ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Simulation Modelling Practice and Theory xxx (2005) xxx–xxx

SIMULATION MODELLING PRACTICE AND THEORY

www.elsevier.com/locate/simpat

Agents, games and HLA ☆

M. Lees [a], B. Logan [a,*], G.K. Theodoropoulos [b]

[a] University of Nottingham, School of Computer Science and Information Technology, Nottingham, NG8 1BB, United Kingdom
[b] University of Birmingham, School of Computer Science, Edgbaston, Birmingham B15 2TT, United Kingdom

Received 15 September 2005; accepted 14 October 2005

Abstract

Over the past decade, there has been a growing interest in utilising intelligent agents in computer games and virtual environments. At the same time, computer game research and development has increasingly drawn on technologies and techniques originally developed in the large scale distributed simulation community, such as the HLA IEEE standard for simulator interoperability. In this paper, we address a central issue for HLA-based games, namely the development of HLA-compliant game agents. We present HLA_AGENT, an HLA-compliant version of the SIM_AGENT toolkit for building cognitively rich agents. We outline the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA, and show that, given certain reasonable assumptions, all necessary code can be generated automatically from the FOM and the object class publications and subscriptions. The integration is transparent in the sense that the existing SIM_AGENT code runs unmodified and the agents are unaware that other parts of the system are running remotely. We present some preliminary performance results, which indicate that the overhead introduced by the HLA extension is modest even for lightweight agents with limited computational requirements.

Keywords: Distributed simulation; Games; Agent-based systems; High level architecture

1. Introduction

There has been considerable recent interest in *agent-based systems*, systems based on autonomous software and/or hardware components (agents) which cooperate within an environment to perform some task. An *agent* can be viewed as a self-contained, concurrently executing thread of control that encapsulates some state and communicates with its environment and possibly other agents via some sort of message passing [44]. The environment of an agent is that part of a physical or computational system in which the agent is embedded, that is from which it can obtain information and on which it can act. Agent-based systems offer advantages when independently developed components must inter-operate in a heterogeneous environment, e.g., the

☆ This work is part of the PDES-MAS project and is supported by EPSRC research grant no. GR/R45338/01 (http://www.cs.bham.ac.uk/research/pdesmas/).
* Corresponding author.
E-mail address: bsl@cs.nott.ac.uk (B. Logan).

Internet, and agent-based systems are increasingly being applied in a wide range of areas including telecommunications, business process modelling, control of mobile robots and military simulations [6,20].

One novel application area of agents is computer games. The search for enhanced believability and the falling cost of hardware has increasingly led games developers to exploit agent technology in games. Research in intelligent agents in areas such as generating real-time responses to a dynamic environment, handling multiple conflicting goals, working in teams, integrating models of personality, emotion and social role and their effect on behaviour and so on, addresses many of the issues which are relevant to games developers. Conversely, much recent work in intelligent agents has drawn heavily on games and game-like environments, e.g., [3,9,21,25]. Modern computer games which have evolved from simple, single user games with primitive 2D graphics, to sophisticated virtual worlds realised in 3D graphics. While such games are still simpler than the real world, they provide a range of locations, situations, objects, characters and actions, which present agents in the game with a complex, dynamic environment. Many computer games are real-time, and the environment can be changed by a human player or other characters. Games therefore present a challenging agent design problem, and the lessons learned in these simplified environments can in many cases be transferred to more general AI problems [24].

A key factor in the adoption of games as an agent development platform has been changes in the software architecture of games. Many modern computer games adopt a client–server architecture, in which users in remote locations (clients) interact in a common virtual environment maintained by a central *game server* [29]. Client server architectures allow agents running as remote processes to connect to the game in a way similar to human players, interacting with the game world via networking middleware which presents a well-defined interface for sensing and action. One example of such an approach is *Gamebots*, a socket-based interface to the Unreal Tournament (UT) game engine [21]. Gamebots provides an agent with data that approximates to that available to a human player, and allows the agent to perform all of the actions available to a human player through the standard game interface.[1]

Gamebots has been successfully applied in a number of game agent research projects, e.g., developing real-time architectures for game agents [17], and reporting on games tournaments [14]. However, approaches such as Gamebots have a number of drawbacks as a basis for game agent development. Such systems are specific to a particular game or game engine: agents and other components developed for one game can not be easily incorporated into a game produced by a different software developer, with the result that code reuse is limited. Moreover, the available systems tend to focus on real time interaction, e.g., in first person shooter games, and do not support other synchronisation schemes which may be appropriate for different game types, e.g., strategy games or persistent games which may be played over a period of days or even longer. As an example, consider a football management game (such as the "Championship Manager" series),[2] in which a user manages a team of football player agents. In such a scenario, the task of the agent development is split between commercial software developers (perhaps different developers for different teams), and the user whose role is to train (or evolve) the basic team provided by the developer by adjusting the individual agent's playing behaviour and/or playing their team against other teams.[3] Both initial development of the basic team by a software developer and training by the user is likely to involve a mixture of real time and virtual time simulation to test/debug the behaviour of individual players and to test or develop the behaviour of the team as a whole. Agents and environments (stadia) developed by different software developers in different programming languages must be able to inter-operate with each other both when playing on the user's system and remotely, e.g., as part of a league or when controlled by their user-managers. If there are a large number of teams in the league it may be advantageous to play many of the games in virtual time, whereas "championship" games or games where the team is controlled by the user may be played in real time. Finally, systems such as Gamebots which rely on a single game server to maintain the game state often have limited scalability.[4] There is therefore a need for a standardised, flexible, scalable middleware for medium to large-scale distributed games.

[1] Other examples include interfaces to Quake II [25], and Half-Life [22].
[2] See http://www.sigames.com/.
[3] See, for example, the Co-Evolutionary Robot Soccer Show http://www.legolab.daimi.au.dk/cerss/.
[4] For example, the Unreal Tournament game server is limited to 16 simultaneous connections (participants).

As computer games become more and more complex they increasingly resemble in complexity, functionality and requirements, Large Scale Distributed Simulation (LSDS) systems. Much of the current work in LSDS has centred around the High Level Architecture (HLA), a framework for simulation reuse and interoperability developed by the US DoD Defence Modelling and Simulation Organization[5] and since adopted (with minor revisions) as an IEEE standard (IEEE 1516) [19]. Using HLA, a large-scale distributed simulation can be constructed by linking together a number of geographically distributed simulation components (or federates) into a single, larger simulation (or federation).

The increasing overlap between computer games and LSDS has led a number of researchers and developers to explore the potential benefits of utilising HLA in games. The flexibility of the HLA standard means that it can be used to support a wide range of game types, from large scale virtual worlds in which players interact in real time over WANs, to games which involve a mixture of simulation and execution. The last few years has seen an increasing amount of work in this area, both in the form of academic research, e.g., [7], and commercial systems, e.g., Mak Technology's HLA Game-Link HLA/DIS adapter[6] which allows games developed using the Epic Games' Unreal Engine[7] to inter-operate with DIS and HLA compliant simulations, Magnetar Games Chronos networking engine for games[8] which implements HLA/RTI using DirectPlay, and Cybernet's OpenSkies massively multiplayer networking infrastructure[9] which is based on HLA.

