Mutation for Multi-Agent Systems

Zhan Huang

PhD

University of York

Computer Science

March 2016

Abstract

Although much progress has been made in engineering multi-agent systems (MAS), many issues remain to be resolved. One issue is that there is a lack of techniques that can adequately evaluate the effectiveness (fault detection ability) of tests or testing techniques for MAS. Another is that there are no systematic approaches to evaluating the impact of possible semantic changes (changes in the interpretation of agent programs) on agents' behaviour and performance. This thesis introduces syntactic and semantic mutation to address these two issues.

Syntactic mutation is a technique that systematically generates variants ("syntactic mutants") of a description (usually a program) following a set of rules ("syntactic mutation operators"). Each mutant is expected to simulate a real description fault, therefore, the effectiveness of a test set can be evaluated by checking whether it can detect each simulated fault, in other words, distinguish the original description from each mutant. Although syntactic mutation is widely considered very effective, only limited work has been done to introduce it into MAS. This thesis extends syntactic mutation for MAS by proposing a set of syntactic mutation operators for the Jason agent language and showing that they can be used to generate real faults in Jason agent programs.

By contrast, semantic mutation systematically generates variant interpretations ("semantic mutants") of a description following a set of rules ("semantic mutation operators"). Semantic mutation has two uses: to evaluate the effectiveness of a test set by simulating faults caused by misunderstandings of how the description is interpreted, and to evaluate the impact of possible semantic changes on agents' behaviour and performance. This thesis, for the first time, proposes semantic mutation for MAS, more specifically, for three logic based agent languages, namely Jason, GOAL and 2APL. It proposes semantic mutation operators for these languages, shows that the operators for Jason can represent real misunderstandings and are practically useful.

Table of Contents

List of Figures and Tables

Acknowledgement

I would like to thank my supervisor, Rob Alexander for his invaluable suggestions for improving, evaluating and publishing every idea. I would also like to thank my second supervisor, John Clark for his useful suggestions for research directions.

I am grateful to Rafael Bordini for helping us collect relevant data from his students. I appreciate that Koen Hindriks and Michael Winikoff answered a lot of useful questions.

I give my greatest thanks to my wife and parents for their unconditional support and company.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Some of the work presented in this thesis has appeared in my publications listed below:

1. Huang, Z., Alexander, R., Clark, J.: Mutation testing for Jason agents. In: Dalpiaz, F., Dix, J., van Riemsdijk, M. (eds.) EMAS 2014. LNCS, vol. 8758, pp. 309–327. Springer, Heidelberg (2014)
2. Huang, Z., Alexander, R.: Semantic mutation testing for multi-agent systems. In: Baldoni, M., Baresi, L., Dastani, M. (eds.) EMAS 2015. LNCS, vol. 9318, pp. 131–152. Springer, Heidelberg (2015)

1. Introduction

Although much progress has been made in engineering multi-agent systems (MAS), many issues remain to be resolved. Two of these issues are as follows:

1. **There is a lack of techniques that can adequately evaluate the effectiveness (fault detection ability) of tests or testing techniques for MAS.**
2. **There are no systematic approaches to evaluating the impact of possible semantic changes (changes in the interpretation of agent programs) on agents' behaviour and performance.**

Some recent work introduces *syntactic mutation* [23, 66], a very effective technique for evaluating test effectiveness, to address Issue 1. Traditionally, syntactic mutation generates slightly modified versions of a program, known as *syntactic mutants*, following a set of rules called *syntactic mutation operators*. Each syntactic mutant simulates a simple fault in the program, therefore, the effectiveness of a test set can be evaluated by checking whether it can detect each simulated fault, in order words, distinguish the original program from each mutant. This thesis introduces syntactic mutation into the Jason agent language [20].

This thesis also, for the first time, introduces another approach to mutation, namely *semantic mutation* [23], into MAS, more specifically, into three logic based agent languages, namely Jason, GOAL [37] and 2APL [25]. Typically, semantic mutation generates variant interpretations of the same program, known as *semantic mutants*, following a set of rules called *semantic mutation operators*. Each semantic mutant simulates a fault caused by a misunderstanding of how the program is interpreted, therefore, the effectiveness of a test set can be evaluated by checking whether it can distinguish the original interpretation from each mutant. Semantic mutation shows potential in simulating hard-to-detect faults and in complementing syntactic mutation [23].

Semantic mutation can be used to evaluate not only test effectiveness, but also the robustness to and the reliability of semantic changes: If a *trusted* test set cannot detect a change to the interpretation of a program, the program is robust to the change, in other

words, the change is reliable for the program. It is possible to use semantic mutation to further check whether this change leads to better agent performance.

Syntactic/semantic mutation is effective in evaluating tests if it can generate mutants that are *representative* of real faults (i.e., mutants whose detection is strongly correlated with real fault detection), and is *powerful* enough to generate hard-to-detect mutants for indicating the weaknesses in testing.

The remainder of this chapter is organized as follows: Sect.1.1 and 1.2 respectively introduce some background of MAS engineering and testing relevant to this thesis; Sect. 1.3 describes the issues addressed by this thesis; Sect. 1.4 describes the contributions and structure of this thesis.

1.1. Engineering Multi-Agent Systems

This section starts with the concept of *agent*. An agent is an *autonomous* computer system situated in an environment. Being autonomous an agent can control its own actions in order to achieve its delegated objectives. An agent can act in a *reactive* or/and *proactive* way: Being reactive an agent can respond in a timely fashion to environmental changes; being proactive an agent can *take the initiative* to further its objectives. A multi-agent system (MAS) is a distributed system composed of *interacting* agents — agents that can interact with each other in order to achieve their own or common objectives [111].

MAS have become a promising paradigm for developing distributed systems that involve autonomous components, or that are situated in environments where human intervention is difficult or expensive. They have been used in some fields such as scientific simulation, electronic commerce and traffic control [71]. For instance, Smith et al. [100] address the issue of optimizing vehicle traffic network by deploying agents in each intersection and each decision point in the network.

There are many models [111], methodologies [15, 34], languages [18] and tools [54] for building MAS. This thesis focuses on a widely studied class of agents – cognitive agents, and a dominant class of languages for programming them – declarative agent languages.

1.1.1. Cognitive Agents

Our behaviour is often determined by *cognition*, which can be described in *cognitive terms* such as beliefs and intentions. For instance, I *believe* that it is not raining now, so I *intend* to go shopping. Agents that can determine their own behaviour by cognition are called *cognitive agents* [42]. There are many *cognitive architectures* that specify the underlying infrastructure for cognitive agents [55], among which an influential class mainly involves three cognitive terms, namely beliefs (B), desires (D) and intentions (I), therefore, is called BDI architectures [92]. Fig. 1 shows a typical BDI architecture explained below.

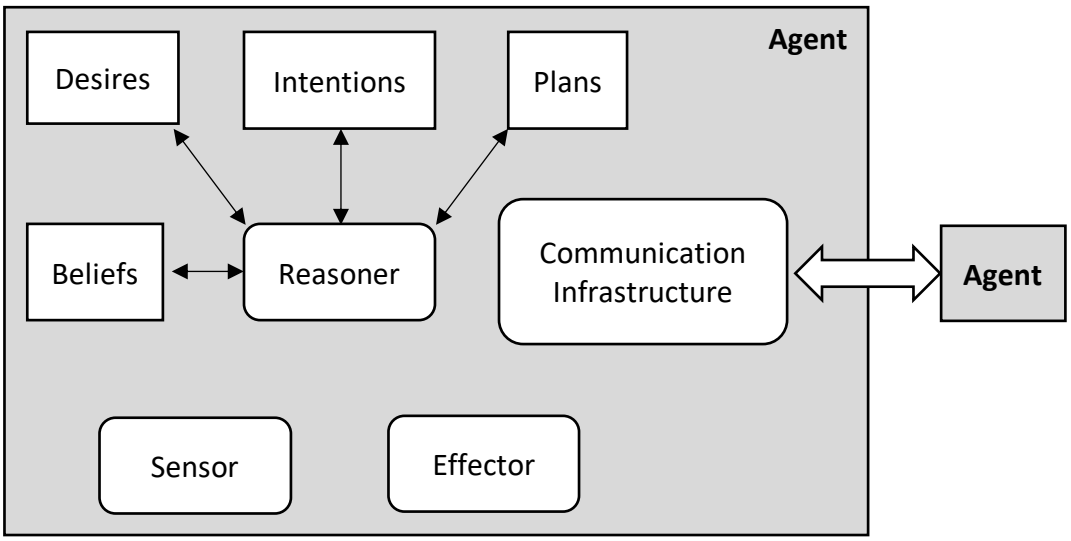


Fig. 1. A typical BDI architecture

In Fig. 1, the rectangles represent four categories of cognitive elements that form the agent's cognitive state, namely beliefs, desires (goals), intentions and plans. Beliefs represent what the agent knows about, which can be the state of the environment, the agent itself or the other agents; desires represent what the agent prefers — the goals that the agent would like to achieve; intentions represent what the agent has committed to do.

Plans are *recipes* for action. A typical plan has three parts, namely event, context and body. The event is the trigger of the plan, which can be some belief change or desire; the context is a set of conditions under which the triggered plan can be adopted; the body is a sequence of actions to carry out in order to change the environment, the agent's cognitive state, or send messages to the other agents.

The rounded boxes represent the functional components of the agent, namely the sensor, effector, communication infrastructure and reasoner. The sensor perceives the environment in order to update the agent's beliefs; the effector acts on the environment as instructed by the agent's actions; the communication infrastructure sends messages to the other agents as instructed by the agent's actions, and receives messages in order to affect the agent's cognitive state. The reasoner plays a key role in how the agent works: It periodically attempts to find plans that can be triggered and adopted according to the agent's cognitive state, adds them to the agent's intentions and then executes the actions in the uncompleted intentions.

It can be seen that BDI agents show both reactivity and proactivity: They can select and execute plans in response to belief changes caused by environmental changes, and in an attempt to achieve their own desires. Recently BDI agents have been extended by more kinds of cognitive elements such as capabilities [22] and emotions [89].

Cognitive agents provide a natural way to build autonomous systems [42]. However, since cognitive agents need environments to be encoded into symbols for reasoning, their reaction is delayed, in contrast to purely reactive agents that are based on direct stimulus-response mappings. Therefore, cognitive agents may be inappropriate when very fast reaction is required (e.g., when the environment is highly dynamic) [111].

1.1.2. Declarative Agent Languages

Recall from Sect. 1.1.1 that a (BDI-like) cognitive architecture controls an agent by checking and updating cognitive elements regularly. A declarative agent language (DAL) implements an agent's architecture in the interpreter, and it allows the architecture to be initialized by a program that specifies some initial cognitive elements such as beliefs, goals and plans. To get more familiar with DAL, a program for a fighting agent in *AgentSpeak* [91], an influential abstract language for programming BDI agents, is shown in Fig. 2 and interpreted below.

```
//belief
can_fight.

//goal
!fight.

//p1
+!fight : can_fight <- attack; !fight.

//p2
+opponent_down : true <- -can_fight.
```

Fig. 2. A typical AgentSpeak program

An agent initialized by the program in Fig. 2 will start with a belief (*can_fight*), a goal (*!fight*) and two plans (denoted by *p1* and *p2* respectively). The agent will first select *p1* to execute because the initial goal and belief satisfy *p1*'s trigger and context (separated by a colon) respectively. *p1* has two actions: *attack* is an action that will be carried out by the agent's effector; *!fight* is to repost the goal before *p1* is completed so that *p1* can be continuously selected for attacking the opponent (An AgentSpeak goal will be dropped once a plan for handling it completed). When it is perceived that the opponent is down and the corresponding belief (*opponent_down*) is added, *p2* will be selected because the belief change (addition) satisfies *p2*'s trigger (*p2*'s context can always be satisfied if specified *true*). *p2* has only one action, namely *-can_fight* which is to delete the corresponding belief in order to prevent *p1* being selected for more attacks.

DAL can facilitate programming cognitive agents: it does not require programmers to implement the complex, error-prone control part of agents. Instead, it allows them to focus on using common sense notions and high-level abstractions. In addition, DAL provides many practical features, such as handling plan failures [94], connecting/wrapping external/legacy code [16, 20], and customizing how agent programs are interpreted [20, 37].

However, DAL brings some difficulties in programming agents. First, effort has to be made to learn a different programming paradigm than the mainstream ones [39]. Second, attention has to be paid to potential conflicts in parallel elements (e.g., goals/plans pursued/executed in parallel) introduced by DAL [116]. For instance, an action (A1) in an intention (I1) may prevent an action (A2) in another parallel intention (I2), such as by dissatisfying the execution condition (C) of A2. If this conflict causes the incorrect agent behaviour, e.g., I2 fails to continue execution for achieving some required goal, it should be

avoided or properly handled, such as by changing C, requiring I1 to be executed after the completion of I2, or specifying a remedial plan that will be selected for execution once the goal fails.

Given a cognitive agent program, it is sometimes necessary or useful to change its interpretation (the way the specified cognitive elements are controlled by the agent architecture) such as changing how to select plans/intentions, update beliefs/goals, execute actions, in order to customize the agent behaviour [39, 59]. For instance, if a plan has to be executed before another to avoid conflicts, the default interpretation in which they are allowed to be executed in parallel can be changed. It is worth noting that DAL uses a different approach to engineering systems than other declarative languages such as SQL and Prolog: other declarative languages adopt *black-box* approach, which is intended to hide the control as much as possible from programmers. By contrast, DAL uses *glass-box* approach, which requires some parts of the control to be customized in order to retain the intuitive meanings of common-sense notions in agent programs [39].

Some customizations that can be made by changing the interpretation can also be made by changing the program, for instance, in the above example, the two plans can be combined into one to avoid conflicts. However, changing the interpretation is sometimes more convenient because it is a centralized way. For instance, Winikoff [105] introduces a meta-interpreter which can facilitate changing the interpretation of cognitive agents, and he gives an example of its usefulness: to let a debugging agent monitor communication between agents in a MAS (namely let the debugging agent be an extra recipient of each message sent by each agent). Extending the interpretation of the message sending action is much more convenient than adding the debugging agent's ID into each message sending action in each agent program.

However, it is sometimes difficult for programmers to determine a proper or optimal way to change the interpretation of cognitive agent programs [59]. For instance, it is not easy to determine the intention selection order that will not create any conflict, if potential parallel intentions are many. To address this difficulty, some approaches have been proposed to guide the change of the interpretation [59]. For instance, intention selection strategy can be generated by task scheduler [17], reasoning with summary information [24], measuring goal coverage [101], or Single Player Monte Carlo Tree Search [113-116]. Each of

these example approaches is claimed to have advantages such as minimizing conflicts, fairness, efficiency, robustness.

Among many declarative agent languages, a promising class is logic based agent languages (LBAL), in which agent programs consist of sentences in logical form, therefore, are simple and elegant (see Fig.2 for an example). Recent LBAL include Jason (an extended implementation of AgentSpeak), GOAL and 2APL. These languages have similar semantics – an agent selects and executes *rules* (plan-like elements) according to its cognitive elements. They also have similar constructs, namely beliefs, goals and rules.

1.2. Testing Multi-Agent Systems

1.2.1. Principles of Software Testing

Software testing is *the process of executing the program with the intent of finding errors* [72]. Here an *error* refers to a difference between an actual result and the expected result. An incorrect result is called a *failure*. Errors/failures are usually caused by incorrect steps, processes or data definitions, known as *faults* in the program. Faults are caused by programmers' *mistakes*, which may be misunderstandings of the specification, typographic mistakes, etc. [1].

Typically, software testing involves three stages, namely generating tests, executing the program against each test (namely executing each test), and determining whether the tests have *passed* or *failed* [88]. A test represents a requirement that the program must satisfy [49], it usually consists of the following two parts:

1. The input data to the program: It can be a set of simple- or complex-type values or conditions, representing stimuli, the initial state, the execution environment of the program, etc.
2. The expected output of the program: It can be a set of values or conditions representing responses to (or effects on) the external environment, the program's state during/after execution, constraints on behaviour (e.g., the maximum execution time of the program), etc.

A test is considered to have passed if the program executed with the input data is found to have produced the expected output, or failed if the program is found to have produced different output than expected. Failed tests indicate failures/errors/faults. The mechanism for determining whether a test has passed or failed is called an *oracle*, which involves capturing the actual output and comparing it to the expected (The expected output of tests is sometimes specified in oracles for convenience). A testing technique focuses on one or more stages of testing.

Testing is the most important means to providing the confidence in the correctness of the program [10]. The program will become more correct if more faults are detected and eliminated. Mistakes that led to the faults can be further identified and learnt. Testing can be performed on parts or the whole of the program. A recent survey [31] shows that an average of 35% of development time is spent on testing. Testing is not a standalone activity, instead it has to be integrated into the software development process and to cooperate with other activities such as debugging and verification, in order to collectively assure the quality of the program [66].

The key to testing is to generate effective tests, namely tests capable of detecting faults. The effectiveness of a test set is in general defined as the ratio of the faults it can detect to all faults. In software development, it is useful to evaluate the effectiveness of tests generated for the program under development in order to improve them for detecting more potential faults. In research, it is useful to evaluate the tests generated using a testing technique in order to evaluate the effectiveness of the technique. In general, the effectiveness of tests or testing techniques can be evaluated using the following two sets of criteria:

1. Coverage criteria: A coverage criterion requires that a set of elements in a program model (e.g., control flow graph [66], UML diagram [67], input domain [66]) be involved by the test set. For instance, a coverage criterion called Statement Coverage requires that each statement in the program graph be executed at least once [66].

   Coverage criteria are used to measure how thoroughly the program is executed in testing. They can help find uncovered model elements (e.g., statements, execution paths), which may indicate untested aspects of the program behaviour. However, thorough execution of a model element on a test set does not guarantee that any fault

related to the element can be detected (an example is that the expected output of the tests is incompletely specified). Recent empirical studies [44, 104] show that coverage is not strongly correlated with fault detection.

2. Fault-based criteria: A fault-based criterion requires that a test set detect a set of known faults, which can be real ones in evaluating testing techniques, or can be seeded by hand or by mutation in evaluating either testing techniques or tests for the program under development. Note that in evaluating testing techniques, since adequate real faulty programs are hard to find, faults are usually seeded in the correct versions of the subject programs; while in evaluating tests for the program under development, faults are seeded in the program, which is assumed to be correct as long as it has passed the tests being evaluated. This is reasonable, because the aim of fault based criteria is not to detect faults, but to evaluate test effectiveness in terms of the ability to detect known faults.

   In contrast to coverage criteria, fault-based criteria provide a straightforward way to measure fault detection ability. In particular, mutation is better than any other test evaluation criteria, because (1) compared with seeding faults by hand, mutation provides a replicable and fast way to simulate a large number of faults that are more representative of real faults [11]; (2) compared with coverage criteria, mutation is better correlated with real faults [11, 47], and help generate more effective tests [57, 102].

1.2.2. MAS Testing

Intuitively, testing MAS is difficult because of some characteristics of MAS [73]. For instance, the same test input may lead to different output, because the agent may autonomously adapt its own behaviour according to different situations or by learning. Agents interact by exchanging messages, which is different to objects that often interact by invoking methods.

Testing cognitive agents is especially difficult. Miles et al. [68] provide the following reasons by analyzing a cognitive agent program:

1. The actual steps of agent execution are opaque, because they are encapsulated in the interpreter.
2. An execution path is hard to trace, because the agent *reconsiders* at regular intervals.

3. Whether a committed plan will produce the expected effect is hard to predict, because other committed plans and external factors may interfere with this plan by affecting the shared cognitive or environmental state.
4. Possible failures (e.g., in achieving goals, executing plans or actions) and ways to handle them make the agent behaviour harder to predict.

Winikoff and Cranefield [108] make a quantitative study of how difficult it is to test BDI agent programs based on coverage criteria. They choose the strongest coverage criterion called Full Paths Coverage that requires every possible execution path in the program to be executed at least once, quantify possible execution paths[1] in a BDI agent program, and then find that they are more than in an equivalently-sized procedural program, therefore, harder to cover. Winikoff [106] repeats the study but chooses Decision Coverage, the *generally accepted minimum* criterion that requires all branches of each decision point (e.g., branches corresponding to TRUE and FALSE evaluation of each *if* statement) to be executed at least once, and finds that testing BDI systems based on this criteria is still harder.

To resolve the issue that coverage-based testing of BDI agent programs is harder, Winikoff and Cranefield [108] propose some potential solutions including:

1. Avoiding the use of program structures that augment the number of possible execution paths (e.g., deep and wide goal-plan hierarchy, failure handlers). However, avoiding them sacrifices the benefits of the BDI approach, namely flexibility and robustness of the agent behaviour.
2. Treating agents like humans who are in general robust although their behaviour cannot be guaranteed to be appropriate in all situations. Make sure the surrounding system has precautions built in for dealing with possible dire consequences caused by agents.
3. Searching for "good" tests at reasonable test cost. Provided with a mechanism to determine whether a test is good (e.g., whether it represents a challenging environment where the agent being tested is located) and a cost limit (e.g., a time limit), search based optimization algorithms (e.g., genetic algorithms) can be used to select as many good tests as possible from all possible ones within the cost limit.

[1] In counting cyclic paths, Winikoff and Cranefield apply *0-1* rule in order to avoid an impractical number of paths to be covered.

4. Supplementing testing with formal methods that can be used to verify the correctness of some aspects of the program without having to test possible executions involving these aspects. A promising formal method technique to verify agent programs is model checking, which can verify whether (sub) goals can be achieved without having to test possible agent executions to achieve these goals. However, formal methods are considered very expensive to apply to industrial-sized programs.

There are many testing techniques for MAS that address some of the difficulties mentioned above. They focus on different testing levels (i.e., unit, agent, integration and system) and testing stages (i.e., generating tests, executing tests and constructing oracles) [40, 73]. Some testing techniques for MAS use coverage criteria to guide test generation, such as Low et al.'s node and plan based criteria [60], Zhang et al.'s event and plan based criteria [117], Miller et al.'s protocol and plan based criteria [69], and Nguyen et al.'s ontology based criteria [75]. These techniques consider MAS models, however, they do not consider the possible difficulty of coverage based testing [68, 106, 108].

Some work uses fault-based criteria to guide or evaluate their testing techniques for MAS. For instance, Nguyen et al. [76] use standard syntactic mutation operators for Java in evolutionary testing, where the most effective tests — those that can detect the most simulated faults are selected to evolve. This technique attempts to find as many effective tests as possible at an acceptable cost. Zhang et al. [117] and Nguyen et al. [74, 77] use only 1 or 2 faulty agent programs to evaluate their respective testing techniques. Nguyen et al. [76] ask only 3 students to seed faults into an agent program in order to evaluate their testing technique.

However, little work shows the use of faults related to MAS models to guide or evaluate their testing techniques, although there is not much doubt that programmers will introduce a different class of faults when using a different programming model. Poutakidis et al. [90], Padgham et al. [86] and Winikoff [107] identify faults in agent programs and show that most of the faults are related to MAS model elements such as messages, rules and actions.

A few sets of mutation operators have been proposed to simulate faults related to MAS models: Adra and McMinn [7] propose a set of syntactic mutation operator classes for a generic MAS model. Saifan and Wahsheh [93] propose a set of syntactic mutation operators for JADE. Savarimuthu and Winikoff [95, 96] systematically derive respective sets of

syntactic mutation operators for AgentSpeak and GOAL, and evaluate the representativeness of their operators for GOAL by checking whether they can be used to generate faults in GOAL agent programs written by students.

1.3. Issues Addressed by this Thesis

This thesis addresses the following issues on MAS testing and engineering.

1. **There is a lack of techniques that can adequately evaluate the effectiveness of tests or testing techniques for MAS.** Most existing testing techniques for MAS are based on coverage criteria or evaluated by hand-seeded faults. These test evaluation methods, however, are not a good indicator of real fault detection ability. To address this, this thesis introduces mutation, a very effective test evaluation technique. In particular, mutation provides another potential solution to the issue that coverage based testing of BDI agent programs is harder than that of procedural programs [68, 106, 108].

   Declarative agent languages introduce a new class of faults, namely faults caused by misunderstandings of the opaque interpretation of agent programs. This thesis, for the first time, proposes semantic mutation for MAS in order to evaluate the ability of tests to detect such faults.

   Although there are several existing sets of mutation operators for MAS models, they lack evidence of effectiveness. This thesis evaluates the representativeness of the syntactic mutation operators for Jason, and the representativeness and power of the semantic mutation operators for Jason.
2. **There are no systematic approaches to evaluating the impact of possible semantic changes (changes in the interpretation of agent programs) on agents' behaviour and performance.** Recall from Sect.1.1.2 that determining a proper or optimal way to change the interpretation of agent programs is sometimes necessary/useful but difficult [59]. Existing techniques to guide the change of interpretation have the following limitations:

   - They focus on individual semantic aspects (especially plan/intention selection). However, different semantic aspects should be considered together, because

different applications may require changes in different semantic aspects, and different semantic aspects interrelate to determine the agent behaviour [35].

- They fail to consider the role of testing in changing the interpretation. Although some semantic changes are universally applicable or can be verified, some have to be adequately tested for specific applications.

This thesis introduces semantic mutation that can systematically evaluate possible changes in different semantic aspects on a test set.

1.4. Contributions and Structure of this Thesis

Chapter 2 reviews syntactic and semantic mutation. Chapter 3–5 answer the following three questions respectively.

1. **What should be mutated in Jason agent programs?** Chapter 3 systematically derives a set of syntactic mutation operators for Jason, then evaluates their representativeness by checking whether they can be used to generate faults in Jason agent programs written by students and experts.
2. **What should be mutated in the semantics of Jason, GOAL and 2APL (the interpretation of Jason, GOAL and 2APL programs)?** Chapter 4 derives respective sets of semantic mutation operators for Jason, GOAL and 2APL, then checks whether the operators for Jason represent real misunderstandings by asking students about the semantics involved.
3. **Is semantic mutation for Jason practically useful?** Chapter 5 first introduces a semantic mutation system for Jason, then applies it to Jason projects in order to check the ability of semantic mutation to evaluate tests, the robustness to (and the reliability of) semantic changes (changes in the interpretation of agent programs), and the impact of semantic changes on agents' performance.

Chapter 6 summarizes the contributions and suggests future work.

2. Literature Review

This chapter reviews two approaches to mutation, namely syntactic and semantic mutation on which the contributions of this thesis are based. It illustrates their use with examples, discusses their strengths and application difficulties, and introduces ways to derive syntactic and semantic mutation operators (rules that guide the change of a description and its interpretation). This chapter aims to help better understand the main chapters (Ch.3–5), why mutation for MAS is potentially useful and better than existing competitors such as coverage criteria for MAS.

2.1. Syntactic Mutation

2.1.1. Concepts of Syntactic Mutation

Syntactic mutation is a technique for evaluating test effectiveness. Traditionally, syntactic mutation generates slightly modified versions of a program and then checks whether a given test set can distinguish the original version from each modified version. Each modified program is called a *mutant*, which simulates a simple fault. Mutant generation is guided by a set of rules called *mutation operators*. For instance, Fig. 3(a) shows a piece of a program and Fig. 3 (b)–(f) show five of its mutants generated by applying a single mutation operator called *Relational Operator Replacement*, which replaces one of the relational operators (<, ≤, >, ≥, =, $\neq$) by one of the others.

