Monitoring Social Expectations in Second Life

Stephen Cranefield and Guannan Li

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

Abstract. Online virtual worlds such as Second Life provide a rich medium for unstructured human interaction in a shared simulated 3D environment. However, many human interactions take place in a structured social context where participants play particular roles and are subject to expectations governing their behaviour, and current virtual worlds do not provide any support for this type of interaction. There is therefore an opportunity to adapt the tools developed in the MAS community for structured social interactions between software agents (inspired by human society) and adapt these for use with the computer-mediated human communication provided by virtual worlds.

This paper describes the application of one such tool for use with Second Life. A model checker for online monitoring of social expectations defined in temporal logic has been integrated with Second Life, allowing users to be notified when their expectations of others have been fulfilled or violated. Avatar actions in the virtual world are detected by a script, encoded as propositions and sent to the model checker, along with the social expectation rules to be monitored. Notifications of expectation fulfilment and violation are returned to the script to be displayed to the user. This utility of this tool is reliant on the ability of the Linden scripting language (LSL) to detect events of significance in the application domain, and a discussion is presented on how a range of monitored structured social scenarios could be realised despite the limitations of LSL.

1 Introduction

Much of the research in multi-agent systems addresses techniques for modelling, constructing and controlling open systems of autonomous agents. These agents are taken to be self-interested or representing self-interested people or organisations, and thus no assumptions can be made about their conformance to the design goals, social conventions or regulations governing the societies in which they participate. Inspired by human society, MAS researchers have adopted, formalised and created computational infrastructure allowing concepts from human society such as trust, reputation, expectation, commitment and narrative to be explicitly modelled and manipulated in order to increase agents' awareness of the social context of their interactions. This awareness helps agents to carry out their interactions efficiently and helps preserve order in the society, e.g. the existence of reputation, recommendation and/or sanction mechanisms discourages anti-social behaviour.

As the new 'Web 2.0' style Web sites and applications proliferate, people's use of the Web is moving from passive information consumption to active information sharing and interaction within virtual communities; in other words, for millions of users, the Web is now a place for social interaction. However, while Web 2.0 applications provide the middleware to enable interaction, they generally provide no support for users to maintain an awareness of the social context of their interactions (other than basic presence information indicating which users in a 'buddy list' online). There is therefore an opportunity for the software techniques developed in MAS research for maintaining social awareness to be applied in the context of electronically mediated human interaction, as well as in their original context of software agent interaction.

This paper reports on an investigation into the use of one such social awareness tool in conjunction with the Second Life online virtual world. Second Life is a 'Web 3D' application providing a simulated three dimensional environment in which users can move around and interact with other users and simulated objects [1]. Users are represented in the virtual world by animated avatars that they control via the Second Life Viewer client software. Human interaction in virtual worlds is essentially unconstrained—the users can do whatever they like, subject to the artificial physics of the simulated world and a few constraints that the worlds support, such as the ability of land owners to control who can access their land. However, many human interactions take place in a structured social context where participants play particular roles and there are constraints imposed by the social or organisational context, e.g. participants in a meeting should not leave without formally excusing themselves, and students in an in-world lecture should remain quiet until the end of the lecture. Researchers in the field of multi-agent systems have proposed (based on human society) that the violation of social norms such as these can be discouraged by publishing explicit formal definitions of the norms, building tools that track (relevant) events and detect any violations, and punishing offenders by lowering their reputations or sanctioning them in some other way [2]. Integrating this type of tool with virtual worlds could enhance the support provided by those worlds for social activities that are subject to norms.

In this research we have investigated the use of a tool for online monitoring of 'social expectations' [3] in conjunction with Second Life. The mechanism involves a script running in Second Life that is configured to detect and record particular events of interest for a given scenario, and to model these as a sequence of state descriptions that are sent to an external monitor along with a property to be monitored. The monitor sends notifications back to the script when the property is satisfied so that the user can be informed.

