Simulating Agent-Based Systems with HLA: The Case of SIM_AGENT — Part II (03E–SIW–076)

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

Ton Oguara
Georgios Theodoropoulos
School of Computer Science
University of Birmingham
Birmingham, B15 2TT, UK
{txo|gkt}@cs.bham.ac.uk

ABSTRACT *In this paper we outline an approach to the distributed simulation of agent-based systems using the* SIM_AGENT *toolkit and the High Level Architecture (HLA) simulator interoperability framework. Using a simple Tileworld scenario as an example, we show how the HLA can be used to flexibly distribute a* SIM_AGENT *simulation with different agents being simulated on different machines. We outline the changes necessary to the* SIM_AGENT *toolkit to allow integration with the HLA, and briefly describe the simulation cycle of the combined system which we call* HLA_AGENT. *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 present some preliminary experimental results which illustrate the performance of* HLA_AGENT *on a Linux cluster running a distributed version of Tileworld and compare this with the original (non-distributed)* SIM_AGENT *version.*

1 Introduction

The adoption of multi-agent systems 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 [6, 14, 3, 18, 1, 15]. However no one testbed is, or can be, appropriate to all agents and environments, and demonstrating that a particular result holds across a range of agent architectures and environments often requires using a number of different systems. Moreover, 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.

In this paper we present an approach to agent simulation which addresses both inter-operability and scalability issues. We describe HLA_AGENT, a tool for the distributed simulation of agent-based systems, which integrates the SIM_AGENT agent toolkit and the High Level Architecture (HLA) a simulator interoperability framework developed by the US DMSO [10]. HLA allows the inter-operation of various simulators and testbeds supporting different agent architectures and environments to be distributed on different machines to increase the overall performance of the global simulation. The remainder of this paper is organised as follows. 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 on different machines. In section 4 we sketch the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA and describe the resulting library, which we call HLA_AGENT. In section 5 we present some preliminary experimental results to illustrate the performance of HLA_AGENT on a Linux cluster running a distributed version of Tileworld and compare this with the original (non-distributed) SIM_AGENT version of Tileworld. 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 [18, 17][1]. 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 [16], agents which report on activities in collaborative virtual environments [11] (which involved integrating SIM_AGENT with the MASSIVE-3 VR system), and simulation of tank commanders in military training simulations [4] (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, or ruleset, is implemented as a collection of rules in a high-level rule-based language called POPRULEBASE. 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. SIM_AGENT can also be used to simulate the agent's environment, and the toolkit provides facilities to populate the agent's environment with user-defined active and passive objects (and other agents).

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

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).

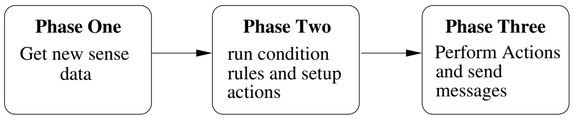
Phase One
Get new sense data
Phase Two
run condition rules and setup actions
Phase Three
Perform Actions and send messages

Figure 1: Logical structure of a simulation cycle

In the first phase each agent's internal database is updated with sensory data 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, which is used to hold the agent's model of the environment, its current goals, plans etc. External data is data which is externally visible to other objects in the environment, e.g., colour, size, shape etc. The agent's sensors update the agent's database with externally visible data. 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 second phase involves decision making and action selection. The contents of the agent's database together with the new facts created in the first phase 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. These rules will typically cause some internal and/or external action(s) to be performed or message(s) to be sent to other agents. 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. External actions typically cause the agent to enter a new state (e.g., to change its location) and hence sense new data.

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.[2] 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`.

2.1 An example: SIM_TILEWORLD

In this section we briefly outline the design and implementation of a simple SIM_AGENT simulation, SIM_TILEWORLD. Tileworld is a well established testbed for agents [14]. 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) [13] and in comparisons of reactive and deliberative agent architectures [14]. SIM_TILEWORLD is an implementation of a multi-agent Tileworld [7], which consists of an environment and one or more agents (see Figure 2).