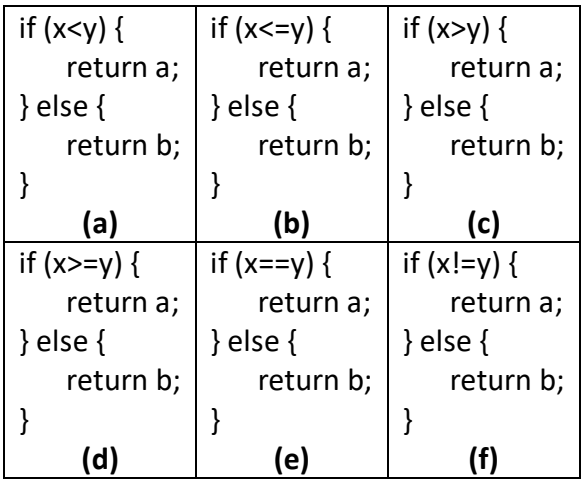


Fig. 3. An example of syntactic mutation

After generating mutants, all tests are executed against the original program and each mutant. If the behaviour of a mutant differs from that of the original program on some test, the mutant will be marked as *killed*, which indicates that the simulated fault can be detected by the test set. Therefore, the fault detection ability of a test set can be measured using *mutation score* – the ratio of the killed mutants to all generated mutants. The mutation score ranges from 0 to 1; the higher the score is, the more effective the test set is.

In the example shown in Fig. 3, a test set consisting of a single test in which the input is *x=3*, *y=5* cannot kill the mutants shown in Fig. 2(b) and 2(f) because on that test these two *live* mutants have the same behaviour as the original program (i.e., *return a*). Therefore, the mutation score is 3/5. According to this result, a test in which the input is *x=4*, *y=4* and another test in which the input is *x=4*, *y=3* can be created to kill these two mutants respectively and get a higher mutation score.

2.1.2. Strengths of Syntactic Mutation

One strength of mutation is that it is flexible. Different mutation operators can be used to target different classes of faults, e.g., faults belonging to different languages (Ada [83], C [8], Java [63], etc.), features (statements [8], inheritance [61], concurrency [21], etc.), testing levels (unit [61, 62], integration [26], system [65], etc.).

Another strength of mutation is that it is a very effective technique for evaluating test effectiveness, which can be illustrated in the following three aspects and by comparing with *coverage criteria*, a widely used class of test evaluation techniques that requires a set of

elements in a program model (e.g., control flow graph [66], UML diagram [67], input domain [66]) to be involved by the test set.

1. **Mutation can *subsume* most of the commonly used coverage criteria.** Criterion C subsumes criterion C' if any test set that satisfies C also satisfies C'. Offutt et al. [84] show that different mutation operators can be substituted for 6 different coverage criteria, namely statement coverage, decision coverage, condition coverage, decision/condition coverage, modified condition/decision coverage (MC/DC) [2] and multiple condition coverage. For instance, the *Logical Connecter Replacement* operator, which replaces one decision (e.g., the evaluation of an *if* statement) in the program by *TRUE* or *FALSE*, can be substituted for decision coverage, which requires that each decision in the program take every possible outcome.
2. **Mutant detection is strongly correlated with real fault detection.** Andrew et al. [11] apply the same test set to a real faulty program and mutants of the corrected version of this program, then find that the difference between the detection rate of real faults and the mutation score is only 1%. Just et al. [47] use a similar method to show that tests that can kill all common mutants can detect 73% of real faults.

   By contrast, Wei et al. [104] continuously generate and execute tests while measuring branch coverage, then find that over 50% of faults are detected in the period when coverage hardly changes. Inozemtseva and Holmes [44] generate a number of different test sets of the same size for the same program, then execute them while measuring statement coverage, branch coverage and multiple condition coverage. They find that the correlation between coverage and test effectiveness is not high: more and stronger forms of coverage do not provide greater insight into the ability of the tests to detect faults.
3. **Mutation is powerful enough to indicate weaknesses in testing.** Baker and Habli [12] show that tests that have been created for some modules in safety critical software and have satisfied statement criterion and MC/DC criterion cannot detect many faults

[2] The MC/DC criterion requires that each decision in the program take every possible outcome (Decision Coverage), and that each condition in a decision take every possible outcome. Consider an *if* statement that has two conditions, denoted by (*C1* || *C2*), (*true* || *true*), (*false* || *true*) and (*false* || *false*) provide MC/DC.

simulated by mutation. Li et al. [57] show that tests satisfying mutation criterion can detect 34%, 39% and 36% more faults than those satisfying edge-pair, all-uses and prime path coverage respectively. Chekam et al. [102] show that tests satisfying mutation criterion can detect 152% and 83% more faults than those satisfying statement and branch coverage respectively.

2.1.3. Difficulties in Applying Syntactic Mutation and Partial Solutions

One difficulty in applying mutation is the computational cost of executing (inc. compiling) a vast number of mutants, each is generated by applying a mutation operator to a relevant point in the program once (as Fig. 3 shows). Acree [9] estimates that the number of mutants is proportional to the square of the number of statements. Offutt and Pan [81] show that applying a standard set of 22 mutation operators produces 951 mutants for a program containing only 28 lines. Derezińska [27] shows that generating 11762 mutants for open source programs and executing them on 2689 tests take about 32 hours.

Many techniques have been proposed to reduce the number of mutants to be executed, while maintaining effectiveness in evaluating tests, more specifically, ensuring that the mutation score of a test set applied to the resultant mutants is approximately equal to that of the same test set applied to the original mutants [45]. The latest review of mutant reduction techniques is [99], from which it can be seen that these techniques can be classified as two classes: those for selecting a subset of mutation operators and those for selecting a subset of the generated mutants. Some representative techniques in each class are described below.

1. **Selecting a subset of mutation operators:** Offutt et al. [82] propose *Selective Mutation*, which selects only a subset of possible mutation operators to guide mutant generation. By analyzing a set of programs under test, they find that removing some mutation operators can significantly reduce the number of mutants while having little effect on the *mutation score* (see its definition in Sect.2.1.1), for instance, removing 6 of the mutation operators causes a 60% reduction in mutants but only a 0.3% reduction in mutation score.

To reduce manual effort in selecting mutation operators for particular programs, Banzi et al. [13] introduce multi-objective search techniques which selects a subset of mutation operators from possible ones in an automatic and smart way for satisfying multiple constraints such as mutation score, number of tests, number of mutants, number of faults detected, and running time. They find that the resultant set causes a significant reduction in mutants but only about 0.1% reduction in mutation score.

2. **Select a subset of the generated mutants:** Acree [9] proposes *Mutant Sampling*, which randomly chooses a subset of the generated mutants. This way is simple, fast and effective. Mathur and Wong [109] find that randomly choosing a certain percentage of mutants only slightly affects mutation score, for instance, randomly choosing 10% of mutants only causes a 2.4% reduction in mutation score.

   Hussain [43] proposes *Mutant Clustering*, which classifies the generated mutants into clusters using a classification algorithm then constructs a set by randomly choosing a small number of mutants from each cluster. This way ensures the diversity of the selected mutants. Hussain shows that the resultant set containing about 13% of all mutants causes less than 1% reduction in mutation score.

   Jiménez et al. [28] introduce a technique which first randomly selects a subset of the generated mutants (denoted by S), and then performs the following steps: (1) find hard-to-kill mutants from S, (2) randomly select some mutants from these hard-to-kill mutants, (3) apply the evolutionary algorithm to produce new mutants from these selected mutants, (4) add these new mutants into S, (5) go to step 1 until the termination condition is met, (6) construct a set that includes all hard-to-kill mutants previously found. Jiménez et al. find that their technique causes less reduction in mutation score than Mutant Sampling.

Note that besides reducing mutants, the mutation execution time can also be reduced by other means [45], such as executing mutants in parallel [97] and generating mutants from the intermediate code in order to avoid spending time on compiling them [63].

Another difficulty in applying mutation is the manual effort to exclude *equivalent mutants* whose generation is unpredictable. Equivalent mutants are those whose behaviour is no different from the original in any way that matters for the correctness of the program. Therefore, there exist no reasonable tests that can kill these mutants. Schuler and Zeller [98]

estimate that the percentage of equivalent mutants of a real-world program ranges from 25% to 70%. Obviously, such a large number of equivalent mutants will significantly affect the precision of mutation score, therefore, should be excluded as much as possible.

Traditionally, equivalent mutants are detected only by manual work, ranging from simply checking the mutation point in the program to tracing the complete execution path [88]. This way is generally effective, however, very costly: It is estimated that testers can judge equivalent mutants correctly 80% of the time by hand [9], however, they have to take an average of 15 minutes to judge a single one [98]. Recently many techniques have been proposed to reduce the manual effort. Kintis et al. [53] recently review these techniques and show that there are two main classes of them: those that can detect some equivalent mutants automatically and those that can suggest possible equivalent mutants. Some representative techniques in each class are described below:

1. **Detecting equivalent mutants automatically:** Offutt and Craft [79] use *compiler optimization techniques* to automatically detect a class of equivalent mutants of Fortran programs – those that can be thought as the program after optimization/deoptimization. For instance, the compiler can detect the dead code in a program, therefore, it can detect a class of equivalent mutants – mutants that differ from the original in the dead code. They find that 10% of all equivalent mutants can be detected by compiler optimization. Similarly, Papadakis et al. [87] use compiler optimization techniques to automatically detect equivalent mutants of C programs. They find that 30% of equivalent mutants can be detected.

   Bardin [14] find that static analysis (such as Value Analysis and Weakest Precondition calculus) can be used to find 95% of the infeasible paths in the program. Therefore, mutants that contain changes in infeasible paths can be identified as equivalent mutants.
2. **Suggesting possible equivalent mutants:** Grün et al. [32, 98] investigate what easily observable characteristics of mutants are strongly correlated with equivalence. They find that 75% of the mutants that do not change code coverage are equivalent, therefore, suggest that such a mutant has a 75% chance to be equivalent. Kintis and Malevris [52] find that almost all mutants that are the same in certain code fragments as the original are equivalent, therefore, suggest that such a mutant is probably equivalent.

Adamopoulos et al.'s work [6] is based on the idea that a mutant is probably equivalent if it cannot be killed by adequate and effective tests. In their technique, tests and mutants are co-evolved with the genetic algorithm (GA). On each generation of the GA, the ability of each test to kill mutants and that of each mutant to avoid being killed are re-measured using fitness functions. When the GA ends, mutants that are never killed by any test, are suggested as equivalent mutants.

2.1.4. Unconventional Forms of Syntactic Mutation

To address the limitations of syntactic mutation, new forms of syntactic mutation have been proposed. They differ from the traditional syntactic mutation in one or more of the following three aspects:

1. **Type of description to be mutated:** A software development process involves various descriptions, ranging from requirement specifications to program source code. Syntactic mutation is traditionally based on programs, but has recently been applied to other specifications such as Finite State Machine [29, 36] and Statechart [48, 103]. This new form is called *specification-based mutation*, which requires tests to kill all mutants of a specification (that can be simulated, executed or formally reasoned about) before being applied to the program [23]. Specification-based mutation is used to evaluate specification-based testing, which attempts to detect faults in the program that are derived from faults in the specification or from misinterpreting the specification. Some of these faults are difficult to detect by testing the program alone. For instance, if the implementation neglects an aspect of the specification, it is possible that there is no evidence in the code itself for this omission.
2. **Order of mutants:** Traditionally, syntactic mutation creates a *first order mutant* by applying a single mutation operator only once, in order to simulate a simple fault. Jia and Harman [46] indicate that many first order mutants simulate faults that are very easy to detect, therefore, weak in evaluating tests.

   To resolve this issue, Jia and Harman [46] replace first order mutants by *valuable higher order mutants*. A higher order mutant is created by applying the same or different mutation operator(s) more than once in order to simulate a complex fault – a

combination of simple faults. A higher order mutant is *valuable* if it is harder to detect than any of its constituent first order mutants. Since there are a vast number of possible combinations of first order mutants, optimization algorithms can be used to select as many valuable ones as possible in a smart way at a reasonable cost [46, 99]. Omar et al. [85] compare the effectiveness of 8 optimization algorithms in finding valuable higher order mutants, and then find that some of these algorithms outperform the others and have their own advantages in the probability/number of the valuable higher order mutants found.

Moreover, Jia and Harman's technique can reduce the cost of mutation. First, it can reduce the number of mutants to be executed. Second, it can reduce the number of equivalent mutants: empirical studies [64, 78] show that the number of equivalent mutants among the generated second order mutants is much less than the number of equivalent mutants among the first order mutants that form the second order mutants.

3. **Process of Killing Mutants:** Traditionally, syntactic mutation requires the original program and each mutant to be executed to completion, in order to compare their final output. Howden [41] proposes the concept of *weak mutation*, and classifies traditional mutation as *strong mutation*. Weak mutation is based on the belief that in most cases a mutant will reach the *anomalous* state as soon as its mutated part is executed, therefore, it chooses only to execute the program and each mutant until the completion of the mutated part, and then compares their immediate state.

   Weak mutation requires less execution time than strong mutation because it does not require the anomalous state to continue propagation until affecting the final output. However, in a few cases the anomalous state may be delayed, therefore, may not be detected upon the completion of the mutated part. Offutt and Lee [80] empirically show that weak mutation causes a very small reduction in mutation score and a significant reduction in mutation time.

   To increase the chance of capturing the delayed anomalous state, Woodward and Halewood [110] propose *firm mutation*, which allows the "compare state" to lie between the execution of the mutated part and the final output of the program. Reales et al. [65] propose *flexible weak mutation* for system level testing, which executes a mutant while regularly comparing its current state to the state of the original program in

the same period, until finding the anomalous state (indicating that the mutant is killed) or the end of the execution (indicating that the mutant is live).

2.1.5. Deriving Syntactic Mutation Operators

The first step of introducing mutation into a new language is to figure out *what to mutate*, more specifically, to derive a set of mutation operators. Three existing derivation methods are described below:

1. **Informal derivation from language constructs/features**: Many designers of mutation operators do not specify their methods for analyzing the language constructs/features. For instance, Agrawal et al. [8] simply mention that their operators for C are the result of *long deliberations amongst the authors in which several aspects of the C language and program development in C were examined*. Therefore, it is reasonable to assume that some sets of operators are informally derived. This method is easy to use.
2. **Derivation from studies of errors/faults/mistakes**: Mutation operators can also be derived by observing what mutants are intuitively related to errors/faults/mistakes. For instance, mutation operators for Fortran have been derived from *studies of programmers' errors and correspond to simple errors that programmers typically make* [51]. Mutation operators for UML class diagrams have been derived from error types in a UML based model [30]. This method can provide some confidence in the representativeness of the mutation operators. However, more confidence must be provided by comparing the detection ratios of mutants and real faults (see point 2 in Sect.2.1.2 for some examples), because a real complex fault may be easier/harder to detect than its constituent simple faults.
3. **Systematic derivation from language constructs/features**: Kim et al. [50] indicate that Method 1 or 2 may miss some potential mutation operators due to insufficient consideration or incomplete categories of errors/faults/mistakes. To derive a relatively complete operator set, they propose a systematic method, where an operator is generated by applying one of the *guide words* (i.e., *none*, *more*, *less*, *as well as*, *part of*, *reverse*, *other than*, *narrowing*, *widening* or *equivalent*) into one of the language constructs/features. For instance, applying *widening* into the *accessibility* feature

produces an operator that can change an access modifier from *protected* to *public*, or from *private* to *public*. Note that not every combination of guide word and construct/feature produces a sensible operator (e.g., in the above example, applying *reverse* into the same feature does not make sense). Therefore, judgements (based on experience, previous similar work, etc.) should be made.

To evaluate this method, Kim et al. [50] use it to derive a set of mutation operators for some constructs in Java and then compare it with four existing sets of mutation operators for the same constructs in other four languages. They show that (1) their set can cover the other four sets, (2) their set includes operators that the other sets do not include, and (3) the other four sets are not equal, which indicates that method 1 or 2 easily leads to incomplete operator sets. In addition, Kim et al. show that their operator set can cover four existing fault taxonomies.

2.2. Semantic Mutation

2.2.1. Concepts of Semantic Mutation

Clark et al. [23] extend the definition of mutation by introducing semantic mutation: suppose N represents a description (a program or specification) and L represents the semantics of the language in which the description is written, the pair (N, L) determines the behaviour. Syntactic mutation generates modified versions of the description namely $N \rightarrow (N_1, N_2, ..., N_k)$ while semantic mutation generates variants of the semantics namely $L \rightarrow (L_1, L_2, ..., L_k)$.

For semantic mutation, $L_1, L_2, ..., L_k$ represent *semantic mutants*, each of which simulates a misunderstanding of how the description is interpreted (and, therefore, the resultant fault). Generation of semantic mutants is guided by a set of rules called *semantic mutation operators*. For instance, Fig.4 shows a semantic mutant generated by applying a single semantic mutation operator that causes the *if* keyword to be used for mutual exclusion (i.e., when an *if* statement is directly followed by another *if* statement, the second *if* is interpreted the same as an *else-if*).

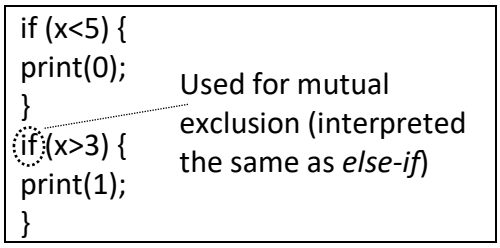


Fig. 4. An example of semantic mutation

After generating mutants, semantic mutation proceeds in a similar way as syntactic mutation, namely 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 Fig.4, 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 live mutant results in the same behaviour as the original interpretation (i.e., only print *0*). Therefore, the mutation score is 0/1 = 0. A test in which the input is *x=4* can be created to kill this mutant (The program will print both *0* and *1* under the original interpretation while only printing *0* under this mutant).

Semantic mutation can be used to evaluate not only test effectiveness, but also the robustness to and the reliability of semantic changes. Robustness and reliability in the context of semantic mutation are defined below:

1. The robustness of a program to a semantic change is the ability of the program to behave as expected under the changed semantics. A program is robust to a semantic change if the change cannot be detected by a *trusted* test set – a test set containing all necessary tests for ensuring the expected behaviour.
2. The reliability of a semantic change for a program is the ability of the semantic change to avoid causing unexpected behaviour of the program. Note that reliability is evaluated in the same way as robustness: a semantic change is reliable for a program if the change cannot be detected by a trusted test set; reliability is used to evaluate semantic changes, while robustness used to evaluate the program.

In evaluating the robustness to or the reliability of semantic changes, an *equivalent* mutant is one that cannot be killed by a trusted test set, therefore, it represents robustness to a semantic change or a reliable semantic change. It is worth noting that equivalent

mutants in evaluating test effectiveness and those in evaluating robustness/reliability are different in the following aspects:

1. Role in evaluation: In evaluating test effectiveness, an equivalent mutant does not represent a reasonable fault. Instead, it represents an equivalent version of the original description. Therefore, it has to be excluded. While in evaluating robustness/reliability, an equivalent mutant is a useful indicator of the robustness to or the reliability of the semantic change.
2. Condition of equivalence: In evaluating test effectiveness, a mutant is equivalent if it cannot be killed by any reasonable test that matters for the correctness of the description. While in evaluating robustness/reliability, a mutant is equivalent if only it cannot be killed by a trusted test set, which does not have to include any reasonable test, instead is allowed not to cover some aspects, or can tolerant some errors, according to particular needs.

It is sometimes useful to make a program robust in a different interpretation with/without the need of maintaining its robustness in the original interpretation. For instance, when teaching programming in general, one unavoidably needs to use a specific language, but it is good if he/she can use the subset of that language that has most in common with other rival languages. When developing a programming language, it is useful to investigate what a program will be or whether it has better performance in a different interpretation. To make the program in Fig. 4 robust in both the original and the changed interpretations, the conditions of the *if* statements can be modified to ensure that at most one branch can be executed.

2.2.2. Strengths of Semantic Mutation

Although semantic changes can usually be simulated by syntactic changes (e.g., in Fig. 4, the second *if* can be changed to *else if*), semantic mutation has the following advantages over syntactic mutation:

1. Semantic mutation can make semantic changes easier than modifying the description, if there is a convenient way (e.g., meta-interpreters [56, 105]) to change the interpretation (see Winikoff's example of using a meta-interpreter in Sect.1.1.2).
2. In contrast to first order syntactic mutation, first of all, semantic mutation has potential to simulate faults that are harder to detect. Clark et al. [23] provide an example in which a statechart (a popular graphical notation for specifying state-based systems) for a cruise-control system can be correctly interpreted under the semantics of *STATEMATE*. By contrast, *UML* and *Stateflow* will give the same statechart slightly wrong interpretations that cannot be detected by testing individual transitions.

   Second, semantic mutation has potential to complement first order syntactic mutation. Clark et al. [23] show that given a set of semantic mutants and a set of similar syntactic mutants, they are not subsumed by each other, that is to say, they simulate different classes of faults.

   Third, semantic mutation normally produces far fewer mutants, therefore, far fewer equivalent mutants. This is because a semantic mutation operator normally only generates a single mutant that contains a single change in the semantics of the description language[3] (as Fig. 4 shows). By contrast, a syntactic mutation operator normally generates many mutants, each contains a single change in a single relevant point in the description (as Fig. 3 shows). From another perspective, this is because a semantic mutant normally simulates a complex fault, therefore, it can be seen as a combination of first order syntactic mutants.

2.2.3. Deriving Semantic Mutation Operators

Clark et al. [23] indicate that in software development, a description N may be transformed into another description N' at the same or lower abstraction level, denoted by N → N'. Typical examples include imaginary description → actual description, specification → program, program in C → program in Java, and program → machine code. They also show

[3] This rule can be relaxed, namely allowing a semantic mutation operator to generate two or more mutants, each of which contains a change in the semantics of a part of the program. This is useful in some cases. For instance, when the program is developed by several persons, semantic mutation can simulate misunderstandings of a single person.

that semantic mutation is useful in the following scenarios which involve the above transformations and from which semantic mutation operators can be derived.

1. **Common misunderstandings**: It is not surprising that a particular group of people (e.g., novice or experienced programmers) have a common set of misunderstandings for a particular language. Semantic mutation can be used to simulate these misunderstandings.
2. **Refinement**: Refinement usually occurs in a transformation from a high level abstraction to a low level one. For instance, generic types specified in UML entities need to be refined to specific types accepted by programming languages (e.g., Age need be refined to Integer or Float). Semantic mutation can be used to explore possible refinements.
3. **Migration**: It is sometimes necessary to migrate a description notation to a different one (e.g., migration from C to Java). The original language and the new one may encapsulate different semantics. If this is the case, there is a danger of mistakes caused by this difference in semantics. Semantic mutation can be used to simulate these mistakes.
4. **Porting of code**: Many programming languages have elements of their semantics that are implementation specific. For instance, C compilers have freedom to decide the evaluation order within an expression. Semantic mutation can be used to explore the freedom of interpretation.

From the scenarios proposed by Clark et al. [23], I obtain three sources from which semantic mutation operators can be derived:

1. **Studies of faults/errors/mistakes**: Some possible misunderstandings can be identified in faults/errors/mistakes. For instance, if observing that one fails to put a *break* statement at the end of a *case* in a *switch* block, it is possible that he/she have the misunderstanding that the *case* label is used for mutual exclusion. An operator can be designed to simulate this misunderstanding.
2. **Semantic differences between similar constructs**: both descriptions involved in a transformation may have similar constructs whose semantics are however different. Simply copying such a construct from one description to the other may be a danger. For instance, copying a C code snippet which contains the end-of-string character (“\0”) to a

Java program may introduce faults because this character is not used to end Java strings. To capture this danger, an operator can be designed to allow the use of this character in Java.

3. **Freedom of interpretation**: some constructs can be interpreted in different ways. For instance, an unbounded type can be retrenched to different bounded types. To explore the impact of different retrenchments an operator can be designed to represent them.

2.3. Summary

This chapter reviews the following two mutation approaches on which the contributions of this thesis are based:

1. Syntactic mutation: Syntactic mutation generates syntactic mutants each of which contains a fault in a description (usually a program), then it evaluates a test set by checking whether it can distinguish the original description from each mutant. Syntactic mutation can be easily adapted to different descriptions, and is more effective in evaluating tests than coverage criteria, a widely used test evaluation technique.

   Since the high cost of executing numerous mutants and excluding equivalent mutants hinder the application of syntactic mutation into practice, various techniques and variant forms of mutation have been proposed to overcome these difficulties. The latest study [70] on the applicability of syntactic mutation shows that when appropriate strategies such as selective mutation operators are used, higher order mutants and multi-threading, syntactic mutation is applicable to industry-sized applications.

   Despite increasing evidence of the potential of syntactic mutation, only limited work has been done to introduce syntactic mutation to MAS, which is a promising paradigm for building autonomous systems but which is hard to test. To address this limitation, Chapter 3 proposes a set of mutation operators for Jason, a popular language for programming cognitive agents, and then it checks how these operators can be used to generate real faults.
2. Semantic mutation: Semantic mutation generates variant interpretations of a description for two uses: (1) to evaluate test effectiveness (the ability of a test set to detect faults) by simulating faults caused by misunderstandings of how the description is

interpreted; (2) to evaluate the robustness of the description to semantic changes or the reliability of semantic changes for the description. Although semantic mutation is a new technique, it has shown potential in simulating hard-to-detect faults and exploring the impact of potential semantic changes on the behaviour of the description.

This thesis, for the first time, introduces semantic mutation into MAS. I believe that semantic mutation is particularly useful in cognitive agent languages because first of all, these languages introduce a new class of faults, namely those caused by misunderstandings of the encapsulated agent control; second, these languages suggest that agent programs should be compact and should focus on the specification of cognitive elements (e.g., beliefs, goals, plans), therefore, there may be more need of variations to semantics in order to meet all desired requirements. Chapter 4 proposes some scenarios where semantic mutation for three cognitive agent languages is useful, derives semantic mutation operator sets for these languages, then checks whether one of the operator sets can simulate real misunderstandings. Chapter 5 evaluates the use of semantic mutation to evaluate tests and the impact of semantic changes.

3. Syntactic Mutation Operators for Jason

Recall from Sect. 2.1.5 that syntactic mutation operators – rules that make simple changes to a program in order to simulate possible faults, can be derived using the following three approaches:

1. **Informal analysis of language constructs/features**. This approach is easy to use, however, it may miss some potential mutation operators due to insufficient consideration.
2. **Studies of faults/errors/mistakes**. This approach can provide some evidence of the representativeness of the mutation operators (the ability of the mutation operators to simulate real faults). However, it may miss some potential mutation operators due to insufficient categories of faults/errors/mistakes.
3. **Systematic analysis of language constructs/features**. This approach can generate a relatively complete set of mutation operators. However, much effort is required to judge whether each guide word application produces a sensible operator.

Savarimuthu and Winikoff [96] combine approach 2 and 3 in the hope of combining their advantages. They first derive mutation operators systematically from the syntax of GOAL and then check how their operators can be used to generate real faults. I use their combined approach to derive syntactic mutation operators for Jason.