Given the increasing prevalence of agents in games, a central issue for HLA-based games is the development of HLA-compliant game agents. While there has been a limited amount of work in this area (see, for example, [42]), this has focused on fairly low level Java agent development tools such as JADE.[10] Such tools are appropriate for game agents, which exhibit simple scripted behaviours. However, more complex behaviours, such as planning, inference and learning, which are increasingly felt to be necessary for interesting and challenging games (see, for example, [5,35,43]) must be coded from scratch in Java.

In this paper, we present HLA_AGENT, an HLA-compliant version of the SIM_AGENT toolkit [40,41].[11] SIM_AGENT is a high-level toolkit designed to support the development of *cognitively rich* agents which integrate a broad range of cognitive capabilities of the sort typically required by modern game agents, such as perception, motive generation, planning, plan execution, execution monitoring and emotional reactions. HLA_AGENT is capable of supporting a wide range of agent applications in HLA-based games, from simulation based games, to the implementation of NPCs in role playing games. The SIM_AGENT toolkit can be used both as a sequential, centralised, time driven simulator for multi-agent systems and as an agent implementation language. In previous work [26–28] we have reported the application of HLA_AGENT for distributed agent simulation. In this paper we focus on the use of SIM_AGENT as an agent implementation language for intelligent, human-like agents, e.g., NPCs in a computer game, and show how multi-agent systems developed using SIM_AGENT can be distributed using HLA/RTI, allowing inter-operation with other HLA-compliant applications, such as game engines.

The rest of the paper is organised as follows: Section 2 provides a brief overview of HLA and summarises the potential benefits of HLA for computer games. In Section 3, we briefly describe the SIM_AGENT toolkit and illustrate its application in a simple Tileworld scenario. In Section 4, we outline how the HLA can be used to distribute an existing SIM_AGENT system with different agents being executed by different federates. In Section 5, we sketch the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA. The integration is transparent in the sense that the existing SIM_AGENT code runs unmodified and the agents are unaware that other parts of the system are running remotely. In Section 6, we present some performance results which indicate that the overhead introduced by the HLA extension is modest even for lightweight agents with limited computational requirements. The paper concludes with a brief description of future work.

---

[5] http://www.dmso.mil/hla

[6] www.mak.com/gamelink.php

[7] www.unrealtechnology.com

[8] www.magnetargames.com/Products/Chronos

[9] www.openskies.net

[10] Another relevant example is HLA_RePast [33], however the RePast toolkit is targeted more at agent-based simulation rather than the development of game agents.

[11] http://www.cs.bham.ac.uk/~axs/cog_affect/sim_agent.html

2. HLA and games

The High Level Architecture (HLA) allows different simulations, referred to as *federates*, to be combined into a single larger simulation known as a *federation* [10]. The federates may be written in different languages and may run on different machines. A federation is made up of:

- one or more federates,
- a Federation Object Model (FOM),
- the Runtime Infrastructure (RTI).

The FOM defines the types of and the relationship among the data exchanged between the federates in a particular federation. Each FOM consists of a set of object classes and a set of interaction classes. Each object class defines a (possibly empty) set of named data called attributes. Instances of these object classes and their associated attribute values are created by the federates to define the persistent state of the simulation. Federates update the state of the simulation by supplying new values for the attributes of object instances. An interaction is a set of named data, called parameters, which forms a logical unit within the federation, e.g., an event within the simulation model. Unlike objects, interactions have no continued existence after they have been received. Object and interaction classes are organised into (separate) inheritance hierarchies, in which each class inherits the attributes (for objects) or parameters (for interactions) of its superclasses. Each federate must typically translate from its internal notion of simulated entities to HLA objects and interactions as specified in the FOM.

The RTI is the middleware that provides common services to simulation systems. Communication between federates and federations is done via the RTI. Each federate contains an RTI Ambassador and a Federate Ambassador along with the user simulation code. The RTI Ambassador handles all outgoing information passed from the user simulation to the RTI. Each call made by the RTI Ambassador typically results in a corresponding callback on other federates. For example, updating the value of an attribute of an instance of an object class defined in the FOM on one federate will result in a callback containing the new value on federates which subscribe to the attribute. It is the task of the Federate Ambassador to handle these callbacks and invoke appropriate code in the user simulation, e.g., update the value of a field or variable representing the attribute. The FOM is passed to the RTI at the beginning of an execution and effectively parameterises the RTI for that federation.

The HLA provides services in six areas, namely Federation Management, Object Management, Declaration Management, Ownership Management, Time Management, and Data Distribution Management [10].

There are three distinct ways in which agent-based computer games (and computer games in general) can benefit by the facilities offered by the HLA.

The first way in which HLA and RTI could be exploited in computer games, is as a standard and middleware engine, respectively. In a climate of ever richer variety of virtual environment systems (in games, arts, performances and entertainment), scenarios can be envisioned where ever more complex and dynamic interactive virtual worlds will be created where individuals will be allowed to build and introduce components at will.[12] In order to compete, a need for an underpinning standard and architecture will arise to support composition and reuse of components for games and HLA/RTI may well serve this purpose [23]. Much of the existing work on applying HLA to games has been in this area.

The second relates to the design and development of the agents themselves. While agents offer great promise, their adoption has been hampered by the limitations of current development tools and methodologies. Multi-agent systems are often extremely complex and it can be difficult to formally verify their properties. As a result, design and implementation remains largely experimental, and experimental approaches are likely to remain important for the foreseeable future. In this context, simulation has a key role to play in the development of agent-based systems, and a wide range of simulators and testbeds have been developed to support

[12] See, for example SecondLife (www.secondlife.com), a (non-HLA-compliant) persistent online 3D virtual environment. The SecondLife environment is extremely open-ended. The environment provides the basic landscape and the laws of physics, and users employ content authoring tools (3D modelling and simple scripting) to create whatever they like.

the design and analysis of agent architectures and systems [1,2,11,36,38,41]. There has also been some work on simulating game environments for development purposes [12,37]. However, no single testbed or simulator is, or can be, appropriate to all agents and environments, and the effort required to develop a new toolkit or testbed from scratch is considerable. There is therefore a strong incentive to reuse existing toolkits and testbeds (or combinations of toolkits and testbeds) for a new problem.

In addition, the computational requirements of simulations of many multi-agent systems far exceed the capabilities of conventional sequential von Neumann computer systems. As computer games become more and more sophisticated, the agents within the game will increasingly be complex systems in their own right (e.g., with sensing, planning, inference, etc. capabilities), requiring considerable computational resources. It is often necessary to simulate many agents to investigate the behaviour of the system as a whole or even the behaviour of a single agent (e.g., when investigating the interaction of a goalkeeper agent and the other agents comprising a football team).

HLA can potentially help to address both of the above problems, as it can facilitate the interoperability of existing (suitably modified) simulators and testbeds which support different agent architectures and environments. Moreover, the component simulations may be distributed on different machines to increase the overall performance of the global simulation.