For the SIM_TILEWORLD example three subclasses of the SIM_AGENT base class `sim_object` were defined to represent holes, tiles and obstacles, together with two subclasses of `sim_agent` to represent the environment and the agents. 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 or more active objects (the environment and the agent(s)) 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 agents' 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

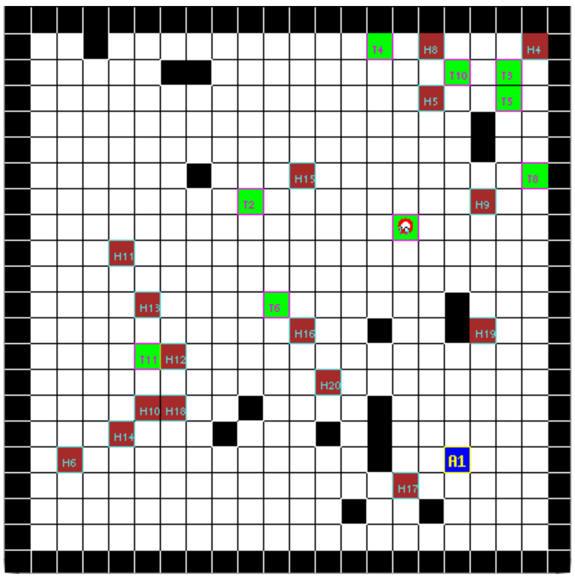

Figure 2: A screen shot of SIM_TILEWORLD

probabilities. The scheduler then runs the agents which perceive the new environment and updates their internal databases with the new sense data. The agents then run 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 with HLA

The High Level Architecture (HLA) allows different simulations, referred to as *federates*, to be combined into a single larger simulation known as a *federation* [5]. The federates may be written in different languages and may run on different machines. More precisely, a federation comprises:

- 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 and is supplied as data to the RTI at the beginning of an execution.

The RTI is the middleware that provides common services to simulation systems, and all 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 (see Figure

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

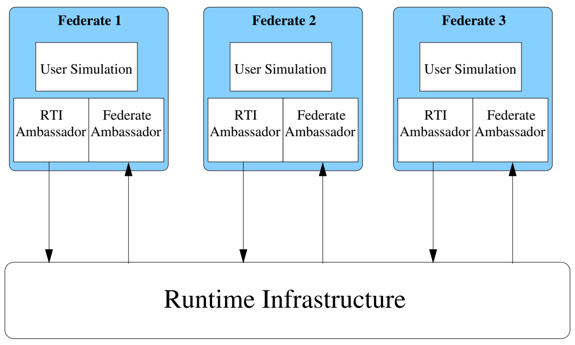

Figure 3: The architecture of an HLA federation

3). 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 to the relevant federates. For example, updating the value of an attribute of an object instance on one federate will result in a callback(s) containing the new value to the relevant federate(s). It is the job of the Federate Ambassador to handle these callbacks and invoke appropriate code in the user simulation, e.g., update a the value of a variable or field representing the attribute.

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. Based on the SIM_TILEWORLD implementation outlined in section 2.1, we chose to split the simulation into $n + 1$ federates, corresponding to n Tileworld agents and the Tileworld environment respectively.

The HLA provides 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 and Data Distribution Management is not used in the current implementation of HLA_AGENT.)

3.1 Object and declaration management

In the HLA, information about objects in the simulation is not held centrally; rather each federate is responsible for maintaining its own, local information about objects of interest simulated by other federates. 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.

A Federate declares its interest in objects and attributes at the beginning of a simulation by *publishing* any attributes it may update during the simulation and *subscribing* to attributes which it would like to receive updates for. A Federate which is subscribed to an attribute of a certain class will also discover any instances of this class when they are created.

In the distributed implementation of SIM_TILEWORLD, the communication between the agent and environment federates is performed via the objects in the FOM, via the creation, deletion and updating of attributes. Figure 4 depicts the FOM for the HLA SIM_TILEWORLD. Two main subclasses are defined: Agent and Object, with the Object class having Tiles, Holes and Obstacles as subclasses. The Agent class is included in the FOM as certain attributes of agents may be accessed by other federates, e.g., the agents need to know the position of other agents in the environment (for sensing).

