Integrating Expectation Monitoring into Jason: A Case Study Using Second Life*

Surangika Ranathunga, Stephen Cranefield, and Martin Purvis

Department of Information Science,University of Otago,
PO Box 56, Dunedin 9054, New Zealand
{surangika,scranefield,mpurvis}@infoscience.otago.ac.nz

Abstract. Previous work on detecting fulfilments and violations of expectations (which may correspond to conventions, norms, commitments or contracts) assumed that information about the world is available as an abstract logical model of observed properties and/or events. There has been little investigation of practical techniques for mapping from sensor readings in a complex environment to such a logical model suitable for monitoring techniques. Moreover, there has been little work on investigating practical techniques for agents to respond to fulfilments and violations of their expectations. This paper investigates these two aspects in the context of interactions involving multiple Jason agents in a Second Life simulation. We present a framework that can be used to connect any agent platform with Second Life, and demonstrate how this framework is integrated with the Jason platform to monitor expectations of agents.

1 Introduction

In open and distributed multi-agent systems, agents must plan and act with incomplete and imperfect information about the environment they are operating in and the other agents that they may need to interact with. To perform efficiently in the face of this uncertainty, agents cannot consider all possible actions of others, but instead must base their decision-making on some observed or declared order in their society. This sense of order may come from published norms, agreed contracts, commitments created through interaction with other agents, or personally inferred regularities of behaviour, and it warrants agents to have expectations about the future behaviour of others or the state of the environment.

However, it is important that an agent is capable of reacting to fulfilments and violations of its expectations. When the agent is operating in a complex environment, this may involve several steps: taking sensor readings from the environment where the multiagent system is deployed, presenting this information as an abstract model, using a monitoring technique to identify the fulfilments and violations based on the created model, and taking reactive measures based on the detected fulfilments or violations.

* This paper is available online at http://eprints.otago.ac.nz/1016/

Previous work on detecting fulfilments and violations of agent expectations has mainly assumed that the information about the world is already available as an abstract logical model of observed properties and/or events [1, 2]. However, when deploying multiagent systems in complex environments, it is important to define mechanisms to get the sensor readings from the environment and map these sensed observations to an abstract model of observations, suitable for the selected monitoring technique. This phase could be non-trivial when considered the number of agents involved and the complexity of interactions.

This paper investigates this challenge in the context of agent interactions that take place in the popular multi-purpose virtual world Second Life [3], which facilitates the creation of complex simulations. We have implemented a framework that can be used to connect any agent platform with Second Life, and this framework is also capable of extracting near accurate agent sensory data from Second Life and encoding this as a sequence of propositional state descriptions.

This paper also addresses the problem of enabling an agent to directly monitor and respond to fulfilments and violations of its expectations. There has been little work on investigating practical techniques in this area. We propose and implement a tight integration of expectation monitoring with the Jason BDI agent model, where domain-specific individual agents can identify the fulfilments or violations of their expectations based on observations of their environment. In our work, the Jason interpreter is extended with built-in actions to initiate and terminate monitoring of expected constraints on the future. New event types are also introduced to the Jason interpreter, representing the addition and deletion of newly triggered fulfilments and violations. This allows plans to be written to respond to such events, either for specific fulfilments and violations or to handle all such events in a generic manner.

Thus our mechanism allows an agent to initiate a service to monitor the expectations that it is interested in, and multiple instances of the monitoring service may be active on behalf of different agents at any time. However, this does not rule out the use of a single designated monitoring agent to monitor expectations (e.g. norms) applying to the whole society. Such a monitor agent can also make use of the techniques discussed in this paper.

In this case study, we integrate this extended Jason BDI interpreter with our Second Life connection framework to implement a Jason agent simulation inside Second Life. A Jason agent wanting to monitor the interactions of other agents and objects can sense the Second Life environment around it and can record these sensor readings, with the use of the framework. In this work, we employ an expectation monitor developed previously [1] to identify the fulfilments or violations of an agent's expectations. The agent's expectations are defined as properties to be monitored in the expectation monitor and the abstract model created by the framework is used by this expectation monitor to identify the fulfilments or violations of the defined expectations.

