are usually supported by distinctive constructs that a programmer would normally take time to learn.
- Focusing on semantic differences between similar constructs. As [5] suggests, such differences easily cause misunderstandings because when writing a program in a new language a programmer may copy the same or similar old constructs without careful examination of their semantics given by the new language.
- Examining both formal and informal semantics of these languages. We start with examining the formal semantics because they can be directly compared. We also verify those that are informally defined through coding and reviewing the interpreter source code.
- Focusing on the default interpreter configuration. The interpreters of these languages are customizable, for instance, the Jason agent architecture can be customized by inheritance of the Java class that implements the default agent architecture; the GOAL rule selection order can be customized in the GOAL agent description. We think the default interpreter configuration is more likely to cause misunderstandings in the context of using a new language because if a programmer customizes an element it suggests he or she is familiar with its semantics.

Table 1. The aspects of the semantics of Jason, GOAL and 2APL (those marked with an asterisk are the ones we focus on)

ID	Aspect	Description
1	Deliberation step order*	Each deliberation cycle consists of a sequence of steps, e.g., rule selection → rule execution is a two-step sub-sequence.
2	Rule selection*	Rule selection is an important deliberation step in which one or several rules are chosen to be new execution candidates.
3	Rule execution*	Rule execution is an important deliberation step in which one or several execution candidates are chosen to execute.
4	Mental state query and update*	Mental states (i.e., beliefs and goals) can be queried in some deliberation steps such as rule selection and updated by execution of rules.
5	Other	Other aspects of the semantics not listed above.

Table 2 shows the semantic differences we found between Jason, GOAL and 2APL. These form the source of semantic changes required to apply SMT in the context of starting to use one of Jason, GOAL and 2APL from one of the others.

Table 2. Semantic differences between Jason, GOAL and 2APL

ID	Source	Jason	GOAL	2APL
1	The order of rule selection and rule execution	select a rule → execute a rule	(select and execute event rules → select and execute an action rule) x Number_of_Modules	select action rules → execute rules → select an external event rule → select an internal event rules → select a message event rule
2	Rule selection	• applicable • linear	• enabled • linear (action rules) and linearall (event rules)	• applicable • linear (event rules) and linearall (action rules)
3	Rule execution	• one rule/cycle • one action/rule	• one rule/cycle (action rules) and all rules/cycle (event rules) • all actions/rule	• all rules/cycle • one action/rule
4	Belief query	linear	random	linear
5	Belief addition	start	end	end
6	Goal query	$E \rightarrow I$; linear	random	linear
7	Goal addition	end of E	end	start or end
8	Goal deletion	delete the event and intention that relates to the goal φ	delete all super-goals of the goal φ	delete only the goal φ, all sub-goals of φ or all super-goals of φ
9	Goal type	procedural	declarative	declarative
10	Goal commitment strategy	no	blind	blind

Difference 1 comes from the order of two important deliberation steps, namely rule selection and rule execution. A Jason agent first selects a rule to be a new execution candidate and then chooses to execute an execution candidate. A GOAL agent processes its *modules* one by one, in each module it first selects and executes event rules and then selects and executes an action rule (both event and action rules are defined in the module being processed). A 2APL agent first selects action rules to be new execution candidates, and then executes all execution candidates, next selects an external event rule, an internal event rule and a message event rule to be new execution candidates.

Difference 2 comes from the rule selection deliberation step. Jason, GOAL and 2APL differ in two aspects of this step, namely the rule selection condition and the

default rule selection order. For the rule selection condition, a Jason or 2APL rule can be selected to be a new execution candidate if both its trigger condition and guard condition get satisfied ("applicable"), while a GOAL rule can be selected if it is applicable and the pre-condition of its first action gets satisfied ("enabled"). For the default rule selection order, Jason rules are selected in linear order (i.e., rules are examined in the order they appear in the agent description, and the first applicable rule is selected), GOAL action rules are selected in linear order while GOAL event rules are selected in "linearall" order (i.e., rules are examined in the order they appear in the agent description, and all enabled rules are selected), 2APL action rules are selected in "linearall" order while 2APL event rules of each type (external, internal, message) are selected in linear order.

Difference 3 comes from the rule execution deliberation step. In this step a Jason agent chooses a single execution candidate and then executes a single action in this candidate, a GOAL agent executes all actions in each selected event rule and each selected action rule[4], a 2APL agent executes a single action in each execution candidate.