3.1. An Introduction to Jason

Recall from Sect. 1.1.1 that a BDI agent operates within an architecture by means of a reasoning cycle that has the following general steps: (1) searching for appropriate plans according to the agent's cognitive state formed by beliefs, goals, etc., (2) adding (some of) these plans as new intentions, and (3) executing the actions in the uncompleted intentions. Jason is a language developed by Hübner and Bordini for programming BDI multi-agent systems (MAS) [20]. It provides an interpreter that can create a customizable architecture

for each agent, and it allows a MAS to be specified in a Jason project using the following three types of files:

1. **Agent programs**: Each agent program specifies an agent's initial beliefs, goals and plans in logical form in order to initialize the agent's architecture.
2. **Java files**: They can be used to customize an agent's architecture, implement the environment and extend an agent's capability.
3. **A configuration file**: It specifies how to construct a MAS from agent programs and Java files and how to execute it.

This chapter focuses on mutating agent programs in a Jason project because they play a central role in determining MAS behaviour. Mutation for the other types of files is left to future work.

To show how Jason agent programs work, a Jason agent program for a gold searching agent is shown in Fig.5 and interpreted below. The details of Jason can be found in [20].

```
//initial belief
capacity(100).

//initial goal
!search_gold.

//p1
+!search_gold : capacity(N) & N=0.

//p2
+!search_gold : gold_piece <- pick; ?capacity(N); -+capacity(N-1); !search_gold.

//p3
+!search_gold : true <- move; !search_gold.
```

Fig. 5. A typical Jason agent program

An agent initialized with the above program will start with an initial belief (*capacity(100)*), an initial goal (*!search_gold*) and three plans (denoted by *p1*, *p2* and *p3* respectively). Assuming there is no gold pieces at the starting position of the agent (in the environment whose specification is not discussed here), the agent will first select *p3* for execution because it is the only *applicable* plan – the plan whose trigger (*+!search_gold*) and context (*true*) are satisfied. *p3* has two actions: *move* is defined in the environment file to let the agent move in the environment; *!search_gold* is to repost the initial goal before *p3* is completed (By default, a Jason goal will be dropped as soon as a plan for handling it is completed) so that the agent can continue searching for gold pieces.

When the agent finds a gold piece, therefore, the corresponding belief *gold_piece* is added (how to map environmental changes to belief changes is specified in the environment file), *p2* and *p3* will become applicable simultaneously. However, by default only the applicable plan that appears first in the program text, namely *p2* is selected for execution. *p2* has four actions: *pick* is defined in the environment file to let the agent collect the gold piece found in the environment; *?capacity(N)* and *-+capacity(N-1)* are to query and then update the belief *capacity(N)* representing that the agent can still collect a maximum number of *N* gold pieces; *!search_gold* is to let the agent keep searching. When the agent holds the belief *capacity(0)*, indicating that it cannot collect gold pieces any more, *p1* will become applicable and be selected for execution (regardless of whether *p2* or *p3* becomes applicable as well because neither appears before *p1* in the program text). Since *p1*'s body has no actions (to repost *!search_gold*), the agent will stop searching and become idle when *p1* is completed.

The structure of Jason agent programs is defined by Jason's syntax, which is based on the syntax of *AgentSpeak* [91] and has been continuously improved and extended in order to enable and facilitate the implementation of practical agents. This chapter focuses on deriving mutation operators from the syntax explicitly defined in the Jason textbook [20], plus the syntax that I create from the standard definitions of the Jason communication actions. The chosen syntax defines constructs that are commonly used in Jason agent programs and that are fairly typical compared with those defined by other cognitive agent languages (e.g., GOAL, 2APL), therefore, the derived mutation operators should be widely applicable. Mutation operators for the constructs not defined by the chosen syntax (e.g., directives, if/for/while statements) are left to future work.

Fig.6 shows the chosen syntax, expressed using Extended Backus-Naur Form (EBNF) rules. These rules are copied from the Jason textbook except rule 10–14 (marked with a grey background) which are extensions of rule 9 for defining communication actions. For better readability, I divided these rules into two parts denoted by Part(a) and Part(b) respectively. Part(a) specifies the syntactic entities involved in high-level agent control, while Part(b) specifies their constituents.

Part (a)

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

Part (b)

15. literal → **[** "~" **]** atomic_formula
16. atomic_formula → **(<ATOM> | <VAR>) [** "(" term **(** "," term **)*** ")" **] [** "[" term **(** "," term **)*** "]" **]**
17. term → literal **|** list **|** arithm_expr **| <VAR> | <STRING>**
18. list → "[" **[** term **(** "," term **)* [** "|" **(** list **| <VAR>)]]** "]"
19. log_expr → simple_log_expr **|** "not" log_expr **|** log_expr "&" log_expr **|** log_expr "|" log_expr **|** "(" log_expr ")"
20. simple_log_expr → **(** literal **|** rel_expr **| <VAR>)**
21. rel_expr → rel_term **[(** "<" **|** "<=" **|** ">" **|** ">=" **|** "==" **|** "\=" **|** "=" **|** "=.." **)** rel_term **]+**
22. rel_term → literal **|** arithm_expr
23. arithm_expr → arithm_term **[(** "+" **|** "-" **|** "*" **|** "/" **|** "div" **|** "mod" **)** arithm_term **]***
24. arithm_term → **<NUMBER> | <VAR> |** "-"arithm_term **|** "(" arithm_expr ")"

Fig. 6. The syntax from which syntactic mutation operators for Jason are derived

3.2. Deriving Syntactic Mutation Operators for Jason

Recall from Sect.2.1.5 that a syntactic mutation operator can be derived by applying a guide word to a language construct/feature. Savarimuthu and Winikoff [95, 96] propose five guide words that can be selected from for deriving operators for AgentSpeak and GOAL, namely *none*, *as well as*, *part of*, *reverse* and *other than*. I believe these guide words are also applicable to the syntax of Jason, because the syntax of Jason is based on that of AgentSpeak and similar to that of GOAL.

By analyzing how Savarimuthu and Winikoff use these guide words, I observe the following:

- Applying *none* or *part of* actually generates an operator that *deletes* an element (e.g., a plan, a plan action);
- Applying *as well as* actually generates an operator that *adds* an element (e.g., a negation operator);
- Applying *reverse* actually generates an operator that *reorders* a set of elements (e.g., plans in an agent program, two adjacent actions in a plan);
- Applying *other than* actually generates an operator that *replaces* an element (e.g., the trigger type of a plan) by another.

It can be seen from these observations that applying *add*, *delete*, *reorder* and *replace*, referred as *keywords*, is a more direct way to derive mutation operators for BDI-like agent languages than applying guide words. Table 1 summarizes these keywords.

Table 1. Keywords used for deriving syntactic mutation operators for Jason

Keyword	Interpretation	Equivalent guide word(s)
add	Add an element	as well as
delete	Delete an element	none, part of
reorder	Change the order of elements	reverse
replace	Replace an element by another	other than

As seen in Fig.6, the syntactic elements of Jason are defined in a top-down way: if there is a EBNF rule R1 specifying that an element E includes another element E'; below this rule,

there may be another rule R2 specifying the constituents of E'. Since I will go through all EBNF rules from top to bottom, it makes sense that when I attempt to apply keywords to R1, I will treat E' as an atom, which means I will not further check its constituents until checking R2. Sometimes a keyword is not applied to R1 but is applied to R2, it is not a contradiction because E and E' usually have different direct constituents arranged in different ways.

The process of deriving mutation operators by applying these keywords to the EBNF rules in Fig.6 are described below.

EBNF rule 1 states that an agent program can include beliefs, goals and plans. By applying the *delete* keyword, I derive the following three mutation operators:

1. **Belief Deletion (BD)**: A single belief in an agent program is deleted.
2. **Goal Deletion (GD)**: A single goal in an agent program is deleted.
3. **Plan Deletion (PD)**: A single plan in an agent program is deleted.

Applying the *reorder* keyword can produce mutation operators that reorder beliefs, goals and plans respectively. Note that if all beliefs/plans are reordered, the following two issues may be raised:

- Numerous mutants will be generated because there are numerous ways to arrange all beliefs/plans (Savarimuthu and Winikoff [96] also note this when deriving a mutation operator for reordering GOAL rules). This makes mutation computationally expensive.
- Many equivalent mutants will be generated, because the agent behaviour is independent of the order of some beliefs/plans. Detecting these equivalent mutants requires much manual effort.

To resolve the above issues, I choose to reorder only *relevant* beliefs/plans – those whose order is very likely to affect the agent behaviour. Relevant beliefs are those that can match the same condition or query in a plan (e.g., *bel(0)* and *bel(1)* can both match *bel(N)* in the context of a plan); relevant plans are those that can become applicable simultaneously. Static program analysis can be used to find (at least some of) relevant beliefs/plans, as Winikoff [107] indicates.

Three mutation operators derived from applying the *reorder* keyword to EBNF rule 1 are

listed below.

4. **Belief Order Change (BOC)**: The order of relevant beliefs in an agent program is changed.
5. **Goal Order Change (GOC)**: The order of goals in an agent program is changed.
6. **Plan Order Change (POC)**: The order of relevant plans in an agent program is changed.

Note that GOC is not required to reorder only *relevant* goals because first of all, usually there is only one or several initial goals in an agent program, in which case the issues of numerous mutants and numerous equivalent mutants will not exist. Second, it is difficult to determine initial goals whose order is not very likely to affect the agent behaviour by analyzing the program alone.

Applying the *replace* keyword can produce a mutation operator that changes a belief to a goal with the same literal or vice versa (literal. → !literal. or !literal. → literal.). Note that if a belief is changed to a goal but there are no plans that can handle this goal, the Jason interpreter will throw a warning/error message that is easy to detect, which means the resultant mutant is not powerful to evaluate tests. To attempt to reduce the number of mutants that cause this message to be thrown, static program analysis can be introduced to check whether the resultant goal can match the trigger of some plan. Therefore, I derive the following mutation operator:

7. **Initial Element Type Replacement (IETR)**: A belief is changed to a goal with the same literal (literal. → !literal.) if the resultant goal can match the trigger of some plan; a goal is changed to a belief with the same literal (!literal. → literal.).

Applying the *add* keyword to EBNF rule 1 will produce mutation operators that have the issue of what belief/goal/plan to add. To resolve this issue, these operators require knowledge about the problem domain (e.g., about possible beliefs held by the agent, possible goals to be achieved), therefore, are not very applicable.

EBNF rule 2 states that a belief can be a literal or a rule that enables a literal to be concluded when some conditions get satisfied. The introduction of rules enables agents to do *theoretical reasoning* [111]. By applying the *delete* keyword, I derive the following mutation operator:

8. **Rule Condition Deletion (RCD)**: The conditions of a rule are deleted.

Note that before applying RCD to a rule, the conclusion of the rule is "conditional"; after applying RCD, the conclusion part becomes a literal independent of the (now deleted) conditions, namely "unconditional". Therefore, if the agent behaviour is influenced by the conclusion, it will probably be different before and after applying RCD.

It is not difficult to see that applying the *add* or *replace* keyword to EBNF rule 2 is not very applicable because it will raise the issue of what logical expression to add to a literal belief or what to replace the literal/logical expression of a belief with. Applying *reorder* to EBNF rule 2 does not make sense because changing the order of the literal and the logical expression in a belief is not allowed by Jason's syntax.

EBNF rule 3 defines a goal as a literal following "!". It is not difficult to see that applying the *add* or *reorder* keyword does not make sense because it will make the program syntactically incorrect. Applying the *delete* or *replace* keyword can produce a mutation operator that changes a goal to a belief with the same literal (!literal. → literal.). However, this operator is not sensible, because it can be *subsumed* by IETR (namely all mutants it generates can be generated by IETR).

EBNF rule 4 states that a plan includes a triggering event and can include a label, a context and a body. The label is the name of a plan, which is used to refer to the plan elsewhere. It can also include meta-level information about the plan, which can be used for practical purposes such as customizing the agent's control or implementing extensions. By applying the *delete* keyword, I derive the following three mutation operators:

9. **Plan Label Deletion (PLD)**: The label of a plan is deleted if specified.
10. **Plan Context Deletion (PCD)**: The context of a plan is deleted if specified but not as *true*[4].
11. **Plan Body Deletion (PBD)**: The body of a plan is deleted if specified but not as *true*.

[4] EBNF rule 7 and 8 allow one to specify the context or body of a plan as *true*, which is equivalent to not specifying it at all.

It is worth noting that PBD is similar, but inequivalent to PD, because a plan whose body has been deleted may still have an effect on the agent behaviour: One possible reason is that its context may include internal actions that may change the agent's cognitive state; another possible reason is that it is still meant to handle some event, which otherwise will be handled by other or no plans (by default, the Jason interpreter will generate a warning/error message if an event cannot be handled by any plan).

It is not difficult to see that applying the *add* or *replace* keyword to EBNF rule 4 is not very applicable because it will raise the issue of what label/context/body to add if it is not specified, or what to replace the label/trigger/context/body of a plan with. Applying *reorder* keyword to EBNF rule 4 does not make sense because changing the order of the components of a plan defined by this rule is not allowed by Jason's syntax.

EBNF rule 5 defines a plan label as an atomic formula following "@". It is not difficult to see that applying any keyword to this rule does not make sense because it will make the program syntactically incorrect.

EBNF rule 6 states that the triggering event of a plan belongs to one of six types: belief addition (+literal), belief deletion (-literal), achievement goal addition (+!literal), achievement goal deletion (-!literal), test goal addition (+?literal) and test goal deletion (-?literal). By applying the *replace* keyword, I derive the following mutation operator.

12. **Triggering Event Type Replacement (TETR)**: The type of a triggering event (+, -, +!, -!, +?, -?) is replaced by another.

Applying the *add* or *delete* keyword to EBNF rule 6 can produce a mutation operator that adds/deletes a goal type (! or ?) into/from a triggering event type (e.g., + → +!, -? → -). This operator, however, is not sensible because it can be subsumed by TETR. As to applying the *reorder* keyword, it does not make sense because changing the order of the components of a triggering event defined by EBNF rule 6 is not allowed by Jason's syntax.

EBNF rule 7 defines the context of a plan as a logical expression (see EBNF rule 19) or *true*. It is not difficult to see that applying the *add*, *delete* or *reorder* keyword to this rule does not make sense because it is not allowed by Jason's syntax. Applying the *replace* keyword can produce a mutation operator that replaces a logical expression by *true* or vice versa. However, replacing a logical expression by *true* is equivalent to not specifying the

context at all, which has been simulated by PCD; replacing *true* by a logical expression will raise the issue of what to replace *true* with.

EBNF rule 8 states that the body of a plan can be a sequence of formulae or *true*. By applying the *delete* and *reorder* keywords, I derive the following two mutation operators:

13. **Body Formula Deletion (BFD)**: A single body formula is deleted.
14. **Body Formulae Swap (BFS)**: Two adjacent body formulae are swapped.

Note that BFS does not aim to change the order of all body formulae in a plan, because it will produce a large number of mutants, as Savarimuthu and Winikoff [96] indicate.

Applying the *add* or *replace* keyword to EBNF rule 8 will produce mutation operators that have the issue of what body formula to add or what to replace a body formula with, therefore, are not very applicable. Applying the *replace* keyword can produce another operator that replaces a formula sequence by *true* or vice versa. However, replacing a formula sequence by *true* is equivalent to not specifying the plan body at all, which has been simulated by PBD; replacing *true* by a formula sequence will raise the issue of what to replace *true* with.

EBNF rule 9 states that body formulae are classified into 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. By applying the *replace* keyword, I derive the following mutation operator:

15. **Body Formula Type Replacement (BFTR)**: The type of a body formula for changing cognitive state (!, !!, ?, +, -, -+) is changed to another. A mental note or test goal is changed to an achievement goal with the same literal only when the resultant goal can match the trigger of some plan.

Two things are worth noting here. First, similar to IETR, BFTR introduces static program analysis in the hope of reducing mutants in which the resultant body formulae will post achievement goals that cannot be handled by any plan. Second, BFTR does not involve three types of body formulae, namely action, internal action and relational expression, because the resultant formula will probably have one of the following consequences:

- It has no effects on the agent behaviour, therefore, applying BFTR is equivalent to applying BFD. A change that probably results in this consequence is one from an internal action (.atomic_formula) to a mental note[5] (+atomic_formula).
- It cannot be processed, therefore, causes errors that are easy to find. A change that probably results in this consequence is one from a mental note (+atomic_formula) to an internal action (.atomic_formula).

The difference between formula "!literal" and "!!literal", which are both used to post a goal, is noteworthy. The former prevents the plan (*p*) where the formula is continuing execution until the posted goal (*g*) is achieved, while the later allows *p* to run alongside the plan for achieving *g*.

It is not difficult to see that applying the *add*, *delete* or *reorder* keyword to EBNF rule 9 does not make sense because it will make the program syntactically incorrect.

EBNF rule 10 defines two internal actions used to send messages to other agents, namely *.send* and *.broadcast*. By applying the *delete* keyword, I derive the following mutation operator:

16. **Message Timeout Deletion (MTD)**: The message timeout specified in a communication action is deleted.

Note that I do not apply the *delete* keyword to the *reply* parameter (which will be associated with the response to the message), because deleting it will probably cause the failure of the following formulae that use it, which is easy to detect. It is not difficult to see that applying the *add* keyword to EBNF rule 10 is not very applicable because it will raise the issue of what timeout to add if it is not specified. Applying the *reorder* keyword does not make sense because changing the order of the parameters in a communication action is not allowed by Jason's syntax. As to applying the *replace* keyword, it can produce a mutation operator that replaces *.send* by *.broadcast* or vice versa. However, I find that this mutation

[5] EBNF rule 15 states that a literal is an atomic formula or its negation. Therefore, an atomic formula following +, -, -+, !, !! or ? is a valid mental note.

operator can be subsumed by the below derived operator which aims to change the range of message recipients.

Rule 11 states that a main parameter in message sending actions – *receiver*, can include one or more recipient agent IDs. By applying the *replace* keyword, I derive the following mutation operator:

17. **Recipient Range Change (RRC)**: The range of the recipients in a *.send* action is changed. Each *.broadcast* action will be transformed into the equivalent *.send* action so that this mutation operator can be applied to it.

Note that applying the *replace* keyword to EBNF rule 11 is equivalent to applying the *add* or/and *delete* keywords once or multiple times. It is not difficult to see that applying the *reorder* keyword does not make sense because changing the order of recipients in a message sending action will not affect the behaviour of the recipients.

Rule 12 defines another main parameter in message sending actions – *illocutionary force*, which represents the intent of sending the message. By applying the *replace* keyword, I derive the following mutation operator:

18. **Illocutionary Force Replacement (IFR)**: The illocutionary force (tell, untell, achieve, unachieve, askOne, askAll, tellHow, untellHow, askHow) in a message sending action is replaced by another.

It is not difficult to see that applying the *add*, *delete* or *reorder* keyword to EBNF rule 12 does not make sense because it is not allowed by Jason's syntax.

Rule 13 states that a third main parameter in message sending actions – *propositional content*, can include one or more propositions that will be sent to the recipient(s). By applying the *delete* keyword, I derive the following mutation operator:

19. **Proposition Deletion (PD2)**: A single proposition in the propositional content is deleted.

Applying the *add* or *replace* keyword to EBNF rule 13 is not very applicable because it will raise the issue of what proposition to add or what to replace a proposition with. It is not

difficult to see that applying the *reorder* keyword does not make sense because changing the order of the propositions in a message sending action is not very likely to affect the behaviour of the recipients.

EBNF rule 14 states that a proposition can be a belief, a triggering event, a plan or a label. It is not difficult to see that applying the *add*, *delete* or *reorder* keyword does not make sense because it is not allowed by Jason's syntax. As to applying the replace keyword, it is not very applicable because it will raise the issue of what to replace the proportion with.

EBNF rule 15 states that a literal is an atomic formula or its strong negation (denoted by "~"). Strong negation enables an agent to *explicitly believe* that something is false. It is introduced to overcome the limitation of default negation, which only allows an agent to *assume* that something is false if it is neither known to be true nor derivable from the known. By applying the *replace* keyword, I derive the following mutation operator:

20. **Literal Form Change (LFC)**: A literal is changed to its negation, or vice versa.

Applying the *add* or *delete* keyword can produce a mutation operator that inserts/deletes the strong negation operator ("~") into/from a literal. This operator, however, is not sensible because it can be subsumed by LFC. As to applying *reorder* keyword, it does not make sense because changing the order of the negation operator and the atomic formula in a literal is not allowed by Jason's syntax.

EBNF rule 16 states that an atomic formula is a predicate that can be applied to a tuple of parameters – terms enclosed in parentheses. By applying the *delete* and *reorder* keywords, I derive the following two mutation operators:

21. **Parameter Deletion (PD3)**: A single parameter in an atomic formula is deleted.
22. **Parameter Swap (PS)**: Two adjacent parameters in an atomic formula are swapped.

Similar to BFS, PS does not aim to change the order of all parameters in an atomic formula, because it will produce a large number of mutants, as Savarimuthu and Winikoff [96] indicate.

An atomic formula can also be followed by a tuple of *annotations* – terms enclosed in square brackets. Annotations are introduced to provide meta information about the atomic

formula for practical purposes such as customizing the agent's control or implementing extensions. By applying the *delete* keyword, I derive the following mutation operator:

23. **Annotation Deletion (AD)**: A single annotation in an atomic formula is deleted.

It is worth noting that there are some predefined annotations handled by the interpreter. *atomic* is one of them, which can be specified in the label of a plan to ensure that this plan can be executed without interleaving with others. Applying AD into this label will delete this annotation so that the execution of this plan will be interleaved with that of others. Another important predefined annotation is *source(S)*, where *S* can be one of three values: *percept*, *self* and an agent ID, which respectively represent that the atomic formula comes from perception of environment, cognition and an agent identified by the ID. By applying the *add* and *replace* keywords, I derive the following mutation operators:

24. **Annotation addition (AA)**: The *atomic* annotation is added to an atomic formula. A possible source is added to an atomic formula if the formula does not have any source.
25. **Source replacement (SR)**: A source of an atomic formula is replaced by another possible one.

Rule 17 states that a term can be a literal, a list, an arithmetic expression, a variable or a predefined string (e.g., agent ids). It is not difficult to see that applying the *add*, delete or *reorder* keyword does not make sense because it is not allowed by Jason's syntax. As to applying the replace keyword, it will raise the issue of what to replace the term with.

Rule 18 states that a list consists of terms and can include the "|" operator which is used to separate the first term in a list from the list of all remaining terms in it (e.g., matching [H|T] with [1,2,3] will instantiate H with 1 and T with [2,3]). By applying the *delete* and *reorder* keywords, I derive the following two mutation operators:

26. **List Term Deletion (LTD)**: A single term in a list in deleted.
27. **List Term Order Change (LTOC)**: The order of terms in a list is changed.

Applying the *add* or *replace* keyword to EBNF rule 18 will raise the issue of what list term to add or what to replace a list term with, therefore, is not very applicable.

EBNF rule 19–20 define logical expressions, 21–22 relational expressions, and 23–24 arithmetic expressions. Inspired by existing, well-established program-level mutation operators applied to these entities [10], I apply corresponding keywords to derive the following six mutation operators for Jason:

28. **Unary Logical Operator Addition (ULOA)**: The unary logical operator (not) is inserted before a logical (sub)expression.
29. **Logical Expression Deletion (LED)**: A single logical subexpression is deleted.
30. **Binary Logical Operator Replacement (BLOR)**: A single binary logical operator (& or |) is replaced by the other.
31. **Relational Operator Replacement (ROR)**: A single relational operator (<, <=, >, >=, ==, \==, =, =..) is replaced by another.
32. **Unary Arithmetic Operator Addition (UAOA)**: The unary arithmetic operator (-) is inserted before an arithmetic expression.
33. **Binary Arithmetic Operator Replacement (BAOR)**: A single binary arithmetic operator (+, −, *, **, /, div, mod) is replaced by another.

Table 2 summarizes the above derived mutation operators.

Table 2. Syntactic mutation operators for Jason

ID	Name	Interpretation
1	Belief Deletion (BD)	A single belief in an agent program is deleted.
2	Goal Deletion (GD)	A single goal in an agent program is deleted.
3	Plan Deletion (PD)	A single plan in an agent program is deleted.
4	Belief Order Change (BOC)	The order of relevant beliefs in an agent program is changed.
5	Goal Order Change (GOC)	The order of goals in an agent program is changed.
6	Plan Order Change (POC)	The order of relevant plans in an agent program is changed.
7	Initial Element Type Replacement (IETR)	A belief is changed to a goal with the same literal (literal. → !literal.) if the resultant goal can match the trigger of some plan; a goal is changed to a belief with the same literal (!literal. → literal.).
8	Rule Condition Deletion (RCD)	The conditions of a rule are deleted.
9	Plan Label Deletion (PLD)	The label of a plan is deleted if specified.
10	Plan Context Deletion (PCD)	The context of a plan is deleted if specified and not specified as *true*.
11	Plan Body Deletion (PBD)	The body of a plan is deleted if specified and not specified as *true*.

12	Triggering Event Type Replacement (TETR)	The type of a triggering event (+, -, +!, -!, +?, -?) is replaced by another.
13	Body Formula Deletion (BFD)	A single body formula is deleted.
14	Body Formulae Swap (BFS)	Two adjacent body formulae are swapped.
15	Body Formula Type Replacement (BFTR)	The type of a body formula for changing cognitive state (!, !!, ?, +, -, -+) is changed to another. A mental note or test goal is changed to an achievement goal with the same literal only when the resultant goal can match the trigger of some plan.
16	Message Timeout Deletion (MTD)	The message timeout specified in a communication action is deleted.
17	Recipient Range Change (RRC)	The range of the recipients in a *.send* action is changed. Each *.broadcast* action will be transformed into the equivalent *.send* action so that this mutation operator can be applied into it.
18	Illocutionary Force Replacement (IFR)	The illocutionary force (tell, untell, achieve, unachieve, askOne, askAll, tellHow, untellHow, askHow) in a message sending action is replaced by another.
19	Proposition Deletion (PD2)	A single proposition in a message sending action is deleted.
20	Literal Form Change (LFC)	A literal is changed to its negation, or vice versa.
21	Parameter Deletion (PD3)	A single parameter in an atomic formula is deleted.
22	Parameter Swap (PS)	Two adjacent parameters in an atomic formula are swapped.
23	Annotation Deletion (AD)	A single annotation in an atomic formula is deleted.
24	Annotation addition (AA)	The *atomic* annotation is added to an atomic formula. A possible source is added to an atomic formula if the formula does not have any source.
25	Source replacement (SR)	A source of an atomic formula is replaced by another possible one.
26	List Term Deletion (LTD)	A single term in a list is deleted.
27	List Term Order Change (LTOC)	The order of terms in a list is changed.
28	Unary Logical Operator Addition (ULOA)	The unary logical operator (not) is inserted before a logical (sub)expression.
29	Logical Expression Deletion (LED)	A single logical subexpression is deleted.
30	Binary Logical Operator Replacement (BLOR)	A single binary logical operator (& or |) is replaced by the other.
31	Relational Operator Replacement (ROR)	A single relational operator (<, <=, >, >=, ==, \==, =, =..) is replaced by another.
32	Unary Arithmetic Operator Addition (UAOA)	The unary arithmetic operator (-) is inserted before a arithmetic (sub)expression.
33	Binary Arithmetic Operator Replacement (BAOR)	A single binary arithmetic operator (+, −, *, **, /, div, mod) is replaced by another.