Therefore, the contribution of this paper is twofold. Firstly, it presents a framework that can be used to deploy any agent platform in Second Life simulations, which is also capable of extracting low level sensory data of agents and

converting them into an abstract logical model of observed properties and /or events. Secondly, it implements a tight integration of expectation monitoring with the Jason BDI model. The case study described in this paper demonstrates both these aspects.

The rest of the paper is organized as follows. Section 2 describes the potential of Second Life as a simulating environment for multiagent systems. Section 3 describes the Jason platform and Section 4 describes the expectation monitor in brief. Section 5 describes the developed system and in Section 6, we demonstrate this developed system by means of two examples. Section 7 discusses some related work and finally, Section 8 concludes the paper.

2 Second Life as a Simulation Environment for Multiagent Systems

Second Life provides a sophisticated and well developed virtual environment to test AI theories, including agent based modeling. With a user base of over 10 million, and with the virtual presence of thousands of organizations, Second Life contains more interaction possibilities, which would inherently lead to the provision of new scenarios to be used in multiagent system testing. Second Life is not restricted to a specific gaming or training scenario. Developers can create a multitude of scenarios as they wish, using the basic building blocks that are provided. For example, in Second Life, these scenarios could be in the areas of education, business, entertainment, health or games. The significance of using Second Life scenarios lies in the fact that they can be carried out between software controlled agents, and also between software controlled agents and human controlled agents. This provides ample opportunities to model intelligent agents by making them observe or interact with human controlled agents [4, 5].

Second Life has been identified as a good simulation platform for testing AI theories [6] and specifically multiagent systems [5]. A detailed analysis on the benefits of using Second Life over the traditional 2D simulations and physical robots has also been done [6], with the main advantage being the ability to create sophisticated test beds in comparison to 2D simulations, and more cost effective test beds when compared to physical robots. However, still we do not see Second Life being used for complex simulations of AI theories or multiagent modeling, despite the existence of some research on creating Intelligent Virtual Agents (IVA) in Second Life [7].

The lack of use of Second Life as a simulation environment for AI research can be, to a certain extent, attributed to the lack of programming interface it provided earlier. The traditional programming in Second Life is done using in-world scripts created using the proprietary language *Linden Scripting Language (LSL)*. These scripts are created inside objects, and in order to use them to control an agent inside Second Life, the objects should be attached to the agent. This approach has lot of limitations, and only very simple AI simulations have been created using this technique so far [6].

However, with the development of the third party library LibOpenMetaverse (LIBOMV), Second Life can now be accessed through a more sophisticated programming interface. "LIBOMV is a .Net based client/server library used for accessing and creating 3D virtual worlds" [8], and is compatible with the Second Life communication protocol. With appropriate programming techniques, the LIBOMV library can be used to create avatars ("bots") that have behavioral abilities similar to those controlled by humans. This includes moving abilities such as walk, run or fly, performing animations such as cry, or laugh, communication abilities using instant messaging or public chat channels, and ability to sense the environment around it. Moreover, just like their human controlled counterparts, LIBOMV based avatars can operate in any Second Life environment or simulation, thus there is no need to create specific simulations to deploy them.

2.1 SecondFootball Virtual Football Simulation in Second Life

We employ the SecondFootball virtual football scenario [9] in Second Life as an example simulation system to portrait how our system could be used in monitoring complex agent interactions in Second Life. SecondFootball is an interesting simulation in Second Life which enables playing virtual football (soccer). This system provides scripted stadium and ball objects that can be deployed inside Second Life, as well as a "head-up display" object that an avatar can wear to allow the user to initiate kick and tackle actions. With a large number of players and a scripted ball that often move around the football field at high speed, SecondFootball matches provide a multitude of challenges when recording events of interest that take place.

First and foremost, it is practically difficult to retrieve accurate position information of the fast moving avatars and the ball. Moreover, in order to make sure that no important event is missed out, recording of data has to be done frequently, which results in a very high volume of low-level position and animation data. High-level, football domain specific events such as passing the ball, tackling to get the ball, or scoring a goal are embedded in this low-level data, making it very difficult to retrieve them in a straightforward manner.

