Simulating Agent-Based Systems with HLA: The case of SIM_AGENT

Michael Lees

Brian Logan
School of Computer Science and IT
University of Nottingham
Nottingham NG8 1BB, UK
{mhl|bsl}@cs.nott.ac.uk

Georgios Theodoropoulos
School of Computer Science
University of Birmingham
Birmingham, B15 2TT, UK
gkt@cs.bham.ac.uk

KEYWORDS

Distributed simulation, agent-based systems, interest management, load balancing.

ABSTRACT ***In this paper we discuss the distributed simulation of agent-based systems in*** **HLA*. Using the*** **SIM_AGENT** ***toolkit and the Tileworld scenario as an example, we present a design proposal showing how the*** **HLA** ***can be used to distribute a*** **SIM_AGENT** ***simulation with different agents being simulated by different federates. 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 simulation are running remotely.***

1 Introduction

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. The environment of an agent is that part of the world or computational system 'inhabited' by the agent. Agents are embedded in an environment. The *environment* may contain other agents whose environments are disjoint with or only partially overlap with the environment of a given agent. Agent-based systems offer advantages when independently developed components must interoperate in a heterogeneous environment, e.g. the internet, and agent-based systems are increasingly being applied in a wide range of areas including telecommunications, business process modelling, computer games, control of mobile robots and military simulations.

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, allowing the agent designer to learn more about the behaviour of a system or to investigate the implications of alternative agent architectures, and the agent researcher to probe the relationships between agent architectures, environments and behaviour. The use of simulation allows a degree of control over experimental conditions and facilitates the replication of results in a way that is difficult or impossible with a prototype or fielded system, allowing the agent designer or researcher to focus on key aspects of the system.

Simulation has traditionally played an important role in agent research and a wide range of simulators and testbeds have been developed to support the design and analysis of agent architectures and systems (Durfee & Montgomery 1989, Pollack & Ringuette 1990, Atkin, Westbrook, Cohen & Jorstad. 1998, Anderson 2000, Schattenberg & Uhrmacher 2000). One such simulator is SIM_AGENT, a toolkit to support research in architectures for intelligent, human-like agents (Sloman & Poli

1996).[1]

However no one testbed is, or can be, appropriate to all agents and environments. Moreover, even if a suitable simulator or testbed can be found for a given problem, the assumptions made by the simulator can make it difficult to generalise the results obtained, and demonstrating that a particular result holds across a range of agent architectures and environments often requires using a number of different systems.

Furthermore, the computational requirements of simulations of many multi-agent systems far exceed the capabilities of conventional sequential von Neumann computer systems. Each agent is typically a complex system in its own right (e.g., with sensing, planning, inference etc. capabilities), requiring considerable computational resources, and many agents may be required to investigate the behaviour of the system as a whole or even the behaviour of a single agent. A solution to this problem is distributed simulation.

The High Level Architecture (HLA), the simulator interoperability framework developed by the US DoD DMSO[2], can help to address both of the above problems, as it can allow the interoperability of various 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. In this paper, we investigate the feasibility of interfacing SIM_AGENT to HLA to support the distributed simulation of agent-based systems. In section 2 we briefly describe the SIM_AGENT toolkit and illustrate its application in a simple Tileworld scenario. In section 3 we outline how the HLA can be used to distribute an existing SIM_AGENT simulation with different agents being simulated by different federates. In section 4 we sketch the changes necessary to the 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 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 simulation are running remotely. We conclude with a brief description of future work.

2 An overview of SIM_AGENT

SIM_AGENT is an architecture-neutral toolkit originally developed to support the exploration of alternative agent architectures (Sloman & Poli 1996, Sloman & Logan 1999). 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 (Scheutz & Logan 2001), agents which report on activities in collaborative virtual environments (Logan, Fraser, Fielding, Benford, Greenhalgh & Herrero 2002) (which involved integrating SIM_AGENT with the MASSIVE-3 VR system), and simulation of tank commanders in military training simulations (Baxter & Hepplewhite 1999) (for this project, SIM_AGENT was integrated with an existing real time military simulation).

