This is the Author's Accepted Manuscript of the following article:
Surangika Ranathunga, Stephen Cranefield & Martin Purvis, Identifying Events Taking Place in Second Life Virtual Environments, Applied Artificial Intelligence, 26(1-2):137-181, 2012.
© 2012 Taylor & Francis Group, LLC.
Available online at http://www.tandfonline.com/doi/abs/10.1080/08839514.2012.629559.

Identifying Events Taking Place in Second Life Virtual Environments

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

Second Life is one of the most popular multi-purpose online virtual worlds, which supports applications in diversified areas relating to real-life activities. Moreover, it is possible to use Second Life in testing Artificial Intelligence theories, by creating intelligent virtual agents. For the successful implementation of many of these applications, it is important to accurately identify events taking place inside Second Life. This involves extracting low-level spatio-temporal data and identifying the embedded high-level domain-specific information. This is an aspect that has not been taken into consideration in the prior research related to Second Life. This paper presents a framework that extracts data from Second Life with high accuracy and high frequency, and identifies the high-level domain-specific events and other contextual information embedded in these low-level data. This is guided by our virtual environment formalism, which defines events and states in a virtual environment. This framework is further enhanced to be connected with multi-agent development platforms, thus demonstrating its use in the area of Artificial Intelligence.

1 Introduction

Multi-purpose online virtual worlds can be considered as a medium that facilitates the next generation of human social interactions. Among the present day multi-purpose virtual worlds that are publicly accessible, Second Life has a clear lead, with a registered user base of over 15 million in mid 2009 (Linden Lab, 2010). An increasing number of individuals and organisations are using Second Life for diverse applications in areas such as education, health, business, and entertainment. For example, many universities are considering the possibility of conducting virtual lectures in their virtual universities in Second Life and some universities and private institutions are using Second Life simulations to train people for tasks for which real life training could be too expensive or dangerous (Diener et al., 2009; Rogers, 2009).

The co-existence of individuals and organizations inside Second Life has led to the creation of virtual human communities, where people with different interests interact with each other to accomplish individual or common goals. This virtual experience is further enhanced by the ability to create custom simulations such as training scenarios, where the participant interactions are governed by the rules defined in that simulation. Moreover, as identified by some researchers, Second Life can provide support for research in the area of Artificial Intelligence (Veksler, 2009) with the ability it provides to create intelligent virtual agents.

However, one very important feature needed by many of these application areas has still not been addressed by the research related to Second Life. Although there are more and more attempts to create sophisticated virtual content and simulations to assist interactions inside Second Life, we do not see much focus on accurately detecting events taking place in these virtual environments to identify 'who did what and when'. Identifying events that take place in Second Life mainly involves two steps: accurately extracting low-level spatio-temporal data from the Second Life servers, and identifying high-level domain-specific information embedded in those low-level data. The framework presented in this paper provides solutions to both these aspects.

Extracting data from Second Life with high accuracy and high frequency is a challenging task, and the related research has not addressed this problem well (Ranathunga et al., 2011a). Second Life introduces various complications in extracting data, depending on the specific characteristics of a simulation. Therefore we introduce two different data extraction mechanisms that work for many simulations in general, and we provide the guidelines to select the best mechanism for a given Second Life simulation.

Data extracted from Second Life mainly include low-level vector coordinates of avatar and object move-

ment (e.g. position, velocity, angular velocity and acceleration), avatar animation information, and messages exchanged in the environment. As with most of the other virtual worlds, the amount of low-level data generated by Second life is high, and these low-level data may not be completely accurate. In the context of many Second Life simulations, information content in low-level data is inadequate for reasoning about what is happening in the environment. For example, in a football scenario, if provided a continuous stream of low-level data about players and the soccer ball, it is required to identify information at different abstraction levels such as which player has the possession of the ball, locations of the players with respect to the football field locations, whether a particular player scored a goal, and whether a team is following any specific game tactic. However we are not aware of much research that focused on identifying the high-level domain specific information in virtual environments, based on these received high amounts of potentially unreliable low-level data. In this work, we mainly focus on the real-time identification of the high-level information. We utilise a Complex Events Detection (CED) mechanism that can identify complex events embedded in these high volumes of low-level data, with very low latency. Consequently, the emphasis on handling the uncertainty of input data is minimal.

Simply using the individually received low-level data items (e.g. position, velocity or animation changes of an avatar) is not enough to identify this type of complex information. Rather, we need a mechanism to generate complete snapshots of the virtual environment, where a snapshot describes the complete state of that environment at a given instance of time. Therefore we present a data amalgamation mechanism that generates complete snapshots of the environment periodically, as well as when a low-level data item is received from Second Life. These snapshots are used by the CED mechanism in our framework to identify high-level domain-specific events embedded in the extracted low-level data. We also present an intermediate processing step that identifies 'contextual information' embedded in the low-level data, thus presenting these received data at a higher abstraction level to CED component, and making it easier to identify complex events.

A general understanding of the dynamism of virtual worlds is beneficial in identifying events taking place in any virtual world. Therefore we propose a general formalism for dynamic virtual environments that encompasses the different aspects of a virtual environment. Our data amalgamation mechanism and the identification of complex events in Second Life simulations is based on this formalism, thus providing a consistence approach for event identification.

In order to demonstrate how the designed framework can be used to accurately extract high-level domain-specific events from a complex simulation, we employ the SecondFootball simulation in Second Life and the Otago Virtual Hospital (OVH) Simulation in the New Zealand Virtual World Grid (NZVWG). The NZVWG is developed using the OpenSimulator virtual world builder toolkit[1], which supports the Second Life client-server protocol.

In designing intelligent agents in Second Life, it is important that they are able to perceive and comprehend the virtual environment at different abstraction levels and initiate new practical reasoning behaviour in response to events they identify in their Second Life environment. Our framework serves well in this purpose, as it is able to report the events and other contextual information to an agent at different granularities. We demonstrate the significance of this research work in creating more rational and believable agents by extending the developed framework to connect any multi-agent development platform with Second Life to create multi-agent simulations. In fact, the ability of our framework to identify complex high-level events and other contextual information makes it possible to present the low-level data received from Second Life at a different abstraction level. It also helps to reduce the number of percepts passed to the agent, thus reducing its cognitive overload. These two issues are identified as key problems faced by researchers developing Intelligent Virtual Agents (Hindriks et al., 2011). We have already connected an extended version (Ranathunga et al., 2011b) of the Jason (Bordini et al., 2007) agent development platform with this framework. The Jason agents employ a expectation monitor that was developed in previous research (Cranefield and Winikoff, 2010) to identify events relating to fulfilments and violations of their expectations. The expectation monitor uses the complex events and contextual information identified by our framework, and the events identified by the expectation monitor introduce an additional level of

[1] http://opensimulator.org/wiki/Main_Page

abstraction of agent percepts.

The rest of the paper is organised as follows. Section 2 contains an introduction to Second Life and Section 3 contains our formalism for virtual environments. Section 4 discusses the challenges and problems in accurately extracting spatio-temporal data from Second Life environments using existing mechanisms. Section 5 presents the new mechanisms that we developed to accurately extract spatio-temporal data from Second Life and Section 6 describes the complete framework designed to identify events in Second Life environments. Section 7 demonstrates the use of this framework in the context of SecondFootball and OVH simulations. Section 8 discusses how the framework is extended to connect multi-agent development platforms with Second Life and a performance evaluation of the system is presented in Section 9. Section 10 discusses related work, and finally Section 11 concludes the paper.

2 Second Life

Second Life is a proprietary product owned by the company Linden Lab. However, it is freely accessible through any client software that adheres to the Second Life communication protocol. Second Life has a client/server architecture with the server carrying out the major part of processing. The Second Life server side is comprised of a grid of server processes called simulators or 'sims'. Each process is responsible for handling one or more regions in the virtual world (Ranathunga et al., 2011a).

2.1 *Benefits of Identifying Events in Second Life*

The co-existence of individuals (known as 'avatars') and organizations has created virtual communities inside Second Life, where residents interact with each other in a structured context to accomplish individual or common goals. In this respect, it is useful to have a mechanism to identify events taking place in these virtual communities in a social setting to find out whether the community members are adhering to the norms of a certain community (Cranefield and Li, 2009). Also, identifying events taking place in virtual training simulations is useful in identifying how effective the simulation was and how effective the participants were. This can replace or enhance manual analysis methods such as getting participant feedback or analysing video recordings of a simulation (Rogers, 2009; Blyth et al., 2010).

Second Life provides a rich AI test bed when compared to simple 2-D simulation environments and a more convenient test bed when compared with physical robots (Veksler, 2009). The multitude of scenarios that can be created in Second Life makes it a better alternative to special-purpose game engines for creating AI simulations (Gemrot et al., 2010). When using Second Life to simulate AI-related theories, two important issues should be addressed: how the actions are performed inside Second Life, and how the sensor readings from Second Life are mapped to an abstract model of observed properties and/or events that serves as the source of percepts for the agent. A robust mechanism to extract events taking place inside Second Life is needed, and this is non-trivial, considering the high volumes of potentially unreliable sensor readings that are received. This would be useful for creating intelligent agents that can initiate practical reasoning behaviour in response to what they perceive in the Second Life environment around them.

3 Dynamic Virtual Environments — A Formal Definition

Virtual worlds have an inherently dynamic nature, due to their real-time and interactive nature. Events taking place in a virtual world change the properties of its entities (objects and avatars), thus changing the state of the virtual world. When trying to identify events taking place in a virtual world, it is beneficial to have a common understanding of what entities a virtual world consists of, how the state of these entities change over time, the static (e.g. structural) and dynamic (e.g. temporal) relationships of these entities and how these lead to the identification of different states and high-level complex events in the virtual environment.

Despite the heterogeneous nature of virtual worlds, many of the virtual worlds share many common attributes. Therefore we present a general formalism for a dynamic virtual environment, encompassing the aforementioned aspects. We also pay a special attention to aspects specific to Second Life. This formalism is used in data amalgamation and event identification in our framework, as described in the sections that follow. As can be seen, having a formal definition for a virtual environment helps to make the event identification process more coherent. It also introduces common terminology to be used throughout our event identification framework.

The terms 'virtual world' and 'virtual environment' have been used interchangeably in the literature (Ellis, 1994; Gemrot et al., 2010). In this work, we opt to use the word 'virtual environment' when referring to the current region that an avatar is logged in to. In our opinion, the Second Life virtual world is the whole Second Life grid, consisting of different regions (running in different simulator processes). An avatar's Second Life environment is a part of the virtual world that it currently has direct access to (i.e. without having to teleport to another region running in a different sim). However, we use the terms 'virtual environment' and 'virtual simulation' synonymously. Our opinion is that a virtual simulation represents a sub-section of a virtual environment, thus any recorded information that falls out of the simulation boundaries can be explicitly ignored.

