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

Dhirendra Singh
RMIT University
Melbourne, Australia
dhirendra.singh@rmit.edu.au

ABSTRACT

Agile software development is a popular and widely adopted practice due to its flexible and iterative nature that facilitates rapid prototyping. Recent work presented an agile approach to capturing requirements in agent systems via *user* and *system stories*. User and system stories present the requirements from the user and system perspective, respectively. Each story contains a set of *acceptance criteria*, which are a set of statements that identify the conditions under which the system behaviour can be accepted by the users or stakeholders. In this paper, we present a novel approach to testing the requirements that are specified via User and System stories in an agent system. We do this by developing a systematic approach to validating the execution traces output by the system against the specified acceptance criteria for each story. The approach identifies acceptance criteria that are met successfully in execution and those that fail. We present a fault model that categorizes the failures providing insight to the developers to address the failed cases. We classify three kinds of faults for a given acceptance criterion: (a) the trigger condition is never met; (b) when the trigger occurs the preconditions are not met; or (c) the trigger and preconditions are met but the resulting actions are not as expected. The motivating application of our work, which is also the test-bed for evaluation, is an agent-based simulation application for modelling the behaviours of civilians in a bushfire emergency scenario that is used in practice. We show our approach is able to successfully test and uncover requirements that were not met in this application.

KEYWORDS

AOSE; Engineering MAS; Agile methodologies; Testing; Requirements

ACM Reference Format:
Sebastian Rodriguez, John Thangarajah, Michael Winikoff, and Dhirendra Singh. 2022. Testing Requirements via User and System Stories in Agent Systems. In *Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022), Online, May 9–13, 2022*, IFAAMAS, 9 pages.

1 INTRODUCTION

Agile software development is one of the most popular approaches today due to its flexible and iterative nature that facilitates rapid prototyping. There has been recent work that has adapted Agile concepts and processes to agent-oriented software engineering (AOSE) [5, 9, 10, 30, 36, 38] including the work of Rodriguez *et al.* [30] that presented an agile approach to capturing requirements in agent systems via *user* and *system stories*.

User Stories and *System Stories* present the requirements from the user and system perspective, respectively. For example, for a personal assistant system, a possible system story is - As the *System* I want to be able to book a calendar appointment. Each story contains a set of *acceptance criteria*, which are a set of statements that identify the conditions under which the system behaviour can be accepted by the users or stakeholders. For example for the system story above an acceptance criterion could be - when an appointment for a time-slot is requested, if the time slot is free then the appointment is booked and the time-slot is no longer free. The *User & system stories* are a valuable tool to capture human behaviors in a format that *Domain Experts* and *Agent engineers* can understand and discuss.

In this paper, we present a novel approach to testing the requirements that are specified via user and system stories in an agent system. We do this by developing an approach that validates the acceptance criteria for each user and system story against the execution traces.

Testing is an important aspect of any software development life cycle to ensure that the system is behaving as expected and there have been several testing approaches proposed in the Agent-Oriented Software Engineering liteature, e.g. [17, 21, 43].

Our test framework is able to test whether the acceptance criteria are met and classify the cases that fail into one of three categories: (a) the trigger condition is never met (e.g., from the example above, an appointment was never requested); (b) when the trigger occurs the preconditions are not true (e.g., the time-slot requested is not free); or (c) the trigger occurs and preconditions are true but the resulting actions are not as expected (e.g. the time slot is still showing as free after the booking).

The motivating application of our work, which is also the test-bed for evaluation, is an agent-based simulation application for modelling the behaviours of civilians in a bushfire emergency scenario that is used in practice. In developing the user and system stories

and the acceptance criteria for this application it was also necessary to extend the representation of acceptance criteria proposed in [30]. Firstly, it was necessary to provide explicit agent-specific constructs such as Agent-Types, Percepts, and Belief Conditions. For example, to be able to specify that a given agent is of type "Resident". Secondly, to include temporal constructs. For example, the ability to specify that a person will *eventually* decide to evacuate the areas in the sight of fire.

In this paper we make the following contributions: (i) extend the definition of acceptance criteria for user and system stories to include agent-specific and temporal constructs; (ii) develop a testing framework for validating the acceptance criteria (including the temporal extensions) against the run-time traces; (iii) present a fault-model for classifying the tests that fail; and (iv) show that our approach is able to successfully test and uncover faults in a real-world application.

In the next section we describe some preliminaries including a description of the Bushfire Simulation application. We present our extended representations in Section 3 followed by our testing approach in Section 4. We present our evaluation and results in Section 5 and conclude with related and future work in Sections 6 and 7 respectively.

2 BACKGROUND

In this section, we present a brief overview of user and system stories, the type of agents (BDI agents) that this work is based on, and detail the Bushfire Simulation application that we use as our testbed.

2.1 User and System Stories

The Agile approach [3] to software development is popular and widely adopted in industry. For example, 71% of the organizations consulted in [22] responded that they use agile approaches in their projects. A key feature of an agile methodology is the way in which requirements are captured - at a higher level at early stages and progressively refined and detailed.

Amongst the various approaches to gather requirements (e.g User Interviews, Questionnaires, Story-Writing workshops) [6], the most common technique in agile frameworks is *User Stories* [2]. A user story is an informal description of a feature described from the perspective of a user of the system. These are represented using the template: *As (role), I want to (do something), so that (reason)* [30]. For example, for a personal assistant system: *as* a user, *I want to* see my daily appointments, *so that* I can plan my day.