As described in the Section 5.2, in order to overcome these challenges, we introduce a combined approach that attaches an LSL script to the monitoring LIBOMV client. These communicate with each other and produce near accurate position information of the players and the ball. Then we employ a complex event detection technique on these retrieved data to identify the domain specific high-level complex events of interest.

3 Jason

Jason [10] is an open source Java interpreter for an extended version of the agent programming language AgentSpeak [11], which is based on the BDI agent model. A Jason agent program consists of plans that are executed in response to events

received. These events are generated by the addition or deletion of the agent's beliefs and goals. A belief is the result of a percept the agent retrieves from its environment or it could be based on a message received from another agent. A goal could be internally generated by the agent, or it could be a goal that was asked to be achieved by another agent. While executing a plan, an agent might generate new goals, act on its environment, create mental notes to itself or execute internal actions.

4 Expectation Monitor

The expectation monitor used in this work [1] aims at monitoring expectations that encode complex temporal constraints on the future.

The language that the expectation monitor accepts includes the following operators relating to conditional rules of expectation:

$$\mathsf{ExistsExp}(\mathit{Condition}, \mathit{Expectation})$$
$$\mathsf{ExistsFulf}(\mathit{Condition}, \mathit{Expectation})$$
$$\mathsf{ExistsViol}(\mathit{Condition}, \mathit{Expectation})$$

where *Condition* and *Expectation* are formulae in a form of linear propositional temporal logic [1]. *Condition* expresses a condition on the past and present, and *Expectation* is a constraint that may express an expectation on the future or a check on the past (or both). An ExistsExp formula states that an expectation currently exists due to the rule defined by its two arguments having fired in a previous or the current state. ExistsFulf and ExistsViol formulae state that the rule defined by the arguments has created an expectation that has become fufilled or violated, respectively, in the current state.

An expectation *exists* when its condition evaluates to true in the current state. We also refer to this as an *active* expectation. The expectation is then considered to be fulfilled or violated if it evaluates to true (without considering any future states that the monitor might already know about if it is running in an "offline" mode).

If an active expectation is not fulfilled or violated in a given state, then it remains active in the following state, but in a "progressed" form. Formula progression involves partially evaluating the formula in terms of the current state and re-expressing it from the viewpoint of the next state [12], e.g. if p holds in the current state, then an expectation that "p is true and in the next state, q is true" will progress to the simpler expectation that "q is true".

Note that although this work is using a specific monitor at present, the interface between Jason and the monitor is designed to be more abstract, so that other monitors could be used. The Jason internal actions that communicate with the monitor are not passed the property in the form given above, e.g. $\mathsf{ExistsExp}(\mathit{Condition}, \mathit{Expectation})$. Instead the condition and expectation of a conditional rule of expectation are separately passed to them, along with

the mode for monitoring the formula (fulfilment or violation of an expectation in the current state, based on the specified rule)[1].

5 System Design

5.1 Main Components of a Monitoring Agent

A monitoring agent in our system consists of three main components.

LIBOMV Client: The LIBOMV client creates and controls an avatar inside the Second Life server. A LIBOMV client can sense the environment around it based on the avatar and object movement updates, and avatar animation updates received by it. It can also communicate with other avatars using the chat channels or instant messaging. It also executes the commands sent by the Jason agent (e.g moving around or playing animations), and reports back the needed information such as the result of the login attempt.

Expectation Monitor: The system employs multiple expectation monitor instances in parallel in order to monitor multiple concurrently active expectations of an agent. This makes it easier to start and stop individual expectation monitor instances dynamically, in order to initiate or terminate monitoring of individual expectations. When an expectation monitor is initially started, it receives the property that it should monitor and the condition to start monitoring. When it starts receiving the states one by one, it matches these against the expectation and progressively simplifies the monitoring expectation and finally deduces fulfilment or violation.

Jason Agent: A Jason agent is the coordinator component of this system. It instantiates the LIBOMV client to create the corresponding Second Life avatar, and it initializes and terminates the expectation monitor instances. The Jason platform is extended with new internal actions to handle this, and the actual initialization and termination of the expectation monitor is encapsulated within these internal actions. This enables the use of different monitoring techniques, without changing the agent's reasoning process.