The rest of this paper is structured as follows. Section 2 describes how we have used the Linden Scripting Language to detect avatars in Second Life and create a sequence of propositional state models to send to the monitor. The architecture for communication between this script and the monitor is presented in Section 3. Section 4 discusses the concept of conditional social expectations used in this work, and the model checking tool that is used as the expectation monitor. Section 5 presents some simple scenarios of activities in Second Life being monitored, and Section 6 discusses some issues arising from limitations of the Linden Scripting Language and the temporal logic used to
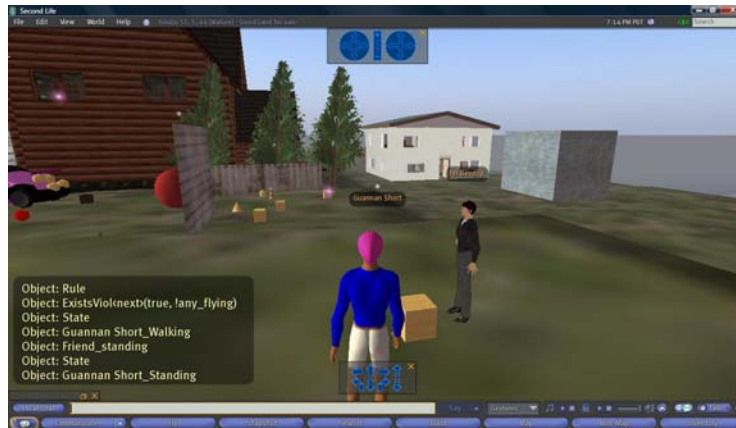
Fig. 1. The Second Life Viewer

express rules. Some related work is described in Section 7, and Section 8 concludes the paper.

2 Detecting events in Second Life

As shown in Figure 1, the Second Life Viewer provides, by default, a graphical view of the user's avatar and other objects and avatars within the view. The user can control the 'camera' to obtain other views. Avatars can be controlled to perform a range of basic animations such as standing, walking and flying, or predefined "gestures" that are combinations of animation, text chat and sounds. Communication with other avatars (and hence their users) is via text chat, private instant messages, or audio streaming. The user experience is therefore a rich multimedia one in which human perception and intelligence is needed to interpret the full stream of incoming data. However, the Linden Scripting Language (LSL [4]) can be used to attach scripts to objects (e.g. to animate doors), and there are a number of sensor functions available to detect objects and events in the environment. These scripts are run within the Second Life servers, but have some limited ability to communicate with the outside world.

LSL is based on a state-event model, and a script consists of defined states and handlers for events that it is programmed to handle. Certain events in the environment automatically trigger events on a script attached to an object. These include collisions with other objects and with the 'land', 'touches' (when a user clicks on the object), and money (in Linden dollars) being given to the object. Some other types of event must be explicitly subscribed to by calling functions such as `llSensor` and `llSensorRepeat` for scanning for avatars and objects within a given arc and range, `llListen` for detecting chat messages from objects or avatars within hearing range, and `llSetTimerEvent` for setting a timer. These functions take parameters that provide some selectivity over what is sensed, e.g. a particular avatar name or object type can be spec-

ified in `llListen`, and `llListen` can be set to listen on a particular channel, for a message from a particular avatar, and even for a particular message.

In this paper we focus on the detection of other avatars via the function `llSensorRepeat`, which repeatedly polls for nearby avatars (we choose not to scan for objects also) at an interval specified in a parameter. A series of `sensor` events are then generated, which indicate the number of avatars detected in each sensing operation. A loop is used to get the unique key that identifies each of these avatars (via function `llDetectedKey`) and the avatar's name (via `llDetectedName`). The key can then be used to obtain each avatar's current basic animation (via `llGetAnimation`). The script can be configured with a filter list specifying which avatar/animation observations should be either recorded or ignored, where the specified avatar and animation can refer to a particular value, or "any". Detected avatar animations are filtered through this list sequentially, resulting in a set of $(avatar\_name, animation)$ pairs that comprise a model of the current state of the avatars within sensor range. Another configuration list specifies the optional assignment of avatars to named groups or roles such as "Friend" or "ClubOfficial". There is currently no connection with the official Second Life concept of a user group (although official group membership can be detected). Group names can also be included in the filter list, with an intended existential meaning, i.e. a pair $(group\_name, animation)$ represents an observation that *some* member of the group is performing the specified animation. The configuration lists provide scenario-specific relevance criteria on the observed events, and are read from a 'notecard' (a type of avatar inventory item that is commonly used to store textual configuration data for scripts), along with the property to be monitored.

When the script starts up, it sends the property to be monitored to the monitor. It then sends a series of state descriptions to the monitor as sensor events occur. However, we choose not to send a state description if there is no change since the previous state, so states represent periods of unchanging behaviour rather than regularly spaced points in time. State descriptions are sets of proposition symbols of the form *avatar_animation* or *group_animation*.