Table 3 indicates which keyword—rule combinations produce the operators derived above. A tick indicates that applying the corresponding keyword to the corresponding rule produces one or more mutation operators. The last row indicates the number of rules each keyword is used to. Note that the *delete* and *replace* keywords are more frequently used than the *add* and *reorder* keywords. This is not surprising, because applying the *add* keyword easily raises the issue of what to add. As for the *reorder* keyword, its application easily raises the issue of too many mutants, and it can only be applied to limited syntactic elements whose order is not restricted by the syntax.

Table 3. Keyword-Rule combinations that produce syntactic mutation operators for Jason

Keyword / Rule ID	add	delete	reorder	replace
1		√	√	√
2		√		
3				
4		√		
5				
6				√
7				
8		√	√	
9				√
10		√		
11				√
12				√
13		√		
14				
15				√
16	√	√	√	√
17				
18		√	√	
19	√	√		√
20				
21				√
22				
23				√
24	√			
Count	3	9	4	10

3.3. Generating Realistic Faults using Syntactic Mutation Operators for Jason

3.3.1. Methodology

What are the faults introduced by Jason programmers? To answer this, I intend to identify and classify faults in two groups of executable Jason projects described in Table 4. The first group consists of 7 assignments of the Multi-Agent Systems course (that involves much Jason programming), which are written by master students taught by Bordini, one of the

authors of Jason, and which are identified by S1–S7. Each student is asked to develop any MAS he/she likes and write a report in which he/she has to document the requirements and design using appropriate techniques (e.g., Prometheus diagrams). The second group consists of 6 examples written/verified by the authors of Jason (experts) and identified by E1–E6.

Table 4. Descriptions of the selected Jason projects

ID	Project name	Description
S1	Election	This project aims to simulate an election process. Three different agents randomly move in a gridded area in their own free time. Each agent will be given a task at a random time so that it will stop moving and then take some time to handle the task in its own way. If an agent passes another that is handling the given task, the former will stop moving and then help the latter. An agent's current influential action, namely handling its task or helping another, will be evaluated by others nearby. When voting takes place, each agent will move to a poll to vote for one of the others according to its own prior evaluation. The simulation ends with the voting results.
S2	Flood Robots	In a flooded area modelled as a gridded one, several boats leave from their base to randomly move around in order to find victims. If one finds a victim, it will tell other boats that the corresponding location has been visited, and contact the unmanned aerial vehicle (UAV) in the base, which will then fly to and take photos of the victim. A boat/UAV's move will consume a unit of fuel, and if the boat/UAV finds that its remaining fuel is just enough to return to the base, it will immediately return to the base to refill the fuel.
S3	Rescuing Agent	A rescuing agent searches for and saves prisoners in a partially observable gridded area, in which obstacles may exist.
S4	Agent Combat	A monster and a hunter randomly move in a cave modelled as a gridded area. When they meet outside the monster's home or the exit, one of them will initiate a fight, in which they attack each other in turn. A fight is over if the monster/hunter's HP is (less than) 0, or if the monster successfully runs away (i.e., continues moving randomly without initiating a fight) when its HP is lower than a certain value. When the monster returns its house, its HP will be recovered. The game is over if the monster reaches the exit, or if the monster/hunter dies in a fight.
S5	Protocol Communication	This project specifies an environment implementing a serial communication channel and an agent for testing this channel.
S6	Supply Chain Integration	This project aims to simulate file transfer along the supply chain. A file will be transferred/processed in some order between/by four agents: the client, the server, the orchestrator, and the portal. The processing includes encrypting/decrypting, compressing/decompressing, importing/exporting, translating, authorizing, and publishing, and sending. The monitor agent keeps an eye on the transfer process and generates a report when an error occurs.
S7	Online Store	This project aims to simulate an online transaction. The order processing agent sends an order to the inventory agent, which will then check whether the item in the order is available in stock: if yes, it will request the transportation agent to deliver the item; otherwise, it will ask the distribution agent to replenish the stock with such items and then request the transportation agent.
E1	Cleaning Robots	A cleaning robot searches for pieces of garbage located in a gridded area and carries them to an immobile incinerator robot.
E2	Domestic Robot	A domestic robot repeats a process of getting beer from the fridge and then serving its owner the beer, until it finds and tells that the owner has reached a certain limit of drinking. The robot will ask the supermarket to deliver beer when finding the fridge empty.
E3	Mining Robots	In a gridded area, an immobile building robot requires different types and numbers of resources to do some construction work. Therefore, several mining robots search for these resources and carry them to the building robot.
E4	Blocks World	This project has five agent programs, each used to create an agent that adopts a different strategy to restack a set of blocks as required, by a series of actions that can include picking up and putting down a single block.
E5	Gold Miners	In a gridded area in which obstacles may exist, several mining agents search for gold pieces and carry them to a depot. They communicate with each other and a leading agent in order to improve efficiency. Gold pieces can be generated with the area or during MAS execution by clicking on empty grids.
E6	Gold Miners II	This project can be viewed as an extended and improved version of E5, which allows two teams of mining agents to compete for collecting gold. One team consists of agents similar to but more robust than the agents in E5. The other team consists of dummy agents, which just randomly move around in order to find gold, without any communication.

Table 5 shows the size of the selected projects. The size of the Java and configuration files in each project is not shown because of first of all, I focus on testing the agent programs in order to use the detected faults to evaluate my derived mutation operators for agent programs. Second, I do not check many places in the Java and configuration files (e.g., places where visualization and user interaction are implemented); instead, I modify these files in order to generate most of the test inputs. The number of agent roles represents how many roles the agents play in the MAS (e.g., In the MAS specified by E5, the agents play two roles: miner and leader). The number of lines in agent programs is counted by `wc`, in the same way as Winikoff's [107].

Table 5. Size of the selected Jason projects

ID	No. of agent roles	No. of lines in agent programs	If involving communication
S1	3	162	
S2	2	168	Y
S3	2	40	
S4	2	62	Y
S5	1	16	
S6	5	173	
S7	4	76	Y
E1	2	50	
E2	3	107	Y
E3	2	78	Y
E4	5	158	
E5	2	372	Y
E6	3	717	Y
Total number of lines:		2179	

To identify and classify faults in each project, I use Winikoff's process [107] shown in Fig.7 and described below.

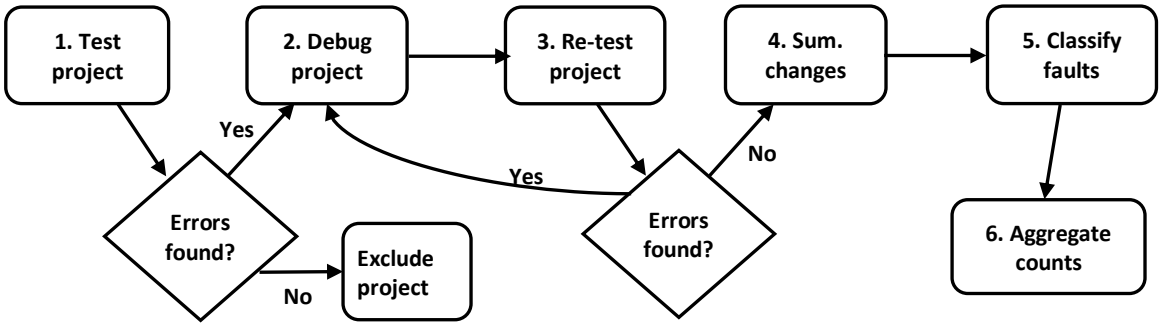


Fig. 7. The process of identifying and classifying faults in a Jason project

1. **Test**: I select system level testing because complex and possibly emergent MAS behaviour is of particular interest [108], and because the selected projects are simple, therefore, faults at unit/integration level can probably be detected by system-level tests. In order to test the project effectively and efficiently, I develop a testing system whose components are described below. If no errors are found by this system, the project will be excluded.

   1.1. **The test component:** it includes the following three parts that have to be customized for a particular project:

   - *Labelled input values*: The main input to each selected MAS is the initial state of the MAS, which consists of a set of values distributed in different project files and representing number/locations of agents/objects, map size, etc. In order to change this input, some of its constituent values can be labelled[6] so that the testing system can locate and replace them by new values.
   - *A set of tests*: A test includes a unique ID and a test input. The ID is automatically generated when the test is created, and it is used to refer to the test elsewhere in the testing system (see later). The test input consists of (label, newValue) pairs, each instructs the testing system to replace the corresponding labelled input value by a new one. As to the expected output of the test, it is specified in the

[6] Labels are comments of the form `/*label*/` so that they will not affect the compilation/interpretation of project files and the MAS behaviour.

test oracle introduced below (as Sect. 1.2.1 indicates, the expected output of a test can be specified in the test oracle).

Tests can be generated manually or by a random test generator (most tests for the selected projects are generated in the latter way). Given the number (N) of tests to be generated and the ranges (R) of the labelled input values, the generator can produce N tests, each aims to replace all the labelled input values and satisfies R. In addition, it ensures that the generated tests cover the valid boundaries of R.

- *An oracle method*: When a test is executed, this method will be regularly called with an argument – the ID of the test. If it returns *true*, the test will terminate and be marked as passed, after which the next test will start execution; if it returns false, the test will continue execution. A command can be called in this method to terminate the current test, mark it as failed and then start executing the next test. Therefore, oracles can be constructed in this method for the generated tests.

When customizing the test component, I consider the specifications in each student's report and in all available documents associated with each Jason example (i.e., Jason textbook [20], project webpages [3], documents in project packages). I also make the following decisions:

- I do not intend to test the visualization and user interaction implemented in some projects.
- In determining which input values to label and their ranges, I obey the constraints of the agent logic. For instance, since the requirements of E5 does not explicitly or implicitly restrict the number of mining agents (N), I naturally expect N to range from 1 to a computationally acceptable number. However, I find that the agent logic restricts N to 4: the agent operates based on the assumption that there are 4 sub areas, each is assigned to an agent. In this case, I give up labelling N rather than adapt the agent logic for making N changeable.

- I determine the number of the automatically generated tests according to the size of the project and the number and ranges of the labelled input values.
- In specifying the expected output of each test in the oracle method, I assign each test a *lifetime* as part of its expected output. A lifetime is a constraint specifying the maximum execution time of the test (as Sect. 1.2.1 indicates, there are different types of expected output, including constraints on behaviour). When the lifetime expires, the test will terminate and be marked as failed, after which the next test will start. Lifetime is introduced to avoid endless waiting for the MAS to produce the expected output. In this thesis, I do not test the execution efficiency of the project, therefore, I assign the same long enough lifetime to each test for the project. This lifetime is derived by 1) creating a test that requires more execution time than any test for the project (e.g., a test that creates a more challenging environment for agents), 2) executing this test, and then 3) doubling the execution time required by this test[7].
- If it is hard to construct an automated oracle for a test, I construct a human one instead: I define a thread that is created only the first time the oracle method is called with the ID of the test. When this thread is created, it will wait for my input for confirming that the test has passed. It will terminate when my confirmation is made or the lifetime of the test expires, in the former case, the oracle method will return *true* and the next test will start.

Appendix 1 gives an overview of the test component customized for each project.

1.2. **The controller component:** it controls the testing process described by the pseudo-code in Fig. 8. It uses JRebel [5] to rapidly reload the Java classes modified by tests.

[7] Given the same test for a MAS, its execution time usually varies in different runs. In this study, since there is not much uncertainty in the MAS behaviour that needs to be checked, it is not very likely that the maximum execution time of a test is more than twice the minimum execution time. Therefore, giving a test's lifetime twice the time of one of its executions is reasonable.

```
1:  While there are still unmarked tests:
2:      Deploy the next unmarked test
3:      Execute the test until finding that it has passed or its lifetime has
        expired
4:      Mark the test as "passed" or "failed"
5:  Derive the testing result
```

Fig. 8. The process of testing a Jason project

2. **Debug**: I attempt to fix the project according to the found errors. One issue here is that there are often alternative fixes that have to be selected from. In order to resolve this, I use Winikoff's strategy [107]: I select the simplest fix(s) with the intent of following the assumption that programmers normally create programs close to being correct [66]. When there is more than one simplest fix, I select the one changing the places closest to the first identified source of the errors.
3. **Re-test**: I re-run the tests to check whether the project has been fixed, if not, return to the debug step.
4. **Summarise changes**: I compare the original and fixed versions of the project to identify differences as faults. Note that besides identify faults, Winikoff [107] further identify the runtime failures caused by the identified faults. By contrast, this thesis does not identify failures because it aims to evaluate the proposed syntactic mutation operators using the identified faults only.
5. **Classify faults**: In order to understand and quantify the identified faults, I attempt to classify them using an evolving taxonomy as Winikoff [107] does. This taxonomy is first drafted based on Jason's constructs and features, then, if necessary, modified and extended based on new identified faults. When drafting a taxonomy, I imitate Winikoff's "coarse grained" taxonomy of faults in GOAL agent programs while considering Jason-specific features, in order to facilitate generalization of the findings.
6. **Aggregate counts**: I keep count of how many instances of each fault category can be found in the identified faults.

After identifying and classifying faults in the projects, I check which of my derived mutation operators can be used to generate instances in the fault categories.

3.3.2. Results

5 of the 7 students' projects (S1, S2, S4, S6, S7) and 4 of the 6 experts' projects (E2, E4, E5, E6) are found faulty. Among these faulty projects, all the students' and 2 experts' (E2, E5) have faults in the constructs defined by the syntax from which my mutation operators are derived. Table 6 shows my derived fault categories and the numbers of their instances. It also shows which of my derived mutation operators can be used to generate these instances. Appendix 2 shows how the category instances manifest as failures in the MAS.

Table 6. Categories of faults in the selected Jason projects and syntactic mutation operators for Jason that can be used to generate category instances

Fault category	No. of instances	Applicable mutation operator
a. wrong belief rule condition (EO)		
b. additional goal (EO)	1	
c. wrong plan order (EO)	3	POC
d. missing plan (D)	6	PD
e. wrong triggering event (D)		
f. wrong plan condition (D)	1	
g. missing plan condition (D)	9	LED
h. additional plan condition (D)	7	
i. wrong body formula (D)		
j. wrong body formula operator (subcategory of i)	2	BFTR
k. missing body formula (D)	11	BFD
l. additional body formula (D)	2	
m. missing parameter (separated from e)	1	PD3
n. missing annotation (separated from f)	1	AD
o. wrong source (separated from e)	1	SR
p. wrong relational operator (separated from a and f)	2	ROR
q. wrong directive for plan patterns	1	
r. wrong perception update	3	
s. wrong action definition	4	

To indicate how the fault categories are derived in Table 6, their names are followed by different annotations (in parentheses) which are explained below:

1. Annotation “D” indicates that the category is drafted by imitating Winikoff’s taxonomy and considering Jason’s features.
2. Annotation “EO” indicates that the category come from empirical observation.
3. Annotation “subcategory of” indicates that the category is a subcategory of another category (i.e., “j” is a subcategory of “i”), it is separated (as well as the number of its instances) because its instances can be generated using my derived mutation operators.
4. Annotation “separated from” means that a category (denoted by *Cat1*) is separated from another category (denoted by *Cat2*) as well as the number of its instances. This annotation is used because the instances of *Cat1* can belong to other categories besides *Cat2* (e.g., in Table 6, instances in “n” can not only belong to “f” but also belong to “e” and “i”) or can be generated using my derived mutation operators.

Note that the numbers of the instances in some categories are left blank, because all their instances are separated.

Categories below the dotted line and in italic, namely “q”, “r” and “s”, include instances that are not from the constructs defined by the syntax from which the mutation operators are derived: “q” includes wrong *plan pattern directives*, a class of constructs for facilitating implementation of strategies for achieving goals (e.g., putting a plan for achieving a goal *g* between *{bc(g)}* and *{end}* allows the plan to be retried once the plan fails to achieve *g*); “r” includes faults in the *updatePercepts* function defined in Java environment files, such as not adding or not clearing some percepts; “s” includes faults in the *executeAction* function defined in Java environment files, such as wrong conditions or effects of actions. The derivation of mutation operators from these constructs/faults is left to future work.

To better understand the derived fault categories, each non-empty one is illustrated in Table 7 with one of its instances. In the “Example Fault Instance” column, the code marked with a strikethrough represents it should be present but is missing. In the “Correction” column, the code marked with a strikethrough represents it should not be present, therefore, is deleted; the symbols marked with a grey background highlight the difference between the fault and its correction.

Table 7. Examples of the identified fault instances

Category	Example Fault Instance	Correction	Project	Fault Description
b	!execute(file).	~~!execute(file).~~	S6	The additional goal "!execute(file)" will mask the fault in category "c" below, so that the fault will not always be detected on the same test.
c	+!execute(file) : importable(file) +!execute(file) : translatable(file)	+!execute(file) : translatable(file) +!execute(file) : importable(file)	S6	The wrong order of the two plans will cause the agent to process the file in wrong order.
d	~~+around(X,Y) : true<- +around(X,Y).~~	+around(X,Y) : true<- -+around(X,Y).	E5	The absence of this plan will prevent the miner agent continuing moving in few cases when none of the other plans with the same triggering event is applicable.
f	+!search(exit): at(hunter) & lifeIs(Value) & Value > 0	+!search(exit): at(hunter) & lifeIs(Value) & Value > 9	S4	The wrong condition "Value>0" will prevent the monster agent running away when its HP is less than the specified value, i.e. 9.
g	+voteTime(true): ~~not voted~~	+voteTime(true): not voted	S1	The absence of the condition "not voted" is likely to cause the agent to vote twice.
h	+help(N) : not at(neutralag,N)	+help(N) : ~~not at(neutralag,N)~~	S1	The additional condition "not at(neutralag, N)" will prevent the agent helping another if both are just in the same place.
j	!combat(hunter);	!!combat(hunter);	S4	The wrong formula operator "!" will make the monster agent cannot execute the next action to run away on time when its HP is not enough.

k	~~-+fuel(Quantity-1);~~	-+fuel(Quantity-1);	S2	The absence of the action "-+fuel" will make the UAV agent not consume fuel occasionally.
l	?reportedVictim(VX,VY);	~~?reportedVictim(VX,VY);~~	S2	This additional test action will re-associate "VX" and "VY" (that have been associated with the location of the victim just found by the agent) with the location of the victim just broadcasted by another agent, so that this location will be broadcasted again.
m	+piped(file)	+piped(file,_)	S6	The absence of the parameter in the plan "+piped" will cause the plan not to be triggered to process the file.
n	+init_pos(S,X,Y)[source(A)] : .count(init_pos(S,_,_)~~[source(_)]~~,4)	+init_pos(S,X,Y)[source(A)] : .count(init_pos(S,_,_)[source(_)],4)	E5	In the leader agent, the absence of this annotation will cause the condition ".count" to fail because the beliefs "init_pos" sent from the four miner agents will not be counted. As a result, the plan "+init_pos" will not be selected to assign the miner agents the areas they are responsible for.
o	+!envia(transportador,Pedido) [source(master)]	+!envia(transportador,Pedido) [source(_)]	S7	The wrong source of the plan "+!envia" will make the plan cannot be triggered by the stock agent to send the order after the stock is replenished.
p	too_much(B) :- QtdB > Limit	too_much(B) :- QtdB >= Limit	E2	The wrong relational operator ">" will cause the agent to drink beer one more time after the drinking limit is reached.

q	{begin bc(clear(X), dg(clear(X)))}	{begin bc(clear(X), bdg(clear(X)))}	E4	The wrong goal pattern “dg” will not cause the goal “clear(X)” to be retried for moving blocks in a few cases when the goal failed to be achieved.

r	void updatePercepts() {
	~~clearAllPercepts();~~
	}
		void updatePercepts() {
		clearAllPercepts();
		}
			S2	The absence of this statement for regularly clearing all previous percepts will make the boat agent hold outdated beliefs about locations so that it may run out fuel outside the base.

s	boolean executeAction(String agent, Structure action) {
	~~removePercept("orchestrator", Literal.parseLiteral("piped(file,port)"));~~
	}
		boolean executeAction(String agent, Structure action) {
		removePercept("orchestrator", Literal.parseLiteral("piped(file,port)"));
		}
			S6	The absence of this statement for removing the percept “piped” will make future addition of the same percept cannot trigger the plan “+piped”.

Let F denote the total fault instances identified in the constructs defined by the syntax from which the mutation operators for Jason are derived, two findings are summarized below:

Finding 3.1: 9 out of 33 mutation operators for Jason can be used to generate 77% of F (36 out of 47). In particular, 4 operators, namely POC, PD, LED and BFD (marked with a grey background in Table 6) can be used to generate 61% of F (29 out of 47).

Finding 3.2: Simulating the other 23% of F (11 out of 47) using mutation is difficult, because it requires much knowledge of the problem domain (e.g., knowledge about what to add/substitute for). For instance, in E2 there is a plan with trigger "+voteTime(true)". The aim of this plan is to let the agent to stop what it is doing and go to the polling station to vote when it perceives that it is time to vote. However, a condition "not at(ag, votePlace)" was wrongly added into the context of the plan so that the plan will not be selected for voting when the agent is just at the polling station. Simulating this fault requires the information about possible beliefs held by the agent. Appendix 3 shows why each of these fault instances is hard to simulate.

3.4. Related Work

This chapter derives a set of syntactic mutation operators for Jason by the methodology proposed by Savarimuthu and Winikoff [95, 96] to derive mutation operators for AgentSpeak and GOAL. This chapter makes the following extensions to Savarimuthu and Winikoff's work:

1. Extensions to the work on deriving mutation operators for AgentSpeak: Savarimuthu and Winikoff [95] develop a set of generic mutation schemas, then give several examples of how to derive mutation operators for AgentSpeak by instantiating these schemas. However, they do not present the final operator set. Since Jason is based on AgentSpeak, my derived operator set completes their work. My operator set also involves constructs that theirs does not involve, namely initial beliefs, initial goals and some Jason-specific constructs (i.e., belief rule, plan label, body formula operator "!!",

message timeout, strong negation, annotation). In addition, I verify the representativeness of some operators, some of which involve constructs that their set does not involve (see Table 6 and 7).

2. Extensions to the work on deriving mutation operators for GOAL: Savarimuthu and Winikoff [96]'s mutation operator set for GOAL does not target initial beliefs and goals, while my operator set does. They evaluate their mutation operators for GOAL using a set of students' projects, each specifies a single agent for solving the same problem (Blocks World). By contrast, although the total size of my selected projects is not as large as theirs, my selected projects are more diverse: They are written by two groups of programmers – students and experts, to solve different problems (see Table 4), and some of them involve agent communication (see Table 5). I found that my operators can be used to generate some faults introduced by experts, some related to agent communication and some related to initial beliefs and goals (see Appendix 2).

It is useful to compare Savarimuthu and Winikoff [96]'s and my mutation operators which target similar constructs and whose representativeness has been shown. Table 8 shows these operators and how useful they are to generate the detected faults in terms of the ranking of the number of times they are applied. It can be seen that the 1^{st} – 4^{th} most frequently applied mutation operators for Jason are similar to the 1^{st} – 8^{th} most frequently applied mutation operators for GOAL except the 7^{th} (namely the operator that changes *if-then* to *forall-do*) which is applied to GOAL specific constructs, as detailed below.

Table 8. Similar syntactic mutation operators for Jason and GOAL

Mutation operator for Jason	Frequency ranking	Mutation operator for GOAL	Frequency ranking
A. Body Formula Deletion	1	a. Swapping two adjacent action rules	1
B. Logical Expression Deletion	2	b. Dropping an action rule	2
C. Plan Deletion	3	c. Dropping a literal in a conjunction	3
D. Plan Order Change	4	d. Dropping an action	4
		e. Dropping a mental literal	5
		f. Moving an action rule to the start	6
		g. Moving an action rule to the end	8

1. "A" is similar to "d", because a body formula in Jason is similar to an action in GOAL.

2. “B” is similar to “c” and “e”, because both sides involve deleting part of the conditions on an element. To further explain this, let the conditions on an element be denoted by *Cond*, we can see that:
   - In Jason, the syntax of *Cond* permits nested logical expressions [58] (see rule 19—20 in Fig.6). Expressions at the bottom level are simple logical expressions such as literals. “B” can delete one logical expression at one level at a time.
   - In GOAL, *Cond* consist of mental literals (namely groups of beliefs/goals), each includes a set of literals. “e” can “c” can respectively delete a single mental literal and a single literal at a time.
3. “C” is similar to “b”, because a plan in Jason is similar to an action rule in GOAL.
4. “D” is similar to “a”, “f” and “g”, because reordering plans/action rules involves swapping or/and moving them.

From the above comparison, a finding is summarized below.

Finding 3.3: Mutation operators for deleting an action, part of conditions, a plan and reordering a plan seem to be most useful in simulating faults in logic based agent programs.

In addition to the comparison to Savarimuthu and Winikoff’s work, it is also useful to compare my mutation operators to other existing fault categories and mutation operators related to MAS models. Poutakidis et al. [90] derive five fault categories related to agent communication from students’ agent programs: the instances of *Uninitialized Agent* (which cause the agent whose initialization is delayed not to receive a message in time) and *Wrong Recipient* can be simulated by my RRC operator. The instances of *Failure to Send* (which cause a message not to be sent) can be simulated by my BFD operator. As to the instances of *Additional Message Sent* and *Same Message Sent Multiple Times*, their simulation by mutation is hard because it requires knowledge about possible messages to be sent and possible elements (e.g., beliefs, goals, actions) to be added to cause the same message to be sent multiple times.

Adra and McMinn [7] propose four mutation operator classes for a generic agent model. Their class for agent communication (Miscommunication, Message Corruption) corresponds to my RRC, IFR, PD2 and other operators for the constituents of the agent communication actions (e.g., literals in the propositional content). Their class for an agent's memory corresponds to my BD and BOC operators. Their mutation operator class for agent execution does not directly correspond to any of my operators since the agent execution in their model is very different from that in BDI. As to their mutation operator class for environments, it does not correspond to any of my operators which only target agent programs.

Saifan and Wahsheh [93] propose a set of mutation operators for JADE mobile agent systems. I find it difficult to compare their and my operator sets because they are based on very different programming models.

Padgham et al. [86] derive fault categories from a suite of BDI agent systems, in order to evaluate their oracle generation technique. I find it difficult to compare my mutation operators with their categories because some of their categories are actually failure categories, whose instances can be caused by different simple or complex faults in different constructs. For instance, an instance of one of these failure categories, *plan failure*, can be caused by different faults such as missing or wrong beliefs or actions. Because of this difficulty, the comparison is left to future work.

4. Semantic Mutation Operators for Jason, GOAL and 2APL

Semantic mutation operators are rules for making changes to the interpretation of a program. This chapter proposes semantic mutation operators for three similar logic based agent languages, namely Jason, GOAL and 2APL. Then, it evaluates the ability of the operators for Jason to simulate real misunderstandings. From the derived semantic mutation operator sets and the evaluation results, this chapter finally proposes a systematic approach to generating semantic mutation operators for logic-based agent languages.

4.1. Deriving Semantic Mutation Operators for Jason, GOAL and 2APL

4.1.1. Methodology

Recall from Sect. 2.2.3 that semantic mutation operators can be derived from semantic changes required in the scenarios in which semantic mutation is useful. Sect. 4.1.2–4.1.6 propose five scenarios in which semantic mutation for Jason, GOAL and 2APL is useful, identify sources of semantic changes in these scenarios, and then derive semantic mutation operators from these sources. Sect. 4.1.7 summarizes these operators.

