A Behaviour-Driven Approach for Testing Requirements via User and System Stories in Agent Systems

Sebastian Rodriguez  
RMIT University  
Melbourne, Australia  
sebastian.rodriguez@rmit.edu.au

John Thangarajah  
RMIT University  
Melbourne, Australia  
john.thangarajah@rmit.edu.au

Michael Winikoff  
Victoria University of Wellington  
michael.winikoff@vuw.ac.nz

ABSTRACT
Testing is a critical part of the software development cycle. This is even more important for autonomous systems, which can be challenging to test. In mainstream software engineering, Behaviour-Driven Development (BDD) is an Agile software development practice that is well accepted and widely used. It involves defining test cases for the expected system behaviour prior to developing the associated functionality. In this work, we present a BDD approach to testing the behavioural requirements of an agent system specified via User and System Stories (USS). USS is also based on established Agile processes and is shown to be intuitive and readily mapped to agent concepts. More specifically we extend USS so that they can be used for testing, and develop a behaviour-driven testing framework based on USS. We show how test cases can be developed, and how to evaluate the test cases using a state-of-the-art mutation testing system, PITest, which we have integrated into our test framework. A key feature of our work is that we leverage a range of state-of-the-art development tools, inheriting the rich set of features they provide.

KEYWORDS
User Stories; System Stories; AOSE; Behaviour-Driven Development; Test-Driven Development Testing; Requirements testing

1 INTRODUCTION
Testing and validation is a critical aspect of any form of software development, and even more so for autonomous systems to build trust. To this end, there have been a number of testing approaches presented over the years by the agent oriented software engineering (AOSE) community [2, 8, 9, 17, 19]. In this work we contribute to that body of work and present a novel behaviour-driven approach to testing the requirements of an agent system that are specified via User and System Stories (USS) [29].

The alignment of AOSE techniques with mainstream software engineering is also an important consideration for our community, to not only support the adoption of AOSE, but also to inherit and build on well-established, demonstrated, and accepted techniques.
(http://cucumber.io), to specify USS and develop test cases to validate the requirements implemented in the SARL [28] agent programming language (http://www.sarl.io). The Cucumber tool also enables the generation of developer-friendly reports. As the mutation testing system, we use PTest (http://pitest.org), which is state of the art.

A key feature of our approach is the use and integration of modern software tools that are widely adopted in the industry as the gold standard. This provides a number of benefits (e.g. developer familiarity, reliable and usable tools) which we discuss in Section 5.

2 BACKGROUND AND RELATED WORK

In this section we provide a brief introduction to behaviour driven development which is well established in mainstream software engineering. We recap the notion of User and System stories as presented in [29]; summarize the behaviour-driven development approach; and provide an introduction to the goals and plans in agent systems which we build on in our work.

2.1 User and System Stories (USS)

A user story [10] is an informal, natural language specification of the requirements of one or more features of a software system. It is a popular tool used in Agile software development to capture a simplified description of requirements from an end-user perspective.

Rodriguez et al. [29] introduced the concept of a system story which further refines user stories from the system’s perspective. They showed how user and system stories (USS) can be used to manage requirements in AOSÉ.

USS are represented using the Gherkin syntax template: As (role), I want to (do something), So that (reason). Scenario 1 (first 4 lines) shows an example user story of a drone operator coordinating drones in a search and rescue operation.