Difference 4 comes from the belief query. In a Jason or 2APL agent, beliefs are queried in linear order (i.e., beliefs are examined in the order they are stored in the belief base, and the first matched belief is returned). In a GOAL agent, beliefs are queried in random order (i.e., beliefs are randomly accessed, and the first matched belief is returned).

Difference 5 comes from the belief addition. In a Jason agent, a new belief is added to the start of the belief base. In a GOAL or 2APL agent a new belief is added to the end of the belief base.

Difference 6 comes from the goal query. In a Jason agent, since goals exist in related events and intentions, the agent queries a goal by first examining its event base then its intention set following linear query order. In a GOAL agent, goals are queried in random order. In a 2APL agent, goals are queried in linear order.

Difference 7 comes from the goal addition. In a Jason or GOAL agent, a new goal is added to the end of the event or goal base. In a 2APL agent, a new goal is added to the start or the end of the goal base according to the relevant agent description (i.e., *adopta* or *adoptz*).

Difference 8 comes from the goal deletion. Given a goal φ to be deleted, a Jason agent deletes the event and intention that relates to φ, a GOAL agent deletes all goals that have φ as a logical sub-goal, a 2APL agent deletes only φ, all goals that are a logical sub-goal of φ, or all goals that have φ as a logical sub-goal according to the relevant agent description (i.e., *dropgoal*, *dropsubgoal* or *dropsupergoal*).

Difference 9 comes from the goal type. Jason adopts procedural goals – goals that only serve as triggers of procedures although it supports declarative goal patterns. GOAL and 2APL adopt declarative goals – goals that also represent states of affairs to achieve.

[4] Unlike Jason and 2APL, a GOAL agent has no intention set or similar structure, so a GOAL rule is immediately attempted to execute to completion once selected.

Difference 10 comes from the goal commitment strategy. Jason doesn't adopt any goal commitment strategy (i.e., a goal is just dropped once its associated intention is removed as the result of completion or failure) although it supports various commitment strategy patterns. GOAL and 2APL adopt blind goal commitment strategy, which requires a goal is pursued until it is achieved or declaratively dropped.

3.2 Evolution of Languages

When a programmer moves a program from a language to its successor (either a different language or a newer version of the same language), he or she may have misunderstandings that come from the semantic evolution, or may want to examine whether a program is robust to the semantic evolution or whether the semantic evolution is reliable. To derive semantic changes required to apply SMT in these cases, we should first find out their source, namely the semantic differences between the language and its successor. We take 2APL and its predecessor 3APL [7], and different versions of Jason as examples: Table 3 shows some semantic differences between 2APL and 3APL; Table 4 shows some semantic differences between different versions of Jason, which are derived from the Jason changelog [11]. We explain these differences as follows.

Semantic differences between 2APL and 3APL

Difference 1 comes from the PR-rules. In 2APL, the abbreviation "PR" means "plan repair", a PR-rule (i.e. an internal event rule) is selected if a relevant plan fails. In 3APL, "PR" means "plan revision", a PR-rule is selected if it matches some plan.

Difference 2 comes from the order of rule selection and rule execution deliberation steps. The order adopted by a 2APL agent has been described in Section 3.1. In contrast, a 3APL agent selects an action rule then a PR-rule to be new execution candidates, then chooses to execute an execution candidate.

Difference 3 comes from the action rule selection order. As described in Section 3.1, 2APL action rules are selected in "linearall" order. In contrast, 3APL action rules are selected in linear order.

Difference 4 comes from the rule execution deliberation step. As described in Section 3.1, a 2APL agent executes all execution candidates in a deliberation cycle. In contrast, a 3APL agent chooses to execute a single execution candidate.

Table 3. Some semantics differences between 2APL and 3APL

ID	Source	2APL	3APL
1	PR-rules	plan repair	plan revision
2	The order of rule selection and rule execution	see Table 2	select an action rule → select a PR-rule → execute a rule
3	Action rule selection	linearall	linear
4	Rule execution	all rules/cycle	one rule/cycle

Semantic differences between different versions of Jason

Difference 1 comes from the belief deletion action. Since Jason v0.95 the belief deletion action *-b* deletes *b* if *b* is a mental note (i.e. *b* has the annotation *source(self)*), while this action deletes *b* wherever it originates from before that version of Jason.