The internal action corresponding to the initialization of expectation monitoring is *start_monitor*. This internal action is used when an agent decides to monitor one of its expectations, and this should not be confused with an expectation logically becoming active due to its condition being satisfied. It is not necessarily be the case that the expectation comes into existence when the monitoring begins, or that an logically active expectation will be monitored, as the existence of an expectation is independent of whether it is being monitored. The

[1] Even though our expectation monitor supports monitoring for the existence of an expectation, when integrating the monitor with Jason, we currently handle only the fulfilment and violation of an expectation.

start_monitor internal action takes in the following parameters:

monitoring_mode: The type of expectation, which can be a fulfilment or violation.

expectation_name: Assigns a name to the expectation, for the ease of future reference.

monitor_tool: Name of the monitoring tool that should be used to monitor this expectation.

condition: Specifies the requirements on the past and present that activates the expectation monitoring.

expectation: Specifies the actual expectation.

context_information_list: Any other information that might be useful for expectation monitoring. This contextual information will be added as annotations to the received fulfilments and violations as they could be useful in plan selection and execution.

The internal action *stop_monitor* stops the monitoring of the expectation. It takes the following parameters:

expectation_name: The name of the expectation.

monitor_tool: The monitoring tool that is currently running the specified expectation. This enables the execution of logic specific to the termination of the given monitoring tool.

Note that stopping monitoring of a particular expectation does not mean that the expectation will no longer be existing, fulfilled or violated, it is just that the agent has no interest in monitoring for the given expectation anymore.

We added components to the Jason interpreter to store the detected fulfilments and violations, similar to how the perceptions are stored. We term these as the *FulfilmentBase* and the *ViolationBase*, which also have functions facilitating the addition, deletion and retrieval of fulfilments and violations, respectively.

The Environment class acts as the interface to integrate the Jason platform with outside simulation environments. Therefore the Environment class was selected as the module that first collects the output of an expectation monitor. Consequently, fulfilments and violations of agent expectations identified by the monitoring tool are added to the Jason environment, to be further processed by an agent's reasoning cycle. The following functions are introduced to add, delete and retrieve fulfilments and violations to and from the Environment class: getFulfilments, getViolations, consultFulfilments, consultViolations, addFulfilment, addViolation, removeFulfilment, removeViolation, removeFulfilmentsByUnif, removeViolationsByUnif, clearFulfilments, clearViolations, containsFulfilment, containsViolation. Most of these method names are self explanatory, and we note that a similar set of functions are available in the Environment class to handle the received percepts.

The agent reasoning cycle is modified so that it retrieves the newly received notifications on the fulfilments and violations of its expectations from the Jason Environment class, similar to how it retrieves the newly received percepts. These fulfilments and violations are then added to the agent's FulfilmentBase and

the ViolationBase using the newly introduced fuf(Fulfilment Update Function) and vuf(Violation Update Function) functions, respectively. Upon the identification of newly received fulfilments and violations, the fuf and vuf functions (respectively) generate corresponding triggering events and eventually plans in the agent's plan library corresponding to these events will get executed[2].

5.2 Overall System Design

Communication Between the LIBOMV Client and the Jason Agent: Communication between the LIBOMV client and the Jason agent is facilitated through sockets, and a pre-defined protocol. This protocol defines how an agent should pass commands to the LIBOMV client, and how an agent platform can interpret a message sent by the LIBOMV client. This decoupling makes it possible to connect any agent platform with the LIBOMV clients easily, and it could well be the case that the LIBOMV clients are connected with agents in different agent platforms.

The module that contains Jason agents is capable of handling multiple concurrent instances of socket connections connected to the Jason agents. Similarly, the module that contains LIBOMV clients is capable of handling multiple concurrent LIBOMV clients and socket connections. A Jason agent connects to its interface socket through the Jason Environment class, and the Jason Connection manager interface. The Jason connection manager and the LIBOMV connection manager together assure that all these individual Jason agents are connected to the correct LIBOMV client, through the interface sockets (in Figure 1, JS represents a socket connecting to a Jason agent, and LS represents a socket connecting to a LIBOMV client).