This process can easily be extended to handle other types of Second Life events that have an obvious translation to propositional (rather than predicate) logic, such as detecting that an avatar has sent a chat message (if it is not required to model the contents of the message). Section 6 discusses this further.

3 Communication between Second Life and the monitor

Second Life provides three mechanisms for communication with entities outside their own server or the Second Life Viewer: scripts can send email messages, initiate HTTP requests, or listen for incoming XML-RPC connections (which must include a parameter giving the key for a channel previously created by the script). To push property and state information to the monitor we use HTTP. However, instead of directly embedding the monitor in an HTTP server, to avoid local firewall restrictions we have chosen to use Twitter [5] as a message channel. An XML-RPC channel key, the property to be monitored and a series of state descriptions are sent to a predefined Twitter account as

direct messages using the HTTP API[1]. The Twitter API requires authentication, which can be achieved from LSL only by including the username and password in the URL in the form http://username:password@.....

The monitor is wrapped by a Java client that polls Twitter (using the Twitter4J library [7]) to retrieve direct messages for the predetermined account. These are ignored until a pair of messages containing an XML-RPC channel key and a property to be monitored (prefixed with "C:" and "P:" respectively) are received, which indicates that a new monitoring session has begun. The monitoring session then consists of a series of messages beginning with "S:", each containing a list of propositions describing a new state. The monitor does not currently work in an incremental 'online' mode—it must be given a complete history of states and restarted each time a new state is received[2]; therefore, the Java wrapper must record the history of states. It also generates a unique name for each state (which the monitor requires).

Each time a state is received, the monitor (which is implemented in C) is invoked using the Java Native Interface (JNI). The rule and state history are written to files and the names passed as command-line arguments. An additional argument indicates the desired name of the output file. The output is parsed and, if the property is determined to be true in any state, that information is sent directly back to the Second Life script via XML-RPC.

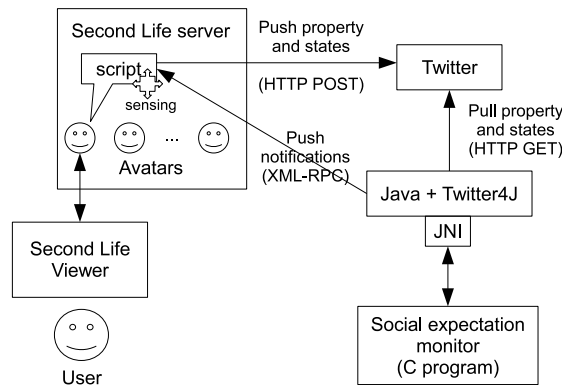Figure 2 gives an overview of the communication architecture.

Fig. 2. The communications architecture

---

[1] Twitter messages are restricted to 140 characters and calls to the Twitter API are subject to a limit of 70 requests per hour, which is sufficient for testing our mechanism. For production use an alternative HTTP-accessible messaging service could be used, such as the Amazon Simple Queue Service [6].

[2] Work is in progress to add an online mode to the monitor.

4 Monitoring social expectations

4.1 Modelling social expectations

MAS researchers working on normative systems and electronic institutions [2] have proposed various languages for modelling the rules governing agent interaction in open societies, including abductive logic programming rules [8], enhanced finite state machine style models, [9], deontic logic [10], and institutional action description languages based using formalisms such as the event calculus [11].

The monitor used in this work is designed to track rules of *social expectation*. These are temporal logic rules that are triggered by conditions on the past and present, resulting in *expectations* on present and future events. The language does not include deontic concepts such as obligation and permission, but it allows the expression of social rules that impose complex temporal constraints on future behaviour, in contrast to the simple deadlines supported by most normative languages. It can also be used to express rules of social interaction that are less authoritative than centrally established norms, e.g. conditional rules of expectation that an agent has established as its personal norms, or rules expressing learned regularities in the patterns of other agents' behaviour. The key distinction between these cases is the process that creates the rules, and how agents react to detected fulfilments and violations.

Expectations become active when their condition evaluates to true in the current state. These expectations are then considered to be fulfilled or violated if they evaluate to true in a state without considering any future states that might be available in the model[3]. If an active expectation is not fulfilled or violated in a given state, then it remains active in the following state, but in a "progressed" form. Formula progression involves partially evaluating the formula in terms of the current state and re-expressing it from the viewpoint of the next state [12]. A detailed explanation is beyond the scope of this paper, but a simple example is that an expectation $\bigcirc \phi$ (meaning that ϕ must be true in the state that follows) progresses to the expectation ϕ in the next state.