The main implication of our formalism is that the state of a virtual world depends on the events taking place there. While there could be virtual worlds that do not explicitly generate events for an external user, many popular virtual worlds such as Second Life and an extended version of UnrealTournament (Adobbati et al., 2001) have an explicit event representation that can be observed by an external user. Moreover, many frameworks attempting to connect agents with virtual worlds have assumed that virtual world state changes are based on events (Dignum et al., 2009; Gemrot et al., 2010). Therefore sensory data generated by a virtual world can be assumed to be mainly comprised of event notifications.

This model is presented from the perspective of an external user such as an agent deployed inside a virtual world. To an external user, only the events published by the virtual world and observable entity properties are visible, but he is not aware of the underlying execution details of the virtual world.

Although the importance of having a higher level of abstraction based on the low-level information of a virtual environment has been identified (Gemrot et al., 2011; Hindriks et al., 2011; Dignum et al., 2009), we are not aware of any general formalism for dynamic virtual environments that is presented in the perspective of an external user. Gemrot et al. (2011) have attempted to formally describe game engines with the purpose of connecting agent systems with game engines. However this formalism mainly focuses on game engine internals and agent internals. Although it talks about game engine *facts*, and *situations*, no emphasis is given to the events taking place in a game engine that result in the change of game engine facts dynamically.

Therefore we use the simulated environment formalism presented by Helleboogh et al. (2007) to define a virtual environment. Although this model is not specifically based on virtual environments or a generic definition of environment, it contains concepts related to dynamic environments that we can utilise in the context of virtual environments. It contains only the basic definition for entities and properties in a simulated environment, and we extend this definition to include the avatars and objects in a virtual environment, along with their properties. We further expand this formalism by defining events and states in an environment.

A virtual environment En is comprised of entities and properties:

$En = \langle E, P \rangle$

where $E = \langle e_1, e_2, ..., e_n \rangle$ is the set of environmental entities,

and $P = \langle p_1, p_2, ..., p_m \rangle$ is the set of environmental properties.

An environmental property is a system-wide characteristic of the environment. For example, a Second Life environment has the property *sunlight*, and depending on this property, a Second Life environment can have different times of day (midday, night, sunrise, sunset, etc.). In our work, we do not consider the environmental properties in event detection, thus we will hereafter use the simpler definition of the environment as:

$En = E$.

Normally in a virtual environment, entities comprise the set of objects and the set of avatars. Hence,

here we expand the original definition of entities by Helleboogh et al. and define the set of environmental entities as

$E = Av \cup Ob$,

where

$Av = \{av_1, av_2, ..., av_n\}$ is the set of avatars (both human-controlled avatars and software-controlled bots),

and $Ob = \{ob_1, ob_2, ..., ob_m\}$ is the set of objects present in the environment.

Following the formalism by Helleboogh et al., it is also possible to categorise these objects and avatars into different types, where a type represents a subset of avatars and/or objects, depending on the simulated domain.

We further expand our environment formalism by defining the properties of entities. An entity has the basic properties[1]:

ID - The unique identifier given to the entity inside the virtual world (e.g. UUID - the Universally Unique Identifier in Second Life)

name - The name assigned to the entity (e.g. the user name of the Second Life avatar)

type - The simulation-specific group or category to which the entity belongs

$position \in \mathbb{R}^3$ - The x,y,z coordinates of the entity

$velocity \in \mathbb{R}^3$ - The speed and direction of movement of the entity

$acceleration \in \mathbb{R}^3$ - The rate of change of velocity of the entity

Here, the first three properties can be considered as static properties, and the latter three properties can be considered as dynamic properties. For Second Life objects, the *name* property can be changed even after their creation, but we assume that the name of an object does not change during a simulation. If the dynamic properties of an entity do not change during a simulation (e.g. for a building or a tree), it is possible to treat the whole object as static.

In order to denote properties of an entity, we define an entity as a tuple:

$e = \langle ID, name, type, position, velocity, acceleration \rangle$,

and we write $e_{property_name}$ to denote a single property value of an entity.

Apart from these general properties of entities, avatars and objects can have properties specific to them, depending on the virtual world or the selected simulation. For example, an avatar may have dynamic properties such as *animation* (denoting the animation it is currently playing), and if in a gaming simulation, *health* (denoting the current health level). As for objects, they can have a *size* property, and an *owner* to whom the object belongs.

In the event and state definitions that follow, we use the basic entity definition in the context of objects. However an avatar av is represented as:

$av = \langle ID, name, type, position, velocity, acceleration, Animations \rangle$

In other words, to define an avatar in a virtual environment, we add the new dynamic property *Animations* to the basic entity definition.

Here, $Animations = \{an_1, an_2, ..., an_n\}$ is the set of animations currently played by an avatar.

3.1 *Events in a Second Life Environment*

In our opinion, events are the root cause of dynamism in virtual environments, as they can change the state of the environment.

An event ev can be simply defined as an occurrence of interest in time, and denoted as:

$ev = \langle EType, AttVals, Es, t_s, t_e \rangle$.

Here, $EType$ is the event type. Each event type is associated with a set of attribute names representing the attributes that all entities of that type possess, denoted by $attNames$(EType)[2]. In an event, $AttVals$ is a map that associates a value with each attribute in $attNames$(EType). Es is the set of constituent events of this event. t_s and t_e are the start and end timestamps of an event, respectively.

[1]In the current model, we omit some properties such as angular velocity, for simplicity

[2]Event types could be extended to have typed attributes, but for simplicity we use untyped attributes in this paper.

An event instance is uniquely identified using the event type name, t_s, t_e, and the assigned values to its attributes, i.e., two event instances belonging to the same event type can be said to be equivalent if they started and ended at the same time, and had identical attribute values.

An event can either be a primitive event or a complex event, which we next explain in detail.

3.1.1 ***Primitive Events.*** Primitive events are instantaneous, meaning that they occur at a point of time.

A primitive event pe is defined as an event $pe = \langle EType, AttVals, \{\}, t, t\rangle$, where t is the single timestamp of the primitive event. In Second Life, t refers to the time value at which a client received the corresponding primitive event. Thus the timestamp of a primitive event is not a globally set value. For a primitive event, Es is empty. We abbreviate primitive events using the notation $pe = \langle EType, AttVals, t\rangle$.

A primitive event represents a state change of the environment[1]. A primitive event in a virtual environment can be defined as a change in at least one of the dynamic properties of an entity, or a change that affects the environment as a whole such as posting a message on the public chat channel, entity appearance and disappearance, or avatars clicking objects.

The set of attribute names defined for primitive event types that represent entity property changes must include "*e*"—this represents the identifier for the entity affected by the primitive event (we assume a primitive event represents a change to a single entity only). The other attribute names defined by a primitive event type are the names of the entity properties that can be affected by that type of event.

Depending on the supported dynamic entity properties, different virtual worlds may generate different types of primitive events. Below we define the primitive events generated by the Second Life server to represent the changes of dynamic properties of an entity.

Animation update events have the following type:
$\langle animation_update, \{e, Animations\}\rangle$.

The name of this event type is *animation_update*, and it has two attributes; the entity (which has to be an avatar), and the set of animations it is performing. The Second Life server generates an animation update whenever there is a change in an avatar's animation.

An event instance of this event type can be written as
$\langle animation_update, \{e \mapsto av_1, Animations \mapsto \{walk\}\}, 00{:}47{:}22.34\rangle$,
where the involved avatar is identified as av_1, and it just changed its animation to $walk$.

Movement update events have the following type:
$\langle movement_update, \{e, velocity, position, acceleration\}\rangle$.

The Second Life server generates a movement update whenever there is a change in the velocity or angular velocity of an entity[2].

An event instance of this event type can be written as,
$\langle movement_update, \{e \mapsto e_1, velocity \mapsto \langle 1.5, 2.6, 0\rangle, position \mapsto \langle 10, 2, 0\rangle, acceleration \mapsto \langle 0, 0, 0\rangle\}, 00{:}47{:}22.34\rangle$.

From the events that affect the virtual environment as a whole, below we only define those that relate to message events, due to space limitations.

Entities in a virtual environment can exchange messages with each other, and post on public chat channels. In Second Life, both avatars and objects can post messages in the public chat channel. Avatars also can exchange instant messages with each other and receive messages from objects. Avatars and objects also can post messages on private chat channels. Though it may not be appropriate to treat these internal messages as primitive events that change an environment state, we would like to record whatever message information that becomes available. In particular, recording messages exchanged in private chat channels is useful in analysing participant interactions in simulations such as the OVH simulation. When defining primitive events related to exchanged messages, we do not treat a message as a property of any specific entity, simply because it is not a persistent property of an entity that changes over time.

[1]In practice, it may not be possible to guarantee that all received 'primitive events' do correspond to a change. Rather, a primitive event could be a reassertion of a property value

[2]Even though the change of position of an entity should ideally be considered as an event, this is not explicitly reported by the Second Life server, as position change is perceived as a continuous function.

Message events have the following type:
$\langle message, \{e_s, e_r, content\}\rangle$,
where e_s and e_r represent the message sender and receiver, respectively.
An event instance of this event type can be written as,
$\langle message, \{e_s \mapsto av_1, e_r \mapsto av_2, content \mapsto 'hello'\}, 00{:}47{:}22.34\rangle$.
Apart from these primitive events, Second Life also has primitive events generated by avatars clicking on objects or entities exchanging money. Although there may be a change of dynamic properties of an object being clicked, the actual click event is perceivable only by the scripts running in that object and the person who clicked that object. Similarly, exchanging money is considered as a private interaction. Therefore, if a certain simulation requires the detection of these types of primitive events, the involved entities should pass a corresponding message into a chat channel, in which case they should either be treated as message events, or as a new primitive event type representing a specific type of avatar action, depending on the requirements of the application domain.

3.1.2 ***Contextual Information.*** Before moving on to complex events, we present the formalism for what we call "contextual information" of a virtual environment. Contextual information introduces an intermediary processing step before using low-level data received from a virtual world in complex event identification. Essentially, contextual information allows the interpretation of these low-level data at a much higher level of abstraction, thus making it easier to identify complex events.

Contextual information can be identified in conjunction with some externally supplied static information. The abstraction level of this static information may vary depending on the selected simulation domain. For example, this static information may contain location data, or position information of static objects such as buildings. A generic function to derive contextual information can be defined as
$f(e_{pr_1}, [e_{pr_2}, ..., e_{pr_n}], [sm_1, sm_2, ..., sm_n])$
In other words, a function that derives a piece of contextual information may have as arguments at least one property of an entity, optionally other entity properties, and an optional set of parameters that represents the set of static information input.

For example, the position property $av_{position}$ of an avatar can be used to derive the simulation-specific location of the avatar, using the computation
$av_{location} = Location(av_{position}, building_locations)$
Here, the static information relates to the positions of buildings in the given simulation.
Similarly the computation $av_{move_direction} = MoveDirection(av_{velocity}, land_marks)$
can be used to identify towards which land mark an avatar is moving.