There are three kinds of sources of semantic changes: (1) studies of faults/errors/failures, (2) semantic differences between different languages or versions of the same language, and (3) freedom in interpretation. Due to the availability of sources, this chapter does not intend to derive a complete set of semantic mutation operators for each selected language, instead it aims for demonstrating how to derive operators for these languages and deriving initial operator sets for further study.

The main source of semantic changes required in the proposed scenarios is semantic differences, whose identification requires comparison of the semantics. Since a language usually has many semantic elements, I use the following comparison strategies in relevant scenarios.

1. Focusing on comparing four semantic elements that are common and important to logic based agent languages, namely *deliberation step order*, *rule selection*, *rule execution*, and *cognitive element query and update*, as marked with asterisks in Table 9. As to other common semantic elements (e.g., semantics of communication actions), they will be roughly checked and they belong to "other" in Table 9. The detailed comparison between them is left to future work.

   In Table 9, the term *rule* is used as a common name of a Jason plan, a GOAL rule and a 2APL rule. I will use standard terms defined by these languages later when describing the identified semantic differences and the derived semantic mutation operators.

Table 9. Common semantic elements of Jason, GOAL and 2APL

ID	Element	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 more rules are selected.
3	Rule execution*	Rule execution is an important deliberation step in which one or more rules that were selected are executed.
4	Cognitive element query and update*	Cognitive elements (e.g., beliefs and goals) can be queried somewhere such as in rule selection and updated somewhere such as in rule execution.
5	Other	Other common elements not listed above.

2. Focusing on comparing semantic differences between similar constructs. As Clark et al. [23] indicate, such differences are hard to notice because they are often small (although there may be exceptions), and because a programmer is likely to judge a similar construct simply by appearance.
3. Checking both formal and informal semantics for comparison. I start with checking the formally defined semantics because they can be compared directly. I also check those that are informally defined, by writing code to test the behaviour in specific cases, and by reviewing the source code of the interpreter.
4. Focusing on comparing default semantics. Some semantic elements are customizable, for instance, the Jason event selection order can be customized by overriding the corresponding method in the Java class implementing the agent architecture; the GOAL

rule selection order can be customized in the GOAL agent program. This thesis, as an initial exploration of semantic mutation for agent languages, does not consider customized semantics when comparing the semantics of different languages (or different versions of a language), because some semantic elements can be customized in many ways (e.g., Jason plan selection), which makes comparison difficult.

The derived semantic mutation operators for Jason, GOAL and 2APL are respectively identified by J, G and A followed by indexes. For instance, the second derived operator for Jason is identified by J2. In some cases, extending an existing derived operator is more succinct than deriving a new operator. The extended operators are identified by the identifiers of the original operators with single quotes (e.g., J2').

4.1.2. Using a New Language

When a programmer who has used one of Jason, GOAL and 2APL starts to use one of the others, he/she may have misunderstandings caused by semantic differences between the two languages. In order to capture such misunderstandings, I first identify semantic differences between Jason, GOAL and 2APL using the methodology proposed in Sect. 4.1.1. The semantic differences found are shown in Table 10 and described below.

Table 10. Some semantic differences between Jason, GOAL and 2APL

ID	Source	Jason	GOAL	2APL
1	The order of rule selection and rule execution	select a plan → execute a plan	(select and execute event rules → select and execute an action rule) x Number_of_Modules	select PG rules → execute rules → select an external event rule → select an internal event rule → select a message event rule
2	Rule selection	• applicable • linear	• enabled • linear (action rules) and linearall (event rules)	• applicable • linear (event rules) and linearall (PG rules)
3	Rule execution	• one plan/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 all events and intentions that relate 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 lies in the order of two deliberation steps, namely rule selection and rule execution. A Jason agent first selects a plan to be a new intention and then executes one of existing intentions. 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. A 2APL agent first selects PG rules (rules for handling goals) to be new intentions[8], and then executes all intentions, next selects an external event rule, an internal event rule and a message event rule to be new intentions (by default, the first *applicable* rule for each event is selected). From difference 1, I derive the following semantic mutation operators for GOAL and 2APL:

G1. **Deliberation step order change (DSO)**: Two steps in a GOAL reasoning cycle, namely "select an event rule" and "select an action rule" are swapped.

A1. **Deliberation step order change (DSO)**: Two steps in a 2APL reasoning cycle, namely "select PG rules" and "select event rules" are swapped.

[8] In fact, the authors of 2APL use another term – *plans* rather than intentions to represent rules being executed. However, I use intentions here because (1) it is a term that is more commonly used to express this meaning in the BDI model on which 2APL is based and which is introduced by this thesis, and (2) it can avoid the confusion caused by the differences between Jason and 2APL plans.

Difference 2 lies in the rule selection deliberation step. Jason, GOAL and 2APL differ in two aspects of this step, namely rule selection condition and rule selection order. Regarding rule selection condition, a Jason plan or a 2APL rule can be selected if both its trigger and guard conditions 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"). From these differences in rule selection condition, I derive the following semantic mutation operators for GOAL and 2APL:

G2. **Rule selection condition change (RSC)**: The condition of selecting a GOAL rule is changed from "enable" to "applicable".

A2. **Rule selection condition change (RSO)**: The condition of selecting a 2APL rule is changed from "applicable" to "enable".

Note that I do not derive a semantic mutation operator for Jason to change the plan selection condition from applicable to enable, because unlike GOAL and 2APL actions, Jason actions have no (explicit) structure for specifying pre- and post- conditions.

Regarding rule selection order, Jason plans are selected in linear order (i.e., plans are checked in the order they appear in the agent program text, the applicable plan first found is selected), GOAL action rules are selected in linear order while GOAL event rules are selected in "linearall" order (i.e., rules are checked in the order they appear in the agent program text, and all enabled rules found are selected), 2APL PG rules are selected in "linearall" order while 2APL event rules of each type (external event, internal event, message) are selected in linear order. From these differences in rule selection order, I derive the following semantic mutation operators for Jason, GOAL and 2APL:

J1. **Plan selection order change (PSO)**: The order of selecting Jason plans is changed from "linear" to "linearall".

G3. **Rule selection order change (RSO)**: The order of selecting action or event rules is changed from "linear" to "linearall", or vice versa.

A3. **Rule selection order change (RSO)**: The order of selecting PG or event rules is changed from "linear" to "linearall", or vice versa.

Difference 3 lies in the rule execution deliberation step, in which a Jason agent executes a single action in a single intention, a GOAL agent executes all actions in each selected event/action rule[9], a 2APL agent executes a single action in each intention. From difference 3, I derive the following semantic mutation operators for Jason and 2APL:

J2. **Plan execution order change (PEO)**: The order of executing Jason plans is changed from one plan per cycle to all plans per cycle.

J3. **Plan execution order change 2 (PEO2)**: The interleaved execution of Jason plans is changed to the non-interleaved execution.

A4. **Rule execution order change (REO)**: The order of executing 2APL PG rules is changed from all rules per cycle to one rule per cycle.

Difference 4 lies in the belief query. In a Jason/2APL agent, beliefs are queried in linear order (i.e., beliefs are checked in the order they are stored in the belief base, and the matching belief first found is returned). In a GOAL agent, beliefs are queried in random order (i.e., beliefs are randomly checked, and the matching belief first found is returned). From difference 4, I derive the following semantic mutation operators for Jason, GOAL and 2APL:

J4. **Belief query order change (BQO)**: The Jason belief query order is changed from linear to random.

G4. **Belief query order change (BQO)**: The GOAL belief query order is changed from random to linear.

A5. **Belief query order change (BQO)**: The 2APL belief query order is changed from linear to random.

Difference 5 lies in the belief addition. In a Jason agent, a new belief is added to the start of the belief base. In a GOAL/2APL agent a new belief is added to the end of the belief base.

[9] Unlike Jason and 2APL, a GOAL agent has no intention set or similar structure, so once selecting a rule, it will immediately attempt to execute the rule to completion.

From difference 5, I derive the following semantic mutation operators for Jason, GOAL and 2APL:

J5. **Belief addition position change (BAP)**: The position where a Jason belief is added is changed from the start to the end of the belief base.

G5. **Belief addition position change (BAP)**: The position where a GOAL belief is added is changed from the end to the start of the belief base.

A6. **Belief addition position change (BAP)**: The position where a 2APL belief is added is changed from the end to the start of the belief base.

Difference 6 lies in the goal query. A goal query is used to search for a current goal that satisfies a certain condition in order to act on this goal (e.g., delete it) or get the data contained in this goal for future use (e.g., unify a variable with a value associated with this goal). In a Jason agent, since goals are embodied in related events and intentions, the agent searches for a goal first in the event base then in the intention set, the one first found is returned. In a GOAL agent, goals are queried in random order. In a 2APL agent, goals are queried in linear order. From difference 6, I derive the following semantic mutation operators for GOAL and 2APL:

G6. **Goal query order change (GQO)**: The GOAL goal query order is changed from random to linear.

A7. **Goal query order change (GQO)**: The 2APL goal query order is changed from linear to random.

Difference 7 lies in the goal addition. In a Jason/GOAL agent, a new goal is added to the end of the event or goal base. In a 2APL agent, a new goal can be added to the start or end of the goal base using different actions (i.e., *adopta* or *adoptz*). From difference 7, I derive the following semantic mutation operators for Jason and GOAL:

J6. **Goal addition position change (GAP)**: The position where a Jason goal is added is changed from the end to the start of the event set.

G7. **Goal addition position change (GAP)**: The position where a GOAL goal is added is changed from the end to the start of the goal base.

Difference 8 lies in the goal deletion. Given a goal φ to be deleted, a Jason agent deletes all events and intentions that relate to φ; a GOAL agent deletes all goals that each include φ as a logical sub-goal; a 2APL agent can delete only φ, all goals that each are a logical sub-goal of φ, or all goals that each include φ as a logical sub-goal using different actions (i.e., *dropgoal*, *dropsubgoal* or *dropsupergoal*). From difference 8, I derive the following semantic mutation operator for GOAL:

G8. **Goal deletion semantics change (GDS)**: Given a goal φ to be deleted, a GOAL deletes only φ or all goals that each are a logical sub-goal of φ, rather than deleting all goals that each include φ as a logical sub-goal.

Note that I do not derive a semantic mutation operator for Jason to change the semantics of goal deletion, because unlike a GOAL and 2APL goal, Jason goals do not represent states of affairs, therefore, have no logical relationship.

Difference 9 lies in the goal type. Jason adopts procedural goals – goals that only serve as triggers of procedures. GOAL and 2APL adopt declarative goals – goals that also represent states of affairs to achieve. From difference 9, I do not derive any semantic mutation operator for changing the goal type from procedural goals to declarative goals or vice versa, because I think the resultant operators will probably cause behavioral changes that are very easy to detect, which means these operators are probably not powerful enough to evaluate tests (as Sect. 2.1.2 indicates, mutation should be powerful enough to reveal weaknesses in testing).

Difference 10 lies in the goal commitment strategy. Jason does not 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). GOAL and 2APL adopt blind goal commitment strategy, which requires that a goal is pursued until it is achieved or explicitly dropped. From difference 10, I do not derive any semantic mutation operator for changing the goal commitment strategy, because I think the resultant operators will probably cause

behavioral changes that are very easy to detect, which means these operators are probably not powerful enough to evaluate tests.

4.1.3. Language Evolution

When a programmer moves a program from one of Jason, GOAL and 2APL to its successor (either a different language or a newer version of the same language), he/she may have misunderstandings caused by the semantic changes in the evolution of the language, or may want to check whether a program is robust to the semantic changes or whether the semantic changes are reliable for existing programs (see Sect. 2.2.1 for the discussion of robustness and reliability). To derive semantic changes required in these cases, I should first find out semantic differences between the language and its successor. I take 3APL [38] and its successor 2APL, and different versions of Jason as examples: Table 11 shows some semantic differences between 2APL and 3APL, which are derived from the methodology in Sect.4.1.1; Table 12 shows some semantic differences between different versions of Jason, which are derived from the methodology in Sect.4.1.1 and the Jason changelog [2]. I explain these differences as follows.

Semantic differences between 2APL and 3APL

Table 11. Some semantic 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 10	select an PG rule → select a PR-rule → execute a rule
3	PG rule selection	linearall	linear
4	Rule execution	all rules/cycle	one rule/cycle

Difference 1 lies in the PR rules (internal event rules). In 2APL, the abbreviation “PR” means “plan repair”, a PR rule is selected when a relevant plan fails. In 3APL, “PR” means “plan revision”, a PR rule is selected when it matches some plan. From this difference, I derive the following semantic mutation operator for 2APL:

A8. **Rule selection condition change 2 (RSC2)**: The condition of selecting a 2APL PR-rule is changed from “when a relevant plan fails” to “when it matches some plan”.

Difference 2 lies in the order of rule selection and rule execution deliberation steps. The order adopted by 2APL has been described in Sect. 4.1.2. By contrast, a 3APL agent selects a PG rule then a PR rule to be new intentions, then executes an existing intention. From this difference, I choose to extend DSO for 2APL as follows:

A1’. **Deliberation Step order change (DSO)**: In a 2APL reasoning cycle, the sequence “select PG rules → execute rules → select event rules” is changed to “select PG rules → select event rules → execute rules” or “select event rules → select PG rules → execute rules”.

Difference 3 lies in the PG rule selection order. As described in Sect. 4.1.1, 2APL PG rules are selected in “linearall” order. By contrast, 3APL PG rules are selected in linear order. I find that RSO for 2APL can be used to make this difference.

Difference 4 lies in the rule execution deliberation step. As described in Sect. 4.1.1, a 2APL agent executes all intentions in a deliberation cycle. By contrast, a 3APL agent executes a single intention. I find that REO for 2APL can be used to make this difference.

Semantic differences between different versions of Jason

Table 12. Some semantic differences between different versions of Jason

ID	Source	Before some version	Since that version
1	Belief deletion action	*-b* deletes any *b*	*-b* deletes *b* if *b* is a mental note
2	Drop desire action	Remove only related events	Remove related events and intentions

Difference 1 lies in the belief deletion action *-b*. Before Jason v0.95 *-b* will delete *b* regardless of its source. After that version, *-b* deletes *b* only if *b* is a mental note (i.e., if *b* has the annotation *source(self)*). From this difference I derive the following semantic mutation operator for Jason:

J7. **Belief deletion action semantics change (BDAS)**: The Jason belief deletion action, *-b*, removes *b* under any condition rather than only if *b* is a mental note.

Difference 2 lies in the drop desire action. Before Jason v0.96 this action drops only related events. After that version this action drops related events and intentions. From this difference I derive the following semantic mutation operator for Jason:

J8. **Drop desire action semantics change (DDAS)**: The Jason drop desire action, *.drop_desire(φ)*, removes only related events rather than both related events and intentions.

4.1.4. Common Misunderstandings

A programmer may have semantic misunderstandings common to a particular group of people he/she belongs to. Such misunderstandings can be identified by analyzing these people's common faults. I take GOAL as an example: I select some of the common GOAL novice programmers' faults identified by Winikoff [107]. These faults are selected because it is easy to think of possible misunderstandings that cause them. These faults and the corresponding possible misunderstandings are shown in Table 13 and explained below.

Table 13. Some possible common misunderstandings of GOAL

ID	Fault	Misunderstanding
1	Wrong rule order	Rules are selected in a different order than the linear order
2	Using "if then" instead of "forall do"	A rule "if q then a" is applied for all instances for which the query "q" and the pre-condition of "a" holds (as "forall q do a")

Misunderstanding 1 is that by default, rules are selected in a different order than linear order. Note that although GOAL supports customization of rule selection order using special constructs, this misunderstanding may still occur in some cases (e.g., when a novice programmer does not study these special constructs, or when a programmer thinks that the default rule selection order defined by GOAL is the same as the default PG rule selection order defined by 2APL that he/she has used). To simulate this misunderstanding, I choose to extend RSO for GOAL as follows:

G3'. **Rule selection order change (RSO)**: The order of selecting action or event rules ("linear", "linearall", "random" and "randomall") is changed to one of the others.

Misunderstanding 2 comes from a rule "if q then a", which is applied for only one instance for which the query "q" and the pre-condition of "a" holds. A programmer may think this rule applied for all such instances (as "forall q do a"). To simulate this misunderstanding, I derive the following semantic mutation operator for GOAL:

G9. **"if then" semantics change (ITS)**: "if then" is interpreted the same as "forall do".

4.1.5. Semantic Ambiguity

A programmer may have misunderstandings of the ambiguous semantics. Bordini and Hübner [19] give two examples of such misunderstandings of Jason as shown in Table 14. I explain these misunderstandings as follows.

Table 14. Some possible misunderstanding of the Jason's informal semantics

ID	Source	Misunderstanding
1	Goal deletion event	"when an intention fails" → "when an intention is removed"
2	Test action	Generate no event when the action is executed → Generate a test goal addition event when the action is executed

Misunderstanding 1 may arise from the ambiguous semantics of a goal deletion event. A goal deletion event (*-!e* or *-?e*) can be generated when an intention that includes the corresponding goal addition triggering event (*+!e* or *+?e*) fails. A programmer may have the misunderstanding that this event can be generated when this intention is removed as the result of completion or failure. I do not derive any semantic mutation operator to simulate this misunderstanding because I think the agent behaviour is mainly driven by posting and removal of goals; if the removal of every goal generates a goal deletion event, it will probably cause significant behavioral changes that are very easy to detect (e.g., if a goal deletion event -!g cannot be handled by some plan, the Jason interpreter will throw a warning/error message).

Misunderstanding 2 may arise from the ambiguous semantics of a test action. A test action (*?e*) generates no event when it is executed. A programmer may have the misunderstanding that a test action generates a test goal addition event when it is executed, which is similar to an achievement goal action (*!e*). To simulate this misunderstanding, I derive the following semantic mutation operator for Jason:

J10. **Test action semantics change (TAS)**: When a test action is executed, a test goal addition event is generated.

Note that simulating either misunderstanding 1 or 2 involves generation of goal related events. However, I only choose to simulate misunderstanding 2, because I think simulating it only involves generation of events related to test goals, therefore, is less likely to cause significant behavioral changes.

4.1.6. Customizing the Interpretation

The interpretation of a Jason, GOAL or 2APL agent program can be customized by modifying the interpreter or selecting between the interpretation options. A programmer may want to check whether a customized interpretation results in the same agent behaviour as the original interpretation, or whether it leads to an optimization of performance (recall from Sect.2.2.1 that semantic mutation can be used not only in testing). Semantic mutation can be used in this context to represent possible alternative interpretations. I take Jason as an example: Table 15 shows some options for interpreting Jason agent programs, which are derived from the Jason changelog updated over time [2].

Table 15. Some options for interpreting Jason agent programs

ID	Option
1	Enable/disable tail recursion optimization for sub-goals
2	Enable/disable cache for queries in the same cycle
3	Treat an event generated by a belief revision action as internal/external
4	Treat an event generated by an achievement goal action as internal/external

From the above options, I derive the following semantic mutation operators for Jason:

J10. **TRO enable/disable (TRO)**: The option for tail recursion optimization ("enabled" or "disabled") for sub-goals is changed to the other.

J11. **Query cache enable/disable (QC)**: The cache for queries in the same cycle is enabled/disabled.

J12. **Belief Revision Action Semantics Change (BRAS)**: A Jason belief revision action generates an external event rather than an internal event.

J13. **Achievement Goal Action Semantics Change (AGAS)**: A Jason achievement goal action generates an external event rather than an internal event.

4.1.7. Summary of the derived semantic mutation operators

Table 16–18 show the derived semantic mutation operators for Jason, GOAL and 2APL, and which scenario(s) they are derived from (but not limited to). Due to data availability, this thesis does not intend to derive a complete set of semantic mutation operators for each language. Instead, it aims for demonstrating how semantic mutation operators are derived and deriving initial sets for further study. The current operator sets can be improved when more data is available (e.g., when studies of faults in 2APL programs are available).

Abbreviations for the scenarios in which semantic mutation for Jason, GOAL and 2APL is useful

Using a New Language: UNL
Common Misunderstandings: CM
Customizing the Interpretation: CI
Language Evolution: LE
Semantic Ambiguity: SA

Table 16. Semantic mutation operators for Jason

Abbr	Semantic mutation operator	Description	Context
PSO	Plan selection order change	linear → linearall	UNL
PEO	Plan execution order change	one plan/cycle → all plans/cycle	UNL
PEO2	Pule execution order change 2	interleaved execution of plans → non-interleaved execution of plans	UNL
BQO	Belief query order change	linear → random	UNL
BAP	Belief addition position change	start → end	UNL
BRAS	Belief revision action semantics change	generate internal events → generate external events	CI
BDAS	Belief deletion action semantics change	*-b* deletes *b* if *b* is a mental note → *-b* deletes any *b*	LE
GAP	Goal addition position change	end → start	UNL

AGAS	Achievement goal action semantics change	generate internal events → generate external events	CI
DDAS	Drop desire action semantics change	remove the related events and intentions → remove only the related events	LE
TAS	Test action semantics change	generate no event when the action is executed → generate a test goal addition event when the action is executed	SA
TRO	TRO enable/disable	enable/disable tail recursion optimization for sub-goals	CI
QC	Query cache enable/disable	enable/disable cache for queries in the same cycle	CI

Table 17. Semantic mutation operators for GOAL

Abbr	Semantic mutation operator	Description	Context
DSO	Deliberation step order change	select and execute event rules then an action rule → select and execute an action rule then event rules	UNL
RSC	Rule selection condition change	enabled → applicable	UNL
RSO	Rule selection order change	change between linear, linearall, random and randomall	UNL, CM
BQO	Belief query order change	random → linear	UNL
BAP	Belief addition position change	end → start	UNL
GQO	Goal query order change	random → linear	UNL
GAP	Goal addition position change	end → start	UNL
GDS	Goal deletion semantics change	"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
ITS	"if then" semantics change	make "if then" interpreted the same as "forall do"	CM

Table 18. Semantic mutation operators for 2APL

Abbr	Semantic mutation operator	Description	Context
DSO	Deliberation step order change	change the original order "select PG rules → execute rules → select event rules" to "select PG rules → select event rules → execute rules" or "select event rules → select PG rules → execute rules"	UNL, EL
RSC	Rule selection condition change	applicable → enabled	UNL
RSC2	Rule selection condition change 2	PR rule selected when a relevant plan fails → selected when it matches some plan	EL
RSO	Rule selection order change	change between linear and linearall	UNL, EL
REO	Rule execution order change	all rules/cycle → one rule/cycle	UNL, EL
BQO	Belief query order change	linear → random	UNL
BAP	Belief addition position change	end → start	UNL
GQO	Goal query order change	linear → random	UNL

4.2. Evaluating the Representativeness of the Semantic Mutation Operators for Jason

The choice of evaluating the representativeness of semantic mutation operators for Jason is motivated by data availability. In order to answer "Can these operators represent real misunderstandings?", I perform a study detailed in Sect. 4.2.1–4.2.2.

4.2.1. Methodology

7 Bordini's master students participated in this study (their coursework was earlier used to evaluate my syntactic mutation operators for Jason, see Sect.3.3). Here I designed questions about the Jason semantics for these students to answer, and I expected these students to give some incorrect answers, which indicate misunderstandings. In designing questions, there were many issues that had to be considered and resolved, as described below.

1. **When to let the students answer the questions**: If the students were given inadequate time to get familiar with the Jason semantics, their answers would not be very interesting. Therefore, Bordini did not give the questions until the students had finished the course (including all lectures, all lab exercises and the coursework).
2. **How to let the students be serious and motivated**: If the students paid insufficient attention to the study, their answers would not be very reliable. Therefore, I asked Bordini to (1) emphasize the significance of the study during teaching, (2) to administer the study as a classroom test, in which the students have to answer the questions independently, under supervision and without assistance from books and computers, and (3) to convey my appreciation and commitment to offer three £10 vouchers as rewards.
3. **Which semantic parts to ask about**: Each question aims to test the understanding of a reasonable semantic part targeted by a semantic mutation operator for Jason. A semantic part is reasonable if it is included in the teaching materials (e.g., lecture notes, exercise sheets).

   Note that only asking the semantics targeted by semantic mutation operators is not enough to evaluate the representativeness of the operators. I adopt this method

because of the limitation of the scope of the study. In the future, a systematic survey of the understanding of Jason has to be conducted.

4. **How to ask the questions**: Each question takes the form of multiple choice, which reduces the difficulty of the question and thus the students' negative attitude, and which facilitates result analysis. Instead of directly asking about the targeted semantics, most questions each first present a program whose semantics include the targeted ones, and then present possible execution results as options some of which reflect the understanding of the targeted semantics. This way of asking can lower the students' guard and make them naturally show their misunderstandings. Each program is made simple in order to ensure that the students are familiar with the constructs involved, reduce their negative attitude and minimize the number of possible misunderstandings that each relevant option can reflect. Even so, several questions still ask about the targeted semantics directly, because it is very difficult to design simple programs to do so.

   Each question provides three categories of options. The first category includes the correct option. If the question involves a program, this option was verified by running the program; otherwise, it was verified from the Jason textbook [20].

   The second category includes one or more incorrect options, which are used to make the students distracted and thus show their misunderstandings easily, and to capture their misunderstandings adequately. The design of these options is guided by the following principles:

   - These options must be reasonable. An incorrect option is reasonable if it can be caused by making one or two small changes to the semantics involved. If the question involves a program, these options must be verified by modifying the interpreter and then running the program.
   - At least one option is (probably) caused by applying the semantic mutation operator.

   The third category includes the "I do not know" option. Based on the assumption that the students are generally honest, this option can be used to improve the accuracy of the study. Attempts have been made to increase the likelihood of this assumption, such as emphasizing the significance of the study, committing to reward the students

and reducing the students' negative attitude. In addition, in the beginning of the question sheets, I requested the students to do select this option if they were unsure about the correct answer.

5. **How much time to give to answer the questions**: The answers given in a hurry are not very reliable. Therefore, the students were given enough time to read, analyse and answer each question.

Before this study could be conducted, it had to be approved by the ethics committees of my and Bordini's universities. Therefore, we took the following measures to satisfy ethical requirements:

1. **Keeping data anonymous**: Bordini asked students not to leave their names when answering questions, and he identified students using numbers (1 – 7) when sending their answers to me for analysis.
2. **Asking for explicit consent**: During the course of Multi-Agent Systems, Bordini provided students with my designed consent form, which clearly states the purpose and content of the study, and that the participation is totally voluntary. This consent form had been approved by the ethics committees.

I designed 11 questions that involve the semantics targeted by my semantic mutation operators for Jason (see Table 16), except those involved by TRO and QC because they were not sensible sources of students' misunderstandings. I gave the students 30 minutes to answer these questions. These questions are shown and described below. In each question, the options in bold are correct answers, those in italic can be caused by applying the semantic mutation operators. Note that some questions ask for "likely" output in order to imply that the output may be non-deterministic because of random query order.