Difference 2 comes from the drop desire action. Since Jason v0.96 the drop desire action *.drop_desire(d)* removes the event and intention that is related to *d*, while this action removes only the related event before that version of Jason.

Table 4. Some semantic differences between different versions of Jason

ID	Source	Before some Version	Since that Version
1	Belief deletion action	*-b* deletes *b* wherever it originates from.	*-b* deletes *b* if *b* has the annotation *source(self)*.
2	Drop desire action	Remove only the related event.	Remove the related event and intention.

3.3 Common Misunderstandings

A programmer may have semantic misunderstandings that are common to a particular group of people he or she belongs to. Such misunderstandings can be identified by analysis of these people's common mistakes or faults. We take GOAL as an example: Table 5 shows some possible misunderstandings of the GOAL's semantics, which are derived from some common faults made by GOAL novice programmers [20]. We explain these misunderstandings as follows.

Possible misunderstanding 1 comes from the fault of the wrong rule order. If a programmer makes this fault in the GOAL agent description, he or she may have the misunderstanding that rules are selected in another available order[5] by default, e.g., action rules are selected in "linearall" order rather than linear order.

Possible misunderstanding 2 comes from the fault of a single rule including two user-defined actions. If a programmer makes this fault, he or she may have the misunderstanding that this is allowed like other agent languages.

Possible misunderstanding 3 comes from the fault of using "if then" instead of "forall do". If a programmer makes this fault, he or she may have the misunderstanding that "if then" is interpreted the same as "forall do".

Table 5. Some possible novice programmers' misunderstandings of GOAL

ID	Fault	Possible Misunderstanding
1	Wrong rule order	By default rules are selected in another available order.
2	A single rule including two user-defined actions	A rule can have more than one user-defined action.
3	Using "if then" instead of "forall do"	"if then" is interpreted the same as "forall do".

[5] GOAL supports four available rule evaluation orders: linear, linearall, random and randomall.

3.4 Ambiguity of Informal Semantics

A programmer may have misunderstandings of the semantics that are imprecisely or informally defined. For instance, [3] gives two examples of such misunderstandings of Jason as shown in Table 6. We explain these misunderstandings as follows.

Possible misunderstanding 1 comes from the goal deletion event. A goal deletion event (*-!e* or *-?e*) is generated if an intention that has the corresponding goal addition triggering event (*+!e* or *+?e*) fails. A programmer may have the misunderstanding that this event is generated if this intention is removed as the result of completion or failure.

Possible misunderstanding 2 comes from the test action. A test action (*?e*) generates a test goal addition event if it fails. A programmer may have the misunderstanding that a test action generates a test goal addition event if it is executed, which is similar to an achievement goal action (*!e*).

Table 6. Some possible misunderstanding of the Jason's informal semantics

ID	Source	Possible Misunderstanding
1	Goal deletion event	"if an intention fails" → "if an intention is removed"
2	Test action	Generate a test goal addition event if the action fails → Generate a test goal addition event if the action is executed

3.5 Customization of the Interpreter

The interpreters of Jason, GOAL and 2APL can be customized through modifying/overriding the functions of the interpreter or choosing between the provided options that can change the interpreter behaviour. Given an agent description, a programmer may want to know whether a custom interpreter provides an alternative to the original interpretation of the description. (The programmer may further examine whether the alternative interpretation leads to better performance, e.g., higher execution efficiency.) SMT can be applied in this context to represent potential customizations of the interpreter. We take Jason as an example: Table 7 shows some Jason interpreter configuration options, which are derived from the Jason changelog [11].

Table 7. Some Jason interpreter configuration options

ID	Option Description
1	Enable/disable tail recursion optimization for sub-goals.
2	Enable/disable cache for queries in the same cycle.
3	Choose whether the event generated by the belief revision action will be treated as internal or external.

3.6 Discussion

SMT is interesting to Jason, GOAL and 2APL in the contexts discussed above considering:

- These languages are declarative languages. They provide a focus on describing capabilities and responsibilities of an agent in terms of beliefs, goals, plans, etc., while encapsulating in the interpreter how an agent goes about fulfilling the responsibilities using the available capabilities. As a result, programmers are likely to pay insufficient attention to how an agent works, and therefore have relevant misunderstandings.
- These languages have customizable semantics. Since the semantics affects the agent behaviour and performance as well as the agent program, it is useful to explore different customizations of the semantics.