3.1.3 ***Complex Events.*** Complex events are those for which the set Es of constituent events is non-empty. These events are generated by recognising particular patterns of simpler events (both primitive and complex), other low-level data, and derived contextual information. We do not attempt to define a generic formalism for the definition of complex events, but we illustrate the use of one approach to complex event detection (CED) in Section 6.3.

3.2 *State of a Second Life Virtual Environment*

When identifying complex events taking place in a virtual environment, it is necessary to base the reasoning on the full set of information that represents the state of the environment at a given instance. For example, consider an avatar performing a "hit" animation that generated a primitive event containing only that information. If a complete description of the environment state at that time can be obtained, it may reveal the presence of another avatar close by, thus it can be identified that the first avatar actually attacked the second avatar.

We define the state of a virtual environment S_{En} as the combined state of all of its constituent entities (S_E), together with the set of messages (Msg) exchanged in the environment at the time instance represented by the state. Therefore a state S_{En} is defined as

$S_{En} = S_E \cup Msg$
where $S_E = \bigcup S_{e_i \in E}$

Essentially, a primitive event taking place in a virtual environment represents a state change in the environment.

Whether a primitive event received from a virtual world can be directly used to define an environment state depends on the actual set of information contained in the update generated by the server. If the primitive event contains only the dynamic property value that was changed, or if it contains only the property values of the entity that changed, this information is not adequate to define the full state of the environment. On the other hand, if a virtual environment passes on the states of all the entities in that environment to the clients whenever a primitive event takes place, this information is adequate to define the environment state. For example, in Second Life, primitive events related to animation updates have the animation property value only, while the primitive events related to movement updates contain position, velocity and acceleration property values of that entity.

In some virtual worlds such as Second Life and an extended version of UnrealTournament (Adobbati et al., 2001), it is possible to periodically extract state information. Periodically extracted state information contains the state of many (possibly all) entities in that environment. The state of an entity does not necessarily have to be different from the same in the previous state. Rather, it may be a reassertion of the entity property values. However, if this periodically extracted state information suggests that the state of the entity has been changed, an appropriate primitive event should be generated.

4 Mechanisms to Extract Data From Second Life

There are two mechanisms available to extract data from Second Life. One important limitation of both these approaches is that they can extract only low-level data. Moreover, the retrieved data may not contain complete state information. Therefore, despite the data extraction mechanism selected, a complex event processing mechanism has to be used on the retrieved data, to identify high-level domain-specific events, along with a mechanism to generate complete state information. Below we provide an overview of these two data extraction mechanisms along with their limitations and drawbacks, and refer interested readers to Ranathunga et al. (2011a) for further information.

4.1 *Linden Scripting Language (LSL)*

Traditional programming in Second Life is done using in-world scripts written in the proprietary *Linden Scripting Language* (LSL).

Spatio-temporal data extraction using LSL scripts is heavily affected by generic LSL limitations plus limitations of the sensor function used for data extraction. The sensor function can be executed periodically, and it can detect, among other things, position and velocity data of entities and names of the animations that the avatars are currently playing. Thus a sensor function is capable of reporting the state of entities in a Second Life environment. However, the sensor function is inefficient, when executed continuously at a high frequency[1]. It also exhibits limitations with respect to the number of entities it can detect, and sensing range. Processing the extracted information inside Second Life is affected by the imposed memory limitations and the unavailability of many useful programming constructs (Ranathunga et al., 2011a). The main limitation of this approach is its inability to detect movement and animation updates of entities as and when they occur.

[1] http://lslwiki.net/lslwiki/wakka.php?wakka=lag

4.2 *LIBOMV Open Source Library*

LibOpenMetaverse or LIBOMV (OpenMetaverse Organization, 2010) is an open source .NET-based library used for accessing and creating 3D virtual worlds[2]. It is compatible with the Second Life protocol and can be used for creating clients (bots) in any virtual world that supports the Second Life protocol, such as those that are implemented with OpenSimulator.

Advantages of using LIBOMV include the ability to detect movement and animation changes of entities as they occur, avoidance of any additional lag being introduced to the Second Life servers, and the ability it provides to process the retrieved data outside the Second Life servers (Ranathunga et al., 2011a).

However, this approach does have its own limitations. The Second Life server sends information to the LIBOMV client only if there is any change in the environment perceived by the bot. Therefore, to identify movement information of objects and avatars that are moving with constant velocity, the client has to employ an extrapolation mechanism, which may not be completely accurate. Also, unlike an LSL script, a LIBOMV client cannot listen on private chat channels, meaning that information related to events such as avatars clicking objects and exchange of money should be published in the public chat channel. Another limitation of the LIBOMV approach when compared to the script-based approach is that the updates received by the LIBOMV client, by themselves, are not enough to generate environment state, because they only contain property values of a single entity (Ranathunga et al., 2011a).

5 A Robust Approach to Extract Data from Second Life

As discussed in Section 4, when recording state information from Second Life environments, both available approaches exhibit their own strengths and weaknesses. Moreover, the most suitable data extraction method should be decided based on the type of simulation scenario because Second Life simulations vary in many aspects (Ranathunga et al., 2011a). Therefore we have introduced a novel data extraction mechanism that combines the strengths of both these data extraction mechanisms, as well as an enhanced version of the LIBOMV-based data extraction. It is possible to decide on the most suitable approach from these two, based on the characteristics of the given simulation.

The mechanisms we use for data extraction are based on the concepts we defined in the formal model. Both these new data extraction mechanisms make sure that periodic state information is recorded, while also capturing primitive events, which are instantaneous. This guarantees that state changes taking place between periodic updates are not missed out. However, these two data extraction mechanisms do not create identical states of the environment, because of the different techniques they use for data extraction (Ranathunga et al., 2011a).

When obtaining data from a virtual world, three mechanisms can be used (Gemrot et al., 2011):

- Primitive events sent by the virtual world can be detected without a request from an avatar (*passive sensing*, or the *push* strategy)
- Information can be periodically requested and retrieved by an avatar (*active sensing* or the *pull* strategy)
- Information that was not provided by the virtual world can be inferred, based on the already received information (the *inference* strategy)

In the following two data extraction mechanisms we present, we use a combination of all three strategies to extract state information from Second Life environments.

5.1 *A Combined Approach to Extract Data from Second Life*

In this combined approach, we attach a simple scripted object to a LIBOMV bot. Based on the received animation and movement updates, the LIBOMV client continuously senses the entities present in the environment and sends the UUIDs of these entities to the script via a private message channel. Since the script receives the UUIDs of the entities, it can use the light-weight `llGetObjectDetails` LSL function

[2]We currently use LIBOMV 0.8 version

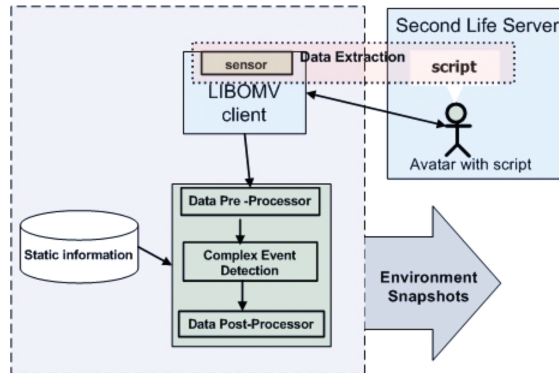

Figure 1. Main components of the framework

to record movement updates for the corresponding entity. This uses the *pull* strategy defined above. The script executes the `llGetObjectDetails` function periodically and sends the recorded entity property values to the LIBOMV client. Avatar animation updates and message updates posted in the public chat channel are captured directly by the LIBOMV client. Messages exchanged in private chat channels can be captured by the script. Receiving these primitive events can be categorised as using the *push* strategy.

This approach extracts more accurate movement information. Thus it is suitable for a simulation where accurate movement updates is important. However, this approach may not be suitable for a highly lagged simulation. Moreover, there can be some simulations that do not allow their participants to wear scripts (Ranathunga et al., 2011a).

5.2 *An Enhanced LIBOMV Approach to Extract Data from Second Life*

In this approach, we use the inference strategy and keep on extrapolating position values of entities based on the latest information received. Once an update is received from the server, the extrapolated value is updated with the server-sent value. More details on this can be found in Section 6.2.2.

The main drawback of this approach is that the LIBOMV bot may not generate correct information when entities go out of the range of the bot, or when they move constantly. In order to record chat exchanged in private chat channels, the LIBOMV client should still wear a scripted object, although the script is not involved in extracting movement updates (Ranathunga et al., 2011a).

6 The Designed Framework to Identify High-Level Events From Second Life

Figure 1 shows the designed framework to record state information from Second Life simulations. This framework is divided into four parts, and is connected to an external database that contains the static information of the given Second Life environment.

6.1 *Data Extraction Module*

The data extraction module is responsible for extracting data from a Second Life environment, using either one of the data extraction mechanisms described in the previous section. The data extraction module encompasses the sensory mechanism of the LIBOMV client and the script attached to the corresponding bot's avatar.

6.2 *Data Pre-Processor*

The main task of the data pre-processor is to amalgamate data received through different primitive events and periodically generate coherent snapshots of the Second Life environment to be used for complex event

detection. A snapshot generated by the pre-processor consists of a set of individual entity states, a set of received and inferred primitive events, and a set of messages.

The data pre-processor is also responsible for deriving the contextual information using the generated snapshot. Domain-specific static and/or semantic information can be stored in external storage and used in this process. Currently we use simple C# logic to identify contextual information, and store only static information. This static information mainly includes the positions of static objects and land marks in a given environment. Note that by pre-recording information of static objects in an environment, we can also significantly reduce the amount of information included in a snapshot, by excluding the state of these static entity states. Vosinakis and Panayiotopoulos (2003) have proposed a similar approach to identify high-level spatial relationships from virtual worlds using a rule-based scripting language. In the presence of semantic information, it is also possible to experiment with ontological reasoning mechanisms.

The following subsections describe how the data pre-processor logic for snapshot generation is designed based on the two data extraction mechanisms.