Communication Between the LIBOMV Client and the Second Life Server: Even though a LIBOMV client is capable of sensing the environment around it better than an LSL script as mentioned in Section 2, it has some limitations when recording position information of fast moving avatars and objects, and of small objects that are out of its visibility range[3]. Therefore we have come up with a combined approach to extract data from Second Life. In this new approach, a scripted object is attached to the LIBOMV client. Detection of the avatars and objects to be monitored is done at the LIBOMV client side. Identification information for these is then sent to the script. As the script already knows what to be tracked, a more efficient, light-weight function can be used to record the position and velocity information, instead of the normal sensor function. Position and velocity data is recorded at regular time intervals (every 500ms for the SecondFootball domain) and sent back to the LIBOMV client.

[2] Note that in this case study we currently only communicate the output of the expectation monitor to the Jason agent. However, it is equally possible to communicate the observed properties directly to the Jason agent, to be treated as regular perceptions that result in agent beliefs

[3] Technical details of these limitations are out of the scope of this paper

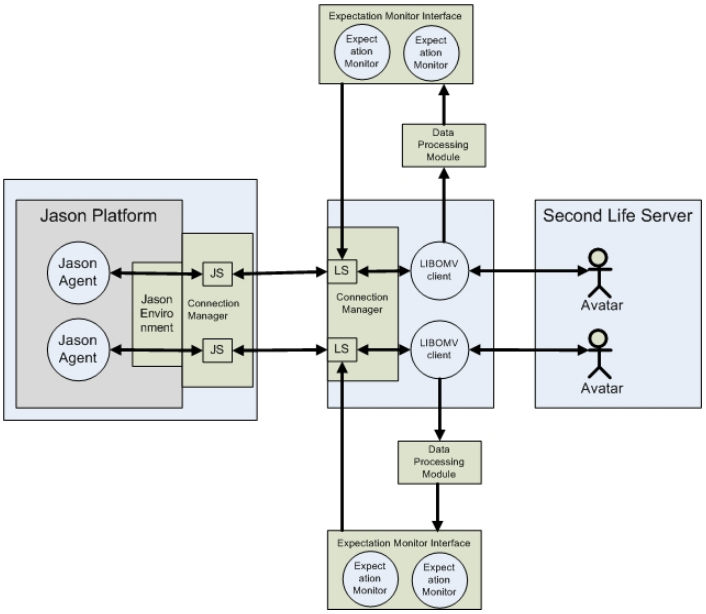

Fig. 1. Overall System Design

Avatar animation data are directly captured by the LIBOMV client to make sure instantaneous animations are not missed out, as well as the communication messages that are exchanged.

Communication Between the LIBOMV Client and the Expectation Monitor Interface: This is handled by the data processing module, shown in Figure 1. Its responsibility is to identify the high-level domain specific events embedded in the recorded sensory data and creating snapshots of the environment that correspond to the abstract model required by the expectation monitor, as the expectation monitor takes a sequence of states as input. The data processing module consists of three components:

Data Pre-processor: The low-level data received from the Second Life server is used to deduce basic high-level information of the avatars and objects, e.g. whether an avatar is moving and if so in which direction and the moving type. Other contextual information such as the location of the avatar can also be attached to this retrieved information. The output of the data pre-processor is a snapshot that contains everything that took place in the Second Life environment at a given unit of time, which includes position of avatars and objects, animations of avatars, and messages that were exchanged. This snapshot is used to generate events corresponding to the position, animation, and communication information, to be sent to the complex event detection module described next.

Complex Event Detection Module: An event stream processing engine called Esper[4] is used to identify the complex high-level domain-specific events embedded in the event stream of position events and animation events. Event identification is done in several levels to facilitate the detection of events with a duration. For example, when identifying the domain-specific high-level events embedded in a SecondFootball match described in Section 2.1, the first level includes identification of events such as a player kicking the ball, a player taking the ball down the field, and a player doing a tackle. Then these events are further processed to identify more complex events. For example, a player kicking the ball could result in a successful or an unsuccessful pass to another player, or it could result in a goal being scored.

Event Post-processor: The event post-processor assigns the identified complex events into the correct snapshot, which corresponds to the exact time that the complex event was generated. With this, a snapshot now includes all the low-level and high-level events that took place in a given unit of itme. Also, for those events that depend on previous other high-level events (e.g. a goal score event is associated with a kick event that took place sometime before it in the SecondFootball scenario), a reference to the dependent state should also be added. Once processed, a state of a SecondFootball match sent to the expectation monitor would look like: "Kick,PenaltyB", which states that a kick was done on the ball inside PenaltyB area. Another example could be: "Kick-GoalScore(9),GoalB", which states that a goal was scored at the Goal B, and this was initiated by a kick action that happened in the state 9.