The third way in which HLA can contribute to games development is as a research platform. A characteristic example is the work that has been undertaken in the context of HLA for Data Distribution Management. Currently, most existing popular game engines and MMOGs are based on the client/server model [29]. The client computers support the individual users while all the *environment* of the game, namely the shared state space of the simulation, is all hosted and maintained by a central server. This arrangement works satisfactorily in current systems, as the shared state is usually small and the client processes interact with each other in a small number of well-defined ways. Even "massively" multiplayer online games, such as World of Warcraft[13] typically offer only a limited degree of interaction between the players. Modern MMOGs are experienced through one of many world "instances" (or "shards"); copies of the game world running on a single server cluster, usually with language localisation for the users to which it is physically close. In areas of the game world (shard) where high levels of interaction are likely (e.g., a quest location), a further level of instancing is often employed, with a new "copy" of the location being spawned for each user or party of users. In addition, non-instanced areas of a shard are typically (statically) partitioned into regions. This rejoining technique is usually non-transparent to the user, with movement from one region to another being explicit. Interaction between users in different shards and/or neighbouring regions is usually limited or non-existent. Within a single region/shard, the number of users actually interacting in real time is small enough that traditional interest management techniques can be applied. These are generally variants of well established techniques based on visual attenuation and server-mediated dead reckoning used in small-scale LAN-based multiplayer games such as Unreal Tournament.

However as games become more complex, with an increasing number of users operating in a complex environment and interacting with it in complex and dynamic patterns (as is the case in MMOGs, for example), the shared state which could, in principle, be accessed or updated by a single player or group of players will increase to a degree that will render the encapsulation of the shared state in a central server a major bottleneck.[14] Distributing the state all across the clients will typically result in frequent all-to-all communication and broadcasting, which is extremely costly and may result in a severe drop of the performance of the underlying system.

This problem has long troubled the LSDS community and has been addressed by means of Interest Management techniques. In the context of the HLA, Interest Management is referred to as Data Distribution Management. The aim of Data Distribution Management (DDM) is to limit the amount of data exchanged between the federates in the simulation. This is achieved through the specification of subscription and publishing regions in routing spaces, with each region implicitly defining an interconnection pattern between federates, and the assignment of multicast groups to these regions. HLA's DDM has been used successfully in

[13] www.blizzard.com

[14] For example, in early 2005 players protesting unfair rules in World of Warcraft gathered together in large numbers on a single server, causing the interest management to fail and the server to crash http://terranova.blogs.com/terra_nova/2005/02/the_right_to_as.html.

virtual environments with thousands of concurrent entities [8], and research efforts to extend or replace the static routing spaces used by HLA, e.g., [32,34], are potentially of interest in game development.

In this paper we focus on the use of the HLA in the development of agents for games (e.g., as NPCs) and (indirectly) on its role as standard for game development. We assume that the federates run asynchronously on different processors, with all synchronisation handled by HLA/RTI. That is the primary motivation behind the work is to obtain the first (RTI as a middleware) and second (agent development and deployment) benefits of using HLA identified above.

3. An overview of SIM_AGENT

SIM_AGENT is an architecture-neutral toolkit originally developed to support the exploration of alternative agent architectures [40,41]. It can be used both as a sequential, centralised, time-driven simulator for multi-agent systems, e.g., to simulate software agents in an Internet environment or physical agents and their environment, and as an agent implementation language, e.g., for software agents or the controller for a physical robot. SIM_AGENT has been used in a variety of research and applied projects, including studies of affective and deliberative control in simple agent systems [39], simulation of tank commanders in military training simulations [4] (which involved integrating SIM_AGENT with an existing real time military simulation), agents which report on activities in collaborative virtual environments and computer games [14,15,31] (which involved integrating SIM_AGENT with the MASSIVE-3 VR system and the Unreal Tournament game engine), and as the implementation language of the GRUE architecture for real-time agents in computer games [16–18].

In SIM_AGENT, an agent consists of a collection of modules representing the capabilities of the agent, e.g., perception, problem-solving, planning, communication, etc. Groups of modules can execute either sequentially or concurrently and with differing resource limits. Each module is implemented as a collection of rules in a high-level rule-based language called POPRULEBASE. However, the rule format is very flexible. Both the conditions and actions of rules can invoke arbitrary low-level capabilities, allowing the construction of hybrid architectures including, for example, symbolic mechanisms communicating with neural nets and modules implemented in procedural languages. The rulesets which implement each module, together with any associated procedural code, constitute the *rulesystem* of an agent. The toolkit can also be used to simulate or implement the agent's environment, and SIM_AGENT provides facilities to populate the agent's environment with user-defined active and passive objects (and other agents).

Execution proceeds in three logical phases: sensing, internal processing and action execution, where the internal processing may include a variety of logically concurrent activities, e.g., perceptual processing, motive generation, planning, decision making, learning, etc. (see Fig. 1).

In the first phase each agent's internal database is updated according to what it senses and any messages sent at the previous cycle. Within a SIM_AGENT execution, each object or agent has both externally visible data and private internal data. The internal data can be thought of as the agent's working memory or *database*. The database is used to hold the agent's model of the environment, its current goals, plans, etc. The internal data is "private" in the sense that other objects or agents have no direct access to it. The external data is data, which conceptually would be externally visible to other objects in the environment, things such as colour, size, shape, etc. For example, if an agent's sensors are able to see all objects within a pre-defined distance, the internal database of the agent would be updated to contain facts which indicate the visible attributes of all objects which are closer than the sensor range.

The next phase involves decision making and action selection. The contents of the agent's database together with the new facts created in phase one are matched against the conditions of the condition-action rules which constitute the agent's rulesystem. It may be that multiple rule conditions are satisfied, or that the same rule is

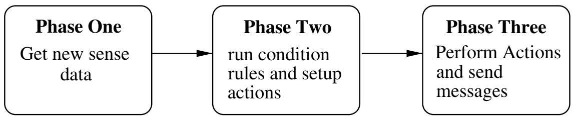
Phase One
Get new sense data
Phase Two
run condition rules and setup actions
Phase Three
Perform Actions and send messages

Fig. 1. Logical structure of a SIM_AGENT cycle.

satisfied multiple times. SIM_AGENT allows the programmer to choose how these rules should run and in what order. For example, a certain program may require that only the first rule matched runs or that every satisfied rule should run. It is also possible to build a list of all the runnable rules and then have a user-defined procedure order this list so that only certain rules (e.g., the more important rules) are run or are run first. These rules will typically cause some internal and/or external action(s) to be performed or message(s) to be sent. Internal actions simply update the agent's database and are performed immediately. External actions change the state of the environment and are queued for execution in the third phase.

The final phase involves sending the messages and performing the actions queued in the previous phase. These external actions will usually cause the object to enter a new state (e.g., change its location) and hence sense new data.

The three logical phases are actually implemented as two scheduler passes for reasons of efficiency. In the first pass, the SIM_AGENT scheduler, `sim_scheduler`, processes the list of agents. For each agent, the scheduler runs its sensors and rulesystem. Any external actions or messages generated by the agent in this pass are queued. In the second pass, the scheduler processes the message and action queues for each agent, transferring the messages to the input message buffers of the recipient(s) for processing at the next cycle, and running the actions to update the environment and/or the publicly visible attributes of the agent.

3.1. An example: SIM_TILEWORLD

In this section we briefly outline the design and implementation of a simple SIM_AGENT application, SIM_TILEWORLD. The Tileworld is a well established testbed for agents [36]. It consists of an environment consisting of tiles, holes and obstacles, and an agent whose goal is to score as many points as possible by pushing tiles to fill in the holes. The environment is dynamic: tiles holes and obstacles appear and disappear at rates controlled by the user. Tileworld has been used to study commitment strategies (i.e., when an agent should abandon its current goal and replan) and in comparisons of reactive and deliberative agent architectures. SIM_TILEWORLD is an implementation of a single-agent Tileworld,[15] which consists of an environment and one agent (see Fig. 2).

