Integrating Expectation Handling into Jason

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. Although expectations play an important role in designing cognitive agents, agent expectations are not explicitly being handled in most common agent programming environments. There are techniques for monitoring fulfilment and violation of agent expectations, however they are not linked with common agent programming environments so that agents can be easily programmed to respond to these circumstances. This paper investigates how expectation monitoring tools can be tightly integrated with the Jason BDI agent interpreter by extending it with built-in actions to initiate and terminate monitoring of expectations, and demonstrates how an external expectation monitor is linked with Jason using these internal actions.

1 Introduction

Expectations represent the anticipatory mental component of an agent, thus they resemble an important part of cognitive agents. When an agent bases its practical reasoning on the assumption that one or more of its expectations will hold, it somehow has to ensure that it is aware of when these expectations are fulfilled and/or violated.

Although much research can be found on techniques for monitoring fulfilment and violation of various types of future expectation such as those based on norms, commitments, and contracts(see [8] for a brief survey of the existing monitoring techniques), we do not see much research on providing support for these in common agent programming environments. However, to successfully implement normative multiagent systems using these agent programming environments, it is important that they support techniques to monitor for fulfilments and violations of these expectations to help in the development of socially aware multiagent systems, and to provide better testbeds for experimenting with new monitoring techniques.

In this work, we present an approach for tightly integrating expectation monitoring with the Jason [4] Belief-Desire-Intension (BDI) agent interpreter, by extending it with built-in actions to initiate and terminate monitoring of expected constraints on the future and by defining specific belief types to represent detected fulfilments and violations of expectations. With the introduction of these built-in actions, any third party monitoring tool can be "plugged in" to the Jason environment, and in this paper we demonstrate this with an expectation monitor

developed in previous research [7]. Moreover, we present extended operational semantics for Jason, which incorporates expectation handling.

Our mechanism allows agents to choose to delegate to an expectation monitor service the monitoring of rules that specify conditional constraints on the future. These rules may be based on published norms, agreed contracts, commitments created through interaction with other agents, or personally inferred regularities of behaviour, and multiple instances of the monitoring service may be active on behalf of different agents at any time.

The benefit of using a monitoring service available within Jason rather than using an external monitoring agent is that it is easier to apply this monitoring mechanism to different applications. The only requirements for agent system developers to understand and use our approach are understanding of the abstract idea of monitoring for fulfilments and violations of future-oriented expectations and the signature of two new internal Jason actions, and to be provided with the customized (Java) logic needed to connect a given monitoring technique with Jason. This is in contrary to monitoring mechanisms based on specialised monitoring agents, such that presented by Meneguzzi et al. [10]. In that work, a norm monitoring tool for a specific domain was implemented as an agent and agent-level communication was used between the monitor agent and its client (a Jason agent). This can be seen as an application pattern that can be reused for different domains, but this reuse requires understanding of the function of the monitor agent, the protocols used for communication, and the Jason plans used to handle communication with the monitor agent. Furthermore, while this approach is suitable for providing an official monitor for norms and contracts defined at the institutional level, it would introduce undesirable communication overhead if used as an architecture for agents wanting their own individual expectations monitored, for use in their own personal reasoning processes.

However, it should be noted that our approach does not rule out the use of a single designated monitoring agent to monitor expectations (e.g. norms) applying to a whole society. Such a monitor agent can also make use of the techniques discussed in this paper.

The rest of the paper is organized as follows. Section 2 gives an overview of agent expectations and their significance. Section 3 gives an overview of the Jason platform, and Section 4 contains an overview of the expectation monitor used in this work. Section 5 lays out the introduced extensions to Jason, and in Section 6 we demonstrate this by means of an example. Finally, Section 7 concludes the paper.

2 Expectations of Cognitive Agents

Expectations represent the anticipatory mental component of an agent. In a theoretical perspective, expectations are "hybrid mental configurations whose components entail not only beliefs but also converging goals that those beliefs will be realized" [6]. Despite several definitions on how expectations are really

formed based on beliefs and goals, it is the common agreement that beliefs and goals are the two elements that form expectations of an agent.