6 Example Scenarios

6.1 Jason Agents Deployed in Second Life in a Leader-Follower Scenario

We employ a simple leader follower scenario to explain how the overall system works. Below given is the Jason agent code snippet for the leader agent[5]. The rule condition and expectation parameters shown in italics are meta-variables that stand for specific formulae that are listed after the Jason code. Here the leader is represented by the agent Ras_Ruby and the follower by the agent Sura_Haroldsen (with the abbreviated name SH). We have a simple reputation system where the leader has an initial belief that represents the reputation it has for its follower, which is set to 10. If the leader detects the follower violating any of its defined expectations, he reduces the follower's reputation by one.

When the Jason system starts, it will log in the LIBOMV clients in the Second Life server. When the follower identifies the presence of the leader, he notifies the leader about his presence. The leader has an expectation that the

[4] Esper Tech, Esper tutorial. `http://esper.codehaus.org/tutorials/tutorial/tutorial.html`

[5] To save space, we have only shown plans that are required to explain the newly added functionality of handling fulfilments and violations.

follower should "walk" from section1 (where the follower currently stands) to section0, without stopping. The leader asks the follower to walk to section0, and at the same time starts monitoring for his expectation. The leader also has an expectation that his follower should never fly. The follower reaches section0, and when the leader detects this, it asks the follower again to "run" back to section1. The stubborn follower, at this point decides that it would fly to section1, and when he starts flying, the leader identifies this violation, decides to reduce the reputation it had for its follower.

Leader Reasoning Code

```
/* Initial beliefs */
reputation("sura_haroldsen", 10).

/* Plans */
+logged_in(Avatar) [source(Avatar)] <-
    monitor.start_monitor(
        "fulf", "walk_to_target", "expectation_monitor",
         Condition1 ,  Expectation1 ,  ContextInfoList);
    monitor.start_monitor(
        "viol", "fly", "expectation_monitor",
         Condition2 ,  Expectation2 ,  ContextInfoList);
    // ask the follower agent to walk to section0
    .send(Avatar, achieve, walk("section0")).

/* Detects the fulfilment of the expectation "walk_to_target" */
+fulfill("walk_to_target")[agent_name("sura_haroldsen")] <-
    // Follower walked to target, ask him to run back to section 1
    .send(Name, achieve, run("section1")).

/* Detects the violation of the expectation "fly" */
+violate("fly")[agent_name("sura_haroldsen")] <-
    monitor.stop_monitor("walk_to_target","expectation_monitor");
    .print("He flew");
    -reputation(Name, R);
    +reputation(Name, R-1).
```

As shown in the leader's code above, the internal function start_monitor is used to initiate monitoring for constraints on the future. The parameter "expectation_monitor" specifies that the expectation monitor described in Section 4 is used for monitoring. We name the first expectation "walk_to_target", which checks for a fulfilment of an expectation where *Condition1* is a string containing the formula `('and','SH_stand','SH_Section1')`[6], meaning that the follower is standing inside Section 1, and *Expectation1* is a string containing the following formula:

[6] The formulae are currently encoded as strings containing Python expressions, for ease of integration with our Python-based expectation monitor.

```
('Next',('U',('and','SH_AdvanceToSection0','SH_walk'),
             ('and','SH_Section0','SH_stand')))
```

This states that from the next state onwards, the expectation monitor should receive states in which the propositions `SH_AdvanceToSection0` and `SH_walk` are true until it receives a state in which `SH_Section0` and `SH_stand` hold. This means that the follower should keep on walking until it reaches section0 and stand there.

The second expectation, named "fly", checks for a violation of the expectation that the follower should never fly. *Condition2* is a string containing the proposition `'SH_fly'` and *Expectation2* is a string containing the Boolean value `False`, i.e. this expectation will be violated immediately whenever it is triggered in a state in which `SH_fly` holds.