6.2.1 ***Combined Approach.*** In order to create a complete snapshot, it is always necessary to cache the last created snapshot (*ls*). The script periodically sends the recorded movement updates. However this information is not adequate for generating a snapshot, as it does not contain the animations the avatars are currently playing. The following algorithm adds this missing information to the received movement updates and generates a new snapshot *ns*. This algorithm (and the other two algorithms that follow) also try to infer any events that are implicit in the generated snapshot. Currently we infer events related to entity appearances and disappearances, and events related to velocity changes. This is done by the *infer_events* method, using the current snapshot and the last created snapshot.

```
generate_snapshot_from_movement_updates(movement_updates)
{
 ns = new Snapshot(current_time)

 FOREACH(movement_update mu in movement_updates)
  Entity e_new = new Entity()
  e_new.id = mu.e.id
  e_new.name = mu.e.name
  e_new.position = mu.position
  e_new.velocity = mu.velocity
  e_new.acceleration = mu.acceleration
  // for objects, animation is an empty string
  e_new.animations = ls.get_entity(mu.e.id).animations
  ns.add_entity(e_new)
 ns = infer_events(ns)
 ls = ns
}
```

An animation update by itself is also not enough to generate a complete snapshot, because position information of the corresponding avatar and other entity information is not contained in the received animation update. The following algorithm extrapolates the position and velocity values for this particular avatar, as well as for all the other entities when generating the new snapshot. The newly received animation update also results in generation of a new animation_update primitive event.

```
generate_snapshot_from_animation_update(animation_update au)
{
 ns = new Snapshot(current_time)

 FOREACH(Entity ent in ls)
  Entity e_new = new Entity()
  e_new.id = ent.id
  e_new.name = ent.name
  e_new.position = extrapolate_position(ent,current_time,ls.timestamp)
  e_new.velocity = extrapolate_velocity(ent,current_time,ls.timestamp)
  e_new.acceleration = ent.acceleration
```

```
   IF(ent.id == au.e.id)
    e_new.animations = au.animations
    ns.add_event("animation_update",ent.id,"Animations"->au.animation_list ,current_time)
   ELSE
    e_new.animations = ent.animations
   ns.add_entity(e_new)
  ns = infer_events(ns)
  ls = ns
}
```

Similarly, when a communication update is received by the client, position and velocity values of all the entities in ls are extrapolated. The generated snapshot additionally contains the received message. Receipt of messages also result in generation of new message events.

6.2.2 ***Extended LIBOMV Approach.*** Just like in the combined approach, here also the data pre-processor keeps track of the last created snapshot (ls). A timer is used to extrapolate position and velocity values of entities recorded in ls periodically to generate snapshots as shown in the following algorithm.

```
generate_periodic_snapshot
{
  ns = new Snapshot(current_time)
  FOREACH(entity ent in ls)
   Entity e_new = new Entity()
   e_new.id = ent.id
   e_new.name = ent.name
   e_new.position = extrapolate_position(ent,current_time,ls.timestamp)
   e_new.velocity = extrapolate_velocity(ent,current_time,ls.timestamp)
   e_new.acceleration = ent.acceleration
   e_new.animations = ent.animations
   ns.add_entity(e_new)
  ns = infer_events(ns)
  ls = ns
}
```

Whenever a primitive event (related to animation updates, movement updates and messages) is received from the server, this is used to generate a new snapshot containing the most current information using algorithms similar to the algorithms given above.

6.2.3 ***Preparing the Snapshots to be sent to the CED Module.*** The generated snapshots are then sent to another sub-component of the data pre-processor which again processes them to be sent to the CED module. For the CED module we currently use, this sub-component decomposes the generated snapshot into separate objects for each entity (including contextual information), event and message. The input to the CED module is therefore a stream of entity, event and message objects and we term these objects 'basic data items'. All these basic data items inherit the timestamp of the original snapshot, thus making it possible to use this temporal information for complex event detection.

6.3 *Complex Event Detection (CED) Module*

The responsibility of the CED module is four-fold. First, it should identify high-level complex events embedded in the low-level snapshot data. Second, it should be able to handle an incoming stream of data generated by the data pre-processor, with a frequency of at least two, or possibly more snapshots per second. Thirdly, it should be able to produce results with low latency. Having low latency is an important requirement of the CED module, especially if the output is being used for agent perception as described in Section 8, or if used for online monitoring of virtual environments. Finally, it should be able to cater for the uncertainty present in the basic data items it receives.

In order to fulfill many of these requirements, we selected the event stream processing engine called Esper[1] (EsperTech Inc., 2010), which has both CED capabilities and event stream handling capabilities. In fact, Esper is considered as the leading open source Complex Event Processing (CEP) solution provider (Ranathunga et al., 2010).

With respect to performance, Esper shows very good results, where latency is normally reported to be below 10 microseconds. The number of events processed per second is well above 100000, while reporting a low CPU usage[1].

When compared with the other open-source CEP applications such as TelegraphCQ (Chandrasekaran et al., 2003), Cayuga (Brenna et al., 2007), and Sase+ (Gyllstrom et al., 2008), the language used by Esper is more expressive. For example, TelegraphCQ does not support the conjunction, disjunction, negation, or sequence of events, while Cayuga does not support negation (Ranathunga et al., 2010). These operations on events are part of the standard set of event operators for complex event processing (Chakravarthy et al., 1994). SASE+ does not support projection (extracting a partial set of information from the selected item), or item renaming, nor does it support GROUP BY or ORDER BY operators on data streams. A survey on the available data stream processing and complex event processing techniques has been conducted by Ranathunga et al. (2010).

Esper has all the aforementioned features, plus it supports other features that are useful in identifying complex events such as sliding windows, ability to extract data from persistent storage or use of complex logic in the underlying programming language, parameterisation of selection queries, and the ability to customise event selection and consumption policies.

Being commercially supported, Esper also has other strengths in terms of quality of supporting documents, timely technical support, a large active user community, and seamless integration with Java and C# programmes, as opposed to other open source software mentioned above, which are mainly research prototypes.

As mentioned above, Esper supports both event stream queries (using selection, aggregation, ordering and grouping operators), and event patterns (using operators such as conjunction, disjunction, sequence and negation). Event identification using patterns can be done in several levels to facilitate the detection of events with a duration. Output of the earlier levels are subsequently passed on to the levels that follow, thus building up hierarchical patterns to detect complex events that are composed of primitive and other complex events.

Identification of complex events in Esper has two stages, following the general model presented by Ranathunga et al. (2010). Here, the Esper engine is responsible for specifying the complex patterns and identifying the data items that satisfy these patterns, resembling the *detection phase* of this model. Esper also has a listener that is responsible for the *production phase*. The output of the Esper engine is an object that contains all the constituent events and the explicitly specified properties of the identified pattern. In our system, the listener uses these pattern properties and generates a complex event object that is in the form $ce = \langle EType, AttVals, Es, t_s, t_e \rangle$, and it contains all the information needed to represent that complex event in the subsequent processing stages.

From the set of responsibilities of CED that we mentioned at the beginning of this section, the only requirement that is not met by Esper is handling of the uncertainty of event recognition. Approaches to address the uncertainty of event recognition have been proposed such as Markov Logic Networks (Tran and Davis, 2008), and Bayesian methods (Hongeng et al., 2004). However, performance of these approaches is not suitable for real-time event recognition, especially when trying to feed the identified information to an agent system for real time operation of agents. In this respect, the latency exhibited by Esper even for the most complex query we specified was about 1 millisecond.

At the implementation level, we have taken several steps to reduce the uncertainty of the extracted low-level data, and this is the main reason we proposed a new data extraction mechanism when accurate movement detection is needed. When there is a clear understanding of the simulation domain, it is also possible to impose more restrictions to improve the reliability of the recorded data, such as the customi-

[1]Nesper 4.0, for C#
[1]http://docs.codehaus.org/display/ESPER/Esper+performance

sation option we describe in Section 6.5. Moreover, in the virtual environment formalism, it is possible to introduce a certainty estimation value for the recorded entity and properties as proposed by Jakobson et al. (2007), where extrapolated entity properties and properties of inferred events may have a lower certainty. This certainty value can then be exploited by the CED mechanism. We leave further investigation of this for future work.

6.4 *Data Post-Processor*

The responsibility of the data post-processor is to re-assign the basic data items and identified complex events into correct snapshots. The data post-processor keeps track of all the basic data items that are received, and amalgamates all the basic data items having the same timestamp into a snapshot again. This way, the snapshot that was broken into four parts (corresponding to avatars, objects, events and messages) while being entered to the Esper engine is re-created.

6.5 *System Customisation and Usability*

The framework can be customised for different scenarios using configuration files that specify the required data extraction mechanism (from the two listed in Section 5), the Esper patterns to be used, and C# scripts to generate contextual information and to provide the Esper listener logic. It is also possible to provide a specific list of avatars and objects to be monitored, which reduces the load on the LSL script compared to the default option of retrieving details of all detected entities. Any simulation-specific animations can also be listed. Depending on the simulation requirements, frequency of periodic snapshot generation and the type of data (e.g. type of primitive event types) that should be recorded can be specified. Finally, as we have observed that the LIBOMV bot sometimes receives redundant updates for some animations, the framework can be configured to ignore these for specific animations.

Because of these customisation options, it is relatively easy to use our framework in identifying events in different simulations. When attempting to use the framework to identify evens in a new simulation, the two main steps to be considered are defining Esper patterns and designing the contextual information logic. If the simulation domain already has an ontology, then the effort needed for these two steps can be substantially reduced. However, when specifying Esper patterns and contextual information generation logic, it is required to have an extensive knowledge about the simulation domain, otherwise simple pattern variations may go unnoticed. Currently the Esper patterns and the contextual information retrieval logic have to be manually specified, however in future work we are planing to automatically generate them from a high-level domain specification. Additional complex logic needed for Esper pattern processing can be included in the C# file 'EsperUtility'. While we have provided the basic logic needed for Esper patterns in general, any simulation specific logic should be added to this file. If the simulation is complex and if it is necessary to identify complex patterns with subtle variations, a testing phase may be required in order to fine tune the parameters.

7 Detecting Events Taking Place in Different Second Life Simulations

We make use of two simulations to demonstrate how the developed framework can be used to identify events in Second Life simulations. These two simulations have separate characteristics with respect to the requirements of event identification from Second Life, providing two different test cases for our framework.

7.1 *SecondFootball Virtual Football Simulation*

The SecondFootball simulation[1] in Second Life facilitates playing virtual football. This simulation contains a football field along with scripted balls. Participants have to wear a Head Up Display (HUD) that is used

[1] http://www.secondfootball.com

to invoke football-specific animations such as tackle, up-kick and down-kick.

The primitive events that are of importance to this simulation are movement updates of the players and the ball, and the animation updates of the players. We use the script-based data extraction mechanism to generate movement updates of players and the ball, because this provided more accurate results than the extrapolation-based technique (Ranathunga et al., 2011a). With respect to avatar animations, the football-specific animations such as tackle, up-kick and down-kick play a very important role. However, primitive events generated by communication messages, or avatars clicking objects do not play any important role in this simulation.

The contextual information important for this simulation are the location of the ball and players inside the football field, and their direction of movement. We pre-record information related to field locations and land marks, and use these as static information for contextual information identification. New logic was also needed to identify the player who was in possession of the ball. A player is considered to be in possession of the ball if he is the closest to the ball and the distance between the player and the ball is less than a given threshold.

For the SecondFootball simulation, the domain-specific high-level complex events we wanted to detect were successful passing of the ball among players by means of up-kicks and down-kicks, goal scores by up-kicks and down-kicks, dribbling goal scores, and successful tackles.

For each snapshot, the Esper engine first receives the data items corresponding to the ball, followed by data items corresponding to the players.

Here we demonstrate how to identify the complex event "goal_score_by_up_kick" in several levels[1].

Level One — Identifying the "upKick" event:

The incoming data stream is analysed to find out whether a player performed the up-kick animation when he has got possession of the ball. This is important to eliminate the possibility of detecting an up-kick animation when the corresponding player did not have possession of the ball.