1. What is the likely output of the following program?

agent.asl:

```
//initial beliefs
bel(0). bel(1).

//initial goals
!goal.

//plan 1
+!goal : bel(X) <- .print(X); !goal.
```

A	**B**	C	D
[agent] 0	**[agent] 0**	[agent] 1	I do not
[agent] 1	**[agent] 0**	[agent] 1	know
[agent] 0	**[agent] 0**	[agent] 1	
[agent] 1	**[agent] 0**	[agent] 1	
…	**…**	…	

Question 1 targets the belief query order. In the presented program, plan 1 will be repeatedly selected for execution in response to the repeatedly posted goal *!goal*. Each time plan 1 is selected, it will query a belief and then print its argument. There are two initial beliefs that can match each query, namely *bel(0)* and *bel(1)*, however, since Jason beliefs are queried in linear order, *bel(0)* will always be returned, and thus "0" will always be printed, as shown in option B. If a student selects option A, he/she is most likely to have the misunderstanding that beliefs are queried in random order, as simulated by BQO. If selecting option C, he/she is most likely to have the misunderstanding that initial beliefs are added to the start of the belief base in the order they appear in the program (they are actually added in the reverse order), so that *bel(1)*, as the one added most recently, will be on top and thus returned.

2. What is the output of the following program?

agent.asl:

```
//initial beliefs
bel(0). bel(1). bel(2).

//initial goals
!goal.

//plan 1
+!goal : bel(0) & bel(1) <- .print(0).

//plan 2
+!goal : bel(1) & bel (2) <- .print(1).
```

A	**B**	C	D
[agent] 0	**[agent] 0**	[agent] 0	I do not
[agent] 1		OR	know
		[agent] 1	

Question 2 targets the rule selection order. In the presented program, plan 1 and 2 will become applicable simultaneously, because their triggers and contexts get satisfied by the initial beliefs *bel(0)*, *bel(1)* and *bel(2)* and goal *!goal*. Since Jason plans are selected in linear order, only plan 1 will be selected for execution, and thus print "0", as shown in option B. If a student selects option A, he/she is most likely to have the misunderstanding that plans are selected in "linearall" order, as simulated by RSO. If selecting option C, he/she is most likely to have the misunderstanding that plans are selected in random order, like the one simulated by RSO for GOAL.

3. How does an agent execute intentions in a SINGLE reasoning cycle?

A. **An agent will execute a single action in a single intention**
B. *An agent will execute a single action in each intention*
C. I do not know

Question 3 directly asks about one part of the rule execution order because it is difficult to design a simple program to do so. A Jason agent will execute a single action in a single intention, as described in option A. Option B describes the misunderstanding simulated by REO.

4. What is the likely output of the following program?

agent.asl:

```
//initial goals
!goal. !goal2.

//plan 1
+!goal : true <- .print(0); .print(1); .print(2); .print(3).

//plan 2
+!goal2 : true <- .print(4); .print(5); .print(6); .print(7).
```

A	*B*	C
[agent] 0	*[agent] 0*	I do not know
[agent] 1	*[agent] 1*	
[agent] 4	*[agent] 2*	
[agent] 2	*[agent] 3*	
[agent] 5	*[agent] 4*	
[agent] 3	*[agent] 5*	
[agent] 6	*[agent] 6*	
[agent] 7	*[agent] 7*	

Question 4 targets the other part of the rule execution order. In the presented program, plan 1 and 2 will be selected for execution in response to the initial goals *!goal* and *!goal2*. Since parallel Jason plans are executed in an interleaved fashion, plan 1 and 2 will

alternatively print numbers, as shown in option A. If a student selects option B, he/she is most likely to have the misunderstanding that parallel plans are executed in a non-interleaved fashion, as simulated by REO2.

5. What is the output of the following program?

agent.asl:

```
//initial beliefs
bel(0).

//initial goals
!goal.

//plan 1
+!goal : true <- +bel(1); !goal2.

//plan 2
+!goal2 : bel(X) <- .print(X).
```

A	**B**	C*	D
[agent] 0	**[agent] 1**	[agent] 0 OR [agent] 1	I do not know

Question 5 targets the belief addition position. In the presented program, plan 1 will be selected for execution in response to the initial goal *!goal*. During execution, it will first add a new belief *bel(1)*, and then call plan 2 that will query a belief and then print its argument. Now there are two beliefs that can match the query, namely *bel(0)* and *bel(1)*. Since a Jason belief is added to the start of the belief base, *bel(1)*, as the one added most recently, will be returned for the query, and thus "1" will be printed, as shown in option B. If a student selects option A, he/she is most likely to have the misunderstanding that a belief is added to the end of the belief base, as simulated by BAP. If selecting option C, he/she is likely to have the misunderstanding that a belief is queried in random order, which is adopted by GOAL.

6. What is the likely output of the following program?

agent.asl:

```
//initial goals
!goal.

//plan 1
+!goal : true <- !goal2; .print(0); .print(1).

//plan 2
+!goal2 : true <- .print(2); .print(3).
```

A	*B*	*C*	*D*
[agent] 2	*[agent] 0*	*[agent] 2*	*[agent] 0*
[agent] 3	*[agent] 2*	*[agent] 0*	*[agent] 1*
[agent] 0	*[agent] 1*	*[agent] 3*	*[agent] 2*
[agent] 1	*[agent] 3*	*[agent] 1*	*[agent] 3*
E			
I do not know			

Question 6 targets the semantics of an achievement goal action. In the presented program, plan 1 will be selected for execution in response to the initial goal *!goal*. During execution, it will first add a new goal *!goal2* and then print "0" and "1". Since adding *!goal2* will generate an internal event that will cause plan 2 to be selected for execution and plan 1 to be suspended until plan 2 is finished, "2" and "3" will be printed by plan 2 before "0" and "1", as shown in option A. If a student selects B, C or D, he/she is most likely to have the misunderstanding that an achievement goal action will generate an external event, as simulated by AGAS. It is worth noting that option C and D also indicate additional misunderstandings such as that of the deliberation step order.

7. What is the likely output of the following program?

agent.asl:

```
//initial goals
!goal.

//plan 1
+!goal : true <- +bel; .print(0); .print(1).

//plan 2
+bel : true <- .print(2); .print(3).
```

A	*B*	*C*	*D*
[agent] 2	*[agent] 0*	*[agent] 2*	*[agent] 0*
[agent] 3	*[agent] 2*	*[agent] 0*	*[agent] 1*
[agent] 0	*[agent] 1*	*[agent] 3*	*[agent] 2*
[agent] 1	*[agent] 3*	*[agent] 1*	*[agent] 3*
E	F		
[agent] 0 [agent] 1	I do not know		

Question 7 targets the semantics of a belief revision (addition) action. In the presented program, plan 1 will be selected for execution in response to the initial goal *!goal*. During execution, it will first add a new belief *bel* and then print "0" and "1". Since adding *bel* will generate an internal event that will cause plan 2 to be selected for execution and plan 1 to be suspended until plan 2 is finished, "2" and "3" will be printed by plan 2 before "0" and "1". Therefore, the correct printing sequence is "2301", as shown in option A. If a student selects option B, C or D, he/she is most likely to have the misunderstanding that a belief addition action will generate an external event, as simulated by BRAS (Like similar options in Question 6, option C and D also indicate additional misunderstandings). If selecting option E, he/she is most likely to have the misunderstanding that a belief addition action will generate no event.

8. What is the output of the following program?

agent.asl:	*agent2.asl*:

//initial goals
!goal.
 	
//plan 1
+!goal : true <-
.send(agent2, tell, bel).
	//plan 1
	+bel : true <-
	.print(0); -bel.
	
	//plan 2
	-bel : true <- .print(1).

A	**B**	*C*	D
NO OUTPUT	**[agent2] 0**	*[agent2] 0* *[agent2] 1*	I do not know

Question 8 targets the semantics of a belief deletion action. In *agent.asl*, plan 1 will be selected for execution in response to the initial goal *!goal*. During execution, it will send a belief *bel* to another agent specified by *agent2.asl*. Then, plan 1 in *agent2.asl* will be selected for execution in response to the received belief. During execution, it intends to print "0" and then delete the received belief using the belief deletion action *-bel*, which, however, will produce no effects because it can only delete a mental note. Therefore, no belief deletion event will be generated to cause plan 2 to print "1", and finally only "0" will be printed, as shown in option B. If a student selects option C, he/she is most likely to have the misunderstanding that *-bel* can delete any *bel*, as simulated by BDAS. If selecting option A, he/she is most likely to have the misunderstanding that the reception of a belief will generate no event.

9. What is the output of the following program?
agent.asl:

```
//initial goals
!goal. !goal2.

//plan 1
+!goal : true <- !goal.

//plan 2
+!goal2 : true <- .print(0).
```

A	**B**	C
NO OUTPUT	**[agent] 0**	I do not know

Question 9 targets the goal addition position. In the presented program, plan 1 will be repeatedly selected for execution in response to the repeatedly posted goal *!goal*. Since a Jason goal is added to the end of the event base (*E*), the other initial goal *!goal2* always has a chance to be moved to the start of *E* and thus be handled and cause plan 2 to print "0", as shown in option B. If a student selects option A, he/she is most likely to have the misunderstanding that a goal is added to the start of *E*, as simulated by GAP.

10. What does the internal action *.drop_desire(d)* do?

A. Drop the events related to d
B. *Drop the intentions related to d
C. **Drop both the events and intentions related to d**
D. I do not know

Question 10 directly asks about the semantics of a drop desire action because it is difficult to design a simple program to do so. This action removes both related events and intentions, as described in option C. Option A describes the misunderstanding simulated by DDAS, and option B describes another misunderstanding.

11. What is the output of the following Jason program?
agent.asl:

```
//initial beliefs
bel(0).

//initial goals
!goal.

//plan 1
+!goal : true <- ?bel(0); ?bel(1).

//plan 2
+?bel(X) : true <- .print(X, s).

//plan 3
-?bel(X) : true <- .print(X, f).
```

A	*B*	**C**	D
[agent] 0s *[agent] 1f*	*[agent] 0s* *[agent] 1s*	**[agent] 1s**	I do not know

Question 11 targets the semantics of a test action. In the program, plan 1 will be selected for execution in response to the initial goal *!goal*. During execution, it will query two beliefs using two test actions, namely *?bel(0)* and *?bel(1)* respectively. *?bel(0)* will succeed because the queried belief exists as an initial belief, while *?bel(1)* will fail and thus cause the test goal addition plan 2 to print its argument followed by "s", as shown in option C. If a student selects option A or B, he/she is most likely to have the misunderstanding that a test goal addition event is generated when the test action is executed, as simulated by TGAS. Option B indicates also the misunderstanding that a test deletion event is generated when the test action fails.

4.2.2. Results

The results of the study are summarized in Table 19, in which S1–S7 represent the students, Q1–Q11 represent the questions, a "x" indicates an incorrect answer, and a grey background indicates a "I do not know" answer.

Table 19. A summary of the results of the study

SID / QID	S1	S2	S3	S4	S5	S6	S7	No. of the students who answered the question incorrectly
Q1	x		x	x	x		x	5
Q2	x				x		x	3
Q3						x		3
Q4				x	x		x	4
Q5		x						2
Q6		x				x		2
Q7	x	x					x	5
Q8	x	x	x	x	x		x	6
Q9		x		x	x	x		6
Q10	x				x			7
Q11	x				x			7
No. of the incorrect answers given by the student	6	5	2	4	7	3	5	**Avg.:** 4.55 / **Avg.:** 4.57

From Table 19, one finding is summarized below:

Finding 4.1: Students misunderstand the Jason semantics easily. This is perhaps because actual agent execution is opaque. Misunderstandings are likely to cause faults, as implied by the above description of the questions.

The details of the misunderstandings that the students may have are given in Table 20, in which misunderstandings on a grey background are the ones simulated by the 11 semantic mutation operators for Jason which are selected in this study (i.e., all operators in Table 16 except TRO and QC, see Sect. 4.2.1).

Table 20. The details of the misunderstandings that the students may have

Semantics	Correct understanding	Misunderstanding	Semantic mutation operator	No. of students who may have this misunderstanding	QID
Belief query order	linear	random	BQO	3	1, 5
Plan selection order	linear	linearall	PSO	2	2
		random		1	
Plan execution order	one intention/cycle	all intentions/cycle	PEO	1	3
	interleaved	non-interleaved	PEO2	3	4
Belief addition position	start	end	BAP	-	5
Achievement goal action	generate an internal event	generate an external event	AGAS	2	6
Belief revision action	generate an internal event	generate an external event	BRAS	2	7
		generate no event		1	
Belief deletion action	-*b* deletes *b* if *b* is a mental note	-*b* deletes *b* with any annotation	BDAS	5	8
Goal addition position	end	start	GAP	4	9
Drop desire action	remove both related events and intentions	remove only related events	DDAS	1	10
		remove only related intentions		1	
Test action	generate no event when the action is executed	generate a test goal addition event when the action is executed	TAS	2	11
	generate a test goal addition event when the action fails	generate a test goal deletion event when the action fails		-	
Initial belief addition order	initial beliefs are added in the reverse order that they appear in the program	initial beliefs are added in the order they appear in the program		3	1
Belief reception	the reception of a belief generates a belief addition event	the reception of a belief generates no event		1	8

From Table 20, one finding is summarized below:

Finding 4.2: 10 out of the 11 selected semantic mutation operators for Jason can simulate misunderstandings that the students probably have. As to the remaining one, namely BAP, it is also likely to simulate a real misunderstanding because a student (S7 in Table 19) seems to be unclear about the semantics it involves.

Note that most incorrect options that are not designed to evaluate the semantic mutation operators for Jason also reflect possible misunderstandings, from which I derive 5 more semantic mutation operators described in Table 21 as an extension of Table 16.

Table 21. More semantic mutation operators for Jason (an extension of Table 16)

ID	Semantic mutation operator	Description	Context
14	Plan selection order change 2 (PSO2)	linear → random	CM
15	Belief revision action semantics change 2 (BRAS2)	generate an internal event → generate no event	CM
16	Drop desire action semantics change 2 (DDAS2)	drop both related events and intentions → drop only related intentions	CM
17	Initial belief addition order change (IBAO)	added in the reverse order that they appear in the program →added in the order they appear in the program	CM
18	Belief reception semantics change (BRS)	generate an internal event → generate no event	CM

4.2.3. Threats to Validity

There are many issues relating to the internal validity of the study, as described below:

1. **Whether the students were serious about the study**: To address this issue, Bordini emphasized the significance of the study, and administered the study as a classroom test; I committed to reward the students, and attempted to reduce the students' negative attitude. An evidence that each student was more or less serious is that the number of "I do not know" answers given by him/her is not extreme (ranges from 0 to 5, see Table 19).
2. **Whether the difficulty of the questions was appropriate**: To address this issue, Bordini gave the students adequate time to get familiar with the Jason semantics; I made sure the semantics to ask about were included in the teaching materials, and made the programs simple. An evidence that the difficulty of the questions was generally appropriate is that there are no questions to which all the answers are "I do not know" and only 2 questions to which no answers are correct (see Table 19).

   Bordini commented in an email that he was horrified how little they learned about Jason. Lots of mistakes he did not think they would make.
3. **In the questions involving programs, whether each incorrect option (execution results) precisely reflects a certain misunderstanding:** To address this issue, I made the programs simple. After checking these incorrect options, I am pretty sure that each can unambiguously or almost unambiguously reflect a certain misunderstanding. Here "almost unambiguously" means that an incorrect option may be caused by several

misunderstandings, which, however, cause the same faults in most cases. For instance, option C in question 5 may be caused by the misunderstanding of random belief query order, or that of random belief addition position. However, I argue that they cause the same faults in most cases.

4. **Whether enough time is given to answer the questions**: I think generally yes, according to what Bordini confirmed in an email that all students spent less than 30 minutes to answer all questions.

There are also some issues relating to the external validity of the study, as described below:

1. **Whether the results can generalize to other agent languages**: Due to data availability, this chapter only evaluates the semantic mutation operators for Jason. Given Jason does differ from other agent languages in some ways, it is not clear to what extent my results generalize to other agent languages. However, the fact that some of the semantic mutation operators for Jason are representative implies the possibility that the semantic mutation operators for GOAL and 2APL which make the similar semantic changes are also representative.
2. **Whether the results from students who are inexperience in Jason have value in testing practical MAS**: Due to data availability, this chapter only checks inexperienced programmers' misunderstandings. The results show some value of the semantic mutation operators in testing Jason programs developed by the inexperience for education or certification purpose. However, it is not clear how much value my results have in testing practical MAS, which is usually developed by experienced programmers.

4.3. A Systematic Approach to Generating Semantic Mutation Operators for Logic Based Agent Languages

Recall from Sect. 2.1.5 that a systematic approach has been used to derive syntactic mutation operators in order to ensure the (relative) completeness of the operator set. However, no systematic approach has been used to derive semantic mutation operators. From existing semantic mutation operators for Jason, GOAL and 2APL in Table 16–18 and 21, I derive four types of semantic changes: *reorder*, *add*, *delete* and *replace*, as described in

Table 22. Table 23–25 show the change type(s) of each derived operator (operators on a grey background are from Table 21). It is reasonable to suggest that semantic mutation operators for logic based agent languages can be systematically derived by applying these change types to a set of semantic aspects. Note that these keywords are the same as those used for deriving syntactic mutation operators for AgentSpeak, Jason and GOAL, therefore, they should also be applied to formal description of the semantics (e.g., transition rules).

Table 22. The change types derived from semantic mutation operators for Jason, GOAL and 2APL

CID	Change	Interpretation
1	reorder	Change the order of elements or alter the positions of elements
2	add	Add more elements
3	delete	Delete/reduce elements
4	replace	Replace elements by others

Table 23. The change type(s) of each semantic mutation operator for Jason

Operator	CID	Interpretation
PSO	2	Select more plans
PEO	2	Execute more plans
PEO2	4	Replace the plan to be executed
BQO	4	Replace the belief to be returned for a query
BAP	1	Alter the position of the added belief
BRAS	4	Replace the type of the generated event
BDAS	3	Remove the conditions of deleting a belief
GAP	1	Alter the position of the added goal
AGAS	4	Replace the type of the generated event
DDAS	3	Reduce the desires that will be removed
TAS	2	Add an event to be generated
TRO	4	Replace the method of posting a sub-goal
QC	4	Replace the method of querying a belief
RSO2	4	Select a different rule
BRAS2	3	Delete the event to be generated
DDAS2	3	Reduce the desires that will be removed
IBAO	1	Change the order of the added initial beliefs
BRS	3	Delete the event to be generated

Table 24. The change type(s) of each semantic mutation operator for GOAL

Operator	CID	Interpretation
DSO	1	Change the order of deliberation steps
RSC	3	Remove a condition of selecting a rule
RSO	4	Select more/less/different rules
BQO	4	Replace the belief to be returned for a query
BAP	1	Alter the position of the added belief
GQO	4	Replace the goal to be returned for a query
GAP	1	Alter the position of the added goal
GDS	4	Replace the condition of deleting a goal
ITS	4	Replace existential quantification by universal quantification

Table 25. The change type(s) of each semantic mutation operator for 2APL

Operator	CID	Interpretation
DSO	1	Change the order of deliberation steps
RSC	2	Add a condition of selecting a rule
RSC2	4	Replace the conditions of selecting a PR-rule
RSO	4	Select more/less rules
REO	3	Reduce the rules to be executed
BQO	4	Replace the belief to be returned for a query
BAP	1	Alter the position of the added belief
GQO	4	Replace the goal to be returned for a query

5. Evaluating the Ability of Semantic Mutation for Jason to Evaluate Tests and Semantic Changes

This chapter first introduces a semantic mutation system for Jason, and then applies this system into Jason projects in order to check the ability of semantic mutation to evaluate tests, the robustness to (and the reliability of) semantic changes, and the impact of semantic changes on MAS execution time.

5.1. A Semantic Mutation System for Jason

Recall from Sect.3.3.1 that I introduce a system for testing Jason projects. This system consists of two components: the test component that specifies tests for a particular project, and the controller component that executes the tests and collects test results. Here I develop a semantic mutation system by extending the testing system: I implement each semantic mutation operator specified in Table 16 in a different Git branch [4] (the semantic mutation operators in Table 21 are left to future work), so that the operator can be changed to another easily using the Git API during the process of semantic mutation. I also extend the controller component as shown in Fig.9.

```
1:      Under the original interpretation:
2:          After deploying the next test:
3:              Execute the test until finding that it has passed or its
                lifetime has expired
4:              If the test has passed:
5:                  Mark it as "passed"
6*:                 Record its execution time
-------------------------------------------------------------------------
7*:     Under each mutant:
8*:         After deploying the next test that has passed under the original
            interpretation:
9*:             Execute the test until finding that it has passed or its
                lifetime has expired
10*:            It the test has passed, record its execution time
11*:            If the lifetime of the test has expired:
12*:                Mark the mutant as "killed"
13*:                Update the set of tests that killed the mutant
14*:    Derive the results of semantic mutation
```

Fig. 9. The process of semantic mutation for Jason

In Fig.9, the steps are divided into two parts by a line; step 6—14 are marked with asterisks, representing the extensions of the testing system introduced in Sect.3.3.1. In the upper part, Step 1–5 execute all tests under the original interpretation in order to obtain the passed ones that will be executed under semantic mutants. Recall from Sect.2.2.1 that semantic mutation compares the behaviour/output of a program under the original interpretation and a mutant, and kills the mutant if the behaviour/output is different. Here if a test passes under the original interpretation of the program, it means that the output of the program is the same as the expected output of the test. Therefore, to compare the output of the program under the original interpretation and a mutant, the same test can be

run under the mutant. If the test still passes, it means that the output is the same; otherwise, the output is different. Note that the tests that fail under the original interpretation are abandoned because if they fail again under semantic mutants, it is not clear whether they are able to detect the semantic changes. Step 6 records the execution time of the passed tests for later use (see below).

The lower part executes the tests (that have passed under the original interpretation) under semantic mutants and then derives the results of semantic mutation. Step 10 records the execution time of the passed tests so that it can be compared to the previously recorded execution time of the same tests under the original interpretation, in order to evaluate the impact of semantic changes on the MAS execution time. If a test fails, the mutant will be marked as "killed" by Step 12 and the test recorded by Step 13.

5.2. Applying Semantic Mutation System

I apply the semantic mutation system to 4 Jason projects, namely *Domestic Robot*, *Blocks World*, *Gold Miners* and *Gold Miners II*, which are respectively identified by E2, E4, E5 and E6 and described in Table 4. I use the tests generated for detecting faults in these projects in Sect.3.3. Note that these tests consider all equivalence classes and boundaries, as Appendix 1 specifies. To facilitate readers, I repeat the descriptions of these 4 projects in Table 26.

Table 26. Jason projects chosen for evaluating semantic mutation

ID	Project Name	Description
E2	Domestic Robot	A domestic robot repeats a process of getting beer from the fridge and then serving its owner the beer, until it finds and tells that the owner has reached a certain limit of drinking. The robot will ask the supermarket to deliver beer when finding the fridge empty.
E4	Blocks World	This project has five agent programs, each used to create an agent that adopts a different strategy to restack 7 blocks as required, by a series of actions that can include picking up and putting down a single block. If the blocks are numbered b1–b7 and a stack is denoted by $S(b_i, \ldots, b_k)$, an example scenario is where the initial stacks are S(b1,b2,b3), S(b4,b5), and S(b6,b7) while the expected stacks are S(b1,b4,b3,b5), S(b2,b6,b7).
E5	Gold Miners	In a gridded area in which obstacles may exist, several mining agents search for gold pieces and carry them to a depot. They communicate with each other and a leading agent in order to improve efficiency. Gold pieces can be generated with the area or during MAS execution by clicking on empty grids.
E6	Gold Miners II	This project can be viewed as an extended and improved version of E5, which allows two teams of mining agents to compete for collecting gold. One team consists of agents similar to but more robust than the agents in E5. The other team consists of dummy agents, which just randomly move around in order to find gold, without any communication.

I select the corrected versions of these projects because it allows the semantic mutation to be better evaluated: First, it enables all previously designed tests to pass and be executed under semantic mutants (see Sect. 5.1 for the description of the semantic mutation process); second, it can check whether tests that can detect the removed faults can detect simulated misunderstandings.

The semantic mutation results obtained by the semantic mutation system and by my further analysis are shown in Table 27 and 28. Table 27 shows the results of evaluating tests and the robustness to (and the reliability of) semantic changes, while Table 28 shows the results of evaluating the impact of semantic changes on the MAS execution time. Sect. 5.2.1—5.2.3 describe these results and the findings from them.

Table 27. The results of evaluating tests and the robustness to (and the reliability of) semantics changes

SMOP	E2		E4-1		E4-2		E4-3		E4-4		E4-5		E5		E6	
	KP	MT	KP	MT	KP	MT	KP	MT	KP	MT	KP	MT	KP	MT	KP	MT
PSO	0	NE	0	E	100%	K	0	E	0	E	0	E	100%	K	100%	K
PEO	0	E	0	E	0	E	0	E	0	E	0	E	0	E	0	E
PEO2	100%	K	0	E	0	E	0	E	0	E	0	E	100%	K	12.25%	K
BQO	0	E	0	NE	0	NE	0	E	7%	K	0	NE	0	E	0	E
BAP	0	E	32%	K	25%	K	0	E	92%	K	37%	K	38%	K	0	E
BRAS	0	E	N/A		N/A		N/A		N/A		N/A		0	E	0	E
BDAS	0	E	N/A		N/A		N/A		N/A		N/A		0	E	0	E
GAP	0	E	0	E	0	E	0	E	0	E	0	E	0	E	0	E
AGAS	19%	K	98%	K	100%	K	0	E	71%	K	96%	K	100%	K	92.5%	K
DDAS	N/A		N/A		N/A		N/A		N/A		N/A		96%	K	61.5%	K
TAS	N/A		N/A		N/A		N/A		N/A		N/A		0	E	0	E
TRO	0	E	0	E	0	E	0	E	0	E	0	E	0	E	0	E
QC	0	E	0	E	0	E	0	E	42%	K	0	E	0	E	0	E

Table 28. The results of evaluating the impact of semantic changes on the MAS execution time

SMOP	E2		E4-1		E4-2		E4-3		E4-4		E4-5		E5		E6	
	ETR	TRP	ETR	TRP	ETR	TRP	ETR	TRP	ETR	TRP	ETR	TRP	ETR	TRP	ETR	TRP
PSO	N/A		-0.97%	4%	N/A		-30.15%	5%	0.18%	35%	-0.14%	15%	N/A		N/A	
PEO	5.46%	100%	27.97%	100%	0.85%	44%	0.22%	47%	28.39%	100%	36.04%	100%	25.78%	100%	32.55%	100%
PEO2	N/A		-0.42%	8%	3.68%	55%	-0.58%	33%	0.45%	38%	15.76%	100%	N/A		N/A	
BQO	0.34%	45%	N/A		N/A		4.6%	50%	N/A		N/A		-1.48%	43.67%	-0.01%	54.25%
BAP	0.29%	47%	N/A		N/A		-0.18%	44%	N/A		N/A		N/A		-3.55%	
BRAS	0.24%	40.5%	N/A		N/A		N/A		N/A		N/A		1.27%	75%	0.14%	
BDAS	0.09%	39%	N/A		N/A		N/A		N/A		N/A		2.01%	62.67%	-0.58%	
GAP	-0.36%	24%	-0.14%	52%	4.73%	73%	-0.71%	44%	0.58%	72%	0.57%	53%	5.96%	75%	-1.36%	
AGAS	N/A		N/A		N/A		-0.12%	47%	N/A		N/A		N/A		N/A	
DDAS	N/A		N/A		N/A		N/A		N/A		N/A		N/A		N/A	
TAS	N/A		N/A		N/A		N/A		N/A		N/A		-1.11%	53.33%	-1.34%	54.25%
TRO	0.15%	40%	0.1%	54%	0.03%	30%	0.36%__	48%	1.08%	80%	0.50%	56%	-0.18%	45.67%	-1.31%	51.75%
QC	0.08%_	39.5%	-0.05%	39%	0.01%	27%	-0.43%	38%	N/A		0.27%	38%	-0.62%	49.33%	-0.26%	50%

5.2.1. Evaluating Tests

In Table 27, E4-1–E4-5 identify 5 different agents, only one of which is allowed in the execution of E4. “N/A” indicates that the semantic mutation operator does not affect the interpretation of the agent programs, which is judged by static program analysis (e.g., it can be judged that BRAS is inapplicable if no belief revision actions are found in the program). “KP” represents the percentage of tests that kill the mutant. “MT” represents the mutant type, which can be one of “K”, “E” and “NE” explained below:

1. “K” represents a killed mutant. This type can be identified by the semantic mutation system.
2. “E” represents an equivalent mutant. This type is identified by static and dynamic program analysis. For instance, I judge that the E2’s mutant generated by GAP is equivalent, by (1) finding in the agent programs that no constructs cause the order of goal related events to matter and (2) verifying this by observing in Jason’s mind inspector the relevant changes in agents’ mental attitudes on all tests.
3. “NE” indicates that the mutant is not equivalent and not killed. This type is identified by deriving a test that can detect the mutant from program analysis. The “NE” mutants, which are marked with a grey background, indicate the weaknesses of tests. In order to kill them, the tests need be improved.

