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
Wellington, New Zealand
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 by 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

ACM Reference Format:
Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. 2023. A Behaviour-Driven Approach for Testing Requirements via User and System Stories in Agent Systems. In *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), London, United Kingdom, May 29 – June 2, 2023*, IFAAMAS, 9 pages.

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. Agile software development is one of the most popular approaches in mainstream software engineering [22]. The use of USS for requirements [29] builds on Agile concepts and processes [3].

User stories and *system stories* present the requirements from the user and system perspective respectively, and are intended to capture behavioural requirements. An example user story is as follows: *as* an owner of an autonomous vehicle, *I want* the vehicle to obey the speed limit, *so that* traffic laws are obeyed. Each story is accompanied by a set of acceptance criteria, which provide guidance to verify that the story is faithfully implemented. For example for the system story above, an acceptance criterion is: *given* the autonomous vehicle is driving at the current speed limit, *when* a new speed limit lower than the current is met, *then* the vehicle slows to the new speed before the new limit applies. The User and system stories are an accepted way to capture requirements in a format that both domain experts and software engineers can collectively understand and discuss, and that is intuitive [29].

Test-Driven Development (TDD) is an Agile practice that is now well accepted. It involves defining test cases before developing the associated new functionality. Behaviour-driven development (BDD) builds on this by having test cases express system behaviour, specified using a domain-expert-friendly format, such as user stories.

There is a range of work on testing requirements against design [1, 14, 15, 36, 37] and against code [2, 8, 9, 17, 19]. Our work is novel in that we adopt a behaviour-driven approach, so we do testing starting with USS. Unlike Rodriquez *et al.* [30], who presented an approach to validate the outputs (i.e. execution traces) of a complete agent system against the requirements specified via USS, our approach can test *parts* of the system (e.g. unit tests), not just the whole complete system. Finally, unlike the Beast methodology [7], we support goals and plans and integrate mutation testing.

We extend USS so that they are amenable to BDD (Section 3). We then show how USS can be used to generate test cases (Section 4) for goals, plans, percepts and beliefs, with the aim of verifying their implemented behaviours against the requirements. That is, we check that the requirements specifications match the code. For example for goals, we check whether the expected plans to achieve the goal are implemented, and whether the conditions that determine the success of goals are correctly implemented. The quality of the testing is reliant on the quality of the test cases that are generated. Hence, we also integrate a mutation testing system, to provide test coverage analysis of the suite of test cases.

We present our BDD approach using an illustrative case study of a simple search and rescue simulation application [29]. We show how we use a popular and well-established BDD tool, *Cucumber*

Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), A. Ricci, W. Yeoh, N. Agmon, B. An (eds.), May 29 – June 2, 2023, London, United Kingdom.

(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 PITest (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 AOSE.

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

Feature: Search for victims
As Drone Operator,
I want to assign drones areas to explore,
So that they find victims and notify me.

Scenario: Drone is idle
Given drone is at base
When it is assigned an area to explore
Then it begins exploring that area autonomously

Scenario: Drone is busy
Given drone is currenty exploring
When it is assigned an area to explore
Then it adds new area to its exploration queue

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) [20] 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[1] (BDI) [5, 25–27] framework, which conceptualises autonomous agents in terms of[2] *beliefs*, *goals*, 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,

[1] It really should be "BDIP": Plans are a key component of the framework [5]
[2] We 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 fulfil), unachieved (dropped when they are fulfilled), possible (dropped when they are impossible to fulfil), 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 sucess condition s when it is considered to be fulfilled, and the failure condition f when it is considered to be impossible to fulfil. 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 fulfil.

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 conecept. 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 a achievement goal, this condition[3] 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[4].

Note that the last two lines of Scenario 2 ("When …") are actually a pattern that is the same for all `@goal-success` conditions. They

Scenario 2: Explore Area goal success acceptance criteria

```
@goal–success
Scenario: Goal success
  Given I believe current_area_explored is greater than 95%
  When I evaluate current_goal success
  Then goal success is true
```

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 3: PlowSweep Plan for the Explore Area goal

```
@goal–plan
Scenario: HIGH priority area
  Given I believe current_area priority is HIGH
  When I adopt the ExploreArea goal
  Then plan PlowSweep is applicable
```

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.