Recently, Rodriguez et. al. [30] introduced the concept of a *System Story* associated with each user story. In essence, the system stories further refine the user story from the system's perspective, identifying the requirement of the system to satisfy the user's needs. They were captured in the same manner as user stories. For example, a system story for the user story is: *as* the system, I want to be able to retrieve daily appointments, *so that* the user can plan her day.

Each story also contains a set of *acceptance criteria*, which are a set of statements that identify the conditions under which the system behaviour can be accepted by the users or stakeholders. These are captured via the scenario-oriented format of *Given/When/Then*, derived from Behavior Driven Development (BDD) [18]. In this format, the *Given* component describes the necessary preconditions for the system to execute the behavior being described; *When* identifies the triggers of the behavior; and *Then* details the resulting effects of executing that behavior. For example, for the above system story, an acceptance criterion might be: *Given* a valid date, *when* a request for daily appointments is made, *then* all appointments for that date are displayed.

Similar to [30], we specify acceptance criteria using an extension (see Section 3) of the *Gherkin* format[1]. In essence, in the Gherkin notation an acceptance criterion is a sequence of steps that include one or more *given* statements which specify a pre-condition, a *when* statement indicating a trigger for the relevant story, and one or more *then* statements indicating a sequence of outcomes, i.e. observable conditions. It is also possible to specify a *background*: a collection of *given* statements that are common to a number of scenarios. Finally, scenarios can be grouped together under a *feature*.

2.2 BDI Agent systems

Similar to [30], our work is grounded in the BDI (Belief-Desire-Intention) agent paradigm [28]. BDI agents are a popular and well established technology for designing [4, 7, 20] and implementing [23, 39] agent systems. A key feature of BDI agents is the cognitive concepts used to specify and implement them.

Some of concepts are, for example, *percepts* in the environment that the agent responds to, *beliefs* about itself and the environment, *goals* that the agents want to achieve, *plans* that are used to achieve the goals and *actions* that change the state of the environment. The Bushfire Simulation application described next is implemented as a BDI agent system.

2.3 The Bushfire Simulation Application

We evaluate our framework on a BDI program to capture the behaviours of residents during wildfires, or bushfires, in the Australian context. The program is part of an evacuation model that combines a fire spread model, a population model, and a traffic simulator [33, 34]. The evacuation model sits within a decision-support system that is being trialled by emergency services to understand the threat to a community from predicted wildfires. This understanding is then used to inform planning and preparedness measures for mitigating the risk to the community.

The web-based decision-support system allows an emergency manager or incident controller to setup and run an evacuation scenario, then visualise and analyse the outputs to gauge the effectiveness of the proposed evacuation plan—a series of zone-based emergency messages that inform the community of the approaching fire threat along with recommendations for appropriate action [29].

Whether or when agents (BDI) in the simulation react to the broadcast emergency messages, or the sight of fire or smoke, depends on how risk averse they are (beliefs) and how anxious they become from the cumulative perception of those events (situation awareness). When agents do decide to react, then what actions they take depends on their situation, such as whether they need to go home before evacuating, and personal circumstances, such as whether they have vulnerable or dependent others that they

[1] https://cucumber.io/docs/gherkin/reference/

Story	::=	**Feature:** *name*, StoryDescription, (AcceptanceCriteria)*
StoryDescription	::=	**As** *role*, **I want to** *task*, **so that** *reason*
AcceptanceCriteria	::=	**Scenario:** *description* , GivenStatement* WhenStatement ThenStatement+
GivenStatement	::=	**Given** (AgentTypeCondition | BeliefCondition)
WhenStatement	::=	**When** Perception | **When** BeliefCondition
ThenStatement	::=	**Then** (**immediately** | **eventually** | **never** | **always**) BeliefCondition
AgentTypeCondition	::=	**Agent is Type** *agentValue*
BeliefCondition	::=	**It believes** (*beliefName* | **current_plan** | **current_goal**) **is** [**less than** | **greater than**] *beliefValue*
Perception	::=	**It sees** *percept*

Figure 1: BNF syntax for capturing a Story and the corresponding set of acceptance criteria. In bold are keywords and in italics are terminals (strings, except for *beliefName* which is an identifier, and *beliefValue* which is string, number, or Boolean).

```
Feature: Handling of dependents for full-time
         residents

As ResidentFullTime,
I want to always attend to my dependents
so that they are safe

Scenario: first response is always to
          attend to dependents (expert)

    Given agent is type ResidentFullTime
    Given it believes HasDependents is true
    When it believes current_goal is
         GoalInitialResponse
    Then eventually it believes status is
         to:DependentsPlace
```

Figure 2: Example user story and acceptance criterion

need to attend to first. Collectively, the agents exhibit the kinds of behaviours that are commonly seen in bushfire situations [35].

The BDI program was created using a process of model co-development with end users. The BDI model is particularly useful here as a way of modelling human decision-making, because it uses the familiar concepts that we use to discuss human planning and acting (e.g. goals, plans, beliefs). Additionally: (i) it is intuitive for domain experts to comprehend and collaborate over, (ii) it can capture rich behaviours in a compact form (e.g. see [42]), and (iii) agreed goal-plan visualisations can readily be converted into a program with minimal errors in translation [19].