SIM_AGENT provides a library of classes and methods for implementing agent simulations. The toolkit is implemented in Pop-11, an AI programming language similar to Lisp, but with an Algol-like syntax. Pop-11 supports object-oriented development via the OBJECTCLASS library, which provides classes, methods, multiple inheritance, and generic functions.[16] SIM_AGENT defines two basic classes, `sim_object` and `sim_agent`, which can be extended (subclassed) to give the objects and agents required for a particular simulation scenario. The `sim_object` class is the foundation of all SIM_AGENT simulations: it provides slots (fields or instance variables) for the object's name, internal database, sensors, and rulesystem together with slots which determine how often the object will be run at each timestep, how many processing cycles it will be allocated on each pass and so on. The `sim_agent` class is a subclass of `sim_object` which provides simple message based communication primitives. SIM_AGENT assumes that all the objects in a simulation will be subclasses of `sim_object` or `sim_agent`.

For the SIM_TILEWORLD example three subclasses of `sim_object` were defined to represent holes, tiles and obstacles, and two subclasses of `sim_agent` to represent the environment and the agent. The subclasses define additional slots to hold the relevant simulation attributes, e.g., the position of tiles, holes and obstacles, the types of tiles, the depth of holes, the tiles being carried by the agent, etc. By convention, external data is held in slots, while internal data (such as which hole the agent intends to fill next) is held in the agent's database.

The system consists of two active objects (the environment and the agent) and a variable number of passive objects (the tiles, holes and obstacles). At startup, instances of the environment and agent classes are created and passed to the scheduler. At each cycle the scheduler runs the environment agent to update the agent's environment. In SIM_TILEWORLD the environment agent has a simple rulesystem with no conditions (i.e., it runs

[15] Multi-Agent Tileworld(s) do exist [13].
[16] OBJECTCLASS shares many features of the Common Lisp Object System (CLOS).

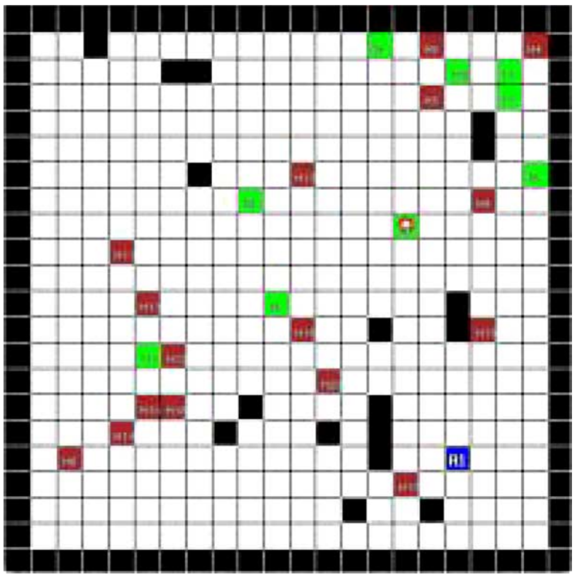

Fig. 2. A screen shot of SIM_TILEWORLD.

every cycle) which causes tiles, obstacles and holes to be created and deleted according to user-defined probabilities. The scheduler then runs the agent, which perceives the new environment and updates its internal database with the new sense data. The sensors of an agent are defined by a list of procedures and methods (conventionally `sim_sense_agent` methods for the classes involved in the simulation, but any procedures can be used). Any object in the simulation objects list which "satisfies" these procedures or methods (in the sense of being an appropriate method for the object class in the case of methods or returning sensor data in the case of procedures) is considered "sensed" by the agent. The agent then runs all rules, which have their conditions satisfied (no ordering of the rules is performed). Some of the rules may queue external actions (e.g., moving to or pushing a tile), which are performed in the second pass of the scheduler at this cycle. This completes the cycle and the process is repeated.

4. Distributing a SIM_AGENT application

There are two distinct ways in which SIM_AGENT might use the facilities offered by the HLA. The first, which we call the *distribution* of SIM_AGENT, involves using HLA to distribute the agents and objects comprising a SIM_AGENT simulation across a number of federates. The second, which we call *inter-operation*, involves using HLA to integrate SIM_AGENT with other simulators. In this paper we concentrate on the former, namely distributing an existing SIM_AGENT implementation using SIM_TILEWORLD as an example.

In the remainder of this section, we outline the role of the HLA services in distributing SIM_TILEWORLD. (We do not consider Federation Management for a distributed SIM_AGENT federation as this is similar to other HLA federations.)

4.1. Object and Declaration Management

Object and Declaration Management enable the federates to share data, providing services for registering, updating, deleting, discovering, reflecting and removing objects as well as subscribing to and publishing data.

Based on the SIM_TILEWORLD implementation outlined in Section 3.1, we chose to split the implementation into two federates, corresponding to the Tileworld agent and the Tileworld environment, respectively. Fig. 3 depicts the FOM for the HLA SIM_TILEWORLD example. Two main subclasses are defined, namely Agent and Object, with the Object class having Tiles, Holes and Obstacles as subclasses.

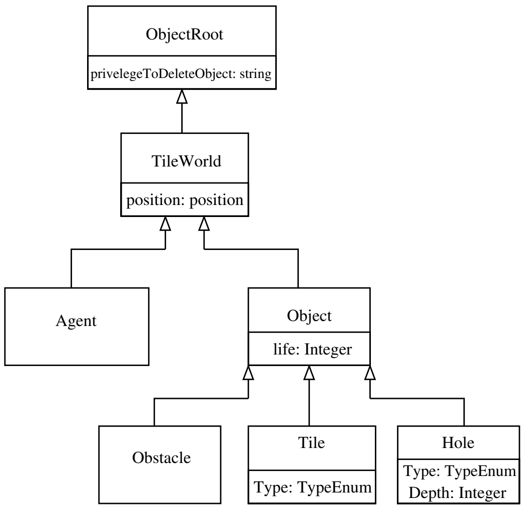

Fig. 3. An example FOM for SIM_TILEWORLD.

In the current implementation of SIM_TILEWORLD, the communication between the agent and the environment federates is performed through the creation and deletion of object instances, and the updating of their attributes. Thus, no interactions are specified in the FOM.

The agent object is included in the FOM as certain attributes of the agent may be accessed by other federates. For example in a multi-agent implementation of SIM_TILEWORLD, the agents would need to know the position of other agents in the environment (for sensing). Table 1 illustrates the corresponding object class publications and subscriptions.