The following complex event pattern identifies this "upKick" event, where *a* and *b* are aliases for two basic data item inputs to the CED module.

```
SELECT *,
       EU.ProcessAttVals(b.AttMap, "entityid") AS player,
       b as constituentEvent,
       a.Timestamp AS startTime,
       a.Timestamp AS endTime
FROM PATTERN
 [EVERY
   a = EntitySnapshot(Entity.Name = 'ball')
  ->
   b = Event(EU.ProcessAttVals(b.AttMap, "animations") LIKE '%up shot%',
       StartTime = a.Timestamp,
       EU.ProcessAttVals(b.AttMap,"entityid") = EU.GetContextVal(a.ContextDic,"possession"))
 WHERE timer:within(0.5 sec)]
```

This complex event pattern searches for a sequence of basic data items where an entity state (belonging to 'EntitySnapshot' stream) corresponding to the soccer ball is followed by (identified by the sequence operator '$->$') an event (belonging to 'Event' stream) corresponding to an animation update. The conditions specify that the list of animations included in the event should contain the 'up_shot' animation, and avatar should have possession of the ball. The check for the equality of timestamps makes sure that we consider basic data items belonging to the same snapshot. Using a timer avoids the necessity of keeping the basic data items related to ball in memory for unnecessarily long periods. The wild card option in the select clause specifies that all the attributes of both 'a' and 'b' should be selected.

Level Two — Identifying the "upKickLand" event: Depending on the strength of the up kick, the ball may travel an arbitrary distance above the ground before landing on the ground. This is captured

[1]In the below patterns, 'EU' represents the EsperUtility class

by the “upKickLand” event described below.

If a is the “upKick” event identified in level one and b and c are basic data items, an “upKickLand” event is identified by the following pattern:

```
SELECT *,
 a.player AS player,
 a.startTime AS startTime,
 c[3].Timestamp AS endTime
FROM PATTERN
 [EVERY
   a = upKick
 ->
  (b = EntitySnapshot(Timestamp > a.endTime
                      Entity.Name = 'ball',
                      Entity.PositionZ > simHeight)
  UNTIL ([4]c = EntitySnapshot(Entity.Name = 'ball',
                               Entity.PositionZ = simHeight,
                               Timestamp > a.endTime)))
 ]
```

Here, after an “upKick” event is detected, we look for one basic data item that records the ball being above the ground level and four basic data items that record the ball at the ground level immediately following it. Ideally, it should suffice to record one basic data item with the ball being above the ground, followed by another one at the ground level. However, testing identified that sometimes there could be about two states where ball is still on the ground after kicking before it starts travelling above the ground (this is possible if two snapshots are generated within a few milliseconds, due to the receipt of updates). While it may seem easy to solve the problem by adding a timestamp comparison between ‘b’ and ‘c’ basic data items, this check fails when the ball starts travelling above the ground immediately after the upKick event. Therefore as a remedy we decided that this requires four basic data items when the ball is at the ground level.

Level Three — Identifying the “goal_score_by_up_kick” event: The upKickLand event is generally used to identify both goal score and successful passes among players. Therefore in order to distinguish a goal score event, this third pattern is used.

```
SELECT *,
 a.player AS player,
 EU.GetContextVal(b.ContextDic, "location") AS goal,
 a.a.constituentEvent AS constituentEvent,
 a.startTime AS startTime,
 b.Timestamp AS endTime
FROM PATTERN
 [EVERY a = upKickLand
 ->
  b = EntitySnapshot(Timestamp > a.endTime,
                     Entity.Name = 'ball',
                     Entity.PositionZ = simHeight,
                     (EU.GetContextVal(b.ContextDic, "location")  ='GoalA'
                     OR EU.GetContextVal(b.ContextDic, "location")  = 'GoalB'))
WHERE timer:within(0.9 sec)]
```

As can be seen, complex events identified in earlier levels are used in the levels that follow. The constituent events and other entity and event properties needed for the following levels are also included in the selection clause. As we claimed in Section 3.2, these patterns need both event information and other entity state information to correctly identify complex events. Moreover, the introduction of the intermediate processing

level to identify contextual information (e.g. goal location and ball possession) is very helpful in keeping the Esper patterns simple.

The Esper engine sends the goal_score_by_up_kick event to the Esper listener, and because its select clause includes all the parameter values needed to generate the corresponding complex event, the listener logic needed to create this complex event is straightforward.

Following the complex event definition given in Section 3.1.3, this complex event is given as:
$\langle goal_score_by_up_kick, \{player_name, goal_name\}, Es, t_s, t_e \rangle$

t_s corresponds to the time point at which the player played the up-kick animation, and t_e corresponds to the timestamp of the final data item included in the third event pattern.

In general, these Esper patterns together are able to correctly identify a goal score by an up-kick. However, in some extreme conditions (e.g. if the ball was very close to the goal when the player kicked it, or the player ran into the goal with the ball after kicking it), it was hard to distinguish whether the goal was scored by an up-kick or down-kick, or whether it was a dribbling goal score. In order to identify the correct goal scoring method in these extreme conditions, more parameter comparisons had to be added to the Esper pattern related to dribbling goal score. While this eliminated most of the false positives and misses, it was hard to obtain complete accuracy.

7.2 *The Otago Virtual Hospital (OVH) Simulation*

The Otago Virtual Hospital Simulation[1] is a "virtual hospital in which medical students, playing the role of junior doctors, solve open-ended clinical cases" (Blyth et al., 2010). Current analysis of participant behaviour has been based on video recordings of in-world activity, patient notes submitted by the participants, and audio recordings of pre- and post-interviews of the students. These analysis methods are cumbersome and time consuming. Moreover, with a manual analysis, it is not easy to identify complex events that arise from the long-term correlations of position, animation, and action information of avatars. Therefore we have used our framework to identify primitive events that take place during an OVH simulation, and the complex high-level events that are of interest. This is still ongoing work, and we report our early results on event recognition in this domain.

The OVH simulation is different from the SecondFootball simulation in many aspects:

- The OVH simulation contains a smaller number of participants, and they exhibit little mobility.
- The OVH simulation contains many scripted objects that are of importance to the simulation. Participants perform clinical observations such as checking the blood pressure, by clicking these objects.
- The chat exchanged in the public chat channel is important, as this information helps to identify the thinking process of the participants.

Because the objects in this simulation do not move, we consider only avatar information when creating snapshots. Static information of objects, such as their position coordinates are separately recorded and stored to be used in generating contextual information such as whether the doctor is close to the bed.

The most important primitive events in this simulation are the communication messages exchanged in the public chat channel and the private chat channels (these contain information about avatars clicking on objects (Ranathunga et al., 2011a)). We used the extrapolation-based data extraction mechanism to generate movement updates, because position information in the OVH simulation does not play a significant role. For the same reason, the frequency of periodic snapshot generation is set to 10 seconds, and the received movement updates are ignored. Animations important to the OVH simulation are washing hands, lying on the bed using different postures, and sitting on chairs.

In addition to the data items corresponding to avatars, a snapshot generated for the OVH simulation contains any messages exchanged at that instance of time. Communication messages related to avatars clicking objects are converted into domain-specific contextual information having the format $examination_name(avatar_name)$ (e.g. $checked_urine_dipstick(house_surgeon)$). However, our framework currently does not possess natural language processing techniques to capture information exchanged

[1] http://hedc.otago.ac.nz/magnolia/ovh.html

in avatar chat communication.

Complex events to be detected in the OVH simulation fall into different levels of abstractions. For example, identifying events such as closing the bed curtain, washing hands before examination, and reacting to sudden changes of the patient are at a much lower abstraction level, while identifying information such as whether the doctor was compassionate to the patient, and determining how the doctors came to the right decision have a much higher level of abstraction, and need long-term analysis and natural language processing capabilities.

Below is the pattern for a simple complex event that checks whether a doctor has carried out the basic clinical examinations within a certain period of time. This pattern uses the data items related to communication messages, when identifying the complex event. The input data stream is termed 'CommunicationMessage'

```
SELECT * FROM PATTERN [
 EVERY (a = CommunicationMessage(Message LIKE '%checked_pulse%')
       AND b = CommunicationMessage(Message LIKE '%checked_BP%')
       AND c = CommunicationMessage(Message LIKE '%checked_ECG%')
       AND d = CommunicationMessage(Message LIKE '%checked_urine_dipstick%'))
 WHERE timer:within(1200 sec)]
```

The listener receives an object that contains all these constituent events (Es), and it generates the corresponding complex event:

$\langle done_basic_clinical_exam, \{avatar_name\}, Es, t_s, t_e \rangle$. Here, t_s refers to the timestamp of 'a', and t_e refers to the timestamp of 'd'.

8 Implementing Intelligent Virtual Agents with Better Cognitive Abilities

An important requirement of intelligent virtual agents is the ability to be believable (Gemrot et al., 2011; Sindlar et al., 2009). In order to achieve this requirement, they should be able to perceive and comprehend the environment around them at different abstraction levels. Our event identification framework provides a solution to this problem for intelligent agents deployed in a Second Life environment.

In order to demonstrate how an agent system can utilise the developed framework to make agents better perceive and comprehend its Second Life virtual environment, we have extended the framework with an interface that facilitates the integration of any agent system with Second Life. Each agent is deployed inside Second Life as an avatar using a dedicated LIBOMV client, and once inside Second Life, the agents use our framework to identify events taking place around them. As shown later in this section, it also provides a solution to the percept overloading of virtual agents, which occurs due to the fact that virtual worlds operate much faster and generate large amounts of low-level sensory data, when compared with multi-agent systems that perceive their environment at a lower frequency, and deal with more declarative information. Solutions proposed in our framework are thus applicable to any other applications that attempt to deploy agents in virtual worlds.

Different agent development mechanisms present different ways for an agent to react to events identified in its environment. From these mechanisms, belief-desire-intention (BDI) agent interpreters such as Jason (Bordini et al., 2007) provide a flexible way for agents to react to events. Therefore we integrated the Jason BDI interpreter with our framework to demonstrate how an agent can react to events it identifies in its Second Life environment.

Jason is an open source Java-based multi-agent systems development platform that incorporates an interpreter for the agent programming language AgentSpeak (Rao, 1996). 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 (explicitly asserted

beliefs) to itself or execute internal actions (Jason's built-in operations).

The Jason platform we have used in this integration is an extended version of Jason, which was developed in our previous research (Ranathunga et al., 2011b). This extended version of Jason implements a tight integration of agent *expectation monitoring* with the Jason BDI agent model. It allows agents to delegate to an expectation monitor the monitoring of rules that specify conditional temporal logic constraints on the future.

An agent may base its practical reasoning on the assumption that one or more of its expectations will hold, while ensuring that it will receive notification events when these rules are fulfilled and/or violated. Therefore, in addition to the continuous stream of domain-specific high-level events and state information that an agent receives, it can dynamically subscribe to fulfilment and violation events for specific rules of expectation that are appropriate to its personal or social context. The rules are defined using temporal logic operators in conjunction with the propositions available in the snapshots of the Second Life state.