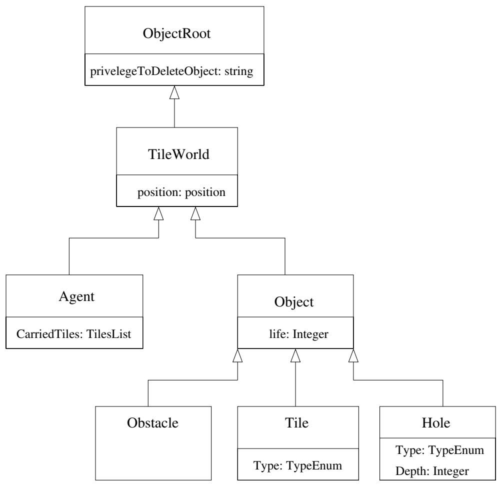

Figure 4: An example FOM for SIM_TILEWORLD

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 when the tile is initially 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.

	Federate	
Object	Environment	Agent
Agent		
privilegeToDelete	publish	publish
position	subscribe	publish
carriedTiles	subscribe	publish
Tile		
privilegeToDelete	publish	publish
position	publish	publish
life	publish	subscribe
type	publish	subscribe
Hole		
privilegeToDelete	publish	publish
position	publish	subscribe
life	publish	subscribe
type	publish	subscribe
depth	publish	publish
Obstacle		
privilegeToDelete	publish	publish
position	publish	subscribe
life	publish	subscribe

Table 1: Object Class Publications and Subscriptions in Tileworld Federation

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. For example, in a multi-agent Tileworld two (or more) agents may try to push the same tile. 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. To ensure that attributes are mutually exclusive, we only allow transfer of ownership once per simulation cycle for any given attribute. A federate will then only relinquish ownership of an attribute if it has not already been updated at the current cycle.

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. We therefore arrange for the Federation to synchronise at the beginning of each cycle, 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 Extending SIM_AGENT

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. Together, the extensions constitute a new library which we call HLA_AGENT. Our aim is to make the distribution transparent to the user simulation and to the existing SIM_AGENT low level scheduler code which processes the agents and objects comprising the simulation. Ideally, the user simulation should run unchanged, with the user providing additional information specifying the number of federates in the federation and how the objects and agents in their simulation are to be assigned to federates so as to make best use of available computing resources. The distribution of the user simulation should also be symmetric in the sense that no additional management federates are required.

In what follows, 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. Each federate corresponds to a single SIM_AGENT process and is responsible both for simulating the local objects forming its own part of the global simulation, and for maintaining *proxy* objects which represent objects of interest being simulated by other federates. Each federate may be initialised with part of the total model or all federates can run the same basic simulation code and use additional information supplied by the user to determine which objects are to be simulated locally. For example, in SIM_TILEWORLD we may wish to simulate the agent(s) on one federate and the environment on another.

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 bookkeeping information etc.;
- we have to handle object creation and deletion by the user simulation, and propagation of object attributes when the user simulation updates externally visible data;
- we have to modify the SIM_AGENT scheduler so that only those agents simulated by this federate are actually run at each cycle, to process the object discovery, object deletion and attribute update callbacks, and to and handle synchronisation at each cycle; and

- we need to add some code to connect to the RTI and initialise the federate's data structures.

The overall organisation of HLA_AGENT is similar to that illustrated in Figure 3. Each SIM_AGENT federate requires two ambassadors: an RTI Ambassador which handles calls to the RTI and a Federate Ambassador that handles callbacks from the RTI. Calls to the RTI are processed asynchronously in a separate thread. However, for simplicity, we have chosen to queue callbacks from the RTI to the Federate Ambassador for processing at the end of each simulation cycle. SIM_AGENT has the ability to call external C functions. We have therefore adopted the reference implementation of the RTI written in C++ developed by DMSO, and defined C wrappers for the RTI and Federate Ambassador methods needed for the implementation. We use Pop-11's simple serialisation mechanism to handle translation of SIM_AGENT data structures to and from the byte strings required by the RTI. All RTI calls and processing of Federate Ambassador callbacks can therefore be handled from SIM_AGENT as though we have an implementation of the RTI written in Pop-11.