4.2 The social expectation monitor

The monitoring tool we have used is an extension [3] of a model checker for hybrid temporal logics [13]. Model checking is the computational process of evaluating whether a formal model of a process, usually modelled as a Kripke structure (a form of nondeterministic finite state machine), satisfies a given property, usually expressed in temporal logic. For monitoring social expectations in an open system, we cannot assume that we can obtain the specifications or code of all participating agents to form our model. Instead our model is the sequence of system states recorded by a particular observer, in other words, we are addressing the problem of *model checking a path* [14]. The task of the model checker is therefore not to check that the overall system *necessarily* satisfies

---

[3] This restriction is necessary, for example, when examing an audit trail to find violations of triggered rules in *any* state. The standard temporal logic semantics would conclude that an expectation "eventually p" is fulfilled in a state s even if p doesn't become true until some later state s'.

a given property, but just that the observed behaviour of the system has, to date, satisfied it. The properties we use are assertions that a social expectation exists or has been fulfilled or violated, based on a conditional rule of expectation, expressed in temporal logic.

The basic logic used includes these types of expression, in addition to the standard Boolean constants and connectives (true, false, $\wedge$, $\vee$ and $\neg$):

- Proposition symbols. In our application these represent observations made in Second Life, e.g. *avatar_name_sitting*.
- $\bigcirc \phi$: formula ϕ is true when evaluated in the next state
- $\Diamond \phi$: ϕ is true in the current or some future state
- $\Box \phi$: ϕ is true in all states from now onwards
- $\phi \, \mathsf{U} \, \psi$: ψ is true at the current or some future state, and ϕ is true for all states from now until just before that state

$\Diamond$ and $\Box$ can be expressed in terms of U and are abbreviations of longer expressions.

The logic also has some features of Hybrid Logic [15], but these are not used in this work except for the use of a *nominal* (a proposition that is true in a unique state) in the output from the model checker to 'name' the state in which a fulfilled or violated rule of expectation became active.

Finally, the logic includes the following operators related to conditional rules of expectation, and these are the types of expression sent from the Second Life script to the model checker:

- ExistsExp($Condition, Expectation$)
- ExistsFulf($Condition, Expectation$)
- ExistsViol($Condition, Expectation$)

where $Condition$ and $Expectation$ can be any formula that does not include ExistsExp, ExistsFulf and ExistsViol.

The first of these operators evaluates to true if there is an expectation existing in the current state that results from the rule specified in the arguments being triggered in the present or past. The other two operators evaluate to true if there is currently a fulfilled or violated expectation (respectively) resulting from the rule.

Formal semantics for this logic can be found elsewhere [3].

The input syntax to the model checker is slightly more verbose than that shown above. In particular, temporal operators must indicate the name of the "next state modality" as it appears in the input Kripke structure. In the examples in this paper, this will always be written as "<next>". Writing "<next>" on its own refers to the operator $\bigcirc$.

5   Two Simple scenarios

A simple rule of expectation that might apply in a Second Life scenario is that no one should ever fly. This might apply in a region used by members of a group that enacts historical behaviour. To monitor this expectation we can use the following property:

```
ExistsViol<next>(true, !any_flying)
```

This is an unconditional rule (it is triggered in every state) stating the expectation that there will not be any member of the group "Any" (comprising all avatars) flying.

If this is the only animation state to be tracked, the script's filter list will state that the animation "Flying" for group "Any" should be recorded, but otherwise all animations for all avatars and other groups should be discarded. On startup, the script sends the property to be monitored to the monitor, via Twitter, and then as avatars move around in Second Life and their animations are detected, it sends state messages that will either contain no propositions (if no one is flying) or will state that someone is flying:

```
S: any_flying
```

These states are accumulated, and each time a new state is received, the monitor is called and provided with the property to be monitored and the model (state history), e.g. $s_1 : \{\}$, $s_2 : \{\}$, $s_3 : \{\texttt{any\_flying}\}$ (the model is actually represented in XML—an example appears below).

For this model, the monitor detects that the property is satisfied (i.e. the rule is violated) in state s_3 and a notification is sent back to the script. How this is handled is up to the script designer, but one option is for the script to be running in a "head-up-display" object, allowing the user to be informed in a way that other avatars cannot observe.