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 the agent's environment. SIM_AGENT provides facilities to populate the agent's environment with user-defined active and passive objects (and other agents).

Simulation 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 Figure 1).

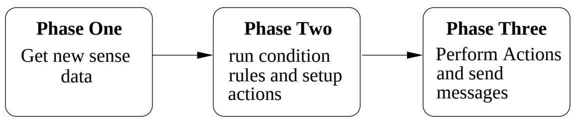
Phase One
Get new sense data
Phase Two
run condition rules and setup actions
Phase Three
Perform Actions and send messages

Figure 1: Logical structure of a simulation cycle

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 simulation, 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

[1]See http://www.cs.bham.ac.uk/∼axs/cog_affect/sim_agent.html
[2]See http://www.dmso.mil/hla

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 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 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.

2.1 An example: SIM_TILEWORLD

In this section we briefly outline the design and implementation of a simple SIM_AGENT simulation, SIM_TILEWORLD. The Tileworld is a well established testbed for agents (Pollack & Ringuette 1990). 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 simulation developer. 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[3], which consists of an environment and one agent (see Figure 2).

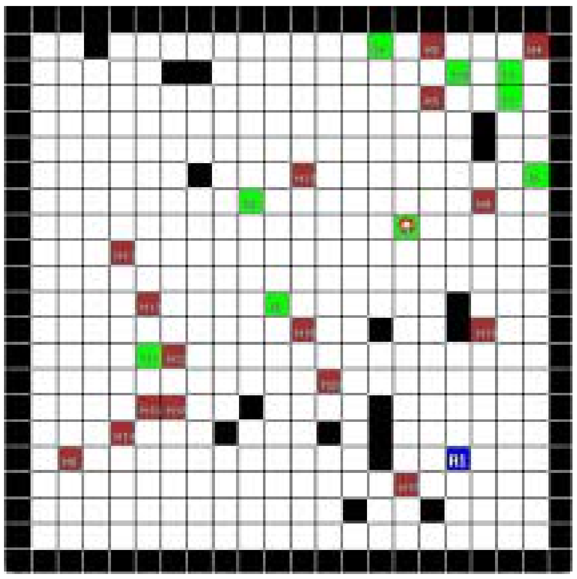

Figure 2: A screen shot of SIM_TILEWORLD

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.[4] 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

[3]Multi-Agent Tileworld(s) do exist (Ephrati, Pollack & Ur 1995)

[4]OBJECTCLASS shares many features of the Common Lisp Object System (CLOS).

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 simulation consists of two active objects (the environment and the agent) and a variable number of passive objects (the tiles, holes and obstacles). At simulation 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 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.

3 Distributing a SIM_AGENT simulation

The High Level Architecture (HLA) allows different simulations, referred to as *federates*, to be combined into a single larger simulation known as a *federation* (DMS 1998). 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. The structure of all FOMs is defined by the Object Model Template (OMT) which ensures federations can communicate with one another. The RTI is the middleware software that provides common services to simulation systems. Communication between federates and federations is done via the RTI. The FOM is supplied as data to the RTI at the beginning of an execution.

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 simulation using SIM_TILEWORLD as an example.

The HLA offers services in six areas, namely Federation Management, Object Management, Declaration Management, Ownership Management, Time Management, and Data Distribution Management. In the remainder of this section, we outline the role of these services in distributing the SIM_TILEWORLD simulation. (We do not consider Federation Management for a distributed SIM_AGENT federation as this is similar to other HLA federations.)

3.1 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 2.1, we chose to split the simulation into two federates, corresponding to the Tileworld agent and the Tileworld environment respectively. Figure 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.