4 Semantic Mutation Operators for Jason, GOAL and 2APL

According to our derived sources of semantic changes required to apply SMT in different contexts, we derive three respective sets of semantic mutation operators for Jason, GOAL and 2APL as shown in Table 8(a) – 8(c). Due to space limitations we don't explain each semantic mutation operator in details.

It is worth noting that each operator set does not cover each context discussed in Section 3, e.g., the operator set for Jason has no operators that are derived from common misunderstandings of Jason. Therefore, we will improve each set when we acquire more sources of potential semantic changes to the corresponding language. In Table 8 each operator is labeled with its context(s) from which it is derived, e.g., the rule selection order change (RSO) operator for Jason is labeled with UNL (use of a new language), which indicates that this operator is derived from and can be used in (but is not limited to) the context of use of a new language discussed in Section 3.1.

Another noteworthy thing is that not every possible semantic change derived from Table 2 – 7 develops into a (or part of a) semantic mutation operator because some of them are considered to be unrealistic. Therefore, these unrealistic changes are adapted or simply discarded. A semantic change is considered to be unrealistic if it satisfies one of the following.

- It requires a significant change in the interpreter. We think that a programmer is not very likely to misunderstand the semantics a lot or to make such semantic change.
- It leads to the significantly different behaviour of each of our selected agent programs written in the corresponding language (i.e., 6 Jason programs, 6 GOAL programs or 4 2APL programs). We think that this semantic change is very easy to detect.

After analysis of these semantic mutation operators we find that most of them concern three kinds of the interpreter behaviour, namely *select*, *query* and *update*[6]. The elements to be selected include deliberation steps, rules, intentions, actions, etc; those to be queried or updated include beliefs, goals, events, etc. We also find that most

[6] We ever considered two more kinds of the interpreter behaviour, namely *transit* (between deliberation steps) and *execute* (a rule or action). However, we find that these two kinds can be classified as *select*, namely select between deliberation steps and select a rule or action to execute. This simplifies our classification.

operators change certain aspects of the interpreter behaviour, i.e., *order*, *quantity*, *position* and *condition*[7]. Table 9(a) and 9(b) list the kinds of the interpreter behaviour and the changeable aspects respectively (*other* kinds and aspects not mentioned above are included in order for completeness). Therefore, we propose a systematic approach to derivation of semantic mutation operators for rule-based agent languages, namely application of a changeable aspect into a kind of the interpreter behaviour. In Table 8 each semantic mutation operator is labeled with the kind of the interpreter behaviour it concerns and the aspect it changes, both of which are identified by their IDs shown in Table 9 (i.e., KID and AID respectively).

Abbreviations for the contexts discussed in Section 3

Use of a New Language: UNL
Evolution of Languages: EL
Common Misunderstandings: CM
Ambiguity of Informal Semantics: AIS
Customization of Interpreter: CI

Table 8(a). Semantic mutation operators for Jason

ID	Semantic Mutation Operator	Description	Context	KID	AID
1	Rule selection order change (RSO)	linear → linearall	UNL	1	1
2	Intention selection order change (ISO)	one intention/cycle → all intentions/cycle	UNL	1	1
3	Intention selection order change 2 (ISO2)	interleaved selection of intentions → non-interleaved selection of intentions	UNL	1	1
4	Belief query order change (BQO)	linear → random	UNL	2	1
5	Belief addition position change (BAP)	start → end	UNL	3	3
6	Belief revision action semantics change (BRAS)	generate internal events → generate external events[8]	CI	3	3
7	Belief deletion action semantics change (BDAS)	*-b* deletes *b* if *b* has the annotation *source(self)* → *-b* deletes *b*	EL	3	4
8	Goal addition position change (GAP)	end → start	UNL	3	3
9	Drop desire action semantics change (DDAS)	remove the related event and intention → remove only the related event	EL	3	2
10	Test goal action semantics change (TGAS)	generate a test goal addition event if the action fails → generate a test goal addition event if the action is executed	AIS	3	4
11	TRO enable/disable (TRO)	enable/disable tail recursion optimization for sub-goals	CI	3	5
12	Query cache enable/disable (QC)	enable/disable cache for queries in the same cycle	CI	2	5

[7] These changeable aspects may have overlaps, e.g., the change "select one rule → select all rules" can be a change to the order or the quantity.

[8] The plan chosen for an internal event will be pushed on top of the intention from which the event is generated; the plan chosen for an external event will become a new intention.