In what follows, we briefly describe the changes to SIM_AGENT in more detail and 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[3].

4.1 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 simulated 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 FOM for the federation, a handle to the local RTI Ambassador etc. `HLA_object` holds RTI bookkeeping information for objects in the simulation, e.g., the unique RTI identifier for the object which is shared by all the federates in the simulation, published and subscribed attributes and a flag which indicates whether the object is local or a proxy. 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.

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

4.2 Creating and deleting objects

We assume that the class definitions for the objects and agents comprising the simulation are available to all federates which publish and subscribe to the corresponding FOM classes and attributes, 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. OBJECTCLASS provides *wrappers*, closures around existing methods which extend or even replace the functionality of the method. The existing SIM_AGENT code is extended with 'new' and 'destroy' wrappers, which intercept calls to class constructors and destructors respectively. When a new object is created by the user simulation, the new wrapper registers it with the RTI, triggering object discovery callbacks and proxy creation on other federates. The new wrapper also handles the bookkeeping information in the associated `HLA_object`. A similar pattern is used with object deletions. When an object in the user simulation becomes garbage, the destroy wrapper is run. This 'undeletes' the object by creating a new (non-weak) reference to it. This ensures that the object persists until the end of the current simulation cycle and allows the object to be deleted on all federates at the same time. If the federate does not own the object being deleted (e.g., if it is a proxy), the destroy wrapper also negotiates with the federate that owns the object for permission to delete it. By convention, a federate will always grant permission to delete an object unless it intends to delete the object itself at this cycle or has already given permission to another federate. Requesting permission to delete an object is therefore sufficient to ensure that the object will be deleted.

4.3 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 situation is complicated by the fact that external actions are usually queued by SIM_AGENT for execution at the end of the current cycle. This avoids the agents in the (local) simulation "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. For example, 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. 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 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. For example, we may 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. In HLA_AGENT, we use 'update' wrappers to "intercept" calls to the slot updater methods and propagate the new value to other federates by making the appropriate RTI calls.

The wrapper for a slot updater first checks to see if the slot corresponds to an attribute of a class in the FOM which is published by the federate. If so, the federate requests the RTI to propagate the update to other federates which have subscribed to this attribute. This may involve negotiating for ownership of the attribute instance with another federate. To ensure that attributes are mutually exclusive, we require that attribute ownership can only be transferred once per simulation cycle, and a federate will relinquish ownership of an attribute only if it has not already been updated at the current cycle. 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 agent's action will be denied ownership and fail.[4] Updates to slots not corresponding to published attributes are assumed to be local to the federate and are not propagated.

To avoid multiple updates to the same attribute at the same cycle being received out of order by other federates, updates to attributes owned by this federate are queued, and only the last update to each attribute is propagated to the RTI. (This is necessary, for example, during object creation to avoid values set by superclass initialisers over-writing a value set by a class initialiser.) Updates to attributes not owned by this federate are further delayed until ownership of the attribute is transferred to the federate. If the federate is unable to obtain ownership of the attribute, i.e., if the attribute has already been updated by another federate at this cycle, then the local update is discarded. All attribute updates are queued by the RTI for delivery at the next timestep; as a result, agents on all federates always see the same simulation state.

4.4 Simulation startup

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

When a HLA_AGENT federate starts up, it loads the user simulation code and runs a user-defined initialisation procedure to create instances of all the objects and agents that are to be run locally. The federate registers the objects created with the RTI and flushes the initial values of their published slots to the RTI for propagation to other federates. The RTI allocates a unique identifier to each local object in the simulation, and notifies other federates of their existence via 'object discovery' callbacks to the Federate Ambassadors. Any objects created during initialisation of the local simulation which are not to be simulated on this federate are assigned proxies and the values of subscribed slots are initialised following attribute updates from the federate simulating the object to which the proxy corresponds.

4.5 A cycle of SIM_AGENT in HLA

Once all federates have initialised, each federate enters the main simulation loop as detailed below:

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

[4]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.

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.