We now consider a slightly more complex example where there are two groups (or roles) specified in the script's group configuration list: leader (a singleton group) and follower. We want to monitor for violations of the rule that once the leader is standing, then from the next state a follower must not be sitting until the leader is sitting again. This is expressed using the following property:

```
ExistsViol<next>(
  leader_standing,
  <next>(U<next>(!follower_sitting,
                leader_sitting))
)
```

The filter list can be configured so that only the propositions occurring in this rule are regarded as relevant for describing the state.

Suppose the scenario begins with the leader sitting and then standing, followed by the follower sitting, and finally the leader sitting again. This causes the following four states to be generated:
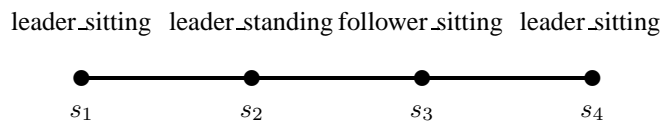
This is represented in the following XML format to be input to the model checker:

```
<hl-kripke-struct name="M">
  <world label="s1"/>
  <world label="s2"/>
  <world label="s3"/>
  <world label="s4"/>
  <modality label="next">
    <acc-pair to-world-label="s2"
              from-world-label="s1"/>
    <acc-pair to-world-label="s3"
              from-world-label="s2"/>
    <acc-pair to-world-label="s4"
              from-world-label="s3"/>
  </modality>
  <prop-sym label="leader_standing"
            truth-assignments="s2"/>
  <prop-sym label="leader_sitting"
            truth-assignments="s1 s4"/>
  <prop-sym label="follower_sitting"
            truth-assignments="s3"/>
  <nominal label="s1" truth-assignment="s1"/>
  <nominal label="s2" truth-assignment="s2"/>
  <nominal label="s3" truth-assignment="s3"/>
  <nominal label="s4" truth-assignment="s4"/>
</hl-kripke-struct>
```

The output of the model checker is:

```
s3: (s2, U<next>(!(follower_sitting),
                 leader_sitting))
```

This means that a violation occurred in state s_3 from the rule being triggered in state s_2. The violated expectation (after progression to state s_3) is:

```
U<next>(!(follower_sitting), leader_sitting)
```

This information is sent to the script.

6 Discussion

As mentioned in Section 2, our detection script currently only detects the animations of avatars within sensor range. This limits the scenarios that can be modelled to those based on (simulated) physical action. However, it is straightforward to add the ability to detect other LSL events, provided that they can be translated to a propositional representation. Thus we could detect that an avatar has sent a chat message, but we can't provide a propositional encoding that can express all possible chat message contents. However, the addition of new types of configuration list would allow additional flexibility. For example, regular expressions or other types of pattern could be defined along

with a string that can be appended to an avatar or group name to generate a proposition meaning that that avatar (or a member of that group) sent a chat message matching the pattern.

A significant limitation of the Linden Scripting Language is that the events that a script can detect are focused on the scripted object's own interactions with the environment—there is no facility for observing interactions between other agents, except for what can be deduced from their animations and chat. For many scenarios, it would be desirable to detect these interactions, for example, passing a certain object or sending money from one avatar to another might be a significant event in a society. One way around this problem would be to add additional scripted objects to the environment and set up the social conventions that these objects must be used for certain purposes. For example, an object in the middle of a conference table might need to be touched in order to request the right to speak next. These objects would generate appropriate propositions and send them to the main script via a private link.

The logic used currently is based on a discrete model of time, which can cause problems in some scenarios. For example, in the leader/follower scenario, it would be reasonable to allow the follower some (short) amount of time to stand after the leader stands. However, the moment that if a follower stands and another does not stand within the granularity of the same sensor event, then that second follower will be deemed in violation. It would be useful to be able to model some aspects of real time. This could be done by moving to a real-time temporal logic (which would involve some theoretical work on extending the model checker), or by some pragmatic means such as allowing the configuration parameters to define a frequency for regular "tick" timer events.

7 Related work

There seems to be little prior work that has explored the use of social awareness technology from multi-agent systems or other fields to support human interaction on the Internet in general, and in virtual worlds in particular.

A few avatar rating and reputation systems have been developed [16] to replace Second Life's own ratings system, which was disestablished in 2007. These provide various mechanisms to allow users to share their personal opinions of avatars with others.