Along with perceptions, expectations play a very important role in generating emotions of agents such as hope, fear, frustration, disappointment, and relief [5]. For example, if the agent had been expecting something bad, and the received punishment is less than what was expected, it generates the emotion *relief*. On the other hand, if the agent achieved a lesser result than what was expected, it leads to the emotional state *disappointment*.

Expectations also have an important role to play in the social context of multi-agent systems. They play a fundamental role in defining social norms, conventions, and commitments, and on the other hand, can arise due to these normative components [6]. Expectations are also the root course in generating social trust, where trust can be defined as a complex form of expectation [9]. Therefore properly analysing and modeling individual agent expectations is beneficial in developing more reactive and emotional agents as well as normative multi-agent societies.

3 Jason

Jason [4] is a Java-based open source interpreter for an extended version of the AgentSpeak agent programming language [12], which is based on the BDI model.

A Jason agent program consists of plans that are executed in response to the events received. These events are generated by the addition or deletion of the beliefs and goals of an agent. 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.

3.1 Jason Agent Reasoning Cycle

An agent is operated in a continuous cycle called the reasoning cycle, operating on a stream of belief update events produced as the agent perceives its environment. The Jason semantics define the following steps for interpreting AgentSpeak programs, as shown in Figure 1 [4]. First, the reasoning cycle checks for messages received by the agent and selects one of the messages to process further (ProcMsg). Then one event from the many pending events is selected to be processed further in that reasoning cycle (SelEv). Then is the step of selecting the set of relevant plans for the given event (RelPl). From the relevant plans, a subset of plans are again retrieved, which are currently applicable (ApplPl), meaning that their context conditions evaluate to true given the agent's current beliefs. Then, one plan from the set of applicable plans (the intended means) is selected for execution (SelAppl), and the new intended means is added to the set of intentions (AddIM). Then the reasoning cycle chooses one of the pending

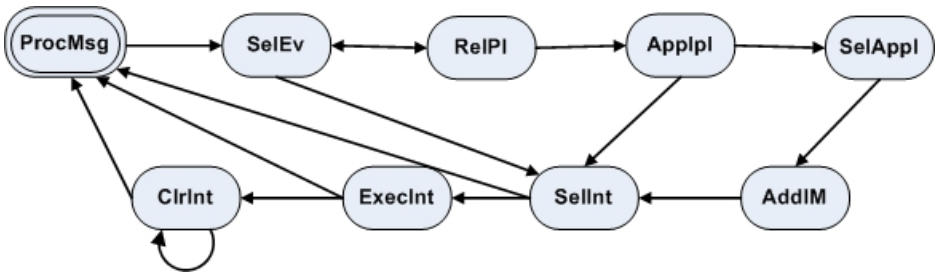

Fig. 1. Possible state transitions within one reasoning cycle [4]

intentions (SelInt), executes one step of that intention (ExecInt), and clears the intended means that may have finished in the previous step (ClrInt).

Most of the aforementioned steps are directly customizable and since the Jason source is freely available, other functionalities of the Jason interpreter can also be changed as needed.

4 The Monitoring Tool

In this paper we selected an expectation monitor developed in previous research [7] to be integrated with the Jason platform. This expectation monitor aims at monitoring expectations that encode complex temporal constraints on the future. It uses a language based on hybrid temporal logic to facilitate this encoding. With the use of temporal logic, this expectation monitor is able to deal with expectations with complex temporal aspects, as opposed to many other monitoring techniques that handle only propositions that must come true by a deadline.

The language that the expectation monitor accepts includes the following operators relating to conditional rules of expectation[1]:

- $\mathsf{Exp}(Condition, Expectation)$
- $\mathsf{Fulf}(Condition, Expectation)$
- $\mathsf{Viol}(Condition, Expectation)$

where *Condition* and *Expectation* are formulae in a form of linear propositional temporal logic. *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). Expectations come into existence when their condition evaluates to true in the current state. These expectations are then considered to be fulfilled or violated if they evaluate to true in a state (without considering any future states that the monitor might already know about, e.g. if it is running in an "offline" mode to analyse an audit trail).

[1] In previous work these operators were named ExistsExp, ExistsFulf and ExistsViol, but we use simplified names here.

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 [3], e.g. if p is true in the current state, an expectation that "p is true and q is true in the next state" will progress to the expectation that "q is true in the current state".