The attribute *position* of the Agent class is published by the Agent federate as this federate updates the position of the Agent. The same applies to the *carriedTiles* attribute for the Agent class. The *position* attribute for the Tile class is published by both the Environment and the Agent federate. This is because initially, when the

Table 1
Object class publications and subscriptions in Tileworld Federation

Object	Federate	
	Environment	Agent
Agent		
PrivelegeToDeleteObject	Publish	Publish
Position	Subscribe	Publish
CarriedTiles	Subscribe	Publish
Tile		
PrivelegeToDeleteObject	Publish	Publish
Position	Publish	Publish
Life	Publish	Subscribe
Type	Publish	Subscribe
Hole		
PrivelegeToDeleteObject	Publish	Publish
Position	Publish	Subscribe
Life	Publish	Subscribe
Type	Publish	Subscribe
Depth	Publish	Publish
Obstacle		
PrivelegeToDeleteObject	Publish	Publish
Position	Publish	Subscribe
Life	Publish	Subscribe

tile is created, the Environment federate will set the Tile's position. However, when the Agent federate picks up the Tile it will start to update the *position* attribute. Similarly, the *depth* attribute of the Hole class will be updated when the agent places a tile in a hole. Initially, when the hole is created, the Environment federate will set the depth of the hole. As the Agent federate places tiles in the hole it will change the *depth* attribute. *privilegeToDelete* is a special attribute, predefined by the HLA. A federate which owns the *privilegeToDelete* attribute of an object can delete the object from the simulation (ownership management is discussed in more detail below). The other attributes are largely self-explanatory.

4.2. Ownership Management

HLA rules require federates to own attribute instances before they can update their value. This ensures that at any point in time only one federate may update an attribute and is achieved via Ownership Management services. In HLA_AGENT we use ownership management to manage conflicts between actions proposed by agents executing on different federates. For example, two (or more) agents may try to push the same tile at the same timestep. Once the tile has been moved by one agent, subsequent moves should become invalid, as the tile is no longer at the position at which it was initially perceived. We therefore extend current practice in SIM_AGENT and require that attribute updates be *mutually exclusive*. A mutually exclusive attribute is one, which cannot be updated twice in the same cycle. To ensure that attributes are mutually exclusive, we require that attribute ownership is only transferred at most once per simulation cycle, and that a federate relinquishes ownership of an attribute only if it has not already been updated at the current cycle. (If multiple attributes are updated by the same agent action, we require that federates acquire ownership of the attributes in a fixed order to avoid deadlock.) For example, if two agents running on different federates try to move a given tile at the same cycle, whichever agent's action is processed first will acquire ownership of the tile and succeed, while the other will be denied ownership and fail.

4.3. Time Management

Time Management services in the HLA perform two main roles, namely, coordinating the advancement of logical time in federates and controlling delivery of time-stamped events to prevent federates receiving "old" events, i.e., events with logical time less than the federate's current logical time.

SIM_AGENT is a centralised, time-driven system in which execution advances in timesteps, referred to as cycles. As explained in Section 3, at the end of a cycle a series of actions may change some aspects of the agents or environment. These changes are then perceived by all agents at the beginning of the next cycle. It therefore makes sense for the Federation to synchronise at the end (or beginning) or each cycle. This can be achieved by making the all federates time-regulating and time-constrained. This ensures that the federates will proceed in a timestep fashion, alternating between performing their external actions and perceiving changes.

4.4. Data Distribution Management

As the sensors used in SIM_AGENT can be restricted to a certain range, DDM is not required for the early stages of the integration exercise described in this paper. Federates will send information to one another based on the publish-subscribe information provided in Table 1 and it will be up to the individual agents to "sense" and filter the relevant information. However DDM is considered crucial for the efficient distributed execution of large agent-based systems and further work will seek to incorporate DDM [32].

5. Extending the SIM_AGENT toolkit

In this section we briefly sketch the extensions necessary to the SIM_AGENT toolkit to allow an existing SIM_AGENT system to be distributed using the HLA. We assume that we have an existing SIM_AGENT application (e.g., SIM_TILEWORLD) that we want to distribute by placing disjoint subsets of the objects and agents comprising the system on different federates. Our aim is to make this distribution transparent to the SIM_AGENT low level scheduler code and agents and objects comprising the application.

The general picture is as follows:

- we extend SIM_AGENT to hold additional data about the federation and the federate in which the SIM_AGENT process is running, e.g., the FOM, the agents to be executed by this federate, proxies for agents executed by other federates, RTI bookkeeping information, etc.;
- we extend the SIM_AGENT base classes so that updates to publicly visible attributes by agents executed by this federate (i.e., updates to public data corresponding to attributes published by this federate) are propagated to other federates;
- we need to add some code to connect to the RTI and initialise the federate's data structures; and
- we have to modify the SIM_AGENT scheduler so that only those agents executed by this federate are actually run at each cycle. We also have to handle object discovery, propagation of object attributes, and synchronisation at each cycle.

SIM_AGENT has the ability to make simple calls to functions written in C. We have therefore interfaced SIM_AGENT with the C++ version of the RTI. RTI Ambassador and Federate Ambassador methods required for the implementation have been given appropriate C wrappers. RTI calls can therefore be made from SIM_AGENT as though we have an implementation of the RTI written in Pop-11.

In what follows, we briefly describe each of these changes in more detail. In Section 5.4, we outline the operation of the modified scheduler over a single simulation cycle (see Fig. 1). It turns out that the changes to the scheduler are confined to the first (sensing) and third (action) phases. The second phase involves only the internal operation of the agent and updates to the agent's private database. Such updates are typically invisible to other agents, and can be ignored for the purposes of distribution.[17]

5.1. Representing the federation

At each federate, we partition the objects managed by the federate into *local objects* and *proxy objects*. Local objects are instances of the standard `sim_object` and `sim_agent` classes and their subclasses which are being run by SIM_AGENT on this federate. Proxy objects represent those local objects being executed by other federates which this federate knows about (e.g., via object discovery) and are used as targets for the sensors and actions of local objects. Note that a federate may not know about all the objects in the simulation (and in the limit case, none of the federates knows about all the objects in the simulation).

We define two new classes, `HLA_federate` and `HLA_object`. `HLA_federate` contains slots to hold the relevant data for a SIM_AGENT process running on a particular federate, e.g., the scheduler and simulation objects lists, the FOM for the federation, a handle to the local RTI ambassador, etc. `HLA_object` holds RTI bookkeeping information for each object in the simulation, e.g., the unique RTI identifier for the object which is shared by all the federates in the simulation. All instances of the existing SIM_AGENT classes (i.e., `sim_object` and `sim_agent` and their subclasses) need to hold this bookkeeping information. We can accomplish this in a straightforward way by declaring `sim_object` to be a subclass of `HLA_object`—in OBJECTCLASS there is no root class from which all other classes descend, and a new class definition can "adopt" an existing class and its subclasses.

We assume that all the class definitions for the objects and agents comprising the simulation are available to all federates, and that all federates can create instances of these classes to represent agents being simulated by the federate and as proxies for agents being simulated by other federates.

5.2. Propagating the effects of actions

As stated in Section 3, agents can perform two different types of actions: internal actions which update the agent's private database, and external actions which update publicly visible attributes of an object. Internal actions only affect the state of the agent and are processed immediately, since the effects of the action (i.e.,

[17] We have not considered the distribution of the components of a single agent across multiple federates.

changes to the contents of the agent's database or working memory) typically form part of a larger decision making process within the agent. However, in the case of external actions, it is necessary to propagate the update to other federates which subscribe to the attribute. This involves calls to the RTI to acquire ownership of the attribute (if it is not currently owned by this federate) and to do the update. The aim is to make these additional calls transparent to the Pop-11 code that implements the action.