Table 8(b). Semantic mutation operators for GOAL

ID	Semantic Mutation Operator	Description	Context	KID	AID
1	Rule selection and execution order change (RSEO)	select and execute event rules then an action rule → select and execute an action rule then event rules	UNL	1	1
2	Rule selection condition change (RSC)	enabled → applicable	UNL	1	4
3	Rule selection order change (RSO)	change between linear, linearall, random and randomall	UNL, CM	1	1
4	Belief query order change (BQO)	random → linear	UNL	2	1
5	Belief addition position change (BAP)	end → start	UNL	3	3
6	Goal query order change (GQO)	random → linear	UNL	2	1
7	Goal addition position change (GAP)	end → start	UNL	3	3
8	Goal deletion semantics change (GDS)	"delete φ' if it is a super-goal of φ" → "delete φ' if it is φ" or "delete φ' if it is a sub-goal of φ"	UNL	3	4
9	The maximum number of user-defined actions change (MNUA)	1 → more than 1	CM	4	2
10	"if then" semantics change (ITS)	make "if then" interpreted the same as "forall do"	CM	2	2

Table 8(c). Semantic mutation operators for 2APL

ID	Semantic Mutation Operator	Description	Context	KID	AID
1	Rule selection and rule execution order change (RSREO)	change the original order "select action rules → execute rules → select event rules" to "select action rules → select event rules → execute rules" or "select event rules → select action rules → execute rules"	UNL, EL	1	1
2	Rule selection condition change (RSC)	applicable → enabled	UNL	1	4
3	Rule selection order change (RSO)	change between linear and linearall	UNL, EL	1	1
4	Plan selection order change (PSO)	all plans/cycle → one plan/cycle	UNL, EL	1	1
5	Belief query order change (BQO)	linear → random	UNL	2	1
6	Belief addition position change (BAP)	end → start	UNL	3	3
7	Goal query order change (GQO)	linear → random	UNL	2	1
8	PR-rule selection condition change (PRSC)	select a PR-rule if the relevant plan fails → select a PR-rule if it matches some plan	EL	1	4

Table 9. (a) Kinds of the interpreter behaviour
(b) Changeable aspects of the interpreter behaviour

(a)

KID	Kind
1	Select
2	Query
3	Update
4	Other

(b)

AID	Aspect
1	Order
2	Quantity
3	Position
4	Condition
5	Other

5 Evaluation of Semantic Mutation Operators for Jason

In order to assess the potential of SMT to assess tests, robustness to and reliability of semantic changes, we develop a semantic mutation system for Jason called *smsJason*. *smsJason* has three components, namely *tests*, *semantic mutation operators*, and *controller*, which are explained as follows.

- *tests* contains the following two custom parts for a particular Jason project:
  - A collection of tests. Each test is an array of values that can be used to instantiate the parameterized agent/environment description. A random test generator is employed to generate random tests given the constraints of each parameter. In addition, each test will be assigned a lifetime at runtime. This lifetime equals to the time taken by the Jason project under the original interpretation to pass this test plus a specified generous tolerance value for this test[9]. The Jason project under any mutant on this test will terminate anyhow when reaching this lifetime, if the project does not terminate as the result of passing this test yet.
  - Test pass criteria. The test pass criteria will be constantly examined at runtime in order to judge whether the Jason project has passed the current test. If the Jason project under the original interpretation is found to pass the current test, it will terminate and the lifetime of the test will be derived; if the project under any mutant is found to pass the current test before the lifetime of the test, it will terminate and the mutant will be marked as "live", otherwise it will terminate when reaching this lifetime and the mutant will be marked as "killed".
- *semantic mutation operators* implements our derived semantic mutation operators for Jason as shown in Table 8(a). Each operator leads to a modified version of the Jason interpreter (v1.4.1) which is pointed by a branch in Git [12] and can therefore be switched to another at runtime via Git API.

[9] The tolerance value is added because the exact time taken by the Jason project varies over a limited range in different runs. It is generous because the execution efficiency is not considered as part of the test pass criteria.