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,

[3] 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.
[4] 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

```
Scenario: create Plow waypoints
  Given an area at <x>, <y> of <height> by <width>
  When I start plan
  Then I should create path with '<waypoints>'
```

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

```
@belief–infer
Scenario: Battery calculation
  Given I believe battery_charge is <charge>
  When I query belief battery_level
  Then I should believe battery_level is <level >

  Examples:
    | charge | level |
    | 100    | FULL  |
    | 90     | HIGH  |
    ...
    | 24     | LOW   |
```

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

```
@perception–plan
Scenario: New assignment
  When I receive a new AreaAssignment
  Then I add the area to the exploration queue
  And I acknowledge the assignment
  And I should start ExploreArea
  And I should start DetectVictim
```

4 TESTING FRAMEWORK

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

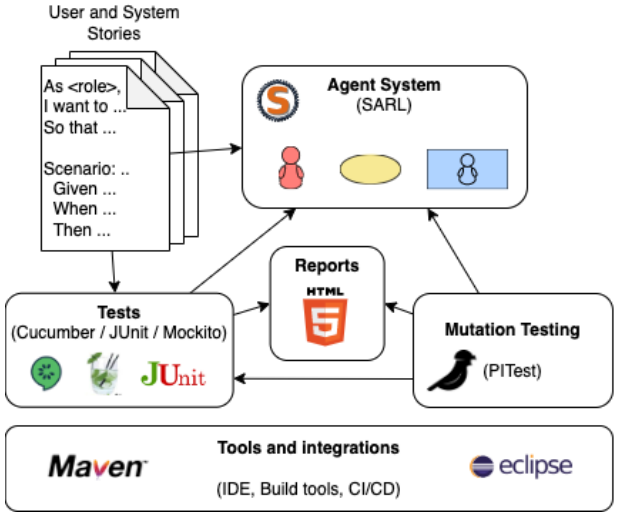


Figure 1: Testing Framework overview

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/

```
skill ExploreArea extends Goal implements AchievementGoal{
  uses SearchRescueBeliefs, DroneState

  def context : boolean {
    batteryLevel != BatteryLevel.LOW
  }
  def success : boolean {
    explorationRate(currentArea) >= 0.95f
  }
  def failure : boolean {
    systemFailure
  }
}
```

Figure 2: SARL implementation of the ExploreArea goal

```
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

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 certain 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. 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, PlanSelectionContraints
  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

the implementation of Scenario 6 (in lines 12-17 of Figure 5). The *on* statement declares the agent's actions to be executed when a given percept is received. The body has four actions, each one mapping directly to each of the action defined in the specification.

Testing this code with the framework yields a test report, shown in Figure 6. The report starts with a summary of the feature's tests, followed by the feature under test. In this case, the scenario is failing

	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
New areas assignments	6	1	0	0	0	7	0	1	1	0.080	Failed

Tags: @perception
Feature New areas assignments
As Drone,
I want to handle new area assignments
so that I can explore them when possible
Background 0.048
Tags: @perception @perception-goals @perception-plan
Scenario New assignment 0.031
Hooks
Steps
When I receive a new AreaAssignment 0.009
Then I add the area to the exploration queue 0.003
And I acknowledge the assignment 0.006
And I should start ExploreArea 0.000
And I should start DetectVictim 0.011
Wanted

```
Wanted but not invoked:
defaultGoalsSkill.post(
    <any searchrescue.DetectVictim>
);
-> at searchrescue.AssignmentSteps.post_explore(AssignmentSteps.java:57)

However, there were exactly 2 interactions with this mock:
defaultGoalsSkill.install();
-> at io.sarl.lang.core.Skill.increaseReference(Skill.java:127)

defaultGoalsSkill.post(
    ExploreArea [
  type = "ExploreArea"
  owner = null
]
);
-> at io.sarl.goals.core.Goals$ContextAwareCapacityWrapper.post(Goals.java:36