Our evaluation (§5) is performed on behaviour traces of BDI agents collected from output log files of the evacuation simulation. The traces include time-stamped information about the reasoning of every agent during the simulation, such as what plans were being considered and selected in response to which triggers when, and what subsequent actions were being taken. These traces were already being used by the BDI programmers to manually verify behaviours in discussions with domain experts [35]. Our intent in this evaluation was to extract first some system and user stories from discussions with domain experts and the programmers, and then automatically parse the simulation log files to check for the existence of those stories in the BDI behaviour traces.

3 REPRESENTATION

In utilising the approach of [30] in the bushfire evacuation simulation system (§2.3), it was necessary to extend the representation of the acceptance criteria to capture what the domain experts wanted to express and subsequently test via the approach we propose. Specifically, it was necessary to extend the notation in two ways: (i) to allow reference to agent-specific aspects; and (ii) to allow temporal expressions.

Figure 1 presents the syntax for capturing a user or system story and the corresponding acceptance criteria. In this representation we introduce agent-specific concepts as follows:

- AgentTypeCondition which allow *Given* statements to refer to the agent type ("Given agent is type *Type*").
- Percept that triggers an acceptance criterion ("When it sees *PerceptName*").
- BeliefConditions where Beliefs can be numerical, true/false, or other types ("…it believes *BeliefName* is (true | false | some_value)", and "…*BeliefName* is [less than | greater than] *numerical_value*".
- BeliefConditions also afford a degree of introspection by allowing the statements to refer to beliefs regarding the current plan and current goal. ("…it believes (current_plan | current_goal) is *Name*").

In addition, we also enrich the language with basic temporal constructs, allowing the *then* statements about the belief updates to include one of "immediately", "always", "never" and "eventually", with their usual meaning.

Figure 2 shows an example story and acceptance criterion as used in our bushfire simulation application. The story captures the expected behaviour of a *Fulltime Resident* with dependents. In line 4 the story description is captured. Aligned with the approach presented in [30], we can extract from the story description a top level goal of the agent to attend to dependents. The current implementation of the simulator uses the generic goal *GoalInitialResponse* to represent this goal. The acceptable behaviors to achieve this goal are described in the acceptance criterion *scenario*. The text after the *Scenario* keyword is the description of the expected behaviour.

This scenario states that *When* the goal *GoalInitialResponse* (i.e. attend to dependents) becomes the *current_goal*; *Then* the agent should eventually be going to its dependents' location (i.e believe its status to be *status=to:DependentsPlace* at some future stage). This example illustrates the need for the temporal constructs. In some cases the agent may need to *immediately* go to the dependents location, and in other cases (such as in this example), the agents may do this at a later stage (i.e. eventually) perhaps after completing some other activities.

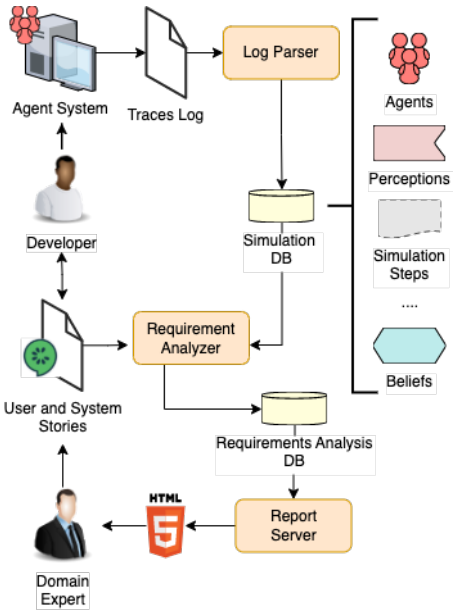

Figure 3: Process overview

4 TESTING APPROACH

A key feature of user and system stories is enabling *domain experts* to intuitively specify requirements and for the *developers* to easily interpret and implement them into the final *agent system*.

The aim of our approach is to validate the implemented system against those requirements. We do this by using the log traces that are produced during execution. These traces contain reasoning events such as belief updates, goal and plan applicability, agent's perceptions, etc. An overview of the process is presented in figure 3. Our prototype system is composed of three top-level components:

(1) Log Parser: Responsible for parsing system logs and input them in a standard format to a simulation history database (see §4.1).
(2) Requirement Analyzer: The core of the system, responsible for transforming user and system stories and their respective acceptance criteria into queries that can be executed against the simulation history database (§4.1). These queries include temporal validations of agents' beliefs (§3).
(3) Report Server: Presents the outcomes of the analysis in a human-friendly format for *Domain Experts* to review and potential enrich the current systems specification (§4.2).

4.1 Traces

We now describe the structure of the traces generated by the agent based simulation system. The traces generated by execution take the form of a list of records, where each record has the following fields, separated by |:

(1) Step number in the simulation, with −1 denoting steps before the start of simulation;
(2) Simulated time;
(3) Agent archetype;
(4) Agent ID; and
(5) Step details (discussed below).

```
42930|11:55:30|ResidentFullTime|8344|saw embers
42930|11:55:30|ResidentFullTime|8344|
        believes anxietyFromSituation=0.3
42930|11:55:30|ResidentFullTime|8344|
        believes futureValueOfVisibleEmbers=0.0
42930|11:55:30|ResidentFullTime|8344|
        believes responseThresholdInitReached=true
42930|11:55:30|ResidentFullTime|8344|
        believes responseThresholdFinReached=true
```

Figure 4: A perception and belief update trace example.

```
...|thinks GoalFullResponse >PlanFullResponse is applicable
...|thinks GoalInitialResponse >
        PlanResponseWhenDependentsAfar is not applicable
...|thinks GoalInitialResponse >
        PlanResponseWhenDependentsNearby is applicable
...|thinks GoalInitialResponse >
        PlanResponseWithoutDependents is not applicable
...|thinks GoalInitialResponse >PlanDoNothing is applicable
...|will go to ... GoalInitialResponse >PlanResponseWhenDependentsNearby
```

Figure 5: Reasoning information traces example (header of trace removed for brevity).