Although this expectation monitor supports the monitoring for the existence (i.e. activation) of an expectation, as well as its fulfilment or violation, in the integration of the monitor with Jason, we currently handle only the fulfilment and violation of an expectation. We intend to modify our extension to support monitoring for the existence of an expectation in future versions.

As an example, consider the the football team play scenario "give and go" illustrated in Figure 2. This team play scenario involves two players where one player (player 1 in the figure) passes the ball to her team mate (player 2). Player 2 then adopts an expectation that player 1 will run down to an advantageous field position (which was agreed upon according to the team tactic). The intention of player 2 is to pass the ball back to player 1, when she fulfills this expectation. On the other hand, if player 1 was unable to fulfill this expectation, player 2 has to initiate a new tactic.

As player 2 has to focus on advancing down the field while avoiding opposition players, the expectation monitor can be delegated to monitor the performance of player 1, to check whether she fulfills (or violates) the defined expectation of player 2. The fulfilment and violation of this expectation can be expressed using the following two formulae, where we assume that player 1 is supposed to advance towards the goal B in the field, *until* she reaches the penalty area in front of goal B.

$$\mathsf{Fulf}(s5, advanceToGoalB(player1)\ \mathsf{U}\ penaltyB(player1))$$
$$\mathsf{Viol}(s5, advanceToGoalB(player1)\ \mathsf{U}\ penaltyB(player1))$$

The first argument in the formulae refers to the condition that triggers the expectation, as explained earlier. This condition becomes true when the proposition $s5$ is true. $s5$ is a nominal, which can be true in only one state (state 5, in this example). The operator U is the standard 'until' operator of temporal logic.

The first formula above evaluates to true in any state in which the rule is fulfilled (i.e. player 1 reaches the penalty B area), and the second formula will be true in any state in which the rule is violated (e.g. player 1 moves in the opposite direction from goal B or stops moving before reaching the penalty area). In these cases, the monitor sends a belief addition event back to player 2 to inform her that this rule was fulfilled or violated.

5 Expectation Handling in Jason

5.1 New Internal Actions to Start and Stop Expectation Monitoring

An important feature of our monitoring mechanism is the ability to use any third-party monitoring tool in conjunction with Jason plans. Therefore the interface

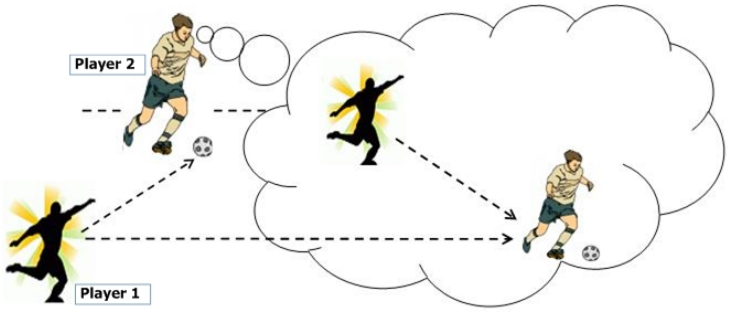

Fig. 2. The "give and go" tactic in football

between Jason and an expectation monitor was designed to be more abstract than the logic described in the previous section. This helps to switch to different expectation monitor techniques without changing the actual agent logic that initializes or terminates expectation monitoring. Our intention is to provide a generic interface that would suit a range of monitoring tools.

Internal actions in the Jason platform help programmers to extend agent capabilities by defining them in the Java programming language. Internal actions are appropriate to use when the corresponding logic cannot be expressed in AgentSpeak language constructs (e.g. integrating Jason with an external program) or involve computations of a procedural nature that are more conveniently expressed in Java.

The new internal actions needed to extend the Jason platform are directly added to the standard internal actions library, enabling any agent program to refer to those. Thus an agent programmer does not have to know how to design these internal actions. However, if the custom logic related to an expectation monitor is included in these internal actions, the agent programmer has to change the standard Jason code each time when integrating a new expectation monitor type. Therefore we have made it possible to store this custom code in a Java class which is stored inside the same Java package as the related agent program. The internal actions expect the existence of this customized class to handle the specific logic related to a given expectation monitor. The internal actions decode the parameter values sent by an AgentSpeak program, and send these values to this customized class, to be processed according to the selected expectation monitor.