In the current implementation of SIM_TILEWORLD, the communication between the agent and the environment federates is performed via the objects in the FOM, via the creation, deletion and updating of 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. The case for this would become clearer in a multi-agent implementation of SIM_TILEWORLD, where 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

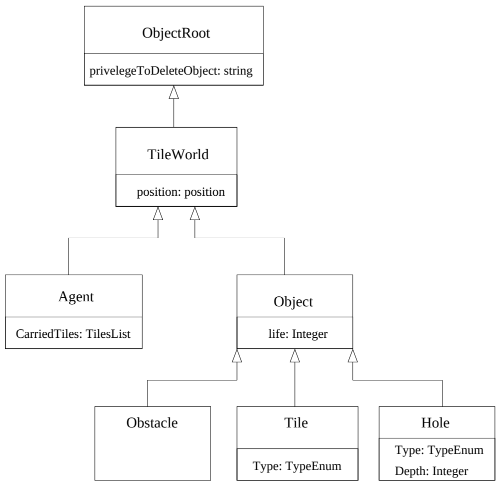

Figure 3: An example FOM for SIM_TILEWORLD

	Federate	
Object	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

Table 1: Object Class Publications and Subscriptions in Tileworld Federation

the 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. The other attributes are largely self explanatory.

3.2 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. Ownership Management plays an important role in a dynamic environment such as Tileworld. In the single agent SIM_TILEWORLD example, the transition of ownership is quite straightforward. However it becomes more complicated as more agent federates are added in the simulation. For instance, in a multi-agent Tileworld two (or more) agents may try to push the same tile. In terms of ownership management this raises important questions. Before an agent federate can move a tile it must obtain ownership of the tile's *position* attribute. Once the tile has been moved by this agent, the second agent's move should become invalid, as the tile is no longer at the position at which the agent initially perceived it.

For other attributes this may not be the case. For example, if the tile has a colour attribute that two agents both wish to change. If the first agent changes the colour to blue and the second agent changes it to red the tile would first change to blue and then to red. The fact that the first agent changes the colour to blue doesn't mean the second agent cannot then change the colour again.

3.3 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 federates current logical time.

SIM_AGENT is a centralised, time-driven system where simulation advances in timesteps, referred to as cycles. As explained in section 2, at the end of a cycle a series of actions may change some aspects of the simulation. These changes are then perceived by all agents at the beginning of the next cycle. It therefore makes sense that the Federation should 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.

3.4 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. Due to the complexity of router configuration and the limited availability of multicast groups, the assignment of multicast groups is static and is based on a priori knowledge of the federates' interconnection patterns (Morse & Zyda 2000). However, as explained in (Logan & Theodoropoulos 2001), in complex agent-based systems it is difficult, if at all possible, to determine an appropriate simulation topology a priori, and therefore, static interest management schemes are inadequate. Various efforts have been and are currently being undertaken to define alternative dynamic schemes for Interest Management (Morse & Zyda 2000, Logan & Theodoropoulos 2001); in (Logan & Theodoropoulos 2001) an approach which combines dynamic interest management and load balancing for the simulation of agent-based systems has been introduced.

The sensors used in SIM_AGENT can be restricted to a certain range. Therefore although using DDM would increase the efficiency of the simulation it is not essential 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.

4 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 simulation to be distributed using the HLA. We assume that we have an existing SIM_AGENT simulation (e.g., SIM_TILEWORLD) that we want to distribute by placing disjoint subsets of the objects and agents comprising the simulation 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 simulation.

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 simulated by this federate, proxies for agents simulated by other federates, RTI book-keeping information etc.;
- we extend the simulation classes so that updates to publicly visible attributes by agents simulated 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 simulated 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. SIM_AGENT will therefore be interfaced with the C++ version of the RTI. Any RTI Ambassador methods and Federate Ambassador methods needed for the implementation will be given appropriate C wrappers. The idea being all RTI calls can be made from SIM_AGENT as though we have an implementation of the RTI written in Pop-11.

In what follows, we briefly describe the necessary changes to SIM_AGENT in more detail. In section 4.5 we outline the operation of the modified scheduler over a single simulation cycle (see Figure 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[5].