Step details can take a number of forms. They can represent a percept ("saw *PerceptName*"), a belief ("believes *beliefName=value*"), and plan selection which includes both indication of which plans are applicable ("thinks *GoalName~>PlanName* is [not] applicable") and which applicable plan is selected ("will *GoalName~>PlanName*"). Figures 4 and 5 shows example traces for perception and belief updates, and reasoning information, respectively. Based on these traces we are able to reconstruct the agent's belief states and reasoning at each step of the simulation.

For every agent, trace information pertaining to the agent is first filtered out from the simulation log. Conceptually, these agent related traces contain the complete evolution of the agent's state from the beginning of the simulation to its end, including when it perceived what external event ('saw' traces), when it updated its beliefs due to external percepts or internal reasoning ('believes' traces), and what course of action it took when ('will do' traces). The parser therefore traverses the traces in order of simulation time stamp, and records belief transitions[2] in the database for each agent for every time step. This greatly simplifies the subsequent step of requirements checking since the full state of every agent at every time step can be directly queried. This is done to improve the performance of the *Requirement Analyzer* and render it more reactive to human interaction. The result is stored in the *Simulation history DB*.

Once the DB is populated, the *Requirement Analyzer* processes the *User and System Stories requirements* and creates queries to verify whether they are respected by the agent traces stored in the

[2]For this exercise, all events related to the agent, such as external percepts received, beliefs updated, and goals/actions triggered, can be treated as state transitions, or simply, belief changes.

Figure 6: Scenario report example (scenario in figure 8)

Simulation DB. The result of this analysis is stored in the *Requirements Analysis DB*.

Finally, the *Report Server* provides the *Domain Expert* with means to explore and understand the outcome. For example, Figure 6 shows the result of a failing acceptance criterion. Using this information the *Domain expert* can review requirements using the fault model presented in the following section.

In practice, we found it useful and often necessary to add user and system stories from the perspective of the developer. This was because the domain expert would typically have a high-level and *incomplete* view of agent behaviours they expected the system to exhibit, without necessarily having thought through any implications of those decisions on the overall behaviour of that agent or groups of agents. The developer on the other hand, was fully involved in the nuances of the modelled behaviours and how they manifested in all possible situations within the simulated evacuation scenarios, and would therefore augment the set with new stories. This further benefited the co-development process between the domain expert and developer, through the expanded set of agreed user stories. The approach then fosters the discussion between experts and developers that is not only encouraged by agile methodologies but also the original intention of user stories.

In applications such as the disaster management simulation, where agents mimic human behaviours in disaster situations, the experts are presented with the simulation results to further refine their understanding of behaviours and improve the quality of the results. This "review" can be associated with practices similar to "Sprint Review" in Scrum, where the experts provide feedback and, often, review the requirements based on the results. Details of this practice can be found in [29].

Intuitively in the process, the *Log Parser* reconstructs the beliefs of the agents in each step of the simulation by updating them in the *Simulation DB* according to the traces. Then the *Requirements Analyzer* translates the *User and System Stories* into database queries to verify that the belief states of the agent comply with the specification. Finally, the *Report Server* presents an interactive view of the outcomes to *Domain Experts* and the *Development Team*.

Fault Name	Fault Type	Interpretation
NO_TRIGGER	Weak	Trigger (perception or belief update) was not observed for any agent in the simulation
TRIGGERED BUT_GIVEN NOT_MET	Weak	Trigger observed, but belief state of the agent did not meet the given conditions
FAIL	Strong	Trigger observed, conditions met, but the observed behavior of the agent does not comply with the specification.
PASS		Trigger observed, conditions met, and the observed behavior of the agent complies with the specification

Table 1: Fault type summary

4.2 Fault Model

The fault model operates at the level of the acceptance criteria, being, *given* some preconditions, *when* a trigger occurs, *then* some behaviour should be observed. In this sense, a fault can potentially exist if the desired criteria are not met together. If *given* and *when* conditions do not hold true together, the reason could be an underlying fault, a poor specification of the conditions, or poor coverage of the behaviours (and the situations that induce them) in the simulation output. For this reason, we call such cases **weak failures**. On the other hand, if both those conditions hold, but the *then* condition fails, it indicates a failed expectation in the traditional sense of software testing, and we call it a **strong failure**.

The checking process is able to detect a number of types of faults.