Initiating Expectation Monitoring: The internal action corresponding to the initialization of expectation monitoring is *start_monitoring*[2]. It takes in the following parameters:

[2] Currently each call to start_monitoring creates a new instance of the monitor. This is due to a current limitation of the monitor implementation that it only handles one rule at a time. In future work we plan to have a single monitor handling multiple rules at a time

monitoring_mode: This is either "fulf" or "viol" to indicate whether the rule of expectation is to be monitored for fulfilment or violation. For example, with respect to the expectation monitor we are currently employing, the logic of the internal action generates a Fulf formula if the parameter refers to "fulf" and a Viol formula if the parameter refers to "viol".

expectation_name: This specifies a name for the expectation, for ease of future reference.

monitor_tool: This identifies the monitoring tool that should be used to monitor this expectation.

condition: This specifies the requirements on the past and present that activate monitoring for the expectation.

expectation: This specifies the actual expectation.

context_information_list: This argument can be used to assign any other contextual information that might be useful for monitoring an expectation. For example, we can specify a specific agent or a group of agents to be monitored. This information will be added to any fulfilment belief or violation belief sent to the agent as a result of monitoring the expectation.

Terminating Expectation Monitoring: We have made it possible for an agent to stop monitoring for an expectation if the need arises to do so during its reasoning process. The internal action *stop_monitoring* 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.

5.2 Representing Expectation Fulfilments and Violations in Jason

An important design consideration is how to encode the fulfilments and violations identified by the external expectation monitor as Jason events to Jason. The Environment class in Jason acts as the interface to integrate the Jason platform with outside simulation environments. Therefore the Environment class was selected as the best option to communicate the detected fulfilments and violations to Jason agents. Just like percepts, these fulfilments and violations result in new beliefs that lead to the execution of plans that handle the detected fulfilment or violation.

We define the structure of beliefs based on the detected fulfilments and violations as follows:

$$\mathsf{fulf}(Name, StateId)[rule(Cond, Exp), rule\_triggered\_in\_state(OldStateId), context(Context)]$$

$$\mathsf{viol}(Name, StateId)[rule(Cond, Exp), rule\_triggered\_in\_state(OldStateId), context(Context)]$$

Here, *fulf* encodes the fulfilment of an expectation, while *viol* represents a violation.

The variable *Name* represents the name assigned to a particular fulfilment or violation detected, and the *StateId* represents the identifier for the state in which the actual fulfilment or violation of the expectation occurred. The notion of a state is important because fulfilments and violations arise in a particular temporal context that is encapsulated by the state identifier. It is up to the monitor to provide an appropriate form of state identifier.

In Jason, a percept with the same content as an already existing belief will not lead to the generation of a new belief. However, we want a fulfilment or a violation detected in one iteration of the Jason reasoning cycle to be distinct from the same fulfilment or violation detected in the previous cycle (e.g. two different robberies in consecutive states are two different crimes). This requirement can be accomplished with the state number associated with the fulfilment (and violation) beliefs.

When creating beliefs in Jason, an agent programmer can add any other variable of importance using 'annotations'. These annotations can be omitted when specifying the triggering event for a plan if the context and the body of the corresponding plan do not need this information. In the above predicates, we have defined three annotations. The first annotation represents the actual rule that was fulfilled or violated. It has two parameters: the condition that triggers the expectation and the actual expectation. These can be defined in any format according to the expectation monitor in use. The second annotation is 'rule_triggered_in_state' which identifies the state in which the condition of the expectation became true. The third annotation is the list of contextual information that is related to this identified fulfilment or violation. The context information list that was generated for the related expectation when it was initiated by star_monitoring internal action is used to provide information in this annotation.

5.3 Extended Jason Semantics

In this section, we present the extended Jason semantics which includes the operation of the expectation monitor.

In Jason, the state of an agent is determined by the *belief base*, the *set of events*, the *plan library* and the *set of intentions*. With our extension, an agent can have a set of expectation monitors that are active on behalf of it, which operate external to the Jason core logic. A monitor has its own state, which is different from an agent's state. For simplicity, we only model a single expectation monitor in the semantics. Incorporating multiple monitors is a straightforward extension.