**Scenario 1: Example user story and acceptance criterion**

<table>
<thead>
<tr>
<th>Feature: Search for victims</th>
</tr>
</thead>
<tbody>
<tr>
<td>As Drone Operator,</td>
</tr>
<tr>
<td>I want to assign drones areas to explore,</td>
</tr>
<tr>
<td>So that they find victims and notify me.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario: Drone is idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given drone is at base</td>
</tr>
<tr>
<td>When it is assigned an area to explore</td>
</tr>
<tr>
<td>Then it begins exploring that area autonomously</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario: Drone is busy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given drone is currently exploring</td>
</tr>
<tr>
<td>When it is assigned an area to explore</td>
</tr>
<tr>
<td>Then it adds new area to its exploration queue</td>
</tr>
</tbody>
</table>

Each story also contains a set of acceptance criteria, which are a set of statements that identify the intended behaviour of the system under various scenarios. These are captured by the Given/When/Then format which is part of the Behaviour Driven Development (BDD) approach - given the necessary preconditions for the system to execute the behavior being described, when the triggers of the behavior are activated, then these results are effected by the successful execution of that behavior. Figure 1 illustrates acceptance criteria for the explore area behaviour of a drone, under two scenarios: drone is idle and drone is busy.

2.2 Behaviour Driven Development

Behaviour-driven development (BDD) [29] is an Agile software development practice that encourages collaboration between the technical system developers and the domain experts, who are often non-technical. BDD encourages rapid prototyping and iterations by continuously refining user requirements into smaller units that are readily implemented and tested in the system. In fact, BDD is a form of test-driven development where test cases are defined before developing the associated new functionality. In BDD the test cases are based on system behaviour, specified using a domain-expert friendly format, such as user and system stories.

The first phase of the (iterative) BDD process is to define the specification of a desired feature of the system in terms of user stories. The second phase involves the development of acceptance criteria collaboratively so all the parties agree on the expected behaviours. In the third phase test cases are created for the behaviours based on the agreed acceptance criteria. Finally, the behaviour is implemented with the test cases guiding the development to ensure the acceptance criteria are met. An additional stage in any test-first development cycle is to refactor the production code to improve its design. This can be done with the knowledge that the behaviour required will be tested.

There are a number of tools that support the BDD process. Of particular importance to our testing framework is Cucumber. Cucumber allows the specification of USS in the Gherkin language and supports the translation of acceptance criteria into test cases. It provides support for most modern programming languages (e.g Java, Python, C++) and also supports the generation of reports in HTML, JSON and other formats. In Section 4 we describe how we integrate Cucumber and other software tools in developing our test framework.

2.3 Goals and Plans

We extend USS to capture key properties of goals, plans, beliefs and percepts (see Section 3). In doing so we build on the prior conception of these entities in the literature.

Percepts are part of the interface between an agent and its environment [21]: a percept is (possibly processed) information from the environment to the agent that triggers a response.

Goals, plans, and beliefs are considered in the context of the Belief-Desire-Intention (BDI) [5, 25–27] framework, which conceptualises autonomous agents in terms of goals, beliefs, and plans.

A belief is simply information represented in the agent that reflects what it believes to be true about the world and/or itself. Beliefs can be represented in a range of forms, often, inspired by logic programming, as sets of terms.

Plans are typically defined (e.g. [4, 18, 23, 24, 34]) in terms of an event that they handle (which might be adding or removing a goal, or adding or removing a belief), a context condition which indicates whether the plan is applicable in the current situation,

1It really should be “BDIP”: Plans are a key component of the framework [5]

2We focus here on the design and implementation view, where we have goals rather than desires, and where the focus is more on plans, with intentions merely being partially instantiated plans at runtime.
and a plan body that is executed over time after the plan is selected, and which may include a range of step types, including posting events or creating goals, which may trigger sub-plans.

Goals are more complex. There are a range of types of goals in the literature [6, 11, 13, 33]. However, implemented platforms typically only support a limited range of goal types, with achievement goals being universally supported, and maintenance goals being commonly supported [6]. An achievement goal specifies a condition to be achieved by performing some tasks, and a maintenance goal specifies a condition that must be maintained by performing tasks to re-establish the condition, if it becomes false.

Goals have a number of key properties [35]: they are persistent (dropped only when they are considered fulfilled or impossible to fulfill), unachieved (dropped when they are fulfilled), possible (dropped when they are impossible to fulfill), known (i.e., represented within the agent), and consistent (i.e., an agent should not simultaneously attempt to pursue inconsistent goals).