In this example, the parameter **ContextInfoList** contains only the name of the agent that should be monitored, in the format `agent_name(sura_haroldsen)`. Note how the annotation generated from this contextual information has been used in the subsequent plans that handle the detected fulfilments and violations in order to react only to fulfilments or violations related to the agent Sura Haroldsen.

In the final plan, the call to the internal function `stop_monitor` stops the monitoring of the fulfilment expectation "walk_to_target".

6.2 Monitoring Expectations in SecondFootball Virtual Football Matches

This example demonstrates how the developed system is capable of monitoring complex agent interactions. In this scenario, a Jason agent is monitoring the interactions of human controlled football players, as our Jason agents are still not capable of handling complex reasoning involved with playing football.

In this football training exercise, the Jason agent (who is a coach) expects the player Su_Monday (with the abbreviated name SM) to advance towards the Goal B with the ball, and while he is in the field section MidfieldB2, he should successfully kick and pass the ball to his teammate Sura_Haroldsen (with the abbreviated name SH) who is in the PenaltyB area. Sura_Haroldsen then should make sure that he moves the ball forward towards the Goal B and kicks the ball into the Goal B while still being inside the Penalty B area.

The following property *f*, which we have broken into sub-parts for convenience, captures the fulfilment of this expectation:

```
f =  ('ExistsFulf',f1,
       ('U',f1,
         ('and',f2,
           ('bind','x',('U',True,('and',f3,f4,('Next',('U',f4,f5)))))))))
f1 = ('and','SM_MidfieldB2','SM_inPossession','SM_advanceToGoalB')
f2 = ('and','SM_MidfieldB2','SM_kick')
f3 = ('Exists','SM_successfulPass y',('@x','y'))
```

```
f4 = ('and','SH_PenaltyB','SH_inPossession','SH_advanceToGoalB')
f5 = ('and',('and','SH_PenaltyB','SH_kick'),
       ('bind','a',
         ('U',True,('Exists','SH_goalScoreGoalB b','@a','b')))))
```

Here, *f1* is the triggering condition. It states that the monitoring should start when Su_Monday takes possession of the ball in MidfieldB2 area and starts moving towards Goal B. Then the first part of the expectation states that Su_Monday should keep on taking the ball down the field towards Goal B, until he kicks the ball towards Sura_Haroldsen, which is denoted by *f2*. Sura_Haroldsen should successfully receive the ball in PenaltyB area. *f3* identifies this successful pass, which is bound to the kick action by Su_Monday that happened sometime ago. Then Sura_Haroldsen has to take the ball towards Goal B, and kick the ball into the goal, and the expectation monitor eventually identifies the fulfilment of the expectation.

The `bind` operator associates the variable that immediately follows it with the state that the kick event happened. For example, in the successful pass section, the state that contains the conjunction of the propositions `'SM_MidfieldB2'` and `'SM_kick'` is associated with the variable x. Then the expectation monitor looks for the existence of the proposition `SM_successfulPass(y)` in an incoming state, where y is initialized with the state number at which the corresponding kick event happened. At the presence of this new state, the `Exists` operator compares the x and y variable values that are now initialized, and if they match, it deduces that this successful pass is a result of the kick event that happened at the state x.

When the system starts, the Jason agent is deployed in the SecondFootball field and the expectation monitor connected to it is passed the above property. Once the human controlled players start playing, this generates the low level information containing the position information of the players and the ball, and agent animation information. The system processes these data and identify the high-level football domain specific events such as a player kicking the ball, a player being in possession of the ball, a player advancing towards a goal, a successful pass and a goal score by a player. These identified events are prefixed with the player's name and are passed into the expectation monitor as propositions, after grouping under states. The expectation monitor matches the incoming states with its defined rule, and when the players correctly finish the scenario, the expectation monitor finally deduces the fulfilment of the agent's expectation.

7 Related Work

There have been a number of works addressing the interplay of the BDI architecture and norms, to make agents norm-aware in their practical reasoning [13–15]. In this paper we have focused on a different issue: the use of a BDI interpreter as a practical way to respond to fulfilments and violations of expectations (which

could be generated by rules representing norms). Meneguzzi et al. [16] have addressed a similar issue in the context of contract-based agent systems. In their work, special agents monitor contracts between other agents, and send notifications to the manager agents that handle violations. These agents, progammed using Jason, have plans that react to the new beliefs resulting from these messages. In our work we aim to have a tighter integration of expectation monitoring with Jason by considering the monitor as a service available within Jason and by introducing new violation and fulfilment event types. Agents are able to initiate and terminate monitoring directly, without the involvement of other agents.