The situation is complicated by the fact that external actions are usually queued for execution at the end of the current cycle. This avoids the agents "seeing" different states of the environment during the first pass of the scheduler, and means that the order in which agents are processed by the scheduler doesn't matter in situations where two agents attempt to update the same attribute. The problem of detecting action conflicts is intractable in general, and it is up to the simulation developer to design a SIM_AGENT simulation so as to avoid conflicts. One way to do this is to arrange for each action to check that its preconditions (i.e., the state the environment was in when the action was selected) still hold before performing the update and otherwise abort the action. However, this is not feasible in a distributed setting, since any attribute updates resulting from the actions of agents simulated by other federates are not propagated until the end of the cycle. We therefore extend the current capabilities of SIM_AGENT by allowing attributes to be declared *mutually exclusive*. A mutually exclusive attribute is one which cannot be updated twice in the same cycle. For example, we may wish to require that the position of an object can only change once in any given cycle. This extension does not solve the ramification problem, it simply provides some additional tools for a simulation developer to manage inconsistent updates.

Attributes in SIM_AGENT are represented by slot values. Each slot has two predefined methods, an *accessor*, which returns the current value, and an *updater*, which sets the value. OBJECTCLASS provides *method wrappers*, methods which extend or even replace the functionality of existing methods. A method wrapper is a closure around a particular method, which can modify the behaviour of that method. Method wrappers can be used to "intercept" calls to the slot updater methods and propagate the new value to other federates by making the appropriate RTI calls.

For each attribute of each class in the FOM that is published by this federate, we define a method wrapper for the slot updater, e.g., the method wrapper for the position slot of the tile object might look like:

```
define :wrapper updaterof position(p, o:sim_tile, upd_p);
  ;;; acquire ownership of this attribute
  ;;; call RTI_update_attribute_values(HLA_identifier(o), [position, p], time)
  ;;; do the local update
  upd_p(p, o);
enddefine;
```

Note that ownership is acquired but not released by the method wrapper. Position is a mutually exclusive attribute, since one agent changing the position of a tile violates the precondition for any other action (by the same or another agent), which attempts to move the tile. If two agents running on different federates try to move a given tile at the same cycle, whichever agent's action is processed first will acquire ownership of the tile and succeed, while the other agent's action will be denied ownership and fail. (Two agents running on the same federate could update the attribute, but in this case, we can use checks on the preconditions of the actions, since the updates are mirrored locally.) Updates to slots not corresponding to published attributes are assumed to be local to the federate and are not propagated.

5.3. System startup

We also need to add some additional initialisation code which loads the FOM, federation description and the parameters for this federate, locates or starts an RTI Ambassador, and creates an `HLA_federate` object for this federate.

When a SIM_AGENT federate starts up, it creates instances of all the objects and agents that are to be run locally, and notifies the RTI of their creation. The RTI creates a unique identifier for each local object in the simulation, and notifies any federates which subscribe to an attribute of the object of its existence via "object discovery" callbacks to the federate's Federate Ambassador. Any objects created or discovered during initialisation which are not to be executed on this federate are assigned proxies, and the values of subscribed slots are initialised when the federate executing the object updates its attributes.

Once the federate is initialised, the main scheduler procedure, `sim_scheduler`, is started and begins to repeatedly execute the main SIM_AGENT cycle.

5.4. A cycle of SIM_AGENT in HLA

The main scheduler cycle proceeds as follows:

(1) Wait for synchronisation with other federates.
(2) For each object or agent in the scheduler list which is not a proxy:
   (a) Run the agent's sensors on each of the objects in the scheduler list. By convention, sensor procedures only access the publicly available data held in the slots of an object, updated in step 5.
   (b) Transfer messages from other agents from the input message buffer into the agent's database.
   (c) Run the agent's rulesystem to update the agent's internal database and determine which actions the agent will perform at this cycle (if any). This may update the agent's internal database, e.g., with information about the state of the environment at this cycle or the currently selected action(s), etc.
(3) Once all the agents have been run on this cycle, the scheduler processes the message and action queues for each agent, transfers outgoing messages to the input message buffers of the recipient(s) for processing at the next cycle, and runs the actions to update objects in the environment and/or the publicly visible attributes of the agent. This can trigger further calls to the RTI to propagate new values.
(4) We then process the object discovery and deletion callbacks for this cycle. For all new objects created by other federates at this cycle we create a proxy instance of the appropriate `sim_object` subclass. If other federates have deleted objects, we delete our local proxies.
(5) Finally, we process the attribute update callbacks for this cycle, and use this information to update the slots of the local objects and proxies simulated at this federate (e.g., if an agent on another federate moves a tile simulated on this federate). The update wrappers are disabled during slot update as these would otherwise trigger a rebroadcast of the attribute updates to the RTI.
(6) Repeat.

The additional generic code, e.g., the definitions of `HLA_federate` and `HLA_object`, extensions to `sim_scheduler`, etc., is loaded as an additional library. The code which is specific to a particular simulation (essentially the method wrappers) is "boilerplate" code which is generated automatically from the FOM and information about which classes in the FOM a federate publishes and which attributes it subscribes to. Different federates therefore generate different code, depending on which attributes they are interested in. However, since Pop-11 uses incremental compilation, none of this boilerplate code needs to be defined in advance: it can be generated on the fly at startup.

6. Performance

To evaluate the performance of HLA_AGENT we implemented a version of SIM_TILEWORLD using HLA_AGENT and compared its performance with the original SIM_AGENT version. The HLA/RTI overhead can be broken down into two parts: the overhead of trapping, e.g., object creation and slot update in SIM_AGENT and servicing the resulting RTI calls and callbacks on the local federate; and the overhead inherent in the RTI exec itself. The latter is inherent in the inter-operation of distributed HLA federates, and depends on the federation topology, the extent to which Interest Management services are used, the particular implementation of the RTI exec chosen, and the bandwidth and latency of the network(s) connecting the federates, and is not considered here. See [26] for a performance analysis of the effects of distributing an HLA_AGENT application over the nodes in a homogeneous cluster.

The hardware platform used for our experiments is a Linux cluster, comprising 64 2.6 GHz Xeon processors each with 512KB cache (32 dual nodes) interconnected by a standard 100Mbps fast Ethernet switch. Our test environment is a Tileworld 50 units by 50 units in size with an object creation probability (for tiles, holes

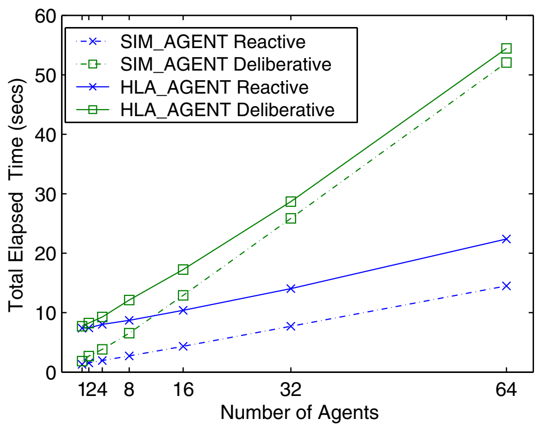

Fig. 4. Total elapsed times for 1–64 Reactive and Deliberative agents in SIM_AGENT and HLA_AGENT (single federate).

and obstacles) of 1.0 and an average object lifetime of 100 cycles, i.e., objects are created and destroyed at approximately the same rate. The Tileworld initially contains 100 tiles, 100 holes and 100 obstacles and the number of agents in the Tileworld ranges from 1 to 64. In the SIM_TILEWORLD federation, the environment was simulated by a single federate while the agents were distributed in one or more federates over the nodes of the cluster.[18] The results obtained represent averages over 5 runs of 100 SIM_AGENT cycles.