An expectation monitor can have many 'monitor tasks', distinguished by their unique name. Each monitor task is comprised of a rule (a rule resembles an expectation, and its triggering condition), and a property which states whether the rule should be monitored for its fulfilment or violation. Associated with a monitor, there is also a history component, which resembles the set of input

states received by the monitor. We also define a *set of notifications*, which becomes the output of the expectation monitor. The set of notifications resembles the set of states where the expectation monitor recorded a fulfilment or violation for any of the monitor tasks that are currently being monitored. These notifications are eventually consumed by the agent.

An expectation monitor is represented by the triple $\langle H, MTs, Ns \rangle$, where:

- H is the history of the monitor. As mentioned earlier, H resembles the set of input states received by the monitor. The input states have a state identifier, and some associated information of the world in a representation specific to the expectation monitor being used.
- MTs is the set of monitor tasks associated with the expectation monitor. A monitor task MT is a 4-tuple of the form $\langle Na, Cn, Ex, Pr \rangle$. Here Na refers to the unique name assigned to the monitor task. The Cn and Ex parameters represent a rule, where Cn represents the condition specifying when an expectation becomes active and Ex refers to the actual expectation. Pr is the property which has the grammar $Pr := FULF|VIOL$, meaning that the property refers to the fulfilment or violation of a rule.
- Ns is the map of notifications generated by the expectation monitor as the output. This map associates state identifiers with sets of pairs $\langle Na, Pr \rangle$ where each pair expresses the information that in the given state the monitor task named Na resulted in a detected event of type Pr ($FULF$ or $VIOL$).

This abstract model of a monitor can be related to the semantics of a specific monitor tool as shown by the following example rule. This shows how the model theoretic semantics (top left) of our chosen monitor [3] is related to the emission of a fulfilment notification. A similar rule can be defined to explain the emission of violation notifications.

$$\frac{H, \varnothing, |H| \models \mathsf{Fulf}(Cn, Ex) \quad \langle Na, Cn, Ex, FULF \rangle \in MTs}{\langle H, MTs, Ns \rangle \rightarrow \langle H, MTs, Ns' \rangle}$$

where

$Ns' = Ns \cup (|H| \mapsto \langle Na, FULF \rangle)$ if $|H|$ is not a key in Notifications,
or
$Ns' = map\_update(Ns, |H|, Ns[|H|] \cup \langle Na, FULF \rangle)$ otherwise.

In this rule, we assume that history states are identified by their (1-based) indices, so $|H|$ (the length of the history H) is the identifier for the final state in the history.

The rule states that when a fulfilment formula logically holds in the logic used by the monitor[3], and the corresponding rule is being monitored, a fulfilment notification is emitted for the current state (the last in the history). The notification map is updated either by adding a new mapping $|H| \mapsto \langle Na, FULF \rangle$ to the monitor notifications, or by adding $\langle Na, FULF \rangle$ to the notifications for state $|H|$ if any exist.

[3] The details of this particular monitor's semantics [3] are outside the scope of this paper.

In the Jason semantics, the transition relation of an agent's configuration is given by a set of conditional rules that change the agent's configuration in each of the steps of the reasoning cycle. The configuration for an agent is represented by the tuple $\langle ag, C, M, T, s\rangle$ [4], where:

- ag refers to the agent program, which consists of a set of beliefs and a set of plans
- C is an agent's circumstance, denoted by the tuple $\langle I, E, A\rangle$, with I being the set of intentions, E the set of events and A being the set of actions to be performed in the environment.
- M is a tuple $\langle In, Out, SI\rangle$ that registers different aspects of communicating agents. Here, In is the message inbox of an agent, Out is the out-going message box, and SI keeps track of the suspended intentions related to the communication messages that are currently being processed.
- T is a structure that stores temporary data required in various steps of the reasoning cycle. This is a tuple $\langle R, Ap, i, \epsilon, \rho\rangle$, where R represents the relevant plans, Ap represents the set of applicable plans, and i, ϵ, ρ respectively represent an intention, event and an applicable plan that are being considered along the execution of one reasoning cycle.
- s is the current step (or state) in the agent reasoning cycle shown in Figure 1, where:
  $s \in \{\mathsf{ProcMsg}, \mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ClrInt}\}$.

Subscripts are used to identify individual components of tuples, e.g. C_E denotes the events set within a configuration C, and the notation $i[p]$ is used to denote an intention consisting of plan p on top of intention i.