4.6 Partitioning the user simulation

To distribute a simulation, the user must define the classes and attributes that constitute the Federation Object Model and, for each federate, provide a mapping between the classes and attributes in the FOM and the SIM_AGENT classes and slots to be simulated on that federate. The additional generic code, e.g., the definitions of `HLA_federate` and `HLA_object`, extensions to `sim_scheduler` etc. are loaded as an additional library. If the user simulation is partitioned so that each federate only creates instances of those objects and agents it is responsible for simulating, then no additional user-level code is required. In the case in which all federates use the same simulation code, the user must define a procedure which is used to determine whether an object should be simulated on the current federate. The user therefore has the option of partitioning the simulation into appropriate subsets for each federate, thereby minimising the number of proxy objects created by each federate at simulation startup, or allowing all federates to create a proxy for all non-local objects in the simulation. For very large simulations, the latter approach may entail an unacceptable performance penalty, but has the advantage that distributed and non-distributed simulations can use identical code.

5 Results

To evaluate the robustness and performance of HLA/RTI and our distribution approach, we have developed a SIM_AGENT Federation using SIM_TILEWORLD as a test case.

The hardware platform used for our experiments is a Linux cluster, comprising 44 1.6GHz AthlonMP 1900+ processors (22 dual nodes, each with 256 KB cache) interconnected by a standard 100Mbps fast Ethernet switch. Our test environment is a Tileworld 20 units by 20 units in size with an object creation probability (for tiles, holes and obstacles) of 0.1. The number of agents in the Tileworld ranges from 1 to 64. In the current SIM_TILEWORLD federation, the environment is simulated by a single federate while the agents are distributed in one or more federates over the nodes of the cluster. In all our experiments, we have only one agent federate per each cluster node we use. We used the timers provided by SIM_AGENT which have a resolution of 1/100th of a second, and the results obtained represent averages over 5 runs of 100 SIM_AGENT cycles.

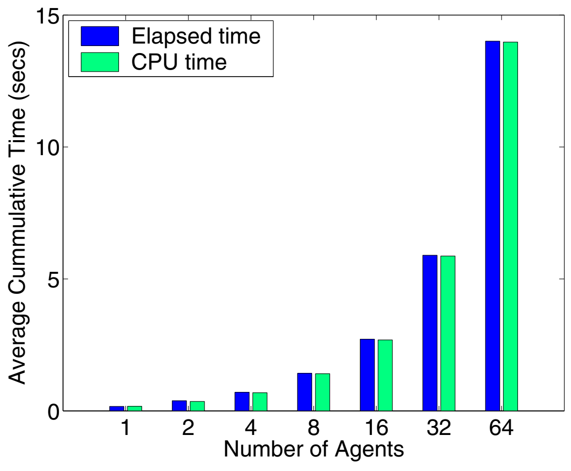

Figure 5: Total CPU and Elapsed Times for 1-64 Agents in SIM_TILEWORLD

For comparison, Figure 5 shows the total CPU and elapsed times when executing 1, 2, 4, 8, 16, 32 and 64 SIM_TILEWORLD agents on a single node using SIM_AGENT. As can be seen, the CPU and elapsed times increase almost linearly with the number of agents, and on an unloaded cluster node, the CPU and elapsed times are very similar.

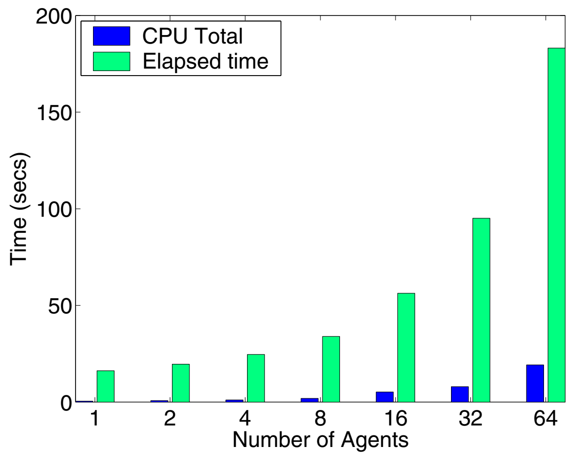

Figure 6: Total Elapsed Times for 1-64 Agents in HLA_AGENT.