(1) When a scenario is triggered (*given* and *when* match trace steps), but the *then* statements do not all match. This is a **(strong) failure** to meet the requirements. The system is doing the wrong thing, and the system needs to be fixed (or, possibly, the scenario specification corrected, e.g. it may have incorrect or additional statements).
(2) When a scenario is never triggered. This is regarded as a **weak failure**: a given desired behaviour is never observed in the given traces. This may reflect an error in the system or in the specification (e.g. too restrictive given/when conditions), or it may reflect that more extensive testing is needed. There are sub-cases here, corresponding to why the scenario is never triggered.
   (a) *when* never occurs - in this case, the behaviours covered by the trace do not test the acceptance criterion in question at all, and either the acceptance criterion is obsolete and should be removed, or, arguably more likely, the range of tests is inadequate and needs to be extended.
   (b) the *when* occurs in the trace, but in each case, the *given* fails to hold. In this sub-case the tests do attempt to exercise the behaviour in question, but there is a mismatch between the specified *given* (precondition) and the observed situation. This may reflect an incorrect (too strict) *given*, or a mistaken assumption about the behaviour that is observed.
(3) In addition to considering scenarios and whether they are met by the trace, we also consider what parts of the trace

are matched against at least one scenario instance. When a trace step is not matched against any scenario, this reflects an *underspecification*, i.e. the specification is missing stories to cover the situations and behaviours in which the trace steps occur.
This situation could also correspond to one where a relevant story does exist, but its trigger and context are too strong (which would be covered by the immediately preceding case). It could also correspond to a situation where a story is missing statements: e.g. if the scenario has "Then t_1; Then t_3" but the trace is t_1, t_2, t_3, or, of course, the trace may have additional incorrect steps. It is important to notice that in the case of temporal statements a particular step might need to comply with multiple requirements.

A summary of the fault types is presented in Table 1. Note that a scenario is reported as failed even if only one step matching the specification fails.

5 EVALUATION

In order to evaluate our approach we used traces generated by the bushfire simulator presented in section 2.3. The simulator has been in development for a number of years based on the requirements expressed by domain experts. However, requirements for the human behaviours of the simulator were not originally captured using *User and System Stories*. Therefore, in collaboration with the simulator developer, we reverse-engineered the requirements in the format of *Stories* and *Acceptance criteria*. We developed 3 *User and System Stories* (i.e. Feature files) containing 9 different *Acceptance Criteria* with 35 Given/When/Then statements. The same set of stories and acceptance criteria were used in the experiments described below.

Our first set of experiments (§5.1) aimed to validate the requirements on a real setting using traces from a simulation previously accepted by domain experts. We refer to this type of simulations as *validated simulations*. These experiments are presented in Section 5.1. The second set of experiments (§5.2) aimed to validate the effectiveness of the approach to detect agent behaviours that does not comply with the requirements. To this end, we intentionally modified agent's traces to introduce faults. In this controlled environment we evaluate the ability of the approach to detect known faults.

User and system stories were created in a *high-level* language that was suitable for discussions with *Domain experts* not familiar to agent terminology.

5.1 Evaluation on validated simulations

In order to validate the accuracy of the requirements represented by the *User and System Stories*, we used traces from a previously validated simulation. The goal of these experiments is twofold: (a) understand the complexity and performance of the process; and (b) ensure that the stories are not inconsistent with the behaviour.

A simulation of this nature contains thousands of agents of various types. We created random samples of agents and extracted their behaviour traces. We obtained 4 sets of simulations traces of 10 agents each. Since the requirements in our application domain represent expected behaviours of each agent, extracting merged agent execution traces do not affect the analysis process. On the other hand, creating smaller sets of agents, make the analysis process faster and more manageable.

A summary of the results is presented in table 2. In all the traces combined we parsed over 41,000 lines of log traces generated by 40 agents. As noted in §4.1 a major step in the process is to expand the belief transitions observed for agents in each simulation step into complete belief states for those agents for that time step for storage in the database that can be queried. This step yields over 5 million belief states. Agents received 12,413 perceptions from their environment. Notice that the perceptions are only external events observed by the agents (i.e. fire, embers, emergency messages, etc.) and does not include belief updates (i.e. internal triggers).

We use the following terminology:

- A *Step* is a simulation tick for a particular agent in the simulation. A step is then a unique combination of a timestamp and an agent.
- A *Triggerable Step* is a step in which all given conditions of a scenario are met. In essence, it captures the step where when the trigger occurs, the behaviour is ready to be executed, as the preconditions are met. Note that the same Step could be *Triggerable* for multiple scenarios.
- An *Event Step* is a Step in which the scenario's trigger happens. A trigger can be a perception or belief update. Notice, that a step where the trigger happens may not initiate the agent's behaviour if the *given* conditions are not met.
- A *Triggered Step:* is a Step that is an Event Step and also a Triggerable Step. These steps are evaluated with the *then* statement.

With that information, the *Requirements Analyzer* used the 3 *Stories* and identified 220,812 Triggerable steps of which 7,462 were actually triggered.

While in most cases the agents behaved according to the specification, 6 of them were flagged as *Fail*.

Step 1116

Agent id : 3

Agent Sim ID : 2

Agent Type : ResidentFullTime

Simulation Step : 48630

Belief	Value BEFORE	Value AFTER
LocationWork	false	false
ImpactFromImmersionInSmoke	false	false
current_goal	FollowRoutine	GoalFinalResponse
current_plan	PlanFollowRoutine	PlanLeaveNow
isStuck	false	false
isDriving	false	false

Figure 7: Step information report example

Upon investigation of those 6 cases, we found that the failing acceptance criterion was the one presented in Figure 8. The associated *Story* is concerned with validating the agent's situational

					Steps					Scenarios	
	Num Traces	**Num Agents**	**Beliefs Expanded**	**Perceptions**	**Total**	**Triggerable**	**Triggered**	**Pass**	**Fail**	**Pass**	**Fail**
Trace 1	9662	10	1410320	2909	10370	60590	1687	1686	1	8	1
Trace 2	11488	10	1319200	3515	9700	55586	2256	2256	0	9	0
Trace 3	5290	10	1077120	1469	7920	48888	1068	1066	2	8	1
Trace 4	14565	10	1358640	4520	9990	55748	2451	2448	3	8	1
Totals	41005	40	5165280	12413	37980	220812	7462	7456	6		

Table 2: Evaluation results for the validated simulation

awareness and in particular, this *Acceptance Criterion* specifies that the agent should eventually start the evacuation of the area (i.e. adopt the goal *GoalFinalResponse*) when the *final response threshold is reached*. In a closer analysis in colalaboration with domain experts, we concluded that in all 6 cases the simulation ended before the agent could adopt the goal. A simulation with an extended time confirmed that all agents completed the goals as expected.