8 Conclusion

In this paper we demonstrated an integration of expectation monitoring with the BDI agent model and presented a practically implemented mechanism to monitor expectations of individual agents in the Jason agent model. We also presented a framework that can be used to connect any agent platform with Second Life, which is also capable of mapping sensor readings from a complex environment to an abstract model which can be used to monitor the fulfilments and violations of agent expectations. Then we demonstrated how this extended version of Jason is integrated with this framework to deploy Jason agents inside Second Life, who are capable of monitoring their expectations based on what is happening in their Second Life environment.

There are a number of directions for further research in this area. First and foremost, the system should be enhanced so that the monitoring agent could first inform the monitored agents about its expectations. Also, the reasoning part of the monitored agent should be improved, so that it considers those during its deliberation process. Currently the developed framework is very much optimized for the SecondFootball domain, therefore in our future work, we intend to make this framework more general to be used with any Second Life simulation.

References

1. Cranefield, S., Winikoff, M.: Verifying social expectations by model checking truncated paths. Journal of Logic and Computation (2010). Advance access, doi: 10.1093/logcom/exq055
2. Spoletini, P., Verdicchio, M.: An automata-based monitoring technique for commitment-based multi-agent systems. In: Coordination, Organizations, Institutions and Norms in Agent Systems IV, *Lecture Notes in Computer Science*, vol. 5428, pp. 172–187. Springer (2009)
3. Second Life: `http://secondlife.com/`
4. Bogdanovych, A., Simoff, S., Esteva, M.: Virtual institutions: Normative environments facilitating imitation learning in virtual agents. In: Intelligent Virtual Agents, *Lecture Notes in Artificial Intelligence*, vol. 5208, pp. 456–464. Springer (2008)

5. Weitnauer, E., Thomas, N.M., Rabe, F., Kopp, S.: Intelligent agents living in social virtual environments – bringing Max into Second Life. In: Intelligent Virtual Agents, *Lecture Notes in Artificial Intelligence*, vol. 5208, pp. 552–553. Springer (2008)
6. Veksler, V.D.: Second Life as a simulation environment: Rich, high-fidelity world, minus the hassles. In: Proceedings of the 9th International Conference on Cognitive Modeling (2009)
7. Ullrich, S., Bruegmann, K., Prendinger, H., Ishizuka, M.: Extending MPML3D to Second Life. In: Intelligent Virtual Agents, *Lecture Notes in Artificial Intelligence*, vol. 5208, pp. 281–288. Springer (2008)
8. OpenMetaverse Foundation: `http://www.openmetaverse.org/projects/libopenmetaverse`
9. Vstex Company: Second Life football system. `http://www.secondfootball.com/`
10. Bordini, R., Hübner, J., M., W.: Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons (2007)
11. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, *Lecture Notes in Artificial Intelligence*, vol. 1038, pp. 42–55. Springer (1996)
12. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. Artificial Intelligence **116**(1-2), 123–191 (2000)
13. Dignum, F., Morley, D., Sonenberg, E.A., Cavedon, L.: Towards socially sophisticated BDI agents. In: Proceedings of the Fourth International Conference on MultiAgent Systems, pp. 111–118. IEEE Computer Society (2000)
14. Meneguzzi, F., Luck, M.: Norm-based behaviour modification in BDI agents. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, pp. 177–184. IFAAMAS (2009)
15. Neto, B.F.S., Silva, V.T., Lucena, C.J.P.: Using Jason to develop normative agents. In: Advances in Artificial Intelligence – SBIA 2010, *Lecture Notes in Artificial Intelligence*, vol. 6404, pp. 143–152. Springer (2011)
16. Meneguzzi, F., Miles, S., Luck, M., Holt, C., Smith, M.: Electronic contracting in aircraft aftercare: a case study. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 63–70. IFAAMAS (2008)