These properties are reflected in each achievement goal being defined in terms of two key conditions: the success condition \( s \) when it is considered to be fulfilled, and the failure condition \( f \) when it is considered to be impossible to fulfill. For maintenance goals, these are defined in terms of the condition \( m \) that it needs to maintain, and the condition \( f \) when it is considered to be impossible to fulfill.

### 3 EXTENDING USS FOR BDD

In this section we extend USS described in [29] to allow information required to test the system to be specified. Different information is required for different entity types, and so we introduce a classification of System Stories. A System Story can be classified as being: a Goal Story (which describes a goal that the agent wants to achieve or maintain), a Plan Story (which describes a plan to be used to achieve a goal or recover a maintenance condition), a Belief Inference Story (which defines relationships between agent beliefs, and how they should be inferred), or a Perception Story (which defines how to handle a particular percept). Although these additional concepts are also associated with design and implementation, the system stories that use them are requirements (see examples later in this section). Rodriguez et al. [29] define mappings from stories to agent concepts. For instance, a What ("I want to …") element of a System Story maps to either an action or a goal.

The remainder of this section defines the additional information required for each agent concept. This additional structure is defined using annotations in the Gherkin language. The additional structure relates primarily to the acceptance criteria, although we also tag a story with its type (e.g., @perception).

**Goal Stories**: describe a goal that an agent wants to achieve or maintain in order to satisfy another goal or handle a perception, for example, Explore Area. Goal Story’s acceptance criteria need to identify and capture what does the goal mean in the domain or in the expert’s view. For instance, in a search and rescue domain, one of the goals of the system will be to Explore Area. But, what does it mean to Explore Area? Under which conditions can the system effectively consider that a given area has been explored to an adequate degree? Since Explore Area is an achievement goal, this condition corresponds to the goal’s success condition [35]. Using the Gherkin language introduced in Section 2.1, we can capture this requirement as in Scenario 2.

Note that the last two lines of Scenario 2 ("When ...") are actually a pattern that is the same for all @goal-success conditions. They define the Given condition (e.g., "I believe current_area_explored is greater than 95%") as being a goal success condition by indicating that the result of the condition being met is that the goal should be considered successful. It is also possible to define syntactic sugar that expands a more concise syntax of the form "@goal-success I believe current_area_explored is greater than 95%" to the syntax of Scenario 2.

In addition to capturing the goal-success condition, for an achievement goal, we can also define the following additional information:

- **goal-context**: the condition under which the goal is able to be adopted (e.g., a drone is not able to adopt the goal Explore Area if the battery level is low).
- **goal-failure**: the condition under which the goal fails (e.g., if there is a system failure).
- **goal-plan**: one or more plans that are relevant to achieve this goal and the condition under which each plan is applicable (See Scenario 3 for an example).

**Scenario 2: Explore Area goal success acceptance criteria**

<table>
<thead>
<tr>
<th>Scenario: Explore Area goal success acceptance criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>@goal-success</td>
</tr>
<tr>
<td><strong>Scenario</strong>: Goal success</td>
</tr>
<tr>
<td><strong>Given</strong>: I believe current_area_explored is greater than 95%</td>
</tr>
<tr>
<td><strong>When</strong>: I evaluate current_goal success</td>
</tr>
<tr>
<td><strong>Then</strong>: goal success is true</td>
</tr>
</tbody>
</table>

A similar approach can be used to capture the requirements for Maintenance goals (e.g., "Maintain Battery Level"). Specifically, in addition to capturing the goal-context (the same as for achievement goals), we also would capture: (i) the condition to be maintained (goal-condition): should this condition become false, it will trigger an attempt to recover (e.g., triggering a recharge plan should the battery level become too low); (ii) a single recovery plan (goal-plan) that is used to recover; and (optionally) (iii) the condition (goal-failure) under which the goal is considered impossible to maintain and should no longer try to recover, e.g., if no charging station is available, or there is a system failure.