To define the semantics of our Jason extension we must address three issues: i) the effect of the new internal actions start_monitoring and stop_monitoring, and ii) how notifications emitted from the monitor are communicated to Jason as beliefs. There is a third issue that we consider out of the scope of these semantics: the process that adds states to the monitor's history. This is because the Jason agent is not responsible for sending percepts to the monitor, and our architecture does not even assume that the monitor receives state information from the Jason environment object—it may have its own separate mechanism for obtaining information from the system in which the Jason agent is situated[4].

In the rules below we define transitions on an extended system configuration comprising the Jason agent and the monitor. This is a pair $\langle AG, EM\rangle$, where $AG = \langle ag, C, M, T, s\rangle$ and EM represents the expectation monitor as defined above.

start_monitoring :

Through the start_monitoring internal action, an expectation monitor is started and is added to the set of active expectation monitors of the agent.

[4] This is the case for our work on integrating this extended version of Jason with the Second Life virtual world [11]

The start_monitoring internal action takes place when the body of an agent plan is being executed, and this internal action becomes the current intended means to be executed. This refers to the ExecInt step in the reasoning cycle in Figure 1. The internal action executes completely (i.e. without suspension, which is the normal procedure for executing internal actions) and returns.

The Jason semantics for this action is shown below.

$$\frac{T_i = i[head \leftarrow start\_monitoring(Mm, En, Mt, Cn, Ex, Cil); h]}{\langle\langle ag, C, M, T, \mathsf{ExecInt}\rangle, EM\rangle \rightarrow \langle\langle ag, C, M, T', \mathsf{ClrInt}\rangle, EM'\rangle}$$

Where:

- Parameters Mm, En, Mt, Cn, Ex and Cil respectively refer to monitoring_mode, expectation_name, monitor_tool, condition, expectation and context_information_list, as defined in Section 5.1
- Here $EM'_{MTs} = EM_{MTs} \cup \{\langle En, Cn, Ex, Mm\rangle\}$
- $T'_i = i[head \leftarrow h]$

As in the standard Jason semantics, where a transition is defined as transforming a structure S into a new version S', all components of S' are assumed to be the same as those in S except where otherwise specified.

stop_monitoring : The stop_monitoring internal action takes place when the body of an agent plan is being executed, and this internal action becomes the current intended means to be executed. This refers to the ExecInt step in the reasoning cycle as in start_monitoring, and it moves the transition to the state ClrInt.

$$\frac{T_i = i[head \leftarrow stop\_monitoring(En, Mt); h]}{\langle\langle ag, C, M, T, \mathsf{ExecInt}\rangle, EM\rangle \rightarrow \langle\langle ag, C, M, T', \mathsf{ClrInt}\rangle, EM'\rangle}$$

where:

- Parameters En and Mt respectively refer to the expectation_name and monitor_tool as defined in Section 5.1
- $EM'_{MTs} = EM_{MTs} \setminus \{MT\}$. In other words, the stop_monitoring internal action removes the monitor task MT referenced by En (here, the expectation_name refers to the unique name of the monitor task) from the expectation monitor.
- $T'_i = i[head \leftarrow h]$

Though not included in the paper, we also modify the condition of the existing Jason semantic rule for handling internal actions to exclude it from applying the standard operational semantics in the case that the selected action a is start_monitoring or stop_monitoring.

Handling Fulfilment and Violation Notifications Whenever the monitor identifies the fulfilment or violation of a rule defined in it, it sends a notification to Jason. These notifications are treated as Jason percepts and subsequently result in new belief events. We use the function *NotBels* to denote the process that converts monitor notifications into belief events using the syntax defined in Section 5.2, and the corresponding rule for this transition can be written as follows:

$$\frac{EM_{Ns} \neq \varnothing}{\langle\langle ag, C, M, T, s\rangle, EM\rangle \rightarrow \langle\langle ag, C', M, T, s\rangle, EM'\rangle}$$

In this rule, $EM'_{Ns} = \varnothing$ and $C'_E = C_E \cup NotBels(EM_{Ns})$.

Here, EM'_{Ns} refers to the set of notifications belonging to all the monitor tasks active for that expectation monitor.