- *controller* implements the process of semantic mutation testing as shown in the following pseudo-code. *JRebel* [13], a powerful class reload technique, is employed to deploy each test (namely each instance of the parameterized agent/environment description) at runtime quickly.

```
1:  On each test:
2:      Run the Jason project under the original
          interpreter until it passes the test
3:      Derive the lifetime of the test

4:  Under each generated mutant:
5:      On each test:
6:          Run the Jason project until it passes the test
              or reaches the lifetime of the test
7:          Mark the mutant as "live" or "killed"
8:          Update the number of tests that killed the
              mutant if the test killed the mutant

9:  Display the SMT result
```

We apply *smsJason* into two Jason projects released with the Jason interpreter, namely *Domestic Robot* (DR) and *Blocks World* (BW). In DR, a robot constantly gets beer from the fridge and then serves its owner the beer until the owner exceeds a certain limit of drinking. The robot will ask the supermarket to deliver beer when the fridge is found empty. In BW, an agent restacks the blocks as required, by a series of actions of carrying or putting down a single block. We specify tests and test pass criteria for DR and BW as summarized in Table 10(a) – 10(b), after which we start the semantic mutation testing for each project. We analyze the SMT results displayed by *smsJason* and present the final results in Table 11.

Table 10(a). The tests and test pass criteria for the *Domestic Robot*

<table>
<tr><th>Parameter</th><th>Constraints</th><th>Test Pass Criteria</th></tr>
<tr><td>Drinking limit (Dl)</td><td>Dl ∈ [0, 16]</td><td rowspan="6">All of the following must be satisfied.
1. The robot is not carrying beer;
2. The robot has advised the owner about having exceeded the drinking limit;
3. The robot has checked the current time as requested by the owner;
4. Dl + 1 = Ib + Db – Rb, where Db is the beer delivered by the supermarket and Rb is the remaining beer in the fridge.</td></tr>
<tr><td>Map size (S x S)</td><td>S ∈ [1, 16]</td></tr>
<tr><td>Initial beer in the fridge (Ib)</td><td>Ib ∈ [0, 16]</td></tr>
<tr><td>Initial position of the robot (Pr)</td><td rowspan="3">Pr, Pf and Po take the form of (X, Y), where X, Y ∈[0, S - 1]</td></tr>
<tr><td>Initial position of the fridge (Pf)</td></tr>
<tr><td>Initial position of the owner (Po)</td></tr>
<tr><td colspan="3">Total number of tests: 160</td></tr>
</table>

Table 10(b). The tests and test pass criteria for the *Blocks World*

Parameter	Constraints	Test Pass Criteria
Original Stacks of Blocks (*OS*)	*OS* or *ES* is a set of lists and a partition of the set {"a", "b", "c", "d", "e", "f", "g"} representing all blocks; $1 \leq |OS|, |ES| \leq 3$	*OS* = *ES*
Expected Stacks of Blocks (*ES*)		
Total number of tests: 80		

Table 11. Results of semantic mutation testing

SMOP	Domestic Robot		Blocks World	
	Percentage of Tests that Kill the Mutant	Mutant Type	Percentage of Tests that Kill the Mutant	Mutant Type
RSO	0	NE	0	E
ISO	0	E	0	E
ISO2	100%	K	0	E
BQO	0	E	0	NE
BAP	0	E	37.5%	K
BRAS	0	N/A	0	N/A
BDAS	0	E	0	N/A
GAP	0	E	0	E
DDAS	0	N/A	0	N/A
TGAS	91.88%	K	0	N/A
TRO	0	E	0	E
QC	0	E	0	E

smsJason identifies the killed mutants (K), and we further classify those live mutants. First, by static analysis of the agent program we find that some live mutants are inapplicable (N/A) because the program has no constructs concerning the mutated semantics. For instance, the BW agent program has no actions of belief revision, belief deletion, drop desire and test goal, hence BRAS, BDAS, DDAS and TGAS are inapplicable to BW. Second, we attempt to identify equivalent mutants (E) among the applicable mutants by static and dynamic analysis of the agent program. For instance, we find that the DR or BW agent program has no constructs that cause the order of goal related events to matter; we also verify this through observing in Jason's mind inspector the relevant changes in agents' mental attitudes on all tests. Therefore, we conclude that GAP probably leads to the equivalent mutant. If we find a mutant likely to be not equivalent we will attempt to improve the tests or test pass criteria in order to kill it and classify it as non-equivalent (NE).