**Scenario 3: PlowSweep Plan for the Explore Area goal**

<table>
<thead>
<tr>
<th>@goal-plan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario</strong>: HIGH priority area</td>
</tr>
<tr>
<td><strong>Given</strong>: I believe current_area_priority is HIGH</td>
</tr>
<tr>
<td><strong>When</strong>: I adopt the ExploreArea goal</td>
</tr>
<tr>
<td><strong>Then</strong>: plan PlowSweep is applicable</td>
</tr>
</tbody>
</table>

Plan stories: each describes a plan to be used to recover a maintenance condition or achieve a goal (e.g., plow sweep story). In writing plan stories, a number of acceptance criteria scenarios might be provided, each defining a behaviour that the system should exhibit.

---

1. It should be a single condition: a goal with more than one success condition may be an indication that the goal should be split into two goals.
2. The complete story is in the supplementary information.
and that can be tested. Following a behaviour-driven approach, these acceptance criteria should capture "high level" observable behaviour of the agents and avoid step-by-step description of the plan’s actions. For example, Scenario 4 shows an acceptance criterion that indicates that when the Drone agent explores an area using plow sweep, it should create a path with certain waypoints (the \(<x>, <y>\) etc. are Gherkin syntax that allows a story to refer to a table of specific values, e.g. see Scenario 5. See supplementary material for the full feature definition.).

**Scenario 4: PlowSweep Plan Story**

<table>
<thead>
<tr>
<th>Scenario: create Plow waypoints</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Given</strong> an area at (&lt;x&gt;, &lt;y&gt;) of (&lt;height&gt; by &lt;width&gt;)</td>
</tr>
<tr>
<td><strong>When</strong> I start plan</td>
</tr>
<tr>
<td><strong>Then</strong> I should create path with ‘(&lt;waypoints&gt;)’</td>
</tr>
</tbody>
</table>

**Belief Inference Stories**: in some scenarios it is important to capture the relationships between beliefs and how we can infer the value of one belief based on another (e.g. inferring Drone battery level of LOW from a battery charge reading of 25%). We use Belief Inference stories, for example Scenario 5, to capture this understanding in terms of test cases.

**Scenario 5: PlowSweep Plan achieves Explore Area**

<table>
<thead>
<tr>
<th>@belief-infer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario:</strong> Battery calculation</td>
</tr>
<tr>
<td><strong>Given</strong> I believe battery_charge is (&lt;charge&gt;)</td>
</tr>
<tr>
<td><strong>When</strong> I query belief battery_level</td>
</tr>
<tr>
<td><strong>Then</strong> I should believe battery_level is (&lt;level&gt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples:</th>
<th>charge</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>FULL</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>HIGH</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>LOW</td>
</tr>
</tbody>
</table>

**Perception Stories** capture how the agent responds to perceptions. For example, when an agent is assigned a new area to explore, an acceptance criterion (see Scenario 6) could indicate that the agent should respond by adding the area to be explored to a queue, acknowledge the assignment, and then start exploring the area, and searching for victims.

**Scenario 6: New area assignments are queued and relevant goals started**

<table>
<thead>
<tr>
<th>@perception-plan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario:</strong> New assignment</td>
</tr>
<tr>
<td><strong>When</strong> I receive a new AreaAssignment</td>
</tr>
<tr>
<td><strong>Then</strong> I add the area to the exploration queue</td>
</tr>
<tr>
<td><strong>And</strong> I acknowledge the assignment</td>
</tr>
<tr>
<td><strong>And</strong> I should start ExploreArea</td>
</tr>
<tr>
<td><strong>And</strong> I should start DetectVictim</td>
</tr>
</tbody>
</table>

4 TESTING FRAMEWORK

In this section we present the testing framework developed to support our proposal.