Figure 6 shows the total CPU and elapsed times when executing 1, 2, 4, 8, 16, 32 and 64 SIM_TILEWORLD agents and the environment in a single federate on a single node using HLA_AGENT. Again, the CPU and elapsed times increase with the number of agents simulated, but in this case the ratio of CPU to elapsed time is much smaller, typically less than 0.1.

Comparing the data for e.g., 64 agents simulated using SIM_AGENT and HLA_AGENT, we can see that the HLA introduces a significant overhead. In SIM_AGENT, simulating 64 agents for 100 cycles takes 13.98 seconds of CPU time and 14.01 seconds of elapsed time, or an average of about 0.138 CPU seconds and 0.138 elapsed seconds per cycle.[5] In HLA_AGENT, simulating 64 agents for 100 cycles requires 19.22 seconds CPU time and 182.14 seconds elapsed time, and the average CPU and elapsed times for each cycle of HLA_AGENT are 0.189 and 1.795 seconds respectively (see Figure 8). These average cycle times can be further broken down into the time for the simulation phase (i.e., running the user simulation) and the RTI phase (i.e., propagating updates, synchronising with other federates etc). The overhead in the simulation phase (basically running the slot update and access wrappers and queueing attribute updates for propagation at the end of the user simulation cycle) is on the order of 0.051 seconds per cycle and accounts for the bulk of the CPU overhead (around 98%). In the RTI phase, queued attribute updates are flushed to the RTI, incoming attribute updates are applied to the slots of local objects and proxies, object deletions are processed, and the federate synchronises with other federates for the start of the next simulation cycle. This accounts for approximately 0.001 seconds per cycle or 2% of the CPU overhead.

[5]The reason the average elapsed time per cycle is not equal to the total elapsed time / 100 is that the total elapsed time includes the time to synchronise the federation and populate the simulation, which takes very little CPU time.

The elapsed time overhead is more significant. Each cycle of HLA_AGENT requires 0.394 seconds of elapsed time for the simulation phase and 1.401 seconds of elapsed time for the RTI phase (see Figure 9). There are two issues here: the first is that the ratio of CPU to elapsed time for the simulation phase is worse than for SIM_AGENT. This is presumably due to the time spent in RTI calls during the simulation phase (e.g., registering object instances, requesting attribute ownership transfers etc.). However the bulk of the overhead is the elapsed time spent in the RTI phase. Detailed analysis reveals that this consists largely of time spent in `tick`. The current implementation of HLA_AGENT calls `tick` after each call to the RTI, e.g., `updateAttributeValues`, `deleteObjectInstance` etc., and uses a `mintick` value of 0.01 and a `maxtick` of 10.0 seconds. Each agent performs at least two updates per simulation cycle, so with 64 agents we will have at least 128 updates per cycle. This gives at least 128 ticks or 1.28 seconds per simulation cycle, which corresponds fairly closely to the elapsed time overhead in our experiments. The value of 0.01 for `mintick` was determined empirically and is the lowest that gave reliable propagation of updates. With lower values (e.g., 0.005 or 0.001), we experienced increasingly frequent loss of updates and failures of the RTI Ambassador and/or the Federation Ambassador.

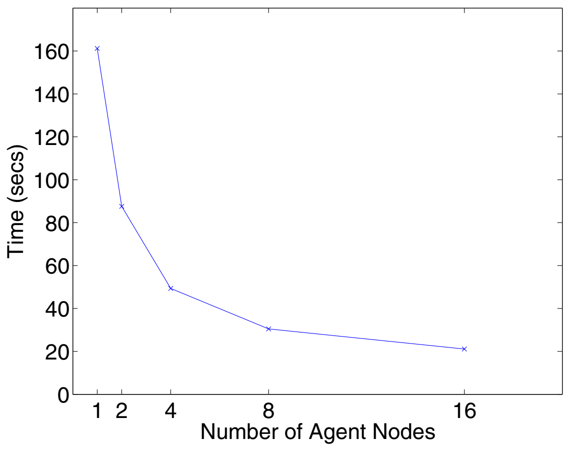

Figure 7: Total Elapsed Time for an Agent Federate (64 Agents Distributed Over 1-16 Nodes).