5.1 Assessment of Tests

The non-equivalent mutants (NE) indicate the weaknesses in the tests or test pass criteria. In order to kill such a mutant that RSO leads to, we need to capture the differences in the resultant agent behaviour between selecting all applicable plans and selecting only the first applicable plan. These plans must have the same triggering event, the contexts that are not mutually exclusive and the ability to affect the agent

behaviour. In the DR agent program, the only two such plans are the robot's plan to get beer when the fridge is empty (*p1*) and the robot's plan to get beer when the owner exceeds the limit of drinking (*p2*). Therefore, we need a test on which the owner just exceeds the limit of drinking when there is no beer in the fridge. This test will cause *p2* to execute twice under the mutant so that the robot will advise the owner twice about having exceeded the drinking limit. We also need to improve the test pass criteria to capture the number of advices given by the robot.

In order to kill the non-equivalent mutant that BQO leads to, we need to capture the differences in the resultant agent behaviour between querying beliefs in linear order and in random order. In the BW agent program, there is only one place that causes the belief order or belief query order to matter, namely the context of the plan (*p*) which is to remove a block from the top of a stack in order to further move a block (*b*) in the same stack. It is worth noting that *b* can belong to more than one stack held by the belief base, for instance, there are two stacks, namely *S(b1, b2, b)* and *S(b2, b)*, where the former contains the latter. In order to move *b*, *b1* has to be removed first.

Under the original interpretation where beliefs are queried in linear order, the context of *p* always returns *S(b1, b2, b)* so that *b1* can be removed. This is because the larger the stack is, the more recently it is added to the start of the belief base, as the result of the application of the belief revision rule to derive stacks. In contrast, under the mutant that BQO leads to, the context of *p* is likely to return *S(b2, b)*, which causes *p* to retry until *S(b1, b2, b)* is returned because *b2* cannot be removed before *b1*. Therefore, we need to improve the tests or test pass criteria in order to capture the retrying of *p*.

5.2 Assessment of Robustness to Semantic Changes

The equivalent mutants (E) indicate that the agent program is robust to the corresponding semantic changes, while the killed or non-equivalent mutants (K or NE) indicate the weaknesses in robustness. In order for the DR agent program to be robust to the semantic change caused by RSO, we can improve the program by ensuring that there is only one applicable non-empty plan at most in every deliberation cycle. As mentioned in Section 5.1, there are only two non-empty plans (*p1* and *p2*) which are likely to become applicable simultaneously in the same cycle, therefore, we can make their contexts mutually exclusive, e.g., by strengthening the context of *p2*.

In order for the BW agent program to be robust to the semantic changes caused by BQO and BAP, we need to make the program's behaviour independent of the order of beliefs or querying beliefs. As mentioned in Section 5.1, there is only one place that causes these orders to matter, namely the context of *p*. Therefore, we can strengthen this context by ensuring that it always returns the largest stack.

As for the semantic changes caused by ISO2 and TGAS, we find it very expensive hence inappropriate to make the agent program be robust to these changes.

5.3 Assessment of Reliability of Semantic Changes

We have improved the DR agent program to resist the semantic change caused by RSO and the BW agent program to resist the semantic changes caused by BQO and BAP, as suggested in Section 5.2. Therefore, RSO, BQO and BAP lead to reliable alternative interpretations of the corresponding agent program as well as the equivalent mutants as shown in Table 11. To further assess the execution efficiency that these reliable alternative interpretations lead to, we make *smsJason* be able to compare the test execution time under the original interpretation and under each reliable alternative interpretation. We present the results of execution efficiency assessment in Table 12.

Table 12. Results of execution efficiency assessment

SMOP	Domestic Robot		Blocks World	
	Percentage of Avg Saved Time	Percentage of Tests that Saved Time	Percentage of Avg Saved Time	Percentage of Tests that Saved Time
RSO	-0.06%	45.63%	-0.33%	41.25%
ISO	7.5%	100%	28.42%	100%
ISO2	N/A		-0.72%	37.5%
BQO	0.49%	53.75%	0.16%	63.75%
BAP	-0.34%	38.75%	-0.15%	41.25%
BRAS	N/A		N/A	
BDAS	-0.01%	43.75%	N/A	
GAP	0.23%	50.63%	0.19%	51.25%
DDAS	N/A		N/A	
TGAS	N/A		N/A	
TRO	0.33%	45.63%	0.08%	50%
QC	0.13%	43.13%	0.27%	46.25%

In Table 12, the inapplicable or unreliable mutants are marked as "N/A". Among the reliable mutants, the one caused by ISO is interesting because it significantly reduces the average execution time of DR and BW by 7.5 and 28.42 percent respectively, and it leads to efficiency improvement on all tests.