Given the above findings, we conclude that we can successfully query large simulation traces to verify that scenarios represent behaviours observed in validated simulations.

It is also important to note the reduction in analysis effort to debug and understand agent behaviours that this approach represents for the agent system developers and domain experts. We were able to reduce the scope from over 220,000 triggerable steps to only 6 cases.

Furthermore, for these 6 cases the step's report (shown in Figure 7) displays investigative information about the beliefs changes in a given step, highlighting modifications. This helps identify possible unwanted behaviours and provides insights to the developers and domain experts.

```
Scenario: when anxiety reaches 2nd limit
                it should start a response

Given agent is type ResidentFullTime
When it believes
        responseThresholdFinalReached is true
Then it eventually believes
        current_goal is GoalFinalResponse
```

Figure 8: Failed Acceptance Criterion in validated simulation

5.2 Evaluation with faulty traces

Our previous evaluation analyzed the capabilities of our approach in a real application setting. We were able to confirm all steps that complied to the specifications are correctly identified. However, while some non-complying traces were found (i.e. 6 cases discussed previously) we need to evaluate the accuracy of our system to ensure that it captures all known errors in the agents' behaviours.

To this end, we manually modified a log of valid traces of 10 agents to deliberately introduce errors. Starting from that valid log where all 10 agents exhibited correct behaviours, we created 4 new trace logs (10 agents each) and manually introduced errors of different characteristics.

Type of error introduced	**Fault Type**	**Total**	**Found**
Then Immediately	Strong	5	5
Then Eventually	Strong	3	3
Then Never	Strong	8	8
Triggered but Given not met	Weak	19	19

Table 3: Summary of errors detected

As presented in Table 3, we introduced 35 errors distributed in the 4 traces and 40 agents. These errors covered all types of Then statements presented in §3. Additionally, we included 19 errors of Event Steps that were not Triggerable (i.e the *given* conditions are not met).

As shown in Table 3, our prototype was able to successfully capturing all types of faults introduced.

Furthermore, the prototype system identified the steps where the faulty behavior was triggered and/or the follow-on steps that caused the fault. This information is reported as shown in Figure 6 for step 38.

These results show that our system is capable of validating correct behaviours against requirements and furthermore capture behaviours non-compliant with the specifications.

6 RELATED WORK

The work of Abushark *et al.* [1] is similar to this paper in that they presented an approach to testing requirements specified via scenarios (use cases) by matching traces against scenario specifications. However, it differs to this paper in that it aims to provide early detection of issues, and so tests requirements against a detailed design (the plans of the agents) prior to implementation. In contrast, our approach validates acceptance criteria specified at the design stage against execution traces generated at run-time.

The work of Poutakidis *et al.* [25–27] is broadly similar in aim to this paper in that it monitors a running multi-agent system against design artefacts to detect mismatches (which may indicate an error in the implementation or in the design artefacts). Specifically, they focus on interaction protocols and monitoring messages, and on detecting cases where the design specifies that a BDI agent should have a particular number of applicable plans for a given goal (e.g. at least 1) but this is violated at runtime. By contrast, this paper focuses on a particular representation for acceptance criteria that derives from agile software development practices.

Thangarajah *et al.* [37] extend the notation of scenarios of the Prometheus methodology, and, amongst other things, look at using the extended scenarios for testing. Scenarios are a sequence of steps

which captures a particular run of the system akin to use cases in UML. Scenarios are much more simplistic and limited in capturing requirements compared to User and System Stories. Their work developed an agent-based simulator to the test the agent system and use the scenarios to generate test cases that are executed. The resulting steps are recorded and validated against the scenario specifications. The testing is limited to one scenario per time, hence does not consider scenarios that may interact.

Our work can be seen as a particular form of run-time verification (e.g. [8, 11, 31]). It differs from standard run-time verification in adopting a *given-when-then* format for requirements that is more accessible to domain experts, and that distinguishes between the trigger, the pre-conditions, and the expected behaviour. By contrast, run-time verification typically uses temporal logic which is less accessible, but richer. Our extended representation has temporal features, but only supports a single temporal operator (e.g. "eventually"), not arbitrary nesting of temporal and logical operators.

One example of run-time verification in a simulation setting is the work of Herd *et al.* [12, 13] which considers how to extend testing a trace of a single entity against a specification (in linear temporal logic) to: (i) deal with traces of hierarchical entities (e.g. agents in a multi-agent system), and (ii) deal with multiple traces, using probabilistic statements.

Finally, one of this paper's contributions is a taxonomy of faults in agent systems.

Winikoff [40] also proposes a taxonomy of faults (and failures) for Goal programs. Unlike this paper, the aim is to understand what faults are made by (novice) agent programmers, and how they manifest as failures, not provide a mechanism for detecting faults.