4.1 Representing the federation

At each federate, we split the objects in the simulation into two lists: the *scheduler list* and the *simulation objects list*. The scheduler list contains instances of the standard `sim_object` and `sim_agent` classes and their subclasses which are being 'run' by SIM_AGENT on this federate. The simulation objects list is used for the sensors and actions of the agents being simulated by this federate. In what follows, we shall refer to the objects in the scheduler list as the "agents being simulated by this federate", always remembering that this covers `sim_object`s too. The simulation objects list contains everything in the scheduler list together with any objects being simulated by other federates which this federate knows about (e.g., via object discovery). 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)

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

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.

4.2 Propagating the effects of actions

As stated in section 2, 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., 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.) At the end of the cycle, when the actions of all the agents have been processed, the scheduler relinquishes ownership of any mutually exclusive attributes acquired by the federate at the current cycle.

The method wrapper for an non-mutually exclusive attribute, e.g., colour, first acquires and then relinquishes ownership of the attribute, allowing other agents running on this or other federates to update the attribute at this cycle.

```
define :wrapper updaterof colour(c, o:sim_tile, upd_p);
    ;;; acquire ownership of this attribute
    ;;; call RTI_update_attribute_values(HLA_identifier(o),
                                         ["colour", c], time);
    ;;; relinquish ownwership of this attribute

    ;;; do the local update
    upd_p(p, o);
enddefine;
```

4.3 Simulation 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 puts them in both the scheduler and simulation object lists. It also notifies the RTI of their creation. The RTI creates a unique identifier for each object in the simulation, and allows other federates to "discover" them. When we get the identifiers back from the RTI, we set the `object_identifier` slots in the locally scheduled objects.

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

4.4 Extending the scheduler

We define a new `sim_scheduler` method which takes an `HLA_federate` object as an argument and calls the existing SIM_AGENT sensor, agent decision making and action methods on the appropriate lists: the simulation object list in the case of sensors and actions, and the scheduler list for agent decision making.[6]

4.5 A cycle of SIM_AGENT in HLA

The main scheduler cycle proceeds as follows:

1. Wait for synchronisation with other federates.
2. At the beginning of each scheduler cycle, we get by discovery all the new objects that have been created by other federates at the last cycle and create an instance of the appropriate `sim_object` subclass. If other federates have deleted objects, we also have to delete our local proxies.
3. We obtain from the RTI all the attribute updates from the last cycle, and use this information to update the slots of the agents simulated at this federate (e.g., if an agent on another federate moves a tile simulated on this federate) and the slots of the proxy agents in the simulation object list. Slot update is accomplished using the `slot_values` method, as calling the normal slot updaters would trigger a rebroadcast of the attribute updates to the RTI.
4. For each object or agent in the scheduler list:
    (a) Run the agent's sensors on each of the objects in the simulation objects list. By convention, sensor procedures only access the publicly available data held in the slots of an object, updated in step 3.
    (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.
5. 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 the environment and/or the publicly visible attributes of the agent. This triggers the calls to `RTI_update_attribute_values`. We then wait until the RTI tells us that it is safe to proceed to the next cycle.
6. repeat.

The additional generic code, e.g., the definitions of `HLA_federate` and `HLA_object`, extensions to `sim_scheduler` etc. can be loaded as an additional library. The code which is specific to a particular simulation (essentially the method wrappers) is "boilerplate" code which can be generated automatically from the FOM and information about which classes in the FOM a federate publishes and which attributes it subscribes to. Different federates would 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.

5 Summary

In this paper, we have sketched an approach to distributing simulations of agent-based systems using the SIM_AGENT toolkit as an example. We showed how the HLA can be used to distribute an existing SIM_AGENT simulation with different agents being simulated by different federates and briefly outlined the changes necessary to the 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.

Although preliminary, the design study presented above highlights many of the issues that are central to any distributed simulation of agent-based systems. Future work will implement the presented design and evaluate the HLA for the simulation of agent based systems, using more complex testbeds. One key problem is the

[6] In the non-distributed implementation of SIM_AGENT, `sim_scheduler` is actually a normal procedure, but procedures and methods can be freely intermixed in OBJECTCLASS, a feature copied from CLOS.

efficient propagation of updates to the shared environment. Our proposal currently makes no use of the DDM services provided by the RTI. This is an area of current work (Logan & Theodoropoulos 2001).