        at searchrescue.AssignmentSteps.post_explore(AssignmentSteps.java:57)
        at ✽.I should start DetectVictim(classpath:searchrescue/features/s1-1
```

Figure 6: Cucumber report - failing feature

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.

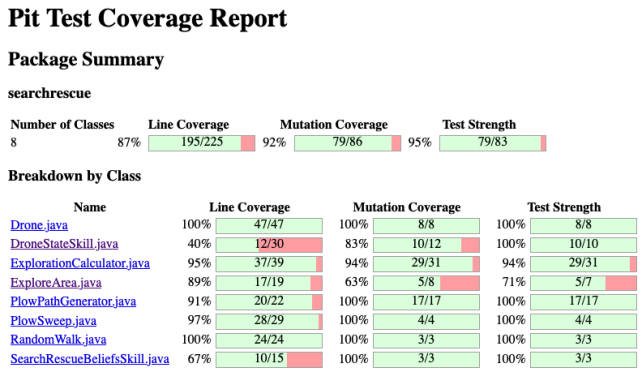
Once the missing step is implemented, successive executions of the test suite will report the scenario is passing.

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.



Pit Test Coverage Report

Package Summary

searchrescue

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	87%	195/225	92%	79/86	95%	79/83

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Drone.java	100%	47/47	100%	8/8	100%	8/8
DroneStateSkill.java	40%	12/30	83%	10/12	100%	10/10
ExplorationCalculator.java	95%	37/39	94%	29/31	94%	29/31
ExploreArea.java	89%	17/19	63%	5/8	71%	5/7
PlowPathGenerator.java	91%	20/22	100%	17/17	100%	17/17
PlowSweep.java	97%	28/29	100%	4/4	100%	4/4
RandomWalk.java	100%	24/24	100%	3/3	100%	3/3
SearchRescueBeliefsSkill.java	67%	10/15	100%	3/3	100%	3/3

Figure 7: Sample PITest Coverage Report

referred to as *killing the mutant*. The test suite is then evaluated by the percentage of mutants they kill.

We chose the PITest mutation testing system, since it provides a gold standard test coverage for Java and JVM in industry practice. PITest offers three measurements of test suite quality:

(1) *Line coverage* - measures the number of lines of code that is covered by the tests;
(2) *Mutation coverage* - measures the number of mutants that were killed out of all the mutants created; and
(3) *Test strength* - measures the number of mutants that were killed out of all the mutants created for which there was test coverage[8]. 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 specifications. 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 purposes, 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.

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 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 test 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 success "True" mutant

 # New Scenario from Mutation: False success
@goal–success
Scenario: Goal success false
Given I believe current_area_exploration_rate is less than 95%
When I evaluate current_goal success
Then goal success is false

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. 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 frameworks 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 constantly, in certain cases, mutations do not modify the behaviour of the system. For instance, consider the case in which we want to ensure 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 $x \leq 0$ yielding

[8]If a mutant survived due to lack of coverage, it will not be counted towards this measurement.

```
26 1. negated conditional → KILLED
   2. replaced boolean return with false for searchrescue/ExploreArea::context → SURVIVED
33 1. changed conditional boundary → KILLED
   2. negated conditional → KILLED
   3. replaced boolean return with false for searchrescue/ExploreArea::success → KILLED
   4. replaced boolean return with true for searchrescue/ExploreArea::success → NO_COVERAGE
38 1. replaced boolean return with false for searchrescue/ExploreArea::failure → KILLED
   2. replaced boolean return with true for searchrescue/ExploreArea::failure → SURVIVED
```

Figure 8: Mutation results for ExploreArea goal