Fulfilments and violations of agent expectations therefore form a new set of complex events that are introduced by the agent dynamically and represent a new level of abstraction above the state descriptions received from our framework. In order to monitor agent-initiated expectations, we have chosen to use a separate event recognition mechanism on top of Esper, which is an expectation monitor that was developed in previous research (Cranefield and Winikoff, 2010).

The use of a separate formalism and event detection technique is due to the higher level of abstraction at which agent expectations are defined. While Esper patterns are well suited for specifying the detailed logic for combining low-level events into structured complex event objects as shown in Section 6.3, they lack formal semantics[1]. In contrast, the expectation monitor described below is based on the propositional representation of states but has formal semantics.

8.1 *Detecting Events Corresponding to Fulfilments and Violations of Agent Expectations Using the Expectation Monitor*

The language that the expectation monitor accepts includes the following operators[2]:

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

where $Condition$ and $Expectation$ are formulae in a form of linear propositional temporal logic (Cranefield and Winikoff, 2010) that together define a conditional rule of expectation. $Condition$ 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 Exp formula states that an expectation is currently active due to the rule defined by its two arguments having fired in a previous or the current state. Fulf and Viol formulae state that the rule defined by the arguments has created an expectation that has become fulfilled or violated, respectively, in the current state.

An expectation is active when its condition evaluates to true in the current state. The expectation is then considered to be fulfilled or violated if it evaluates to true, but as it may contain future-oriented temporal operators, this evaluation must be done without considering any future states (which may be available, for example, if the monitor is examining an audit trail while 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 (Bacchus and Kabanza, 2000), e.g. if p holds in the current state, then an expectation that "p is true and in the next state q will be true" will progress to the simpler expectation that "q is true".

[1]Esper semantics are described by a textual description only.
[2]We use simplified names for the operators compared to the earlier presentation of this logic (Cranefield and Winikoff, 2010).

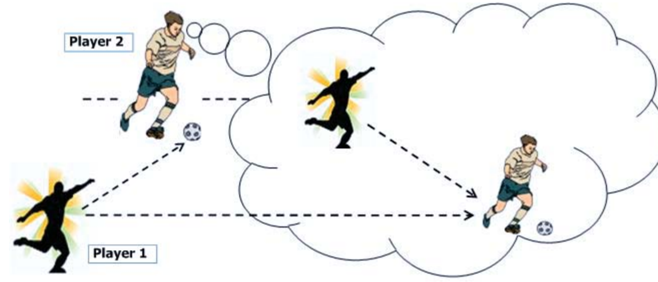

Figure 2. The "give and go" tactic in football

The expectation monitor is a model checker that takes a formula and an observed linear trace as input and determines in which states the formula holds (note that the monitor addresses the computational problem of "model checking a path", and does not determine if the formula holds in all possible traces generated by an input automaton). When the monitor is run in an online mode where new states are appended to the model as they occur, it can be considered as a generator of complex events defined in terms of the activation, fulfilment or violation of rules of expectation[3]. Furthermore, the monitor can be used as an on-demand service by agents, so they can control in which contexts they wish to receive which type of events, and for what rules of expectation.

For example, consider the "give and go" football team play scenario illustrated in Figure 2. The give and go play involves one player (player 1 in the figure) passing the ball to another player (player 2). Player 2 then expects player 1 to run down the field to an advantageous position, and has the intention of passing the ball back to player 1 once that has been achieved. While player 2 has the ball, he will need to focus his attention on controlling the ball and advancing down the field while avoiding the tackles of opposition players. However, player 2 must notice when player 1 reaches the desired position (e.g. the penalty area in front of goal B) so that the ball can be quickly passed back to player 1. Furthermore, if player 1 is unable to complete his part of the play (e.g. if he falls over), then player 2 should also become aware of this fact and initiate a new plan of attack.

With the expectation monitor available, player 2's need to be aware of player 1's performance can be delegated by invoking the monitor to send back a notification when either of the following two formulae becomes true:

$$\mathsf{Fulf}(s97, advanceToGoalB(player1)\,\mathsf{U}\,penaltyB(player1))$$

$$\mathsf{Viol}(s97, advanceToGoalB(player1)\,\mathsf{U}\,penaltyB(player1))$$

The arguments of these formulae represent the following rule: when the proposition *s97* is true, we have the expectation that from then onwards player 1 will be advancing towards goal B until he is in the penalty area for goal B. The proposition *s97* is a *nominal* (a proposition that is true in one state only), and we assume that this refers to the state in which player 2 initiates the monitor, so that the rule is triggered immediately (and then never again). The operator U is the standard 'until' operator of temporal logic. As the logic used is a propositional temporal logic, the parentheses within the two propositions that are arguments to U are uninterpreted—they are characters like any others within the proposition name. However, allowing parentheses within propositions is useful for the integration with the Jason BDI interpreter described in the next section.

The first formula above will be true in any state in which the rule is fulfilled (i.e. player 1 completes his advance to the penalty area), and the second formula will be true in any state in which the rule is violated (e.g. player 1 moves away from the goal or stops moving before reaching the penalty area). In these cases, the monitor will send an event back to the agent to inform it that this rule was fulfilled or violated. This example is continued in Section 8.3 where we show some agent plans in Jason that respond to these events.

[3]In the integration with Jason discussed in the next section we currently only consider fulfilment and violation events.

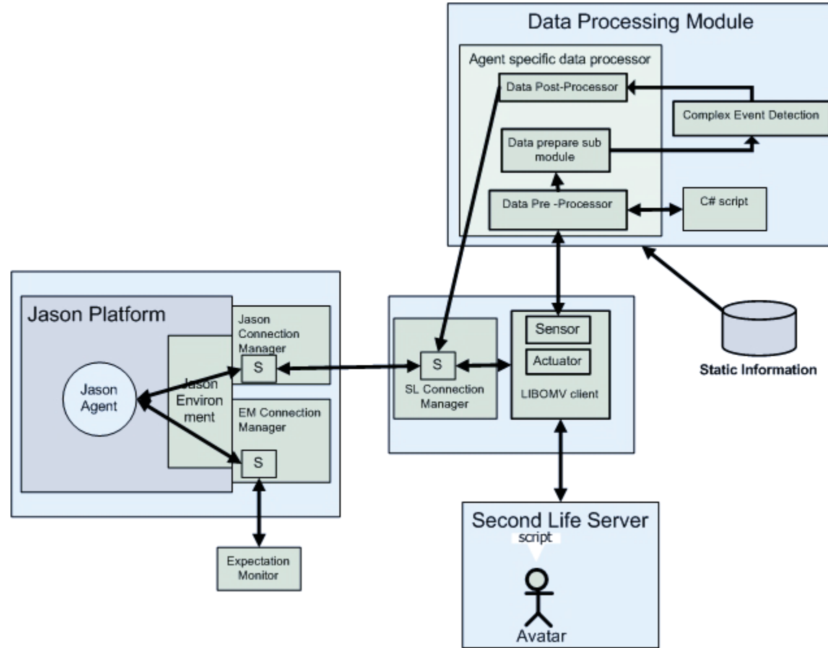

Figure 3. Extending the framework to integrate Jason with Second Life

8.2 *Integrating Jason with Second Life*

Figure 3 shows how the framework is extended with a new interface to connect the Jason multi-agent development framework with Second Life[1], along with a new actuator component to execute agent actions inside Second Life. The actuator module is responsible for making the LIBOMV avatar execute the agent's commands inside Second Life. This is a modified version of the agent connection framework presented by Ranathunga et al. (2011c).

Communication between the LIBOMV client and the Jason agent is facilitated using a pre-defined protocol and they communicate through sockets (denoted by 'S' in Figure 3). The protocol currently defines how an agent should pass commands to the LIBOMV client such as login requests, messages, performing movements and animations. It also defines how an agent platform can interpret a message sent by the LIBOMV client. These messages mainly include state information (snapshots) generated by our framework.

In this extended framework, the CED module serves as a common service for all the LIBOMV clients. Even though this imposes a limitation where all the agents have to share the same set of complex event patterns, it improves the performance of the system. The C# script used for defining contextual information has also been made a common service to all the agents, for similar reasons.

The input to the expectation monitor is a state model, where each state represents the world at a particular time point. Snapshots generated by the data post-processor serve as the ideal candidate to form the state model for the expectation monitor. The data post-processor generates propositions representing only the events and contextual information included in a snapshot, thus eliminating any low-level data. If a snapshot is identical to the previous snapshot, it is simply discarded. While these design choices at the implementation level reduce the amount of percepts being sent to an agent, inclusion of events at different abstraction levels is very much useful for situation comprehension by the agent.

The Jason environment class processes the information received from the framework, and makes it available both to the agents and the expectation monitor. The EM Connection Manager facilitates the connection between the expectation monitor and the Jason environment. This communication is also implemented using a text-based communication protocol.

The temporal logic used by the expectation monitor allows domain information to be recorded using

[1]We show only one agent-avatar pair, for convenience.

ordinary propositions and so-called "state-referencing propositions". The latter provides a way of encoding the occurrence of events that began at a prior state, continued across a range of consecutive states, and were verified to have completed in the current state. For example, consider the events of making a pass and kicking a goal in football. Both are initiated by a kick and are only known to have occurred in some later state when the pass is received or the goal is scored. Not all kicks result in successful passes or goals being scored, so the event can only be recorded in the state where the event completion is detected. The proposition recording the occurrence of such an event therefore takes a parameter recording the state in which the event started.

Each complex event includes information about its constituent events. The data post-processor processes this information and generates state referencing propositions. For example, a successful pass is represented by the proposition 'successfulkick(su_monday,ras_ruby)(9)' which states that su_monday successfully passed the ball to ras_ruby, and this event began in the prior state that has the state number 9 (the state in which su_monday kicked the ball).

The example set of propositions below encodes a state containing the contextual information relating to two players, su_monday and ras_ruby, as well as the soccer ball, in a given instance of time. The propositions describe their locations, move type and style, and the player that is currently in possession of the ball.

```
possession(ras_ruby,ball); location(midfieldB2,ball);
moveType(notmoving,ball); animation(stand,ras_ruby); location(midfieldB2,ras_ruby);
moveType(notmoving,ras_ruby); animation(stand,su_monday); location(midfieldB2,su_monday);
moveType(notmoving,su_monday); successfulkick(su_monday,ras_ruby)(9)
```

The Jason platform, our data processing framework, and the expectation monitor run in three separate threads. Synchronisation of these three processes is important, in order to make sure that agents are behaving in a believable manner. In fact, synchronisation between a virtual world and a BDI-based multi-agent system has been identified as a difficult technical task (Dignum et al., 2009), because virtual worlds have much faster execution cycles than an agent reasoning cycle. For example, the execution cycle of a Jason agent is about 500 milliseconds, while our framework generates snapshots at a much higher frequency, depending on the frequency of updates received.