Another obvious area for future work is *inter-operation*, using HLA to integrate SIM_AGENT with other simulators. This would allow the investigation of different agent architectures and environments using different simulators in a straightforward way. 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 interoperability guidelines for SIM_AGENT simulations.

Acknowledgements

We would like to thank Aaron Sloman, Ron Chrisley and other members of the Agents group at the University of Birmingham for many helpful discussions and comments on earlier versions of the design presented in this paper. This work is part of the PDES-MAS project[7] and is supported by EPSRC research grant No. GR/R45338/01.

References

Anderson, J. (2000), 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, pp. 36–44.

Atkin, S. M., Westbrook, D. L., Cohen, P. R. & Jorstad., G. D. (1998), 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, pp. 89–96. Technical Report WS–98–10.

Baxter, J. & Hepplewhite, R. T. (1999), 'Agents in tank battle simulations', *Communications of the ACM* **42**(3), 74–75.

DMS (1998), 'High level architecture interface specification, version 1.3'.

Durfee, E. H. & Montgomery, T. A. (1989), MICE: A flexible testbed for intelligent coordination experiements, *in* 'Proceedings of the Ninth Distributed Artificial Intelligence Workshop', pp. 25–40.

Ephrati, E., Pollack, M. & Ur, S. (1995), 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, pp. 679–685.

Logan, B., Fraser, M., Fielding, D., Benford, S., Greenhalgh, C. & Herrero, P. (2002), 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, pp. 62–68. Technical Report SS–02–01.

Logan, B. & Theodoropoulos, G. (2001), 'The distributed simulation of multi-agent systems', *Proceedings of the IEEE* **89**(2), 174–186.

Morse, K. L. & Zyda, M. (2000), On line multicast grouping for dynamic data distribution management, *in* 'Proceedings of the 2000 Fall Simulation Interoperability Workshop'. Paper No. 00F-SIW-052.

Pollack, M. E. & Ringuette, M. (1990), Introducing the tileworld: Experimentally evaluating agent architecture, *in* 'National Conference on Artificial Intelligence', pp. 183–189.

Schattenberg, B. & Uhrmacher, A. (2000), 'Planning agents in JAMES', *Proceedings of the IEEE*.

Scheutz, M. & Logan, B. (2001), 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, pp. 1–10.

Sloman, A. & Logan, B. (1999), 'Building cognitively rich agents using the SIM_AGENT toolkit', *Communications of the ACM* **42**(3), 71–77.

Sloman, A. & Poli, R. (1996), 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, pp. 392–407.

[7]http://www.cs.bham.ac.uk/∼gkt/Research/PDES/pdes-mas.html

Author Biographies

MICHAEL LEES is a PhD student studying in the School of Computer Science and IT at the University of Nottingham, UK. He received a joint Honours Computer Science and Artificial Intelligence degree from the University of Edinburgh,UK in 2001. His thesis will be centred of the areas of Multi-Agent systems and distributed simulation.

BRIAN LOGAN is a lecturer in the School of Computer Science and IT at the University of Nottingham, UK. He received a PhD in design theory from the University of Strathclyde, UK in 1986. His research interests include the specification, design and implementation of agent-based systems, including logics and ontologies for agent-based systems and software tools for building agents.

GEORGIOS THEODOROPOULOS received a Diploma degree in Computer Engineering from the University of Patras, Greece in 1989 and MSc and

PhD degrees in Computer Science from the University of Manchester, U.K. in 1991 and 1995 respectively. Since February 1998 he has been a Lecturer in the School of Computer Science, University of Birmingham, U.K. teaching courses on Hardware Engineering and Computer Networks. His research interests include parallel and distributed systems, computer and network architectures and modelling and distributed simulation.