```
19 1. changed conditional boundary → KILLED
   2. negated conditional → KILLED
20 1. replaced return value with null for searchrescue/DroneStateSkill::getBatteryLevel → KILLED
22 1. changed conditional boundary → KILLED
   2. negated conditional → KILLED
23 1. replaced return value with null for searchrescue/DroneStateSkill::getBatteryLevel → KILLED
25 1. changed conditional boundary → KILLED
   2. negated conditional → KILLED
26 1. replaced return value with null for searchrescue/DroneStateSkill::getBatteryLevel → KILLED
28 1. replaced return value with null for searchrescue/DroneStateSkill::getBatteryLevel → KILLED
32 1. replaced boolean return with true for searchrescue/DroneStateSkill::getSystemFailure → NO_COVERAGE
36 1. replaced int return with 0 for searchrescue/DroneStateSkill::getBatteryCharge → NO_COVERAGE
```

Figure 9: Mutation results for Drone State belief set

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 *ExplorationCalulator* 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.

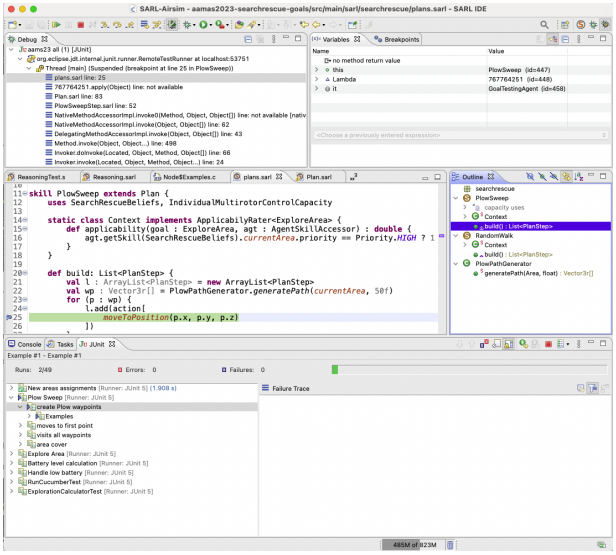


Figure 10: SARL IDE: Eclipse-based with breakpoints and debugging tests

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 represented by the Defence Science and Technology Group of the Department of Defence.

REFERENCES

[1] Yoosef B. Abushark, John Thangarajah, Tim Miller, James Harland, and Michael Winikoff. 2015. Early Detection of Design Faults Relative to Requirement Specifications in Agent-Based Models. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*. ACM, 1071–1079.

[2] Marina Bagić Babac and Dragan Jevtić. 2014. AgentTest: A Specification Language for Agent-Based System Testing. *Neurocomputing* 146 (Dec. 2014), 230–248. https://doi.org/10.1016/j.neucom.2014.04.060

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, and Jeff Sutherland. 2001. *The Agile Manifesto*. Technical Report. The Agile Alliance.

[4] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley.

[5] Michael E. Bratman. 1987. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.

[6] Lars Braubach, Alexander Pokahr, Daniel Moldt, and Winfried Lamersdorf. 2004. Goal Representation for BDI Agent Systems. In *Programming Multi-Agent Systems, Second International Workshop ProMAS 2004 (Lecture Notes in Computer Science, Vol. 3346)*. Springer, 44–65. https://doi.org/10.1007/978-3-540-32260-3_3

[7] Álvaro Carrera, Carlos A. Iglesias, and Mercedes Garijo. 2014. Beast Methodology: An Agile Testing Methodology for Multi-Agent Systems Based on Behaviour Driven Development. *Information Systems Frontiers* 16, 2 (April 2014), 169–182. https://doi.org/10.1007/s10796-013-9438-5

[8] Roberta Coelho, Elder Cirilo, Uira Kulesza, Arndt von Staa, Awais Rashid, and Carlos Lucena. 2007. JAT: A Test Automation Framework for Multi-Agent Systems. In *2007 IEEE International Conference on Software Maintenance*. IEEE, Paris, France, 425–434. https://doi.org/10.1109/ICSM.2007.4362655

[9] Roberta Coelho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. 2006. Unit Testing in Multi-agent Systems Using Mock Agents and Aspects. In *Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems (SELMAS '06)*. ACM, New York, NY, USA, 83–90. https://doi.org/10.1145/1138063.1138079

[10] Mike Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison-Wesley, Boston.

[11] Mehdi Dastani, M. Birna van Riemsdijk, and Michael Winikoff. 2011. Rich goal types in agent programming. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum (Eds.). IFAAMAS, 405–412.

[12] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[13] Simon Duff, James Harland, and John Thangarajah. 2006. On proactivity and maintenance goals. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*, Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone (Eds.). ACM, 1033–1040. https://doi.org/10.1145/1160633.1160817

[14] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. 2001. Model checking early requirements specifications in Tropos. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*. 174–181. https://doi.org/10.1109/ISRE.2001.948557

[15] Paolo Giorgini, John Mylopoulos, and Roberto Sebastiani. 2005. Goal-oriented requirements analysis and reasoning in the Tropos methodology. *Engineering Applications of Artificial Intelligence* 18, 2 (2005), 159–171. https://doi.org/10.1016/j.engappai.2004.11.017

[16] Zhan Huang and Rob Alexander. 2015. Semantic Mutation Testing for Multi-agent Systems. In *Engineering Multi-Agent Systems (Lecture Notes in Computer Science)*, Matteo Baldoni, Luciano Baresi, and Mehdi Dastani (Eds.). Springer International Publishing, Cham, 131–152. https://doi.org/10.1007/978-3-319-26184-3_8