This rule is not executed as part of the Jason agent's reasoning cycle. Rather, it represents a separate process that consumes notifications from the monitor and adds them as new events for the agent to process. This process runs concurrently with the Jason interpreter, and we do not assume any sychronisation between the two processes (except to avoid concurrent modification of the agent's input event set C_E). Therefore this rule can be applied in any state of the agent[5].

6 Example Scenario - A Jason Agent Engaged in the Football Team Play Scenario "Give and Go"

We have integrated this Jason extension with the popular virtual world Second Life [1], with the use of a framework we have developed [11] for integrating agents with Second Life. This enables the implementation and testing of sophisticated Jason agents.

In this example, we demonstrate how the ability of a Jason agent to monitor and detect fulfilments and violations of its expectations is useful in its decision making process. We implement this example in the SecondFootball [2] virtual simulation in Second Life which enables playing virtual football. 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.

In this example, we implement the "give and go" team play scenario described in Section 4. Here, the Jason agent Ras_Ruby is engaged in the team play scenario with the player Su_Monday, who is controlled by a human. When Ras_Ruby receives the ball, it adopts the expectation which states that Su_Monday should run until she reaches the PenaltyB area, so that she can pass the ball back to Su_Monday for her to attempt a goal score at Goal B.

[5] In practice, the monitor's notifications are recorded as percepts in the Jason Environment object, and the agent perceives them via Jason's belief update phase. However, Jason's operational semantics do not include a state for perceiving the environment, so here we model the connection between the monitor and the agent as a separate process that pushes fulfilment and violation beliefs to the agent.

When the system starts, the Jason agent corresponding to Ras_Ruby is initialized. When the Jason agent starts executing, it first tries to log itself into Second Life. After sending its login request, the agent has to wait till it gets the confirmation of the successful login. When it receives the successful login notification, the agent adopts the new goal to walk to the area MidfieldB2. The corresponding plan for this goal addition is shown below (+! denotes a goal addition event, a context condition appears after the colon, and the arrow operator separates the head and body of the plan).

```
+!check_connected: connected
  <-
  action("walk", "MidfieldB2").
```

Once in the area MidfieldB2, the agent Ras_Ruby waits for Su_Monday to kick and pass the ball to it. Once it successfully receives the ball, the agent gets the "successful_kick(su_monday, ras_ruby)" percept (which is generated by our Second Life integration framework and states that Su_Monday successfully passed the ball to Ras_Ruby through a kick), and this triggers the corresponding plan related to this belief addition, as given below.

```
+successful_kick(su_monday,ras_ruby)
   <-
   //internal actions
   .start_monitoring("fulf",
      "move_to_target",
      "expectation_monitor",
      "#once",
      "('U',
         'advanceToGoalB(su_monday)',
         'penaltyB(su_monday)')",
      []);

   .start_monitoring("viol",
       "move_to_target",
       "expectation_monitor",
       "#once",
       "('U',
         'advanceToGoalB(su_monday)',
         'penaltyB(su_monday)')",
      []).
```

This plan starts the internal actions for monitoring for the fulfilment and violation of the agent's expectation. Here, in the first parameter we define the type of expectation; whether it is a fulfilment or a violation. The second parameter assigns a name for the expectation. The third parameter is the name of the expectation monitor used. The fourth parameter is the triggering condition for the expectation, and in this example, it is a keyword with a special meaning

(`#once`). For this scenario the initiating agent wants the rule to fire precisely once, as soon as possible, and this can be achieved in our current expectation monitor by using a 'nominal' (a proposition that is true in exactly one state) for the current state as the rule's condition. However, the BDI execution cycle only executes a single step of a plan at each iteration, and any knowledge of the current state of the world retrieved by the plan may be out of date by the time the monitor is invoked. The `#once` keyword instructs the monitor to insert a nominal for the current state of the world just before the rule begins to be monitored. Here, the actual expectation formula is given by the fifth parameter, and the sixth parameter is a list of optional context information, which we do not utilize in this example.

The fulfilment of this expectation occurs when Su_Monday advances towards GoalB (`advanceToGoalB(su_monday))`, until `('U')` she reaches PenaltyB, denoted by `'penaltyB(su_monday)'`. Similarly, the violation of this expectation occurs if Su_Monday stopped somewhere before reaching PenaltyB, or she moves in the opposite direction before reaching PenaltyB area[6].