Abushark *et al.* define a simple taxonomy of failures and their causes [1, Table 1], which has three cases: (1) The specification has remaining steps to be performed, but the trace is empty, which indicates that the (abstract) trace generated from the plan graph should have more steps. This corresponds to a failure in our taxonomy. (2) The specification has been met, but the trace still has unmatched steps. This corresponds to our third case: steps in simulation that do not match a scenario. (3) The next step in the specification does not match with the next step in the (abstract) trace, which can indicate a missing step, or a difference in ordering. This corresponds to our first case. By contrast, because our specifications have not just expected steps, but also a trigger, we are able to distinguish between cases where an acceptance criterion is triggered and fails, and one where it is never triggered.

Finally, Potiron *et al.* [24] takes a broad taxonomy of faults of systems, which includes not just programming errors, but also hardware errors, and extends it with some additional fault types. For example, an autonomous agent exercising its autonomy in deciding to not respond to a request. The fault model used is quite high level and broad.

7 CONCLUSION AND FUTURE WORK

In this work we presented a novel approach to testing the requirements specified via user and system stories in developing an agent system. We extended the representation of acceptance criteria in [30] to allow agent-specific constructs and temporal constructs that were necessary to capture the requirements and subsequent test cases for our testbed system - a bushfire simulation application. We presented a formal syntax for our representation. We presented the testing process that validates the execution log traces generated by the simulation system against the specified acceptance criteria for each story. Finally, we showed that our testing approach was able to accurately detect and capture faults, when present.

Beyond the debugging features for agent systems, this approach (and support tooling) can help foster discussions between domain experts and the development teams, enhancing the understanding of agents' behaviours in large scale systems. With the benefit of a high-level language (i.e. stories) to describe the expected agent behaviours, the tests can be executed and verified directly without additional steps to interpret them into testing code. This reduces the possibility of human-introduced errors when translating the specifications into relevant tests.

There are a number of directions for future work. Firstly, investigating a new standardized logging format for agent systems to enable improved queries on agent states. This could help in the creation of shared approaches and tooling for heterogeneous application domains. These could build on previous work in this area such as [41].

Secondly, more sophisticated approaches to query simulation traces are required. While our prototype shows that our approach is feasible and sound, expanding the agents' states to a format that can be queried requires large space and computational resources. In future work, we will explore new ways of representing the agents states. In doing this, it may be possible to build on work on efficient omniscient debugging for agent systems [16].

Thirdly, the approach has potential to be used in a wider range of software systems including applications developed using other paradigms and non-simulated environments. Future applications will explore this option.

Fourthly, we plan to extend the approach to take into account interactions between agents. A Story captures the "acceptable" behaviour from one agent's perspective. When there is an interaction between two agents, there will be two scenarios capturing that, one from each agent's perspective. There is future work to be done on deriving the interactions between multiple parties and validating them.