4.1 Overview and tools

In order to support this approach we developed a software framework that enables the development team to use these techniques effectively. The framework integrates and leverages state-of-the-art testing tools widely used in the industry.

As discussed in Section 2.2, the process of BDD is to capture stories, write test cases, implement the test cases, then implement the desired functionality (using the tests to assess the implementation), and finally refactor the code to improve its design. Our framework supports the central steps of implementing tests, and using them to test the system.

An overview of the main components of the framework is presented in Figure 1. Our framework and process are supported by a set of professional tools such as an IDE (e.g. SARL’s IDE is based on Eclipse), build and dependency management tools (e.g. maven), and other software engineering tools.

User and system stories are written in the Gherkin language (see Section 2.1). These stories are used to create executable tests (see Section 4.2), and to this end we integrated the Cucumber and JUnit frameworks, which are widely-used and supported testing frameworks for Java. In addition to these frameworks, we use Mockito\(^5\) to isolate particular components and verify interactions. For instance, verify if a plan calls the appropriate actions.

Once test cases have been created, the production code is (progressively and iteratively) implemented in SARL [28] following the approach presented in [29]. Due to its generic and highly extensible architecture, SARL is able to integrate new concepts and features quickly. In order to faithfully capture the properties of goals and plans, we extended SARL to incorporate a goal-reasoning engine. We intend to contribute this extension to the SARL open source project\(^6\). Features are mapped to agent concepts and then to code, using this extended version of SARL.

\(^5\)Mocking framework for Java https://site.mockito.org/.
\(^6\)https://github.com/sarl/
def explorationRate

uses SearchRescueBeliefs, DroneState

def context: boolean {  
    batteryLevel >= BatteryLevelLOW
}

def success: boolean {  
    explorationRate(currentArea) >= 0.95
}

def failure: boolean {  
    systemFailure
}

Figure 2: SARL implementation of the ExploreArea goal

Figure 2 shows how goals in SARL are implemented, by declaring them as skills of the agent. Goals can require specific capacities from the agent that is going to adopt them. This is done by declaring required capacities via the ‘uses’ keyword. For instance, ExploreArea requires from the agent to have SearchRescueBeliefs and DroneState beliefs in line 2 in Figure 2. Beliefs definitions are then brought into the scope which enables to query them directly. For instance, in line 8 in order to determine if the goal has been achieved, the goal queries the belief of the exploration rate for the currentArea. This approach make the code easy to read and simple to link the requirements specification.

As feature requirements are implemented, they are tested, producing an HTML report (see Section 4.3). SARL is interoperable with Java and its ecosystem, seamlessly integrating with other tools and APIs such as JUnit and Cucumber.

In order to assess the effectiveness of the test scenarios, we use mutation testing (see Section 4.4) using PITest. This provides feedback both in general on the quality of the test suite, but also specifically on areas where additional test cases may be required.

4.2 Implementing Tests

Following a behaviour-driven approach acceptance criteria are translated into executable tests to drive the feature development.

To do so, Cucumber requires us to create Step definitions that will link the feature specification (USS) in Gherkin with executable tests and assertions. This link is created by annotating methods with the following annotations:

- @Given indicates the pre-condition of the scenario, it will configure beliefs with expected values for the scenario;
- @When indicates the trigger of the behaviour, it will fire the agent’s behaviour using perceptions, message or belief updates; and
- @Then indicates desired post conditions, it will use assertions from JUnit or Mockito to verify that the behaviour’s outcome complies to the specification.

SARL’s object orientation support enables us to define these steps using the annotation provided by Cucumber’s JVM implementation.

class NewAreaAssignmentTestSteps {
@Inject
var agt: GoalTestingAgent

var area = new Area(0f, 0f, 10f, 10f, Priority.HIGH)

@When("I receive a new AreaAssignment")
def perceive_assignment{
this.agt.perceive(new AreaAssignment("Alpha", area))
}

@Then("I should start DetectVictim")
def post_detect{
verify(this.agt.goals).post(any(DetectVictim))
}

Figure 3: Test definition in SARL using Cucumber

Additionally, a new testing framework for SARL agents has been developed to facilitate the testing of SARL code using JUnit. This set of extensions allows to mock agents and other agent concepts.