It can be seen that the “NE” mutants include the mutant of E2 generated by PSO, the mutants of E4-1, E4-2 and E4-5 generated by BQO. This finding is summarized below:

Finding 5.1: PSO and BQO have potential to generate hard-to-kill mutants.

In order to kill the mutant of E2 generated by PSO, I need tests that can capture the differences in the agent behaviour between selecting all applicable plans and selecting only the first applicable plan. To design such tests, I look for a set of plans that have (1) the same triggering event, (2) the contexts that are not mutually exclusive and (3) the ability to affect the agent behaviour. I find that 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 reaches the limit of drinking (*p2*). Then, I design a test on which the owner will reach the limit of drinking exactly when there is no beer in the fridge. This test will cause *p2* to execute twice under the mutant so that the robot will tell the owner twice that the drinking limit has been reached. In addition, I also enable the relevant oracles to check the number of times of warning the owner.

In order to kill the mutants of E4-1, E4-2 and E4-5 generated by BQO, I need tests to capture the differences in the agent behaviour caused by querying beliefs in linear order and in random order. In each of these projects, there is only one place that causes the belief query order and belief order to matter, namely the context of the plan (*p*) which aims to move a block (*b*) from a stack to the table by repeatedly removing a block on the top of the same stack. Note that a stack is built from a single block by repeatedly applying the belief revision rule for adding one of the other blocks on the top. After each application, the resultant stack that is part of the stack to be built is called a sub-stack and added to the belief base. Therefore, it is possible that *b* belongs to more than one (sub-)stack, such as *S(b1, b2, b)* and *S(b2, b)*. In order to move *b*, *b1* has to be removed first and then b2.

Under the original interpretation where beliefs are queried in linear order, the context of *p* is always associated with *S(b1, b2, b)* first so that *b1* can be removed first. This is because the larger the (sub-)stack is, the more recently it is added to the start of the belief base by applying the belief revision rule for building stacks. By contrast, under the mutant of BQO, the context of *p* may be associated with *S(b2, b)* first, which will cause *p* to retry until its context is associated with *S(b1, b2, b)*, because *b2* cannot be removed before *b1*.

To capture the retry of *p*, I can improve relevant oracles, or design a test on which *b* is in the stack *S(b1, b2, b3, b4, b5, b6, b)* (recall from Table 26 that there are a total of 7 blocks) as well as in the sub-stacks of *S*, and *b* has to be moved onto the table. The correct order of association of *p*'s context with this stack and its sub-stacks should be from the largest (i.e., *S(b1, …, b6, b)*) to the smallest (i.e., *S(b6, b)*). Under the mutant of BQO, *p* probably has to retry many times for the correct order of association. However, during retry *p* will easily reach the retry limit required by the agent environment specification. If so, the agent will stop moving blocks, therefore, the test will fail.

5.2.2. Evaluating the Robust to (and the Reliability of) Semantic Changes

Recall from Sect. 2.2.1 that it is sometimes useful to make the program robust in different interpretations. For instance, when teaching agent programming in general, one unavoidably needs to use a specific language, but it is good if he/she can use the subset of that language that has most in common with other rival languages.

In Table 27, the equivalent mutants (E) indicate that the agent programs are robust to the semantic changes, or that the semantic changes are reliable for the agent programs. The killed (K) or non-equivalent (NE) mutants indicate the non-robustness/unreliability. The following shows how to improve the robustness to some of the K and NE mutants. These mutants are selected because the improvements only require a small change to the agent programs.

In order for E2 to be robust to the semantic change caused by PSO, the agent programs can be improved to ensure that there is only one applicable *influential plan*[10] at most in every deliberation cycle (Winikoff [107] views such an improvement as a good programming practice for avoiding many condition related faults he identified in agent programs). As mentioned in Sect. 5.2.1, in this project there are only two influential plans (*p1* and *p2*) that are likely to become applicable simultaneously in the same cycle. Therefore, their contexts can be made mutually exclusive, e.g., by strengthening the context of *p2*.

In order for E4-1, E4-2 and E4-5 to be robust to the semantic changes caused by BQO and BAP, the behaviour of the agent programs can be made independent of the order of beliefs or querying beliefs. As mentioned in Sect. 5.1, in each of these projects there is only one place that causes these orders to matter, namely the context of *p*. Therefore, this context can be strengthened to ensure that it is always associated with the largest stack.

---

[10] An influential plan is one that can affect the agent behaviour directly, or indirectly by affecting the agent's cognitive state. By contrast, a non-influential plan is one that has no direct or indirect effect on the agent behaviour. A non-influential plan is sometimes used in the program, e.g., an empty plan can be used to handle a goal deletion event in order to prevent warning messages given by the Jason interpreter. In a deliberation cycle, if there is only one applicable influential plan while the other applicable plans are non-influential, it is not very likely that changing the order of selecting these applicable plans will change the agent behaviour.

5.2.3. Evaluating the Impact of Semantic Changes on MAS Execution Time

Recall from Sect. 1.1.2 that agent implementation should arguably adopt a *glass-box* approach, which considers not only writing agent programs, but also customizing their interpretation (e.g., plan/intention selection). This is because this approach can retain the intuitive meanings of common-sense notions in agent programs [39]. The semantic mutation system introduced in Sect. 5.1 can search for interpretations that can reduce the execution time, an important aspect of performance [112].

Table 28 shows the results of comparing the test execution time under the original interpretations and the semantic mutants. "N/A" indicates one of the following:

1. That the semantic mutation operator has no effect on the interpretation of the agent programs. This indication is the same as that of "N/A" in Table 27.
2. A "K" or "NE" mutant in Table 27. This mutant is not considered in the evaluation because in general, improvements on performance should not affect behaviour[11].

"ETR" represents the average reduction in execution time (negative ETR represents increase in execution time). "TRP" represents the percentage of tests on which the execution time is reduced.

Among the mutants that lead to positive ETR (larger than 0%), I focus on those that can lead to apparent ETR (no less than 3%) and large TRP (no less than 50%), because unapparent ETR is probably just caused by normal variance in the execution time (the execution time of the MAS selected for this study varies within a small range), and relatively small TRP indicates that the mutant is probably not useful in most cases. My focused mutants are marked with a grey background in Table 28 and described below:

1. PEO can reduce the average execution time of E2, E4-1, E4-4, E4-5, E5 and E6 by 5.46%, 27.97%, 28.39%, 36.04%, 25.78% and 32.55% respectively. The reduction occurs in all tests for all these projects.

[11] There are exceptions, e.g., when the effect of the semantic change on the agent behaviour is tolerant.

2. PEO2 can reduce the average execution time of E4-2 and E4-5 by 3.68% and 15.76%. The reduction occurs in 55% of the tests for E4-2 and all tests for E4-5.
3. BQO can reduce the average execution time of E4-3 by 4.6%. The reduction occurs in 50% of the tests.
4. GAP can reduce the average execution time of E4-2 and E5 by 4.73% and 5.96% respectively. The reduction occurs in 73% of the tests for E4-2 and 75% of the tests for E5.

The above findings are summarized below:

Finding 5.2: PEO, PEO2, BQO and GAP have potential to reduce the execution time of MAS without affecting the MAS behaviour. In particular, PEO has great potential to lead to significant reduction.

6. Conclusions

There are many issues of engineering multi-agent systems (MAS), two of which are as follows:

1. **There is a lack of techniques that can adequately evaluate the effectiveness (fault detection ability) of tests or testing techniques for MAS.**
2. **There are no systematic approaches to evaluating the impact of semantic changes (changes in the interpretation of agent programs) on agents' behaviour and performance.**

This thesis introduces syntactic mutation to address issue 1. Syntactic mutation is a technique that systematically generates variants ("syntactic mutants") of a description (usually a program) following a set of rules ("syntactic mutation operators"). Each mutant is expected to simulate a real fault, therefore, the effectiveness of a test set can be evaluated by checking whether it can detect each simulated fault, in order words, distinguish the original description from each mutant. Although syntactic mutation is widely considered very effective, only limited work has been done to introduce it into MAS.

This thesis also, for the first time, introduces another approach to mutation, namely semantic mutation, to address both issues. Semantic mutation systematically generates variant interpretations ("semantic mutants") of a description following a set of rules ("semantic mutation operators"). It can be used to evaluate test effectiveness by simulating a class of faults, namely those caused by misunderstandings of how a description is interpreted. Some of these faults may be hard to detect. In addition, it can be used to evaluate the impact of potential semantic changes on agents' behaviour and performance.

6.1. Summary of Contributions

The contributions of this thesis are summarized below:

1. **Introducing syntactic mutation for Jason**: Chapter 3 systematically derives **33** syntactic mutation operators for Jason from the syntax of constructs that are commonly used in Jason agent programs. Then, it evaluates the representativeness of these operators (the ability of these operators to simulate real faults) by checking whether they can be used to generate faults in Jason agent programs written by students and experts. Results show that **9** operators can be used to simulate 77% of the identified faults in the constructs covered by my operator set; **4** out of these 9 operators are particularly useful. Finally, by comparing mutation operators for similar constructs of Jason and GOAL, **4** kinds of syntactic changes are found to have high potential to simulate faults in logic based agent programs.
2. **Introducing semantic mutation for Jason, GOAL and 2APL**: Chapter 4 derives respective sets of semantic mutation operators for Jason, GOAL and 2APL from different scenarios where semantic mutation for these languages is useful. Then, it evaluates the representativeness of the operators for Jason by checking whether students show the simulated misunderstandings when asked about the semantics involved. Results show that **nearly all** operators for Jason can simulate misunderstandings that students probably have. Finally, this chapter extracts **4** semantic change types from the previously derived operators, and suggests that semantic mutation operators for a logic based agent language can be systematically derived by applying these types into a set of semantic aspects.
3. **Evaluating the use of semantic mutation for Jason**: Chapter 5 checks the ability of semantic mutation for Jason to evaluate tests, the robustness to (and the reliability of) semantic changes, and the impact of semantic changes on the MAS execution time. Results show that **2** out of 11 semantic mutation operators for Jason have potential in generating hard-to-detect mutants; **4** out of 11 have potential in reducing the MAS execution time, particularly, **1** leads to significant reduction.

6.2. Future work

Although the study on mutation for MAS has been furthered by this thesis, they are still not enough. Sect.6.2.1 and 6.2.2 propose some future work on syntactic and semantic mutation for Jason.

6.2.1. Future Work on Syntactic Mutation for Jason

1. Deriving syntactic mutation operators for other Jason constructs and Jason project files that are not covered by this thesis, such as directives, if/for/while statements, environments and configuration files, and mutation operators for other platforms that are integrated with Jason, such as *Moise* and *CArtAgO* [16]. There is not much doubt that programmers may make errors in these places (Table 6 and 7 show some faults identified in directives for plan patterns and environment files). In particular, I believe the connections between different platforms and different components of Jason projects are places where faults can be easily introduced (such as wrong perception or definition of environmental actions as Table 6 and 7 show).
2. Further evaluating the representativeness of syntactic mutation operators for Jason. Due to data availability, this thesis analyzes a small number of small sized faulty Jason projects, many of which are developed by students who do not have much experience in agent programming; it only shows the representativeness of a small percentage of the derived operators. It is necessary to analyze more, larger-sized faulty projects developed by experienced agent programmers. In addition, simply observing how mutation operators can be used to generate real faults as this thesis does is not enough to evaluate the representativeness of mutation operators, the detection ratios of mutants and real faults should be compared when unit testing techniques for Jason are available (see point 2 in Sect.2.1.2 for some examples).
3. Evaluating the ability of syntactic mutation for Jason to generate hard-to-detect faults. This can be done by executing the generated mutants with tests that satisfy some criteria (e.g., coverage criteria) and then checking whether they can be killed (see point 3 in Sect. 2.1.2 for some examples).
4. Investigate the cost of syntactic mutation for Jason. The cost of syntactic mutation is determined by the number of mutants to be generated, the effort to exclude equivalent mutants, and the test execution time. The cost of syntactic mutation for Jason should be measured and compared with the cost of mutation for mainstream languages and other verification techniques (e.g., coverage criteria, model checking). If the cost is found too high, many existing techniques for reducing the cost of mutation [45] can be attempted.

6.2.2. Future Work on Semantic Mutation for Jason

1. Generating more semantic mutation operators for Jason from the following:

    - Semantic changes required in scenarios where semantic mutation for Jason is useful. Chapter 4 derives some semantic mutation operators for Jason from some sources of semantic changes required in some of the proposed scenarios. When more sources of semantic changes (e.g., more studies of faults in Jason projects) are available, more semantic mutation operators for Jason can be derived.
    - The method proposed in Chapter 4 to generate semantic mutation operators for logic based agent languages systematically.

2. Further evaluating the representativeness of semantic mutation operators for Jason. This can be done by asking more (experienced) Jason developers about the semantics targeted by semantic mutation operators, and also by studying faults in Jason projects, e.g., by checking whether a real fault can be simulated by modifying the semantics.
3. Further evaluating the ability of semantic mutation for Jason to generate hard-to-detect faults. This thesis conducts a limited study, where 2 semantic mutation operators show their ability to generate faults that cannot be detected by random testing of a small number of Jason projects. It is necessary to look for more hard-to-kill mutants by testing more Jason projects based on test evaluation criteria (e.g., coverage criteria, syntactic mutation).
4. Further investigating automatic search of interpretations that can improve agents' performance. I argue that logic based agent languages seem to be higher level than general-purpose ones, therefore, may have more need of variations to semantics in order to support all desired program types well. If someone argues that optimizations should preserve the original semantics and be done in code, this search is still useful, because it can provide some insight into how to optimize the code.

    Chapter 5 uses semantic mutation operators to guide the automatic search of interpretations: it checks a small number of individual semantic mutants. In the future it is useful to check combinations of semantic mutants, or extend semantic mutation operators so that each can generate many mutants (e.g., PSO can be modified to

generate every possible selection order of applicable plans). If there are a large number of possible mutants or their combinations to be checked, search-based optimization techniques [33] can be introduced. To facilitate changing the interpretation, meta-interpreters [56, 105] can be introduced.

References

[1] (10 September 2015). *IEEE Standards Definition Database*. Available: http://dictionary.ieee.org/.

[2] *Jason changelog*. Available: http://sourceforge.net/p/jason/svn/HEAD/tree/trunk/release-notes.txt.

[3] *Jason examples*. Available: http://jason.sourceforge.net/wp/examples/.

[4] *JGit documentation*. Available: https://eclipse.org/jgit/documentation/.

[5] *JRebel documentation*. Available: http://zeroturnaround.com/software/jrebel/learn/.

[6] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution," in *Genetic and Evolutionary Computation – GECCO 2004: Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004. Proceedings, Part II*, K. Deb, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1338-1349.

[7] S. F. Adra and P. McMinn, "Mutation Operators for Agent-Based Models," presented at the Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, 2010.

[8] H. Agrawal *et al.*, "Design of mutant operators for the C programming language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[9] J. Allen Troy Acree, "On mutation," Georgia Institute of Technology, 1980.

[10] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[11] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 402-411: ACM.

[12] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering,* vol. 39, no. 6, pp. 787-805, 2013.

[13] A. S. Banzi, T. Nobre, G. B. Pinheiro, J. C. G. Árias, A. Pozo, and S. R. Vergilio, "Selecting mutation operators with a multiobjective approach," *Expert Systems with Applications,* vol. 39, no. 15, pp. 12131-12142, 2012.

[14] S. Bardin *et al.*, "Sound and quasi-complete detection of infeasible test requirements," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, 2015, pp. 1-10: IEEE.

[15] C. Bernon, M. Cossentino, and J. Pavón, "Agent-oriented software engineering," *The Knowledge Engineering Review,* vol. 20, no. 2, p. 99, 2005.

[16] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," *Science of Computer Programming,* vol. 78, no. 6, pp. 747-761, 2013.

[17] R. H. Bordini, A. L. Bazzan, R. de O Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser, "AgentSpeak (XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, 2002, pp. 1294-1302: ACM.

[18] R. H. Bordini *et al.*, "A survey of programming languages and platforms for multi-agent systems," *Informatica (Slovenia),* vol. 30, no. 1, pp. 33-44, 2006.
[19] R. H. Bordini and J. F. Hübner, "Semantics for the Jason Variant of AgentSpeak (Plan Failure and some Internal Actions)," in *ECAI*, 2010, pp. 635-640.
[20] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.
[21] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Mutation Analysis, 2006. Second Workshop on*, 2006, pp. 11-11: IEEE.
[22] P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson, "Structuring BDI agents in functional clusters," in *International Workshop on Agent Theories, Architectures, and Languages*, 1999, pp. 277-289: Springer.
[23] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 100-109: IEEE.
[24] B. J. Clement, E. H. Durfee, and A. C. Barrett, "Abstract reasoning for planning and coordination," *Journal of Artificial Intelligence Research,* vol. 28, pp. 453-515, 2007.
[25] M. Dastani, "2APL: a practical agent programming language," *Autonomous agents and multi-agent systems,* vol. 16, no. 3, pp. 214-248, 2008.
[26] M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE transactions on software engineering,* vol. 27, no. 3, pp. 228-247, 2001.
[27] A. Derezińska, "A quality estimation of mutation clustering in c# programs," in *New Results in Dependability and Computer Systems*: Springer, 2013, pp. 119-129.
[28] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Information and Software Technology,* vol. 53, no. 10, pp. 1108-1123, 2011.
[29] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, 1994, pp. 220-229: IEEE.
[30] M. F. Granda, N. Condori-Fernández, T. E. Vos, and O. Pastor, "Mutation Operators for UML Class Diagrams," in *International Conference on Advanced Information Systems Engineering*, 2016, pp. 325-341: Springer.
[31] M. Grindal, J. Offutt, and J. Mellin, "On the testing maturity of software producing organizations," in *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, 2006, pp. 171-180: IEEE.
[32] B. J. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, 2009, pp. 192-199: IEEE.
[33] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR),* vol. 45, no. 1, p. 11, 2012.
[34] B. Henderson-Sellers, *Agent-oriented methodologies*. IGI Global, 2005.
[35] A. Herzig, E. Lorini, L. Perrussel, and Z. Xiao, "BDI logics for BDI architectures: old problems, new perspectives," *KI-Künstliche Intelligenz,* pp. 1-11, 2016.
[36] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic and stochastic finite state machines," *Journal of Systems and Software,* vol. 82, no. 11, pp. 1804-1818, 2009.

[37] K. V. Hindriks, "Programming rational agents in goal," in *Multi-Agent Programming*: Springer, 2009, pp. 119-157.
[38] K. V. Hindriks, F. S. De Boer, W. Van der Hoek, and J.-J. Ch Meyer, "Agent programming in 3APL," *Autonomous Agents and Multi-Agent Systems,* vol. 2, no. 4, pp. 357-401, 1999.
[39] K. V. Hindriks, F. S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, "Control structures of rule-based agent languages," in J. P. Müller, M. P. Singh and A. S. Rao(eds.), Intelligent Agents V (LNAI 1555), Springer-Verlag: Berlin, 1999, pp. 381-396.
[40] Z. Houhamdi, "Multi-agent system testing: A survey," *International Journal of Advanced Computer,* 2011.
[41] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering,* no. 4, pp. 371-379, 1982.
[42] M. N. Huhns and M. P. Singh, "Cognitive agents," *IEEE Internet computing,* vol. 2, no. 6, pp. 87-89, 1998.
[43] S. Hussain, "Mutation clustering," *Ms. Th., King's College London, Strand, London,* 2008.
[44] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 435-445: ACM.
[45] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering,* vol. 37, no. 5, pp. 649-678, 2011.
[46] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology,* vol. 51, no. 10, pp. 1379-1393, 2009.
[47] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654-665: ACM.
[48] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using UML statechart diagrams," in *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, 2003, pp. 296-300: South African Institute for Computer Scientists and Information Technologists.
[49] H. Kim, B. Choi, and S. Yoon, "Performance testing based on test-driven development for mobile applications," in *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, 2009, pp. 612-617: ACM.
[50] S. Kim, J. Clark, and J. McDermid, "The rigorous generation of Java mutation operators using HAZOP," *Informe técnico, The University of York,* 1999.
[51] K. N. King and A. J. Offutt, "A fortran language system for mutation‐based software testing," *Software: Practice and Experience,* vol. 21, no. 7, pp. 685-718, 1991.
[52] M. Kintis and N. Malevris, "Identifying more equivalent mutants via code similarity," in *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, 2013, vol. 1, pp. 180-188: IEEE.
[53] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE Transactions on Software Engineering,* 2017.

[54] K. Kravari and N. Bassiliades, "A survey of agent platforms," *Journal of Artificial Societies and Social Simulation,* vol. 18, no. 1, p. 11, 2015.
[55] P. Langley, J. E. Laird, and S. Rogers, "Cognitive architectures: Research issues and challenges," *Cognitive Systems Research,* vol. 10, no. 2, pp. 141-160, 2009.
[56] S. Leask and B. Logan, "Programming deliberation strategies in meta-APL," in *International Conference on Principles and Practice of Multi-Agent Systems*, 2015, pp. 433-448: Springer.
[57] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, 2009, pp. 220-229: IEEE.
[58] V. Lifschitz, L. R. Tang, and H. Turner, "Nested expressions in logic programs," *Annals of Mathematics and Artificial Intelligence,* vol. 25, no. 3-4, pp. 369-389, 1999.
[59] B. Logan, "A Future for Agent Programming," in *Engineering Multi-Agent Systems: Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers*, M. Baldoni, L. Baresi, and M. Dastani, Eds. Cham: Springer International Publishing, 2015, pp. 3-17.
[60] C. K. Low, T. Y. Chen, and R. Rónnquist, "Automated test case generation for BDI agents," *Autonomous Agents and Multi-Agent Systems,* vol. 2, no. 4, pp. 311-332, 1999.
[61] Y.-S. Ma and J. Offutt, "Description of class mutation operators for java," *November,* 2005.
[62] Y.-S. Ma and J. Offutt, "Description of method-level mutation operators for java," *Electronics and Telecommunications Research Institute, Korea, Tech. Rep.,* 2005.
[63] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: a mutation system for Java," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 827-830: ACM.
[64] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering,* vol. 40, no. 1, pp. 23-42, 2014.
[65] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at system and functional levels," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 110-119: IEEE.
[66] A. P. Mathur, *Foundations of Software Testing, 2/e*. Pearson Education India, 2008.
[67] J. A. McQuillan and J. F. Power, "A survey of UML-based coverage criteria for software testing," *Department of Computer Science. NUI Maynooth, Co. Kildare, Ireland,* 2005.
[68] S. Miles, M. Winikoff, S. Cranefield, C. D. Nguyen, A. Perini, P. Tonella, M. Harman, and M. Luck , "Why testing autonomous agents is hard and what can be done about it," in *URL http://www.pa.icar.cnr.it/cossentino/AOSETF10/docs/miles.pdf. AOSE Technical Forum*, 2010.
[69] T. Miller, L. Padgham, and J. Thangarajah, "Test coverage criteria for agent interaction testing," in *International Workshop on Agent-Oriented Software Engineering*, 2010, pp. 91-105: Springer.
[70] J. Možucha and B. Rossi, "Is Mutation Testing Ready to Be Adopted Industry-Wide?," in *Product-Focused Software Process Improvement: 17th International Conference,*

PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17, 2016, pp. 217-232: Springer.

[71] J. P. Müller and K. Fischer, "Application impact of multi-agent systems and technologies: a survey," in *Agent-oriented software engineering*, 2014, pp. 27-53: Springer.

[72] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[73] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah, "Testing in multi-agent systems," in *International Workshop on Agent-Oriented Software Engineering*, 2009, pp. 180-190: Springer.

[74] C. D. Nguyen, A. Perini, and P. Tonella, "Experimental evaluation of ontology-based test generation for multi-agent systems," in *International Workshop on Agent-Oriented Software Engineering*, 2008, pp. 187-198: Springer.

[75] C. D. Nguyen, A. Perini, and P. Tonella, "Ontology-based test generation for multiagent systems," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, 2008, pp. 1315-1320: International Foundation for Autonomous Agents and Multiagent Systems.

[76] C. D. Nguyen, A. Perini, P. Tonella, and F. B. Kessler, "Automated continuous testing of multi-agent systems," in *The fifth European workshop on Multi-agent systems*, 2007.

[77] C. D. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman, and M. Luck, "Evolutionary testing of autonomous software agents," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, 2009, pp. 521-528: International Foundation for Autonomous Agents and Multiagent Systems.

[78] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 1, no. 1, pp. 5-20, 1992.

[79] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability,* vol. 4, no. 3, pp. 131-154, 1994.

[80] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering,* vol. 20, no. 5, pp. 337-344, 1994.

[81] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software testing, verification and reliability,* vol. 7, no. 3, pp. 165-192, 1997.

[82] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*, 1993, pp. 100-107: IEEE Computer Society Press.

[83] A. J. Offutt, J. Voas, and J. Payne, "Mutation operators for Ada," Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.

[84] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," 1996.

[85] E. Omar, S. Ghosh, and D. Whitley, "Comparing search techniques for finding subtle higher order mutants," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 1271-1278: ACM.

[86] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, "Model-based test oracle generation for automated unit testing of agent systems," *IEEE Transactions on Software Engineering,* vol. 39, no. 9, pp. 1230-1244, 2013.

[87] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, 2015, vol. 1, pp. 936-946: IEEE.

[88] M. T. Patrick, "Mutation-Optimised Subdomains for Test Data Generation and Program Analysis," University of York, 2013.

[89] D. Pereira, E. Oliveira, N. Moreira, and L. Sarmento, "Towards an architecture for emotional BDI agents," in *Artificial intelligence, 2005. epia 2005. portuguese conference on*, 2005, pp. 40-46: IEEE.