The 500 millisecond execution cycle suggests that a Jason agent cannot consume the snapshots at the rate they are generated, and it can be overloaded with percepts. As a solution to this problem, there is a choice between automatically amalgamating snapshots (possibly using some domain-specific rules), which could lose information about some events having occurred, or each time the agent senses the environment we could return a sequence of snapshot data for each entity and leave it to the application level code to handle this additional complexity.

The expectation monitor may take a longer execution time to process a snapshot than a Jason agent, depending on the complexity of the monitoring rule. This suggests that an agent may not be able to respond to the fulfilment or violation of its expectations in the same reasoning cycle as the one that processes the snapshot that caused the fulfilment or violation. Therefore fulfilment and violation notifications are sent to Jason as beliefs that include a state identifier indicating when the fulfilment or violation occurred.

8.3 *Example — A Jason Agent Engaged in Football Team Play Scenario "Give and Go"*

The following example demonstrates how a Jason agent can engage in a football training scenario with a human controlled player (we have not yet implemented the other agent). In particular it shows how our framework allows Jason plans to respond to domain-specific high-level events in Second Life that have been detected by our framework.

In this example, we outline an implementation of the football team play scenario "give and go" that was discussed in Section 8.1 and shown in Figure 2. Here, the Jason agent ras_ruby is engaged in the team play scenario with the player su_monday, which is controlled by a human. When ras_ruby receives the ball, he adopts the expectation that su_monday should advance up the field continuously until the penaltyB area is reached, so that he can pass the ball back to su_monday, for her to attempt a shot at goal.

When the system starts, the Jason agent corresponding to ras_ruby is initialised and the agent has an

initial goal to start playing (the prefix '!' in the code identifies this as a goal). He also has an initial belief (or a 'mental note', in the Jason terminology) about the current tactic he is supposed to play. Here, the current tactic is to try the give and go team play, between su_monday and himself.

```
/* Initial belief */
 current_tactic(give_and_go(su_monday, ras_ruby)).

/* Initial goal */
!start_playing.
```

The plan responding to the `start_playing` goal logs ras_ruby into Second Life. As soon as ras_ruby logs in to Second Life, he starts receiving snapshots that describe the state of the environment. The agent ras_ruby moves to the area MidfieldB2. Once there, he waits for su_monday to pass the ball to him. When he receives the ball, the corresponding snapshot contains the proposition $successful_kick(su_monday, ras_ruby)$, and by processing this proposition, the percept successful_kick is generated. This percept also has the annotation $state(N)$, which refers to the state at which this complex event is detected.

In the reasoning process of the agent, this percept becomes a new belief. Note that the agent could acquire this higher level belief because our framework is able to identify these high-level events. If the agent had to depend only on low-level sensory data, he is not able to achieve this type of high-level reasoning. As the context condition is instantiated to $current_tactic(give_and_go(su_monday, ras_ruby))\&.my_name(ras_ruby)$, the plan below gets triggered.

```
+successful_kick(OtherAgent, Me)[state(N)] :
     current_tactic(give_and_go(OtherAgent, Me)) & .my_name(Me)
     <-
         .term2string(OtherAgent, OtherAgentStr);

         .concat("('U',",
                    "'advanceToGoalB(", OtherAgentStr, ")',",
                    "'penaltyB(", OtherAgentStr, ")')",
                 Expectation);

         .start_monitoring("fulf", "move_to_target", "expectation_monitor",
                           "#once", Expectation, [N]);

         .start_monitoring("viol", "move_to_target", "expectation_monitor",
                           "#once", Expectation, [N]).
```

During the execution of this plan, ras_ruby starts monitoring for the fulfilment and violation of his expectation. We use the internal action *start_monitoring* defined in the extended version of the Jason platform, and initiate monitoring the rule of expectation, both for fulfilment and violation. The first parameter of this internal action specifies the mode of monitoring. The first call to this internal action specifies 'fulf', meaning that the monitor should check for the fulfilment of the rule. The second call to the internal action has the value 'viol', which means that the monitor should also check for the violation of the rule. When a monitor based on the logic described in Section 8.1 is used, these two cases cause the monitoring of the truth of Fulf and Viol formulae, respectively.

The second parameter of *start_monitoring* assigns a name to the expectation, for the ease of future reference. Here, we use the same name "move_to_target" in both calls. The third parameter is the name of the expectation monitor used, as our framework allows different monitor tools to be used. The fourth parameter is the triggering condition for the expectation to become active, while the fifth parameter is the expectation to be monitored. These two parameters are generally strings containing encodings of the condition and expectation of the rule to be monitored in whatever format is required by the expectation monitor specified by the third parameter. The condition and expectation of the rule provide the arguments

of the Fulf and Viol formulae shown in Section 8.1. In this example, the fifth parameter is a string containing a Python expression suitable for our current Python-based monitor. Before *start_monitoring* is called, Jason's `.concat` internal action is used to assemble a string representation of this expectation formula. We use the sixth parameter to inform the expectation monitor of the state that triggered monitoring.

The keyword `#once` appearing as the condition for the rule of expectation has a special meaning. As discussed in Section 8.1, for this scenario the initiating agent wants the rule to fire precisely once, immediately, and this can be achieved by using a 'nominal' 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. This is generated from the state number associated with the most recent state propositions it has received.

In our scenario, if su_monday fulfilled ras_ruby's expectation, the expectation monitor detects this event and reports back to ras_ruby along with the corresponding state number (denoted by the variable 'X'). The following plan handles this detected fulfilment event and instructs ras_ruby's avatar to turn towards su_monday[1] and carry out the kick action.

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

On the other hand, if su_monday violated the expectation, the expectation monitor reports the identified violation event to ras_ruby, and the first plan below reacts to this violation by creating a goal to choose a new tactic and executing it. The second plan below is then triggered, and ras_ruby adopts the tactic of attempting to score a goal on his own.