As an example, Figure 3 shows a snippet of the step definitions for Scenario 6 (where a new area is assigned to a drone). The steps are defined as annotated methods in a class, in this case NewAreaAssignmentTestSteps. In line 3, a mock agent is created - GoalTestingAgent is defined by SARL’s testing framework and offers functionalities to facilitate agent testing (e.g. perceive events, logging, etc.). Line 4, specifies a value for the area that will be used during the tests.

As Cucumber reads the scenario specification, it matches the text with code annotations. So, in scenario 6, the step When I receive a new AreaAssignment matches with the annotation in line 6, and will therefore execute perceive_assignment in line 7. In this step, using SARL’s testing extensions, we trigger the corresponding agent perception in the production code. The agent will perceive the AreaAssignment event containing the Area defined in line 4.

Similarly, the step I should start DetectVictim of scenario 6 will be matched to post_detect in line 11. Essentially, we verify that the DetectVictim goal is posted using Mockito’s assertions.

Using this same process we can implement the steps in Scenario 2 that define the success condition for goal ExploreArea as shown in Figure 4. The Given annotation allows us to define the pre-conditions of the scenario, including the type of goal we are evaluating (see line 5) and belief values required (e.g. the exploration rate of the area in line 14). Goal success is evaluated during the triggering step (i.e. When) in line 18. Finally, we assert that the value obtained is the expected outcome in line 22.

As shown, the mapping between the acceptance criteria specified in scenario format and the testing code is straightforward, given the framework and integration.

4.3 Running Tests

As the features are developed iteratively and tests are executed, reports are generated to support the developer. For example, consider
class ExploreAreaTestSteps {
    ... @Given("current_goal is ExploreArea")
    def current_goal {
        this.goal = new ExploreArea
        this.goal.owner = this.agt
    }
    @Given("I believe current_area_explored is greater than (int)%")
    def exploration_is_percent (rate: int) {
        val area = new Area(0f, 0f, 10f, 10f, Priority.HIGH)
        doReturn(area).when(this.agt.beliefs).currentArea
        doReturn(rate / 100f).when(this.agt.beliefs).explorationRate(any(Area))
    }
    @When("I evaluate current_goal success")
    def evaluate_goal_success {
        this.evalResult = this.goal.success
    }
    @Then("goal {word} is {word}")
    def evaluation_outcome (cond: String, outcome: String) {
        assertEquals(Boolean.valueOf(outcome), this.evalResult)
    }
}

Figure 4: Test step definition for ExploreArea goal

agent Drone {
    uses Logging, DefaultContextInteractions
    uses Goals, PlanSelectionConstraints
    uses SearchRescueBeliefs

    on Initialize {
        info("Drone Initialized")
        PlowSweep.handles(ExploreArea, new PlowSweep.Context)
        RandomWalk.handles(ExploreArea, new RandomWalk.Context)
    }

    on AreaAssignment {
        addAreaToExplore(occurrence.area)
        emit(new AreaAssignmentAccepted)
        post(new ExploreArea)
        // post(new DetectVictim) // Commented for illustration purposes.
    }
}

Figure 5: SARL Drone agent example

as the final step is not implemented yet. The step causing the issue is highlighted in red, in this case Then I should start DetectVictim. Immediately below (in the grey box) an explanation of the failing assertion is displayed. In this case, a Mockito assertion error is shown detailing that it was expected to receive an invocation of the post(DetectVictim) action\(^7\), but it was not called.

Once the missing step is implemented, successive executions of the test suite will report the scenario is passing.