[90] D. Poutakidis, L. Padgham, and M. Winikoff, "An exploration of bugs and debugging in multi-agent systems," in *International Symposium on Methodologies for Intelligent Systems*, 2003, pp. 628-632: Springer.

[91] A. S. Rao, "AgentSpeak (L): BDI agents speak out in a logical computable language," in *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996, pp. 42-55: Springer.

[92] A. S. Rao and M. P. Georgeff, "Modeling Rational Agents within a BDI-Architecture," *KR,* vol. 91, pp. 473-484, 1991.

[93] A. A. Saifan and H. A. Wahsheh, "Mutation operators for JADE mobile agent systems," in *Proceedings of the 3rd International Conference on Information and Communication Systems*, 2012, p. 16: ACM.

[94] S. Sardina and L. Padgham, "A BDI agent programming language with failure handling, declarative goals, and planning," *Autonomous Agents and Multi-Agent Systems,* vol. 23, no. 1, pp. 18-70, 2011.

[95] S. Savarimuthu and M. Winikoff, "Mutation operators for cognitive agent programs," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 2013, pp. 1137-1138: International Foundation for Autonomous Agents and Multiagent Systems.

[96] S. Savarimuthu and M. Winikoff, "Mutation operators for the GOAL agent language," in *International Workshop on Engineering Multi-Agent Systems*, 2013, pp. 255-273: Springer.

[97] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 297-298: ACM.

[98] D. Schuler and A. Zeller, "(Un-) covering equivalent mutants," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010, pp. 45-54: IEEE.

[99] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology,* vol. 81, pp. 19-35, 2017.

[100] S. F. Smith, G. J. Barlow, X.-F. Xie, and Z. B. Rubinstein, "Smart Urban Signal Networks: Initial Application of the SURTRAC Adaptive Traffic Signal Control System," in *ICAPS*, 2013.

[101] J. Thangarajah, S. Sardina, and L. Padgham, "Measuring plan coverage and overlap for agent reasoning," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 2012, pp. 1049-1056: International Foundation for Autonomous Agents and Multiagent Systems.
[102] T. Titcheu Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption," *An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption,* 2017.
[103] M. Trakhtenbrot, "Implementation-oriented mutation testing of statechart models," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 120-125: IEEE.
[104] Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?," in *Empirical Software Engineering and Verification*: Springer, 2012, pp. 194-212.
[105] M. Winikoff, "An AgentSpeak meta-interpreter and its applications," in *International Workshop on Programming Multi-Agent Systems*, 2005, pp. 123-138: Springer.
[106] M. Winikoff, "BDI agent testability revisited," *Autonomous Agents and Multi-Agent Systems,* pp. 1-39, 2017.
[107] M. Winikoff, "Novice programmers' faults & failures in GOAL programs," in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, 2014, pp. 301-308: International Foundation for Autonomous Agents and Multiagent Systems.
[108] M. Winikoff and S. Cranefield, "On the Testability of BDI Agent Systems," *J. Artif. Intell. Res.(JAIR),* vol. 51, pp. 71-131, 2014.
[109] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software,* vol. 31, no. 3, pp. 185-196, 1995.
[110] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, 1988, pp. 152-158: IEEE.
[111] M. Wooldridge, *An introduction to multiagent systems*. John Wiley & Sons, 2009.
[112] F. Wu, M. Harman, Y. Jia, and J. Krinke, "HOMI: Searching Higher Order Mutants for Software Improvement," in *International Symposium on Search Based Software Engineering*, 2016, pp. 18-33: Springer.
[113] Y. Yao and B. Logan, "Action-level intention selection for BDI agents," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, 2016, pp. 1227-1236: International Foundation for Autonomous Agents and Multiagent Systems.
[114] Y. Yao, B. Logan, and J. Thangarajah, "Intention selection with deadlines," in *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-2016)*, 2016, pp. 1700-1701.
[115] Y. Yao, B. Logan, and J. Thangarajah, "Robust execution of BDI agent programs by exploiting synergies between intentions," presented at the Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, Arizona, 2016.
[116] Y. Yao, B. Logan, and J. Thangarajah, "SP-MCTS-based intention scheduling for BDI agents," in *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, 2014, pp. 1133-1134: IOS Press.

[117] Z. Zhang, J. Thangarajah, and L. Padgham, "Automated Unit Testing for Agent Systems," *ENASE,* vol. 7, pp. 10-18, 2007.

Appendix 1. Test Specifications of the Selected Jason Projects

Each of the following table shows the input values, expected output and number of the tests for each selected Jason project. Oracles for checking the grey parts of the output are human ones.

S1: Election		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [1, 10]$	All the following must be satisfied. 1. Each agent has ended with voting. 2. The voting result is correct. 3. Each agent is able to solve its assigned problem and help another.
Initial locations of the agents (IL_a)	$\forall il_a \in IL_a$, $il_a = (x, y)$, where $x, y \in [0, s)$	
Location of the voting place (l_v)	$L_v = (x, y)$, where $x, y \in [0, s)$	
Total number of tests: 200		

S2: Flood Robots		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [2, 10]$	All the following must be satisfied. 1. All victims have been found and photographed. 2. No boats/UAV stopped working.
Maximum fuel in each boat (mf_b) and the UAV (mf_u)	$mf_b \in [4s - 4, 50]$; $mf_u \in [2s - 2, 30]$	
Extra fuel for safe return (f_r)	$f_r \in [0, 5]$	
Locations of victims (L_v)	$\forall l_v \in L_v$, $l_v = (x, y)$, where $x, y \in (0, s)$; $0 < \lvert L_v \rvert < s^2$	
Total number of tests: 200		

S3: Rescuing Agent		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [2, 10]$	All prisoners have been rescued successfully.
Locations of obstacles (L_o) and prisoners (L_p)	$\forall l_o \in L_o$, $\forall l_p \in L_p$, l_o and l_p are in the form of (x, y), where $x, y \in (0, s)$; $0 < \lvert L_o \rvert, \lvert L_p \rvert \leq s^2 * 0.2$ (round up to 1 if less than 1)	
Total number of tests: 50		

S4: Agent Combat		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [2, 10]$	All the following must be satisfied. 1. The game has ended with the monster/hunter being killed or the monster has reached the exit. 2. No fight is allowed at home/exit. 3. The monster's HP can be recovered at home. 4. The monster and hunter have behaved normally in fight.
Initial locations of the monster's home (l_h), the exit (l_e), the monster (il_m) and the hunter (il_h)	l_h, l_e, il_m, il_h are in the form of (x, y), where x, y $\in$ [0, s); $l_h \neq l_e$; $l_e \neq il_m$	
Maximum HP of the hunter (mhp_h) and the monster (mhp_m)	mhp_h, $mhp_m \in [100, 200]$	
Attack and defence of the hunter (a_h, d_h) and the monster (a_m, d_m)	a_h, d_h, a_m, $d_m \in (0, 200]$ $0 < a_h - d_m \leq mhp_m$ $0 < a_m - d_h \leq mhp_h$	
Speed of the hunter (s_h) and the monster (s_m)	s_h, $s_m \in [50, 150]$	
HP for safe escape (hp_e)	$hp_e \in [0, mhp_h]$	
Total number of tests: 200		

S5: Protocol Communication		
Input values	**Range**	**Expected output**
N/A	N/A	A message has been printed successfully through a serial communication channel
Total number of tests: 10		

S6: Supply Chain Integration		
Input values	**Range**	**Expected output**
N/A	N/A	The file has been processed in a correct way.
Total number of tests: 10		

S7: Online Store		
Input values	**Range**	**Expected output**
Stock availability (sa)	sa $\in$ {true, false}	The inventory agent has sent out the requested item.
Total number of tests: 20		

E1: Cleaning Robots		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [1, 10]$	All garbage have been picked and burned.
Initial locations of the cleaning robot (il_c) and the incinerator robot (l_i)	il_c and l_i are in the form of (x, y), where $x, y \in [0, s)$	
Locations of pieces of garbage (L_g)	$\forall l_g \in L_g$, $l_g = (x, y)$, where $x, y \in [0, s)$; $0 < \lvert L_g \rvert \leq s^2$	
Total number of tests: 100		

E2: Domestic Robot		
Input values	**Range**	**Expected output**
Drinking limit (dl)	$dl \in [1, 10]$	All the following must be satisfied. 1. The robot is not carrying beer. 2. The robot has told the owner that he/she has reached dl. 3. The robot has checked the current time requested by the owner. 4. dl = ib + db - rb, where db is the beer delivered by the supermarket and rb is the remaining beer in the fridge.
Map size (s x s)	$s \in [1, 10]$	
Initial beer in the fridge (ib)	$ib \in [0, 10]$	
Initial locations of the robot (il_r), the fridge (l_f) and the owner (l_o)	il_r, l_f and l_o are in the form of (x, y), where $x, y \in [0, s)$	
Total number of tests: 200		

E3: Mining Robots		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [16, 30]$	All the following must be satisfied. 1. All mines have been collected. 2. All mining robots have returned to the building robot. 3. The number of each type of mines was reduced one by one.
Location of the building robot (l_b)	$l_b = (x, x)$, where $x \in [0, s)$	
Locations of mines (L_m)	$\forall l_m \in L_m$, $l_m = (x, y)$, where $x, y \in [0, s)$; $\neg \exists\ l_m \in L_m$, $l_m = l_b$; $30 \leq \lvert L_m \rvert < s^2$	
Numbers of different types of mines required for construction (N_m)	$\forall n_m \in N_m$, $n_m \in [1, 50]$	
Total number of tests: 150		

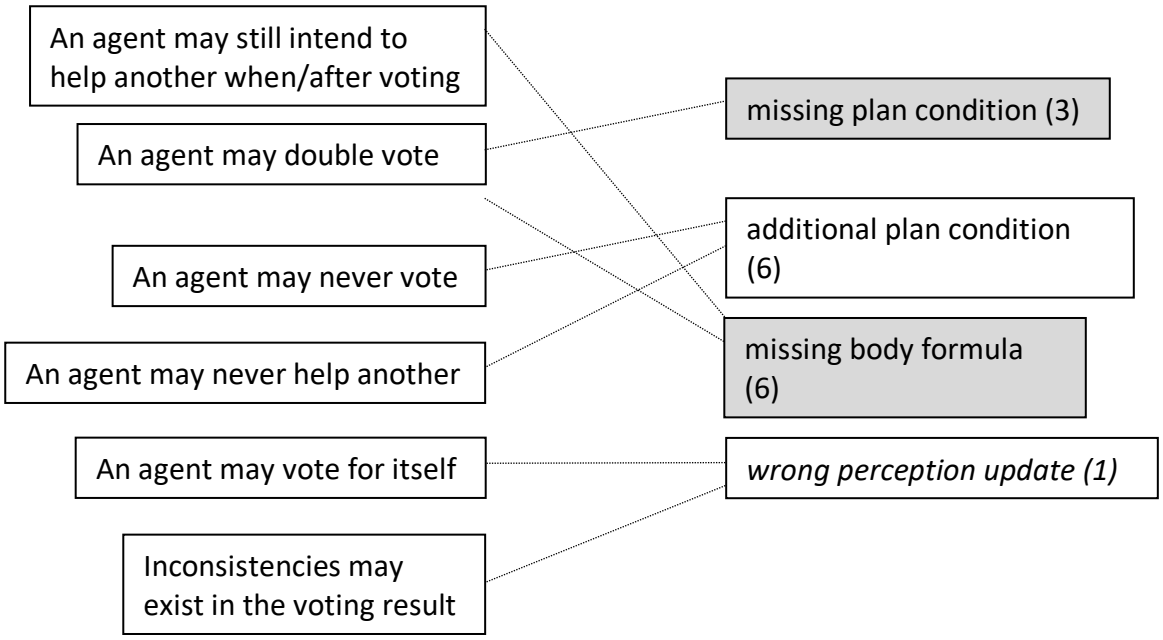
E4: Blocks World		
Input values	**Range**	**Expected output**
Initial stacks of blocks (IS)	IS and ES each is a set of lists and a partition of {“a”, “b”, “c”, “d”, “e”, “f”, “g”} representing all blocks; $1 \leq \lvert IS \rvert, \lvert ES \rvert \leq 3$	IS = ES
Expected stacks of blocks (ES)		
Total number of tests: 100		

E5, E6: Gold Miners, Gold Miners II		
Input values	**Range**	**Expected output**
Map size (s x s)	$s \in [16, 30]$	All the following must be satisfied. 1. All gold pieces have been collected. 2. Each miner can move normally.
Initial locations of the miners (IL_m)	$\forall il_m \in IL_m$, $il_m = (x, y)$, where $x, y \in [0, s)$	
Initial location of the depot (l_d)	$l_d = (x, y)$, where $x, y \in [0, s]$	
Locations of goal pieces (L_g) and obstacles (L_o)	$\forall l_g \in L_g$, $\forall l_o \in L_o$, l_g and l_o are in the form of (x, y), where $x, y \in [0, s]$; $\neg \exists\ l_g \in L_g, l_g = l_d$; $\neg \exists\ l_o \in L_o, l_o \in L_m, l_o = l_d$, or $l_o \in L_g$; $0 < \lvert L_g \rvert, \lvert L_o \rvert \leq s^2 * 0.2$	
Total number of tests: 300 for E5, 400 for E6		

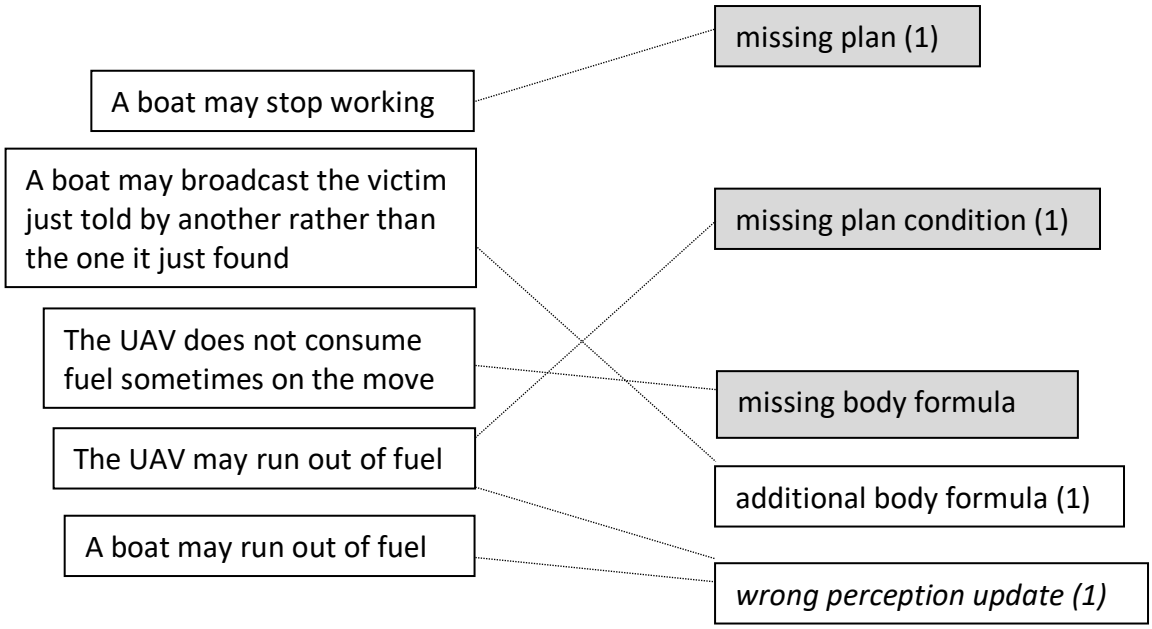
Appendix 2. Failures and Faults Detected by Testing the Selected Jason Projects

In each of the following figures, the left side includes the failures and the right side the faults. Each fault is followed by the number of its occurrences enclosed in parentheses. Faults on a dark background can be generated using my derived syntactic mutation operators for Jason, and those in italic are not in the constructs covered by my operators. Some more specific faults are separated from their parents by "–" as well as the numbers of their occurrences.

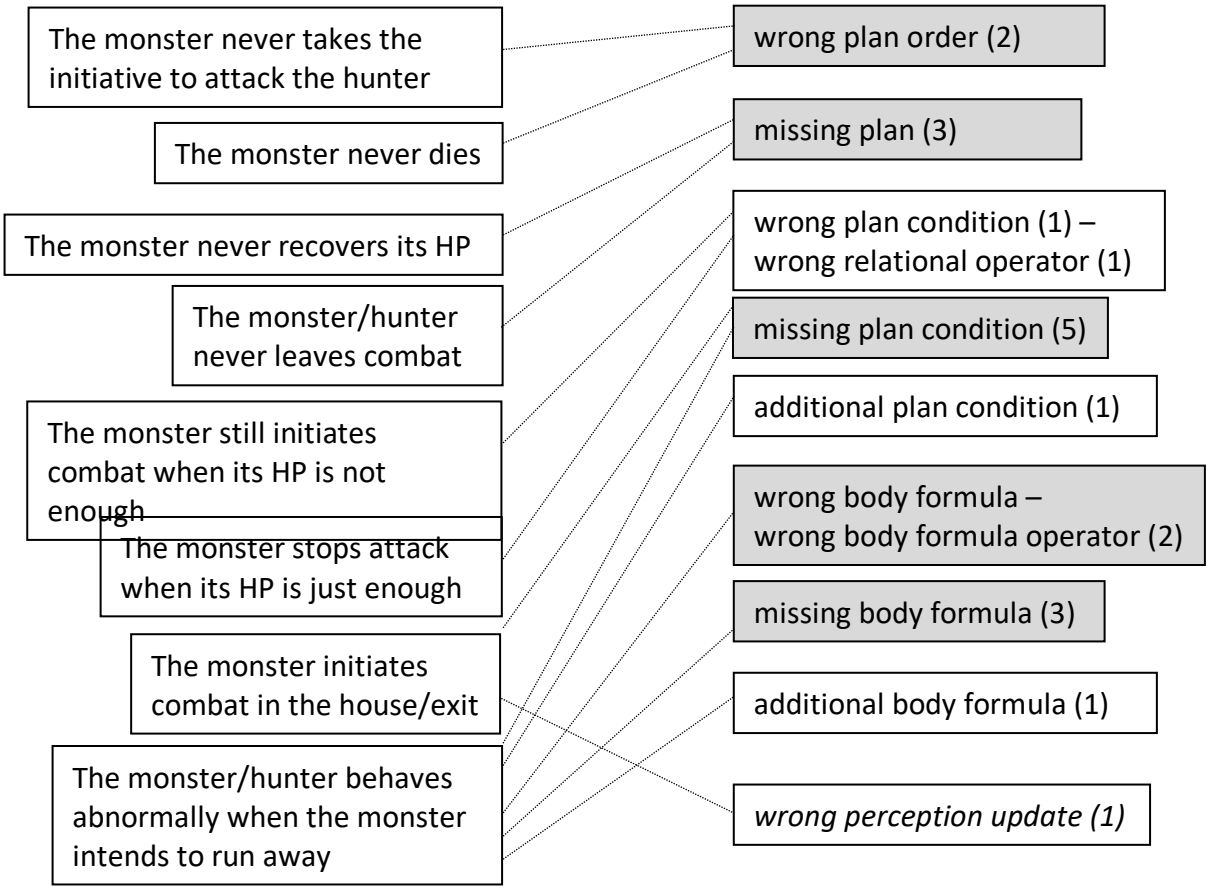
S1: Election



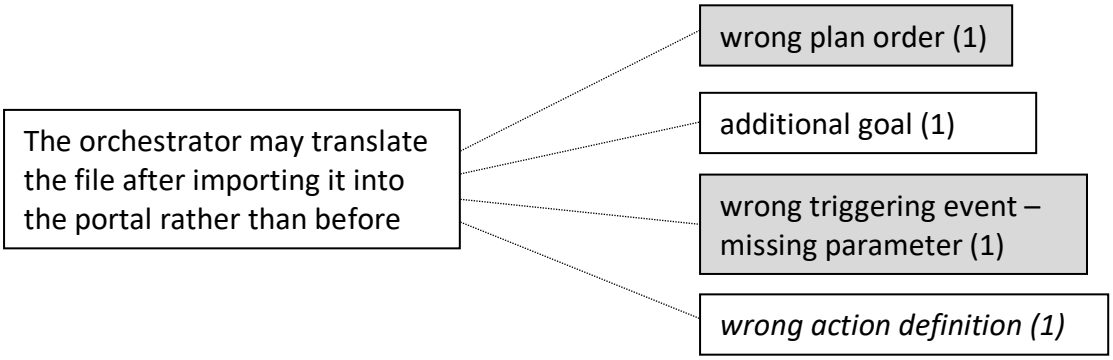
S2: Flood Robots



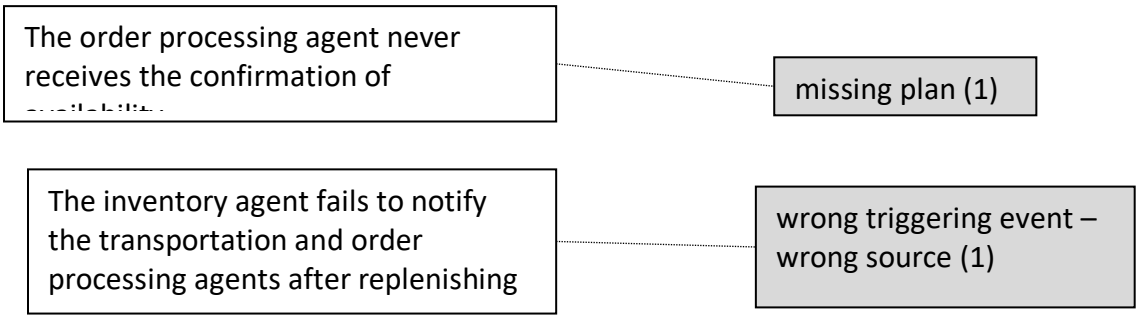
S4: Agent Combat



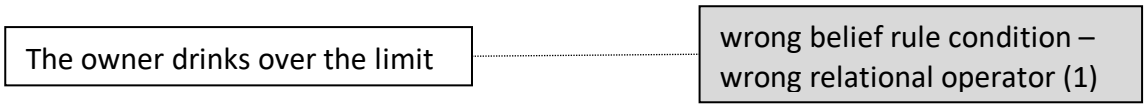
S6: Supply Chain Integration



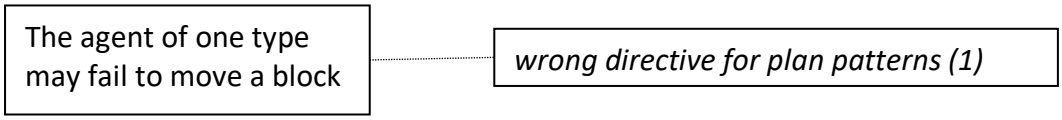
S7: Online Store



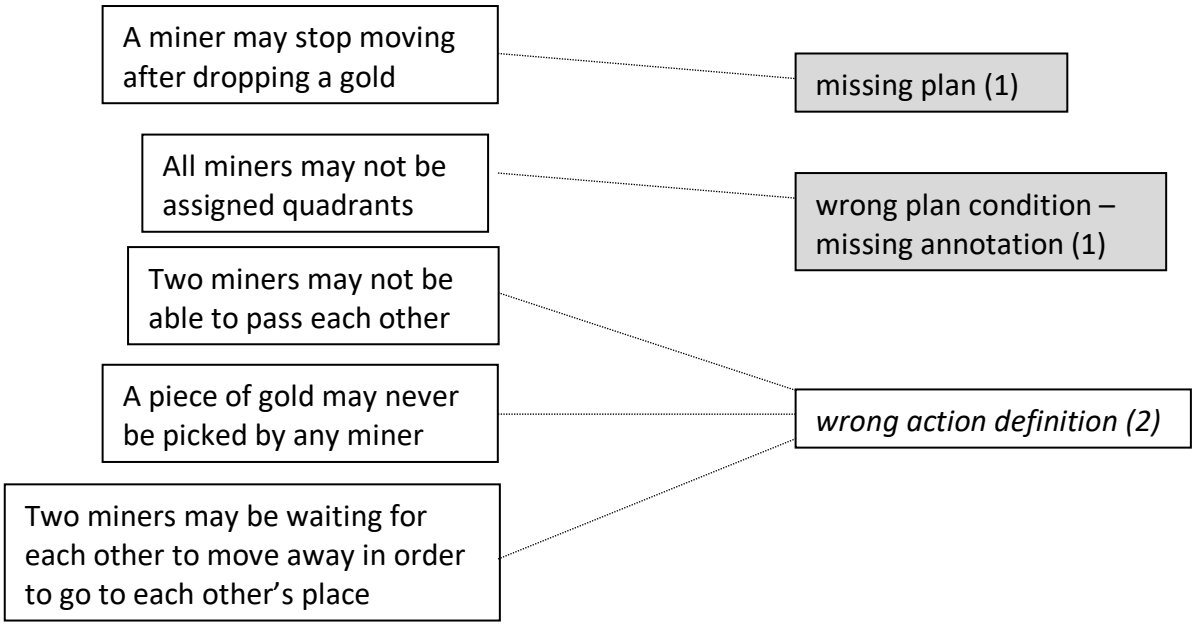
E2: Domestic Robot



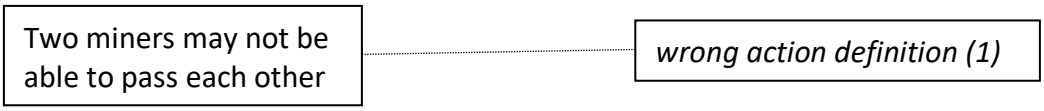
E4: Blocks World



E5: Gold Miners



E6: Gold Miners II



Appendix 3. The detected faults whose simulation by mutation is hard

Fault ID	Fault Category	Project	Faulty block	Correction	Why it is hard to simulate this fault
1	Additional goal	S6	!execute(file).	~~!execute(file).~~	Possible goals to be achieved by the agent need to be known.
2	Wrong plan condition	S4	+!search(exit): at(hunter) & lifeIs(Value) & Value > 0	+!search(exit): at(hunter) & lifeIs(Value) & Value > 9	Ranges of variables need to be known.
3	Additional plan condition	S1	+help(N) : not at(neutralag,N)	+help(N) : ~~not at(neutralag,N)~~	Possible beliefs to be added into the belief base need to be known.
4	Additional plan condition	S1	+help(N) : not at(ethicalag,N)	+help(N) : ~~not at(ethicalag,N)~~	
5	Additional plan condition	S1	+help(N) : not at(unethicalag,N)	+help(N) : ~~not at(unethicalag,N)~~	
6	Additional plan condition	S1	+voteTime(true): not at(neutralag,votePlace)	+voteTime(true): ~~not at(neutralag,votePlace)~~	
7	Additional plan condition	S1	+voteTime(true): not at(ethicalag,votePlace)	+voteTime(true): ~~not at(ethicalag,votePlace)~~	
8	Additional plan condition	S1	+voteTime(true): not at(unethicalag,votePlace)	+voteTime(true): ~~not at(unethicalag,votePlace)~~	
9	Additional plan condition	S4	+!search(robot) : not combat(Ag)	+!search(robot) : ~~not combat(Ag)~~	
10	Additional body formula	S2	?reportedVictim(VX,VY)	~~?reportedVictim(VX,VY)~~	Possible body formulae the agent can execute need to be known.
11	Additional body formula	S4	!search(exit)	~~!search(exit)~~	