We would expect the local HLA overhead to vary with CPU load. With a single federate, the HLA adds a more or less fixed overhead to specific operations (simulation startup, the creation and deletion of objects, updating attributes and synchronisation). As the total CPU required by the application increases, the additional time required for these operations with HLA becomes less significant. We therefore investigated two scenarios: simple reactive Tileworld agents with minimal CPU requirements and deliberative Tileworld agents which use an A^* based planner to plan optimal routes to tiles and holes in their environment [30]. The planner was modified to incorporate a variable "deliberation penalty" for each plan generated and the agents replanned whenever the region of the Tileworld they could sense was changed by the actions of another agent or by the environment itself. In the experiments reported below the deliberation penalty was arbitrarily set at 10 ms per plan.

Fig. 4 shows the total elapsed time when executing 1, 2, 4, 8, 16, 32 and 64 reactive and deliberative SIM_TILEWORLD agents and their environment on a single cluster node using SIM_AGENT and a single HLA_AGENT federate. This gives an indication of the overhead inherent in the HLA_AGENT library itself independent of any communication overhead entailed by the RTI. As can be seen, the HLA overhead diminishes with increasing CPU load. For example, with 64 reactive agents the HLA introduces a significant overhead. In SIM_AGENT, the average elapsed time is 14.5 seconds compared to 22.4 s with HLA_AGENT, giving a total overhead for the HLA of approximately 54.5%. For agents, which intrinsically require more CPU, the overhead is proportionately smaller. With 64 deliberative agents, the average elapsed time is 52.08 s with SIM_AGENT and 54.45 s with HLA_AGENT, giving a total overhead for the HLA of just 4.6%.

7. Summary

As the worlds of computer games and Large Scale Distributed Simulation increasingly overlap, the need for a universal standard and architecture for the composition and reuse of components such as the HLA will grow. In this paper, we have addressed a central issue for HLA-based games, namely the development of HLA-compliant game agents. We described HLA_AGENT, an HLA-compliant version of the SIM_AGENT toolkit for building cognitively rich agents, and showed how the HLA can be used to distribute an existing SIM_AGENT application with different agents being executed by different federates. We briefly outlined the changes necessary to the

[18] For our experiments, only one processor was used in each node.

SIM_AGENT toolkit to allow integration with the HLA. It turns out that, given certain reasonable assumptions, all necessary code can be generated automatically from the FOM and information about which attributes the federate publishes and subscribes to. The integration is transparent in the sense that the existing SIM_AGENT code runs unmodified and the agents are unaware that other parts of the simulation are running remotely.

The design study presented above highlights many of the issues that are central to any distributed execution of agent-based systems. Future work will evaluate the HLA for the execution of agent based systems, using more complex testbeds. One key problem is the efficient propagation of updates to the shared environment. HLA_AGENT currently makes no use of the DDM services provided by the RTI. This is an area of current work [32].

Another area for future work is *inter-operation*, using HLA to integrate SIM_AGENT with HLA-compliant game engines and other HLA-compliant agent toolkits (for example, [42]). HLA_AGENT enables games developers to design agents for games and integrate (for development or playing) these agents with any other HLA compliant software. For example, a developer producing player agents for a football management game in an HLA compliant SIM_AGENT toolkit could be tested their agents in conjunction with agents and environments developed by other developers, using different languages or platforms. Initial investigation suggests that the additional changes to SIM_AGENT required to support inter-operation are relatively straightforward, and the key issue is one of specifying interfaces for sensor and action data. We are currently in the process of developing a set of inter-operability guidelines for SIM_AGENT applications.

Acknowledgement

We would like to thank Rob Minson for his comments on interest management in MMOGs.

References

[1] J. Anderson, A generic distributed simulation system for intelligent agent design and evaluation, in: H.S. Sarjoughian, F.E. Cellier, M.M. Marefat, J.W. Rozenblit (Eds.), Proceedings of the Tenth Conference on AI, Simulation and Planning, AIS-2000, Society for Computer Simulation International, 2000, pp. 36–44.

[2] S.M. Atkin, D.L. Westbrook, P.R. Cohen, G.D. Jorstad, AFS and HAC: Domain general agent simulation and control, in: J. Baxter, B. Logan (Eds.), Software Tools for Developing Agents: Papers from the 1998 Workshop, AAAI Press, Menlo Park, CA, 1998, pp. 89–96, Technical Report WS-98-10.

[3] J. Bates, A.B. Loyall, W.S. Reilly, Broad agents, in: Proceedings of the AAAI Spring Symposium on Integrated Intelligent Architectures, 1991, Sigart Bulletin 2 (4) (1991) 38–40.

[4] J. Baxter, R.T. Hepplewhite, Agents in tank battle simulations, Communications of the ACM 42 (3) (1999) 74–75.

[5] C. Bererton, State estimation for game AI using particle filters, in: D. Fu, S. Henke, J. Orkin (Eds.), Challenges in Game Artificial Intelligence: Papers from the 2004 AAAI Workshop, AAAI Press, Menlo Park, CA, 2004, pp. 36–40, Technical Report WS-04-04.

[6] J. Bradshaw (Ed.), Software Agents, AAAI Press, Menlo Park, CA, 1997.

[7] W. Cai, P. Xavier, S.J. Turner, B.-S. Lee, A scalable architecture for supporting interactive games on the Internet, in: PADS '02: Proceedings of the Sixteenth Workshop on Parallel and Distributed Simulation, IEEE Computer Society, Washington, DC, USA, 2002, pp. 60–67.

[8] J.O. Calvin, C.J. Chiang, S.M. McGarry, S.J. Rak, D.J. van Hook, Design, implementation and performance of the STOW RTI prototype (RTI-s), in: Proceedings of the 1997 Spring Simulation Interoperability Workshop, 1997, pp. 145–154.

[9] M. Cavazza, S. Bandi, I. Palmer, Situated AI in video games: integrating NLP, path planning and 3D animation, in: Papers from the AAAI 1999 Spring Symposium on Artificial Intelligence and Computer Games, AAAI Press, Menlo Park, CA, 1999, pp. 6–12, Technical Report SS-99-02.

[10] DMSO, High Level Architecture interface specification, version 1.3, 1998.

[11] E.H. Durfee, T.A. Montgomery, MICE: A flexible testbed for intelligent coordination experiements, in: Proceedings of the Ninth Distributed Artificial Intelligence Workshop, 1989, pp. 25–40.

[12] H. Engum, J.V. Iversen, Ø. Rein, Zereal: A semi-realistic simulator of massively multiplayer games, Technical report, Department of Computer and Information Science, NTNU, Trondheim, Norway, 2002.

[13] E. Ephrati, M. Pollack, S. Ur, Deriving multi-agent coordination through filtering strategies, in: C. Mellish (Ed.), Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Francisco, 1995, pp. 679–685.

[14] D. Fielding, M. Fraser, B. Logan, S. Benford, Extending game participation with embodied reporting agents, in: Proceedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology (ACE 2004), ACM Press, New York, USA, 2004, pp. 100–108.