If Su_Monday fulfilled Ras_Ruby's expectation, the expectation monitor detects this and reports back to the Jason agent, which results in a fulfilment belief. The following plan handles this detected fulfilment and instructs the avatar to carry out the kick action[7].

```
+fulf("move_to_target", X)
   <-
   //Calculate kick direction and force, turn, then ...
   action("animation", "kick").
```

On the other hand, if Su_Monday violated the expectation, the expectation monitor reports the violation to the Jason agent, generating a violation belief for the agent. The agent uses the first plan below to decide the agent's reaction to the detected violation, which creates a goal to choose a new tactic for execution. The second plan (responding to this new choose_and_enact_new_tactic) is then triggered, and the agent adopts the tactic of attempting to score a goal on its own by running towards the PenaltyB area with the ball.

```
+viol("move_to_target",X)
   <-
   !choose_and_enact_new_tactic.

+!choose_and_enact_new_tactic
   <-
   action("run", "penaltyB").
```

[6] The conditions and expectations are defined in temporal logic and we do not wish to elaborate on them in the scope of this paper. These are written as nested Python tuples, as this is the input format for the expectation monitor written in Python.

[7] Due to technical problems the Second Life avatar cannot currently perform the actual 'kick' animation

7 Conclusion

This paper addressed the importance of agents having a capability to directly monitor their expectations and detect the fulfilments and violations of these expectations, and respond accordingly.

We demonstrated a tight integration of expectation monitoring in the BDI agent model and presented an implemented mechanism to monitor expectations of individual agents in the Jason agent model. Also, we identified this as an approach to focus on monitoring at the individual agent level, as opposed to the organizational level monitoring that has received the main focus in the past research, and noted that our approach also has the flexibility to be used for monitoring in the organization level.

As future work, it is interesting to investigate ways of how agents can publish their expectations to make other agents in the society aware of their personal expectations, and how agents should react to the detected fulfilments and violations of their expectations, both in a social context and with respect to their emotions. Moreover, it should also be investigated how agents can use their expectations as well as expectations of other agents in the society that they are aware of, proactively in their deliberation process.

References

1. Linden Lab. Second Life Home Page. `http://secondlife.com`, August 2010.
2. Vstex Company. Secondfootball Home Page. `http://www.secondfootball.com`, August 2010.
3. F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
4. R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons Ltd, England, 2007.
5. C. Castelfranchi. Mind as an anticipatory device: For a theory of expectations. In M. De Gregorio, V. Di Maio, M. Frucci, and C. Musio, editors, *Brain, Vision, and Artificial Intelligence*, volume 3704 of *Lecture Notes in Computer Science*, pages 258–276. Springer Berlin / Heidelberg, 2005.
6. C. Castelfranchi, F. Giardini, E. Lorini, and L. Tummolini. The prescriptive destiny of predictive attitudes: From expectations to norms via conventions. In *Proceedings 25th Annual Meeting of the Cognitive Science Society (CogSci 2003), Boston, USA, 31 July 2 August*, 2003.
7. S. Cranefield and M. Winikoff. Verifying social expectations by model checking truncated paths. *Journal of Logic and Computation*, 2010. Advance access, doi: 10.1093/logcom/exq055.
8. S. Cranefield, M. Winikoff, and W. Vasconcelos. Modelling and monitoring interdependent expectations. Discussion Paper 2011/03, Department of Information Science, University of Otago, 2011. `http://eprints.otago.ac.nz/1094/`.
9. E. Lorini and R. Falcone. Modeling expectations in cognitive agents. In *AAAI 2005 Fall Symposium: From Reactive to Anticipatory Cognitive Embodied Systems*, 2005.

10. F. Meneguzzi, S. Miles, M. Luck, C. Holt, and M. Smith. Electronic contracting in aircraft aftercare: a case study. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 63–70, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.
11. S. Ranathunga, S. Cranefield, and M. Purvis. Interfacing a Cognitive Agent Platform with a Virtual World: a Case Study using Second Life. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, 2011. To appear.
12. A. S. Rao. BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world: agents breaking away*, pages 42–55. Springer-Verlag Berlin, Heidelberg, 1996.