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

+!choose_and_enact_new_tactic : .my_name(Me)
  <-
    -+current_tactic(solo_run(Me));
    action("run", "penaltyB").
```

9 Performance Evaluation of the Framework

Performance evaluation on the framework has been carried out using two metrics. The first metric records the amount of server lag introduced by the concurrently running scripts that record movement information of avatars and objects. The second metric records the time taken for the framework to process a received primitive event into a snapshot.

9.1 *Measuring the Amount of Server Lag Imposed by Scripts*

We used the time dilation values sent by the server to the LIBOMV client to measure the amount of lag experienced by the server. Time dilation is represented on a scale of 0-1: the higher the value, the lower the lag. The tests were designed in such a way to measure the server lag when the following parameters are varied:

- The number of mobile avatars in the region (ranging from 1 to 5)

[1] We have not yet implemented this part of the tactic.

- The number of data extracting scripts present in the region (ranging from 1 to 5)
- The amount of work done by a script — whether it is simply used as a listener on log channels, or used for movement data extraction
- The script running time (ranging from ten minutes to one hour)

A complete description of the conducted tests is presented by Ranathunga et al. (2011a). In summary, the results of the tests show that there is no direct relationship between the lag experienced by the server and the above parameters. One possible explanation for this behaviour could be due to the changes that are said to be introduced to Second Life servers, which may cause the scripts to run much slower without introducing server lag (Ranathunga et al., 2011a). Supporting this assumption, it can be seen that the time gap between two consecutive messages from the script varies. However it is always concentrated around 500 milliseconds, which is the timer interval we used in the script (Ranathunga et al., 2011a).

Therefore, even if a script may not slow down an entire Second Life server, the change in script execution time suggests that it is not possible to depend on the interval set in a script timer. Moreover, even though a direct connection is not evident between the test parameters and server lag, our Second Life viewer experienced client-side lag when the number of concurrently operating avatars was increased. The amount of experienced client-side lag was the highest when all the five moving avatars were wearing scripts. Therefore we can say that the initial suggestion we made about selecting the type of data extraction mechanism is still valid, despite the fact that the server lag does not show direct evidence for this.

9.2 *Measuring the Efficiency of Data Processing Logic in Generating Snapshots*

Test results show that the framework takes, on average 45 milliseconds to generate a snapshot from an update received from the Second Life server. This result is for one LIBOMV client thread running in an Intel Core2 Quad CPU with 2.40GHz processing power, while handling snapshots containing two avatars. Therefore we can say that the framework has reasonably low latency, when generating snapshots. We have tested the framework with up to six concurrent avatars, and the results did not show any noticeable increase in the snapshot processing time. This indicates that the framework is scalable enough to process snapshots with multiple entities.

The CED module (both the Esper engine and the listener) takes less than one millisecond to process a received basic data item. The data pre-processor and the data post-processor are roughly taking equal times to process a received snapshot.

10 Related Work

Some researchers have tried to use Second Life data to understand the special characteristics of the virtual presence of humans and related social norms (Friedman et al., 2008; Yee et al., 2007; La and Michiardi, 2008; Varvello et al., 2008). However in this research, the explored social behaviours were limited to simple aspects such as interpersonal distance, and eye gaze. Consequently, there was no attempt to extract high-level complex information embedded in these low-level data, to facilitate a comprehensive analysis. Cranefield and Li (2009) have proposed the first research related to extracting temporal data to identify high-level events to deduce whether members in a Second Life virtual community have fulfilled or violated the rules defined for that community. However, this research was carried out in a narrow scope using LSL scripts that only dealt with avatar animations.

Several research that attempted to deploy intelligent agents inside Second Life can also be found, such as in the work by Jan et al. (2009), Ullrich et al. (2008), and Bogdanovych et al. (2010). However they have not focused explicitly on identifying events taking place in Second Life virtual environments.

When looking at the related research on identifying events in virtual worlds in general, we are not aware of any application of complex event recognition mechanisms. As mentioned earlier, Vosinakis and Panayiotopoulos (2003) have proposed the concept of geometric memory to understand high-level information in virtual worlds. Zhang and Hill (2000) also proposed a way to identify spatial structures in virtual worlds; however both these attempts did not take temporally defined events into consideration.

Although it still has not been exploited by research related to virtual worlds, complex event recognition is a rich research on its own. Specifically, complex event detection mechanisms have been heavily employed in the research related to human activity recognition in videos. According to Aggarwal and Cai (1999), research related to human activity or behavior recognition basically fall into two categories, namely state-space approaches and template matching techniques.

For example, the CED mechanism presented in this paper uses a template matching technique. Some attempts to use templates for activity recognition have also been reported by Polana et al. (1994) and Bobick and Davis (1996). As we have already mentioned, the main advantage of using a template-based mechanism is the lower computational cost. When it is important to address the uncertainty of the input low-level data and intermediate processing steps, state-space approaches have been experimented with. These state-space approaches mainly involve different variations of Hidden Markov Model (HMM) techniques. Some such examples are the work of Tran and Davis (2008), Hakeem and Shah (2007), and Biswas et al. (2007). Due to the inherent complexities in activity recognition, most of the activity recognition mechanisms such as those of Nevatia et al. (2003), Hongeng et al. (2004), Hakeem and Shah (2007) propose hierarchical event recognition approaches.

11 Conclusion

This paper addressed the problems of accurately extracting and processing spatio-temporal data from Second Life and identifying the events taking place inside Second Life environments. We have described two mechanisms that can be used to accurately extract spatio-temporal data from most, if not all of the Second Life simulations.

As for the problem of identifying events in Second Life environments, our solution had several components. Firstly, we presented a general formalism for dynamic virtual environments, which provided a common understanding of different concepts of a virtual environment, and their relationships. We proposed a data amalgamation mechanism to generate complete state information, and argued that it is important in identifying complex events at different abstraction levels. We also introduced an intermediate processing step that generates contextual information, which made the complex event identification less complex. We presented a complex event recognition mechanism using the Esper complex event processing engine as a solution to process the large amount of low-level data in real-time. Finally, we presented a temporal logic based event recognition mechanism that identifies events at a more abstract social level.

We incorporated these mechanisms in a framework that can be used to identify low-level and high-level domain-specific events from Second Life environments and demonstrated it in the context of two Second Life simulations with very different characteristics. We also demonstrated its use in implementing more believable and rational virtual agents inside Second Life, by extending the framework with an interface connecting multi-agent systems with Second Life.

Results of our evaluation tests show that there is no direct relationship between the lag present in the Second Life server and the workload introduced by the additional scripts. The framework exhibited very low latency when identifying events based on the received low-level data from Second Life, and also proved to be scalable.

As future work, there is an opportunity to experiment with different event recognition mechanisms in order to identify the best suited approach in identifying high-level complex events embedded in the extracted low-level data, while paying special attention to how to handle the uncertainty of the received low-level data. We also intend to introduce code generation for Esper patterns and contextual information specification.

References

Adobbati, R., A. N. Marshall, A. Scholer, S. Tejada, G. Kaminka, S. Schaffer, and C. Sollitto (2001). Gamebots: A 3D Virtual World Test-Bed for Multi-Agent Research. In *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS.*

Aggarwal, J. K. and Q. Cai (1999). Human Motion Analysis: A Review. *Computer Vision and Image Understanding 73*, 428–440.

Bacchus, F. and F. Kabanza (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence 116*(1-2), 123–191.

Biswas, R., S. Thrun, and K. Fujimura (2007). Recognizing activities with multiple cues. In *Proceedings of the 2nd conference on Human motion: understanding, modeling, capture and animation*, Berlin, Heidelberg, pp. 255–270. Springer-Verlag.

Blyth, P., S. K. Loke, and J. Swan (2010). Otago Virtual Hospital: medical students learning to notice clinically salient features. In *Curriculum, technology & transformation for an unknown future. Proceedings ascilite Sydney 2010*, pp. 108–112.

Bobick, A. and J. Davis (1996). Real-time recognition of activity using temporal templates. In *Proceedings of the 3rd IEEE Workshop on Applications of Computer Vision (WACV '96)*, WACV '96, Washington, DC, USA, pp. 39–. IEEE Computer Society.

Bogdanovych, A., J. A. Rodriguez-Aguilar, S. Simoff, and A. Cohen (2010). Authentic Interactive Reenactment of Cultural Heritage with 3D Virtual Worlds and Artificial Intelligence. *Applied Artificial Intelligence 24*(6), 617–647.

Bordini, R. H., J. F. Hubner, and M. Wooldridge (2007). *Programming Multi-Agent Systems in Agent Speak using Jason*. John Wiley & Sons Ltd, England.

Brenna, L., A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White (2007). Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, New York, NY, USA, pp. 1100–1102. ACM.

Chakravarthy, S., V. Krishnaprasad, E. Anwar, and S.-K.Kim (1994). Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, San Francisco, CA, USA, pp. 606–617. Morgan Kaufmann Publishers Inc.

Chandrasekaran, S., O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah (2003). TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*.

Cranefield, S. and G. Li (2009). Monitoring Social Expectations in Second Life. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pp. 1303–1304. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC.

Cranefield, S. and M. Winikoff (2010). Verifying social expectations by model checking truncated paths. *Journal of Logic and Computation*. Advance access, doi: 10.1093/logcom/exq055.

Diener, S., J. Windsor, and D. Bodily (2009). Design and development of clinical simulations in Second Life. In *Proceedings of the EDUCAUSE Australasia Conference*.

Dignum, F., J. Westra, W. A. van Doesburg, and M. Harbers (2009). Games and Agents: Designing Intelligent Gameplay. *International Journal of Computer Games Technology 2009*, 1–18.

Ellis, S. R. (1994). What are virtual environments? *IEEE Computer Graphics and Applications 14*, 17–22.

EsperTech Inc. (2010). Esper Tutorial. `http://esper.codehaus.org/tutorials/tutorial/tutorial.html`.

Friedman, D., A. Steed, and M. Slater (2008). Spatial Social Behavior in Second Life. In *Proceedings of the 7th international conference on Intelligent Virtual Agents*, pp. 252–263. Springer-Verlag Berlin, Heidelberg.

Gemrot, J., C. Brom, R. Kadlec, M. Bída, O. Burkert, M. Zemčák, R. Píbil, and T. Plch (2010). Pogamut 3 Virtual Humans Made Simple. In J. Gray (Ed.), *Advances in Cognitive Science*, pp. 211–243. The Institution Of Engineering And Technology.

Gemrot, J., C. Brom, and T. Plch (2011). A periphery of pogamut: from bots to agents and back again. In F. Dignum (Ed.), *Agents for games and simulations II*, pp. 19–37. Berlin, Heidelberg: Springer-Verlag.

Gyllstrom, D., J. Agrawal, Y. Diao, and N. Immerman (2008). On Supporting Kleene Closure over Event Streams. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, Washington, DC, USA, pp. 1391–1393. IEEE Computer Society.

Hakeem, A. and M. Shah (2007). Learning, detection and representation of multi-agent events in videos. *Artificial Intelligence 171*, 586–605.

Helleboogh, A., G. Vizzari, A. Uhrmacher, and F. Michel (2007, February). Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-Agent Systems 14*, 87–116.

Hindriks, K. V., B. Van Riemsdijk, T. Behrens, R. Korstanje, N. Kraayenbrink, W. Pasman, and L. De Rijk (2011). UnREAL Goal Bots: conceptual design of a reusable interface. In F. Dignum (Ed.), *Agents for games and simulations II*, pp. 1–18. Berlin, Heidelberg: Springer-Verlag.

Hongeng, S., R. Nevatia, and F. Bremond (2004). Video-based event recognition: activity representation and probabilistic recognition methods. *Computer Vision and Image Understanding 96*, 129–162.

Jakobson, G., J. F. Buford, and L. Lewis (2007). Situation Management: Basic Concepts and Approaches. In *Information Fusion and Geographic Information Systems*, pp. 18–33.

Jan, D., A. Roque, A. Leuski, J. Morie, and D. Traum (2009). A Virtual Tour Guide for Virtual Worlds. In *Proceedings of the 9th International Conference on Intelligent Virtual Agents*, pp. 372–378. Springer-Verlag Berlin, Heidelberg.

La, C.-A. and P. Michiardi (2008). Characterizing User Mobility in Second Life. In *Proceedings of the first workshop on Online social networks*, pp. 79–84. ACM New York, USA.

Linden Lab (2010). Second Life Home Page. `http://secondlife.com`.

Nevatia, R., T. Zhao, and S. Hongeng (2003). Hierarchical Language-based Representation of Events in Video Streams. *Computer Vision and Pattern Recognition Workshop 4*, 39.

OpenMetaverse Organization (2010). libopenmetaverse developer wiki. `http://lib.openmetaverse.org/wiki/Main_Page`.

Polana, R., R. Nelson, and A. Nelson (1994). Low Level Recognition of Human Motion (Or How to Get Your Man Without Finding his Body Parts). In *In Proc. of IEEE Computer Society Workshop on Motion of Non-Rigid and Articulated Objects*, pp. 77–82. Press.

Ranathunga, S., S. Cranefield, and M. Purvis (2010). Processing Flows of Information: From Data Stream to Complex Event Processing. Technical report, Politecnico di Milano.

Ranathunga, S., S. Cranefield, and M. Purvis (2011a). Extracting Data from Second Life. Discussion Paper 2011/07, Dept. of Information Science, University of Otago.

Ranathunga, S., S. Cranefield, and M. Purvis (2011b). Integrating Expectation Handling into Jason. In *International Workshop on Programming Multi-Agent Systems (ProMAS 2011)*.

Ranathunga, S., S. Cranefield, and M. Purvis (2011c). Interfacing a Cognitive Agent Platform with a Virtual World: a Case Study using Second Life. In *International Workshop on the uses of Agents for Education, Games and Simulations (AEGS 2011)*.

Rao, A. S. (1996). 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*, pp. 42–55. Springer-Verlag Berlin, Heidelberg.

Rogers, L. (2009). Simulating clinical experience: Exploring Second Life as a learning tool for nurse education. In *Proceedings of ascilite Auckland 2009*.

Sindlar, M., M. Dastani, and J.-J. Meyer (2009). BDI-Based Development of Virtual Characters with a Theory of Mind. In Z. Ruttkay, M. Kipp, A. Nijholt, and H. Vilhjlmsson (Eds.), *Intelligent Virtual Agents*, Volume 5773 of *Lecture Notes in Computer Science*, pp. 34–41. Springer Berlin / Heidelberg.

Tran, S. D. and L. S. Davis (2008). Event Modeling and Recognition Using Markov Logic Networks. In *Proceedings of the 10th European Conference on Computer Vision: Part II*, Berlin, Heidelberg, pp. 610–623. Springer-Verlag.

Ullrich, S., K. Bruegmann, H. Prendinger, and M. Ishizuka (2008). Extending MPML3D to Second Life. In *Proceedings of the 8th international conference on Intelligent Virtual Agents*, pp. 281–288. Springer-Verlag Berlin, Heidelberg.

Varvello, M., F. Picconi, C. Diot, and E. Biersack (2008). Is There Life in Second Life? In *Proceedings of the 2008 ACM CoNEXT Conference*, pp. 1:1–1:12. ACM New York, USA.

Veksler, V. D. (2009). Second Life as a Simulation Environment: Rich, high-fidelity world, minus the hassles. In *Proceedings of the 9th International Conference of Cognitive Modeling*.

Vosinakis, S. and T. Panayiotopoulos (2003). Programmable Agent Perception in Intelligent Virtual

Environments. In *IVA*, pp. 202–206.

Yee, N., J. N. Bailenson, M. Urbanek, F. Chang, and D. Merget (2007). The Unbearable Likeness of Being Digital: The Persistence of Nonverbal Social Norms in Online Virtual Environments. *CyberPsychology and Behavior 10*, 115–121.

Zhang, W. and R. W. Hill, Jr. (2000). A template-based and pattern-driven approach to situation awareness and assessment in virtual humans. In *Proceedings of the fourth international conference on Autonomous agents*, AGENTS '00, New York, NY, USA, pp. 116–123. ACM.