[15] D. Fielding, M. Fraser, B. Logan, S. Benford, Reporters, editors and presenters: Using embodied agents to report on online computer games, in: N.R. Jennings, C. Sierra, L. Sonenberg, M. Tambe (Eds.), Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004), vol. 3, IEEE, New York, 2004, pp. 1530–1531.

[16] E. Gordon, B. Logan, A goal processing architecture for game agents, in: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2003), ACM Press, Melbourne, 2003, pp. 998–999.
[17] E. Gordon, B. Logan, Game over: You have been beaten by a GRUE, in: D. Fu, S. Henke, J. Orkin (Eds.), Challenges in Game Artificial Intelligence: Papers from the 2004 AAAI Workshop, AAAI Press, Menlo Park, CA, 2004, pp. 16–21, Technical Report WS-04-04.
[18] E. Gordon, B. Logan, Managing goals and resources in dynamic environments, in: D.N. Davis (Ed.), Visions of Mind: Architectures for Cognition and Affect, Idea Group, 2005, pp. 225–253 (Chapter 11).
[19] IEEE, IEEE Standard for modeling and simulation (M&S) High Level Architecture (HLA)—Framework and rules, IEEE, IEEE Standard No. 1516-2000, 2000.
[20] N.R. Jennings, M. Wooldridge, Applications of intelligent agents, in: N.R. Jennings, M. Wooldridge (Eds.), Agent Technology: Foundations, Applications, Markets, Springer-Verlag, Berlin, 1998, pp. 3–28.
[21] G.A. Kaminka, M.M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A.N. Marshall, A. Scholer, S. Tejada, GameBots: A flexible test bed for multiagent team research, Communications of the ACM 45 (1) (2002) 43–45.
[22] A. Khoo, R. Zubek, Applying inexpensive AI techniques to computer games, IEEE Intelligent Systems 17 (4) (2002) 48–53.
[23] F. Kuhl, R. Weatherly, J. Dahmann, Creating Computer Simulation Systems: An Introduction to the High Level Architecture, Prentice Hall, Englewood Cliffs, NJ, 1999.
[24] J.E. Laird, M. van Lent, Human-level AI's killer application: Interactive computer games, AI Magazine 22 (2) (2001) 15–26.
[25] J.E. Laird, It knows what you're going to do: Adding anticipation to a QUAKEBOT, in: Papers from the AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment, AAAI Press, Menlo Park, CA, 2000, pp. 41–50, Technical Report SS-00-02.
[26] M. Lees, B. Logan, T. Oguara, G. Theodoropoulos, Simulating agent-based systems with HLA: The case of SIM_AGENT—Part II, in: Proceedings of the 2003 European Simulation Interoperability Workshop, European Office of Aerospace R&D, Simulation Interoperability Standards Organisation and Society for Computer Simulation International, 2003.
[27] M. Lees, B. Logan, T. Oguara, G. Theodoropoulos, HLA_AGENT: Distributed simulation of agent-based systems with HLA, in: Proceedings of the International Conference on Computational Science (ICCS'04), LNCS, Springer, Krakow, Poland, 2004, pp. 907–915.
[28] M. Lees, B. Logan, G. Theodoropoulos, Simulating agent-based systems with HLA: The case of SIM_AGENT, in: Proceedings of the 2002 European Simulation Interoperability Workshop, European Office of Aerospace R&D, Simulation Interoperability Standards Organisation and Society for Computer Simulation International, 2002, pp. 285–293.
[29] M. Lewis, J. Jacobson, Game engines in scientific research, Communications of the ACM 45 (1) (2002) 27–31.
[30] B. Logan, N. Alechina, A^* with bounded costs, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI-98, AAAI, AAAI Press/MIT Press, Menlo Park, CA/Cambridge MA, 1998, pp. 444–449.
[31] B. Logan, M. Fraser, D. Fielding, S. Benford, C. Greenhalgh, P. Herrero, Keeping in touch: Agents reporting from collaborative virtual environments, in: K. Forbus, M.S. El-Nasr (Eds.), Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Symposium, AAAI Press, Menlo Park, CA, 2002, pp. 62–68, Technical Report SS-02-01.
[32] B. Logan, G. Theodoropoulos, The distributed simulation of multi-agent systems, Proceedings of the IEEE 89 (2) (2001) 174–186.
[33] R. Minson, G. Theodoropoulos, Distributing RePast agent-based simulations with HLA, in: Proceedings of the 2004 European Simulation Interoperability Workshop, Simulation Interoperability Standards Organisation and Society for Computer Simulation International, Edinburgh, 2004, in press.
[34] K.L. Morse, M. Zyda, On line multicast grouping for dynamic data distribution management, in: Proceedings of the 2000 Fall Simulation Interoperability Workshop, Paper No. 00F-SIW-052, 2000.
[35] J. Orkin, Symbolic representation of game world state: Toward real-time planning in games, in: D. Fu, S. Henke, J. Orkin (Eds.), Challenges in Game Artificial Intelligence: Papers from the 2004 AAAI Workshop, AAAI Press, Menlo Park, CA, 2004, pp. 26–30, Technical Report WS-04-04.
[36] M.E. Pollack, M. Ringuette, Introducing the Tileworld: Experimentally evaluating agent architectures, in: National Conference on Artificial Intelligence, 1990, pp. 183–189.
[37] P. Quax, P. Monsieurs, T. Jehaes, W. Lamotte, Using autonomous avatars to simulate a large-scale multi-user networked virtual environment, in: VRCAI'04: Proceedings of the 2004 ACM SIGGRAPH International conference on Virtual Reality Continuum and its Applications in Industry, ACM Press, New York, USA, 2004, pp. 88–94.
[38] B. Schattenberg, A.M. Uhrmacher, Planning agents in JAMES, Proceedings of the IEEE 89 (2) (2001) 158–173.
[39] M. Scheutz, B. Logan, Affective vs. deliberative agent control, in: Proceedings of the AISB'01 Symposium on Emotion, Cognition and Affective Computing, AISB, The Society for the Study of Artificial Intelligence and the Simulation of Behaviour, 2001, pp. 1–10.
[40] A. Sloman, B. Logan, Building cognitively rich agents using the SIM_AGENT toolkit, Communications of the ACM 42 (3) (1999) 71–77.
[41] A. Sloman, R. Poli, SIM_AGENT: A toolkit for exploring agent designs, in: M. Wooldridge, J. Mueller, M. Tambe (Eds.), Intelligent Agents II: Agent Theories Architectures and Languages (ATAL-95), Springer-Verlag, Berlin, 1996, pp. 392–407.
[42] L. Wang, S.J. Turner, F. Wang, Interest management in agent-based distributed simulations, in: 7th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT 2003), 23-25 October 2003, Delft, The Netherlands, IEEE Computer Society, Washington, DC, USA, 2003, pp. 20–29.
[43] B. Wetzel, Step one: Document the problem, in: D. Fu, S. Henke, J. Orkin (Eds.), Challenges in Game Artificial Intelligence: Papers from the 2004 AAAI Workshop, AAAI Press, Menlo Park, CA, 2004, pp. 11–15, Technical Report WS-04-04.
[44] M. Wooldridge, N.R. Jennings, Intelligent agents: theory and practice, Knowledge Engineering Review 10 (2) (1995).