The changes in efficiency that are caused by other reliable mutants are not significant hence may be just caused by normal floating of execution time.

6 Related Work and Conclusions

In Section 2 we compared SMT to traditional mutation testing. Here we compare them in terms of multi-agent systems, by two examples showing that the semantic mutation operators for GOAL as shown in Table 8(b) can simulate some faults that cannot be captured by the traditional mutation operators for GOAL which are derived by Savarimuthu and Winikoff [18].

The RSO semantic mutation operator for GOAL can change the action rule selection order from "linear" to "linearall", which is similar to the change from *else-if* to *if*.

We examine the traditional mutation operators for GOAL and find no operators that can simulate this semantic change. For instance, these traditional mutation operators can *delete* or *swap* an element, however, deleting a single plan or swapping two plans cannot simulate this semantic change.

The BQO semantic mutation operator changes the belief query order from "random" to "linear". Again we cannot find any traditional mutation operator for GOAL that can simulate this semantic change.

In this paper, we applied SMT to Jason, GOAL and 2APL. We showed that SMT for these languages is useful in several contexts, namely use of a new language, evolution of languages, common misunderstandings, ambiguity of informal semantics and customization of the interpreter. We derived sets of semantic mutation operators for these languages, and proposed a systematic approach to derivation of semantic mutation operators for rule-based agent languages. Finally, we used two Jason projects in a preliminary evaluation of the semantic mutation operators for Jason. The results suggest that SMT has some potential to assess tests, robustness to and reliability of semantic changes.

Our future work will focus on further evaluation of the semantic mutation operators for Jason. To further evaluate the ability of these operators to assess tests, we will examine their *representativeness* by comparing to realistic semantic misunderstandings and their *power* by looking for more hard-to-kill mutants (as we have done in this paper), as suggested by [10]. To further evaluate the ability of these operators to assess robustness to and reliability of semantic changes, we will apply them to more Jason projects so as to provide more suggestions on improving program robustness and optimizing interpreter.

References

1. Adra, S.F., McMinn, P.: Mutation operators for agent-based models. In: Proceedings of 5th International Workshop on Mutation Analysis. IEEE Computer Society (2010)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
3. Bordini, R.H., Hübner, J.F.: Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions). In: Proceedings of ECAI'10, pp. 635–640 (2010)
4. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
5. Clark, J.A., Dan, H., Hierons, R.M.: Semantic Mutation Testing. Science of Computer Programming (2011)
6. Dastani M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
7. Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming multi-agent systems in 3APL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 39–67. Springer, Heidelberg (2005)
8. Hindriks, K.V.: Programming rational agents in GOAL. In: Bordini R.H., Dastani M., Dix J., El Fallah Seghrouchni A. (eds.), Multi-agent programming: Languages, platforms and applications, vol. 2, pp. 3–37. Springer, Heidelberg (2009)

9. Houhamdi, Z.: Multi-agent system testing: A survey. International Journal of Advanced Computer Science and Applications (IJACSA) 2(6), 135–141 (2011)
10. Huang Z., Alexander R., Clark J.A.: Mutation Testing for Jason Agents. In: Dalpiaz F., Dix J., van Riemsdijk, M.B. (eds.) EMAS 2014. LNCS (LNAI), vol. 8758, pp. 309–327. Springer, Heidelberg (2014)
11. Jason changelog, http://sourceforge.net/p/jason/svn/HEAD/tree/trunk/release-notes.txt
12. JGit documentation, https://eclipse.org/jgit/documentation/
13. JRebel documentation, http://zeroturnaround.com/software/jrebel/learn/
14. Mathur, A.P.: Foundations of Software Testing. Pearson (2008)
15. 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)
16. 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)
17. 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)
18. Savarimuthu, S., Winikoff, M.: Mutation Operators for the GOAL Agent Language. In: Cossentino M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 255–273. Springer, Heidelberg (2013)
19. Tiryaki A.M., Oztuna S., Dikenelli O., Erdur R.C.: Sunit: A unit testing framework for test driven development of multi-agent systems. In: Agent-Oriented Software Engineering VII. LNCS, vol. 4405, pp. 156–173. Springer, Heidelberg (2006)
20. Winikoff M.: Novice programmers' faults & failures in GOAL programs. In: Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2014), pp. 301–308 (2014)