Closer to our own work, Bogdanovych et al. [17, 18] have linked the AMELI electronic institution middleware [19] with Second Life. However, their aim is not to provide support for human interactions within Second Life, but rather to provide a rich interface for users to participate in an e-institution mediated by AMELI (in which the other participants may be software agents). This is done by generating a 3D environment from the institution's specification, e.g. *scenes* in the e-institution become rooms and transitions between scenes become doors. As a user controls their avatar to perform actions in Second Life, this causes an associated agent linked to AMELI to send messages to other agents, as defined by an action/message mapping table. Moving the avatar between rooms causes the agent to make a transition between scenes, but doors in Second Life will only open when the agent is allowed to make the corresponding scene transition according the rules of the institution.

This approach could be used to design and instrument environments that support structured human-to-human interaction in Second Life, but the e-institution model of communication is highly stylised and likely to seem unnatural for human users. In our work we are aiming to provide generic social awareness tools for virtual world users while placing as few restrictions as possible on the forms of interaction that are compatible with those tools. However, as discussed in Section 6, the limitation of the sensing functions provided by virtual world scripting languages may mean that some types of scenario cannot be implemented without providing specific scripted coordination objects that users are required to use, or the use of chat messages containing precise specified phrases.

8 Conclusion

This paper has reported on a prototype application of a model checking tool for social expectation monitoring applied to monitoring social interactions in Second Life. The techniques used for monitoring events in Second Life and allowing communication between a Second Life script and the monitor have been described, and these have been successfully tested on some simple scenarios. A discussion was presented on some of the limitation imposed by the LSL language and the logic used in the model checker, along with some suggestions for resolving these issues.

References

1. Linden Lab: Second Life home page. http://secondlife.com/ (2008)
2. Boella, G., van der Torre, L., Verhagen, H.: Introduction to normative multiagent systems. In Boella, G., van der Torre, L., Verhagen, H., eds.: Normative Multi-agent Systems. Number 07122 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
3. Cranefield, S., Winikoff, M.: Verifying social expectations by model checking truncated paths. In: Coordination, Organizations, Institutions, and Norms in Agent Systems IV. Volume 5428 of Lecture Notes in Computer Science. Springer (2009) 204–219
4. Linden Lab: LSL portal. http://wiki.secondlife.com/wiki/LSL_Portal (2008)
5. Twitter: Twitter home page. http://twitter.com/ (2008)
6. Amazon Web Services: Amazon simple queue service. http://aws.amazon.com/sqs/ (2008)
7. Yamamoto, Y.: Twitter4j. http://yusuke.homeip.net/twitter4j/en/ (2008)
8. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based software tool. In Trappl, R., ed.: Cybernetics and Systems 2004. Volume II., Austrian Society for Cybernetics Studies (2004) 570–575
9. Esteva, M., de la Cruz, D., Sierra, C.: ISLANDER: an electronic institutions editor. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems, ACM (2002) 1045–1052
10. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Implementing norms in multiagent systems. In: Proceedings of the Second German Conference on Multiagent System Technologies (MATES). Volume 3187 of Lecture Notes in Computer Science., Springer (2004) 313–327

11. Farrell, A.D.H., Sergot, M.J., Sallé, M., Bartolini, C.: Using the event calculus for tracking the normative state of contracts. International Journal of Cooperative Information Systems **14**(2 & 3) (2005) 99–129

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

13. Dragone, L.: Hybrid logics model checker. http://luigidragone.com/hlmc/ (2005)

14. Markey, N., Schnoebelen, P.: Model checking a path. In: CONCUR 2003 – Concurrency Theory. Volume 2761 of Lecture Notes in Computer Science. Springer (2003) 251–265

15. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2001)

16. Second Life: Removal of ratings in beta. `http://blog.secondlife.com/2007/04/12/removal-of-ratings-in-beta/` (2007)

17. Bogdanovych, A., Berger, H., Sierra, C., Simoff, S.J.: Humans and agents in 3D electronic institutions. In: Proceedings of the 4rd International Joint Conference on Autonomous Agents and Multiagent Systems, ACM (2005) 1093–1094

18. Bogdanovych, A., Esteva, M., Simoff, S.J., Sierra, C., Berger, H.: A methodology for 3d electronic institutions. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, IFAAMAS (2007) 358–360

19. Esteva, M., Rosell, B., Rodrguez-Aguilar, J.A., Arcos, J.L.: AMELI: An agent-based middleware for electronic institutions. In: Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems. Volume 1., IEEE Computer Society (2004) 236–243