4.4 Mutation Testing

The aim of our testing approach is to verify that the feature specifications are faithfully implemented in the production code, via the generated test cases. Hence, the strength of the verification is as strong as the test cases. Therefore, another important aspect of our approach is to be able to assess the quality of our test suite using mutation testing.

Mutation testing\[^{12}\] is a technique frequently used in software engineering to evaluate the quality of test suites. In mutation testing, the production code is modified in small ways, introducing a fault. This modified version, called a mutant, changes the behaviour of the system slightly. Then the test suite is executed against the mutant to confirm whether it detects the anomalous behaviour, usually

\(^7\)post(Goal) is provided by the new SARL goal-oriented agents API and posts new goals to the agent’s goal-reasoning engine.
I believe current_area_exploration_rate is less than 95%
I evaluate current_goal success
goal success is false
which results in lower line coverage results.

written by the developer, but also internal code that provides SARL
features, and is not under the programmer’s control. We therefore
consider the mutations for the DroneState components that are building blocks of the agent system. Let us
consider the mutations for the DroneState belief shown in Figure 9.

Whilst most of them are killed, two are reported as not covered
in lines 32 and 36. They are related to beliefs system_failure and
battery_charge. These beliefs are linked to perceptions of the agent’s
body and not the result of any inference process. We refer to these
beliefs as ground beliefs. So, when the belief value is mutated (e.g.
incorrectly returning a battery charge of 0 when the battery still
has charge), the issue is not that the agent reasoning is incorrect,
but that the sensor reporting battery charge is erroneous. Therefore,
these tests should be done at the “battery driver level” to ensure
hardware measurements are correct. Identifying and understanding
the agent’s ground beliefs is critical to determine the boundaries of the
agent system and key integration points to test.

Acceptable behaviours despite mutation survival. Mutation framework
perform several mutation operations on the production code
using heuristics with the objective of creating mutants that will
survive the test suite. Although these heuristics are improving con-
tantly, in certain cases, mutations do not modify the behaviour of
the system. For instance, consider the case in which we want to en-
sure that our drone does not go outside the boundaries of the area. In
the bottom corner, we might achieve this with if (x < 0) then x = 0.
If the mutation changes the condition from x < 0 to x ≤ 0 yielding
results were low, and how to improve them. Specifically, for line 33,
PITest produced four mutants related to the goal’s success condition
(see Scenario 2). Scenario 2 states that when the Explore Area goal is
evaluated for success, if the area explored is believed to be more
than 95%, then the goal should be deemed to have succeeded. The
surviving mutant highlights that we are missing a story: replacing
the test (> 95%) with just “True” means that the goal is dropped
any time it is evaluated for success. However, the story does not
behave for this - it just states that the goal should be deemed successful
in a certain situation, and is missing the converse statement that
the goal should not be deemed successful in any other situations.
This could be addressed by adding a story that indicates that when
the goal is assessed for success, if the area covered is not greater
than 95%, then the goal should not be considered to have succeeded.
Therefore, introducing Scenario 7 covers this situation and kills
the mutant. A similar situation is shown in line 38 for the failure
condition and line 26 for the context condition.

Scenario 7: Additional Scenario for ExploreArea to kill suc-
cess "True" mutant

This highlights a very important feature of our approach. We
can relate and trace back issues in the system’s behaviour directly
to particular requirement specifications. These specifications can
then be shown and discussed directly with the experts using their
own domain language. Refinements in the requirements can be
implemented iteratively as discussed in the previous sections.

Identifying Ground beliefs. In particular cases, mutations identify
components that are building blocks of the agent system. Let us
consider the mutations for the DroneState belief shown in Figure 9.

In other words, this metric ignores a mutant that was not killed if the corresponding code was not tested by
the test suite.

Integrating PITest was facilitated by the fact that PITest offers a
Maven plugin to execute mutation testing and coverage reports as
part of the build process, and that SARL also uses Maven to manage
a project’s dependencies and build process.