Finally, conducting further evaluation would be valuable. In particular, it would be useful to conduct evaluations with human subjects to gauge the usability of the tool and approach, and to quantify its benefit. It would also be useful to conduct evaluation by systematically exploring mutations of BDI programs, to assess to what extent a reasonable collection of acceptance criteria are able to detect mutants. In doing this, we would build on existing work on mutation testing for BDI agents [14, 15, 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] Kent Beck and Cynthia Andres. 2005. *Extreme Programming Explained: Embrace Change* (2nd ed ed.). Addison-Wesley, Boston, MA.
[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] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. 2004. TROPOS: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agents and Multi-Agent Systems* 8 (May 2004), 203–236.
[5] 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.
[6] Mike Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison-Wesley, Boston.
[7] Rick Evertsz, John Thangarajah, and Thanh Ly. 2019. *Practical Modelling of Dynamic Decision Making*. Springer. https://doi.org/10.1007/978-3-319-95195-9
[8] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (LNCS, Vol. 11237)*, Christian Colombo and Martin Leucker (Eds.). Springer, 241–262. https://doi.org/10.1007/978-3-030-03769-7_14
[9] Alma María Gómez-Rodríguez and Juan Carlos González Moreno. 2010. Comparing Agile Processes for Agent Oriented Software Engineering. In *Product-Focused Software Process Improvement, 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings (Lecture Notes in Business Information Processing, Vol. 6156)*, Muhammad Ali Babar, Matias Vierimaa, and Markku Oivo (Eds.). Springer, 206–219. https://doi.org/10.1007/978-3-642-13792-1_17
[10] Juan C. González-Moreno, Alma Gómez-Rodríguez, Rubén Fuentes-Fernández, and David Ramos-Valcárcel. 2014. INGENIAS-Scrum. In *Handbook on Agent-Oriented Design Processes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 219–251.
[11] Klaus Havelund and Grigore Rosu. 2018. Runtime Verification - 17 Years Later. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (LNCS, Vol. 11237)*, Christian Colombo and Martin Leucker (Eds.). Springer, 3–17.
[12] Benjamin Herd, Simon Miles, Peter McBurney, and Michael Luck. 2015. Monitoring Hierarchical Agent-based Simulation Traces. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*. ACM, 463–471.
[13] Benjamin Herd, Simon Miles, Peter McBurney, and Michael Luck. 2018. Quantitative analysis of multi-agent systems through statistical verification of simulation traces. *Int. J. Agent Oriented Softw. Eng.* 6, 2 (2018), 156–186.
[14] Zhan Huang and Rob Alexander. 2015. Semantic Mutation Testing for Multi-agent Systems. In *Engineering Multi-Agent Systems - Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers (LNCS, Vol. 9318)*. Springer, 131–152.
[15] Zhan Huang, Rob Alexander, and John A. Clark. 2014. Mutation Testing for Jason Agents. In *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers (LNCS, Vol. 8758)*. Springer, 309–327.
[16] Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. 2017. Omniscient Debugging for Cognitive Agent Programs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 265–272.
[17] 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.
[18] Dan North. 2006. Introducing BDD. https://dannorth.net/introducing-bdd/.
[19] Lin Padgham, Dhirendra Singh, and Fabio Zambetta. 2015. Social Simulation for Analysis, Interaction, Training and Community Awareness. In *Proceedings of the 2015 Winter Simulation Conference (WSC '15)*. IEEE Press, Piscataway, NJ, USA, 3130–3131.
[20] Lin Padgham and Michael Winikoff. 2004. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons.
[21] Lin Padgham, Zhiyong Zhang, John Thangarajah, and Tim Miller. 2013. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. *IEEE Trans. Software Eng.* 39, 9 (2013), 1230–1244. https://doi.org/10.1109/TSE.2013.10
[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*. Springer, 149–174.
[24] Katia Potiron, Amal El Fallah Seghrouchni, and Patrick Taillibert. 2013. *From Fault Classification to Fault Tolerance for Multi-Agent Systems*. Springer.
[25] David Poutakidis, Lin Padgham, and Michael Winikoff. 2002. Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*. ACM, 960–967.
[26] David Poutakidis, Lin Padgham, and Michael Winikoff. 2003. An Exploration of Bugs and Debugging in Multi-agent Systems. In *Foundations of Intelligent Systems, 14th International Symposium, ISMIS 2003, Maebashi City, Japan, October 28-31, 2003, Proceedings (LNCS, Vol. 2871)*, Ning Zhong, Zbigniew W. Ras, Shusaku Tsumoto, and Einoshin Suzuki (Eds.). Springer, 628–632.
[27] David Poutakidis, Michael Winikoff, Lin Padgham, and Zhiyong Zhang. 2009. Debugging and Testing of Multi-Agent Systems using Design Artefacts. In *Multi-Agent Programming, Languages, Tools and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Springer, 215–258.
[28] Anand S. Rao and Michael P. Georgeff. 1991. Modeling Rational Agents within a BDI-Architecture. In *Proc. of KR'91*. Cambridge,MA, 473–484.
[29] Joel Robertson, Dhirendra Singh, Leorey Marquez, Vincent Lemiale, Peter Ashton, Trevor Dess, Justin Halliday, Pawan Gamage, and Mahesh Prakash. 2021. Accounting for seasonal populations in bushfire evacuation modelling and planning: A Surf Coast Shire case study. In *AFAC Conference*. Sydney, Australia.
[30] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. 2021. User and System Stories: An Agile Approach for Managing Requirements in AOSE. In *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé (Eds.). ACM, 1064–1072.
[31] Kristin Yvonne Rozier. 2019. From Simulation to Runtime Verification and Back: Connecting Single-Run Verification Techniques. In *2019 Spring Simulation Conference, SpringSim 2019, Tucson, AZ, USA, April 29 - May 2, 2019*. IEEE, 1–10.
[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] Dhirendra Singh and Lin Padgham. 2015. Community Evacuation Planning for Bushfires Using Agent-Based Simulation: Demonstration. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems (AAMAS '15)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1903–1904.
[34] Dhirendra Singh and Lin Padgham. 2017. Emergency Evacuation Simulator (EES) - a Tool for Planning Community Evacuations in Australia. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Melbourne, Australia, 5249–5251.
[35] Dhirendra Singh, Ken Strahan, Jim McLennan, Joel Robertson, and Bhagya N. Wickramasinghe. 2021. What will they do? Modelling self-evacuation archetypes. *CoRR* abs/2105.12366 (2021). https://arxiv.org/abs/2105.12366
[36] Jan-Philipp Steghöfer, Hella Seebach, Benedikt Eberhardinger, Michael Huebschmann, and Wolfgang Reif. 2015. Combining PosoMAS Method Content with Scrum: Agile Software Engineering for Open Self-Organising Systems. *Scalable Comput. Pract. Exp.* 16, 4 (2015), 333–354.
[37] John Thangarajah, Gaya Buddhinath Jayatilleke, and Lin Padgham. 2011. Scenarios for system requirements traceability and testing. 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, 285–292.
[38] Yves Wautelet, Samedi Heng, Soreangsey Kiv, and Manuel Kolp. 2017. User-Story Driven Development of Multi-Agent Systems: A Process Fragment for Agile Methods. *Computer Languages, Systems & Structures* 50 (Dec. 2017), 159–176.
[39] Michael Winikoff. 2005. JACK Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming*. Springer, New York, NY, 175–193.
[40] Michael Winikoff. 2014. Novice programmers' faults & failures in GOAL programs. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014*, Ana L. C. Bazzan, Michael N. Huhns, Alessio Lomuscio, and Paul Scerri (Eds.). IFAAMAS/ACM, 301–308.
[41] Michael Winikoff. 2017. Debugging Agent Programs with Why? Questions. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS '17)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 251–259.
[42] Michael Winikoff and Stephen Cranefield. 2014. On the Testability of BDI Agent Systems. *J. Artif. Intell. Res.* 51 (2014), 71–131. https://doi.org/10.1613/jair.4458
[43] Zhiyong Zhang, John Thangarajah, and Lin Padgham. 2009. Automated Testing for Intelligent 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)*. Springer, 66–79.