[17] Zhan Huang, Rob Alexander, and John Clark. 2014. Mutation Testing for Jason Agents. In *Engineering Multi-Agent Systems (Lecture Notes in Computer Science)*, Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk (Eds.). Springer International Publishing, Cham, 309–327. https://doi.org/10.1007/978-3-319-14484-9_16

[18] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. 1992. An Architecture for real-time reasoning and system control. *IEEE Expert* 7, 6 (1992).

[19] Cu D. Nguyen, Anna Perini, Carole Bernon, Juan Pavón, and John Thangarajah. 2009. Testing in Multi-Agent Systems. In *Agent-Oriented Software Engineering X - 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers (LNCS, Vol. 6038)*, Marie-Pierre Gleizes and Jorge J. Gómez-Sanz (Eds.). Springer, 180–190.

[20] Dan North. 2006. Introducing BDD. https://dannorth.net/introducing-bdd/.

[21] Lin Padgham and Michael Winikoff. 2004. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons.

[22] PMI. 2017. *Pulse of the Profession 2017*. Technical Report. Project Management Institute.

[23] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. 2005. Jadex: A BDI Reasoning Engine. In *Multi-Agent Programming: Languages, Platforms and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15. Springer, 149–174.

[24] Anand S. Rao. 1996. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96) (Lecture Notes in Artificial Intelligence, Vol. 1038)*, Walter Van de Velde and John Perrame (Eds.). Springer, 42–55.

[25] Anand S. Rao and Michael P. Georgeff. 1991. Modeling Rational Agents within a BDI-Architecture. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, James F. Allen, Richard Fikes, and Erik Sandewall (Eds.). Morgan Kaufmann, Cambridge, MA, USA, 473–484.

[26] Anand S. Rao and Michael P. Georgeff. 1992. An Abstract Architecture for Rational Agents. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, Bernhard Nebel, Charles Rich, and William R. Swartout (Eds.). Morgan Kaufmann, Cambridge, MA, 439–449.

[27] Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Conference on Multiagent Systems*, Victor R. Lesser and Les Gasser (Eds.). The MIT Press, 312–319.

[28] Sebastian Rodriguez, Nicolas Gaud, and Stéphane Galland. 2014. SARL: A General-Purpose Agent-Oriented Programming Language. In *The 2014 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, Vol. 3. IEEE Computer Society Press, Warsaw, Poland, 103–110. https://doi.org/10.1109/WI-IAT.2014.156

[29] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. 2021. User and System Stories: An Agile Approach for Managing Requirements in AOSE. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '21)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1064–1072. https://doi.org/10.5555/3461017.3461136

[30] Sebastian Rodriguez, John Thangarajah, Michael Winikoff, and Dhirendra Singh. 2022. Testing Requirements via User and System Stories in Agent Systems. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS '22)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1119–1127.

[31] Sharmila Savarimuthu and Michael Winikoff. 2013. Mutation operators for cognitive agent programs. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Saint Paul, MN, USA, May 6-10, 2013*, Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M. Jonker (Eds.). IFAAMAS, 1137–1138. http://dl.acm.org/citation.cfm?id=2485109

[32] Sharmila Savarimuthu and Michael Winikoff. 2013. Mutation Operators for the Goal Agent Language. In *Engineering Multi-Agent Systems - First International Workshop, EMAS 2013, St. Paul, MN, USA, May 6-7, 2013, Revised Selected Papers (LNCS, Vol. 8245)*. Springer, 255–273.

[33] M. Birna van Riemsdijk, Mehdi Dastani, and Michael Winikoff. 2008. Goals in agent systems: a unifying framework. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 2*, Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons (Eds.). IFAAMAS, 713–720. https://dl.acm.org/citation.cfm?id=1402323

[34] Michael Winikoff. 2005. JACK^{TM} Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming: Languages, Platforms and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15. Springer, 175–193.

[35] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. 2002. Declarative & Procedural Goals in Intelligent Agent Systems. In *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*, Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams (Eds.). Morgan Kaufmann, 470–481.

[36] Nitin Yadav and John Thangarajah. 2016. Checking the Conformance of Requirements in Agent Designs Using ATL. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016) (Frontiers in Artificial Intelligence and Applications, Vol. 285)*. IOS Press, 243–251. https://doi.org/10.3233/978-1-61499-672-9-243

[37] Nitin Yadav, John Thangarajah, and Sebastian Sardiña. 2017. Agent Design Consistency Checking via Planning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 458–464. https://doi.org/10.24963/ijcai.2017/65