SARL is JVM based and generates Java code from SARL specifi-
cations. This Java code includes code based on the SARL program written by the developer, but also internal code that provides SARL
features, and is not under the programmer’s control. We therefore
configured PITest to ignore internal SARL-specific code, and only
mutate Java code that corresponds to SARL code written by the
developer. Although PITest ignores these methods for mutation pur-
poses, it still counts the lines of code when computing the coverage,
which results in lower line coverage results.

Figure 7, shows an example summary report of evaluating a
test suite generated for our illustrative search and rescue system
described earlier. The evaluation of the test suite produced 87% line
coverage; 92% mutation coverage and 95% test strength.

Acceptable behaviours despite mutation survival. Mutation framework
perform several mutation operations on the production code
using heuristics with the objective of creating mutants that will
survive the test suite. Although these heuristics are improving con-
tantly, in certain cases, mutations do not modify the behaviour of
the system. For instance, consider the case in which we want to en-
sure that our drone does not go outside the boundaries of the area. In
the bottom corner, we might achieve this with if (x < 0) then x = 0.
If the mutation changes the condition from x < 0 to x ≤ 0 yielding

Discovering missing acceptance criteria. One benefit of mutation
testing is that when issues are discovered, the details of the mutants
that failed to be detected by the test suite can guide us in generating
new test cases or refining existing test cases to address the issues.
For example, whilst overall the test suite has high coverage, the
ExploreArea goal had quite low coverage. Looking at the detailed
report for that goal (Figure 8) allows us to understand why these
to if \((x < 0)\) then \(x = 0\), then the overall behaviour of the system remains the same, despite the code mutation. This is the situation observed in ExplorationCalculator belief set, when investigated following the PITest report (Figure 7).

5 CONCLUSION

We have presented an approach for using User and System Stories for Behaviour-Driven Development. Key contributions are an extension to USS for representing additional information (e.g. goal success conditions), and an implemented framework that supports developers to easily implement and run tests.

By using approaches (e.g. stories, BDD) and tools (e.g. Cucumber) that have been widely adopted and refined with extensive use, we can be confident that our approach will be usable and useful. For instance, the use of stories to capture requirements for BDD is widely-adopted and well-accepted. Our support for implementing tests (Section 4.2) is comparable in complexity to using Cucumber to implement tests for (non-agent-based) systems implemented in Java, therefore we can be confident that the tool is usable.

A key feature of our approach is that it uses modern industry-grade development tools. This provides a number of benefits:

1. It facilitates adoption of AOSE by mainstream software developers since the tools used are ones that they are likely to already be familiar with (e.g. Eclipse, Maven, Cucumber, JUnit, Mockito).
2. Since these tools are widely-used and mature, they are well-developed, reliable, and provide a rich set of features. For example, the SARL IDE is based on the Eclipse project, and consequently provides developers with familiar and mature facilities for test execution and step-by-step debugging (see Figure 10).
3. These tools also have a large community of users, which results in good support.
4. Using standard tools makes it easier to integrate with other tools. This was illustrated in our ability to integrate PITest by using Maven. There are also opportunities to integrate other tools that support common modern development practices, such as continuous integration and delivery.

Future work includes adding support for agent-oriented programming languages other than SARL, and developing a program mutation scheme that is specific for SARL. Currently, we mutate the Java code that SARL generates. However, this means that the mutation operators are not specific to SARL. In particular, because it operates on generated Java, PITest is not able to make mutations that are simple in the context of SARL, but more complex in the generated code. One example of this is changing the order in which plans assigned to a given goal are considered. There has been some work on mutation operators for agent-oriented programming languages that we could build on (e.g. [16, 17, 31, 32]).
ACKNOWLEDGMENTS

This research is supported by the Commonwealth of Australia as well as by the Defence Science and Technology Group of the Department of Defence.

REFERENCES