We also investigated the effect of distributing the Tileworld agents across varying numbers of federates. Since the bulk of the overhead in HLA_AGENT is determined by the number of attribute updates and hence ticks, moving agents (and the updates they generate) onto other nodes should give a reduction in elapsed time for any given node.

Figure 7 shows a breakdown of the total elapsed time for an agent federate when distributing 64 agents over 1, 2, 4, 8, 16, 32 and 64 nodes of the cluster. As expected, the elapsed time drops with increasing distribution, and with 16 nodes the elapsed time is comparable to the non-distributed case.

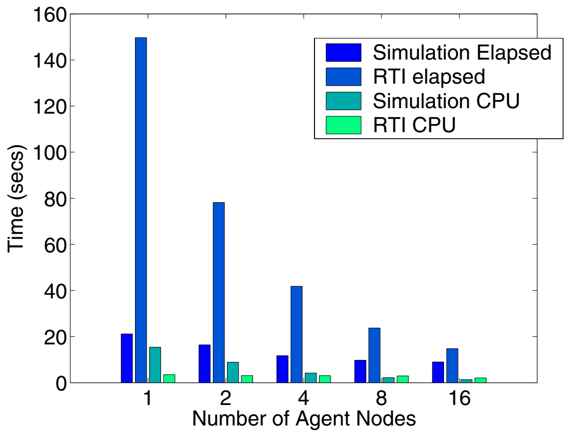

Figure 8: CPU time vs Elapsed Time for an Agent Federate (64 Agents Distributed over 1-16 Nodes).

Figure 8 shows a breakdown of the total CPU and elapsed times for both the simulation and RTI phases of HLA_AGENT for an agent federate for each distribution of 64 agents over nodes of the cluster. The environment was simulated in its own federate on a different cluster node and the CPU and elapsed times for the environment federate for both the simulation and RTI phases are shown in Figure 9.

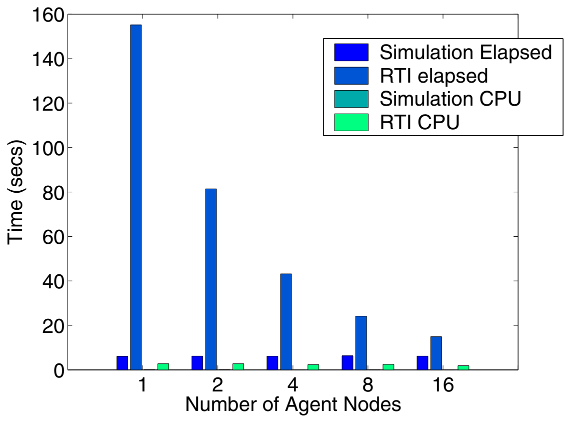

Figure 9: CPU time vs Elapsed Time for the Environment Federate (64 Agents Distributed over 1-16 Nodes).

Although preliminary, our experiments show that, even with relatively lightweight agents like the Tileworld agents, we can get speedup by distributing agent federates across multiple cluster nodes. However processor utilisation is low due to the elapsed time overhead. We would expect to see more favourable results with heavyweight agents which intrinsically require more CPU time. For example, Schattenberg and Uhrmacher [15] report experiments with planning agents which require from 2 seconds to 20 hours of CPU time per cycle.

6 Related Work

There has been relatively little work to date on the distributed simulation of multi-agent systems. In this section, we outline some of the current research in the area.

Schattenberg and Uhrmacher [15, 19] have developed a Java-Based Agent Modelling Environment for Simulation (JAMES). In JAMES, agents are modelled as situated automata, and the agent program is compiled into transition rules. JAMES extends the DEVS formalism [20] and utilises an approach based on *discrete event simulation* rather than the time-stepped simulation used by HLA_AGENT. This can simplify the representation time in simulations of agent behaviour and may also be more efficient if the lower bound on the response time of an agent is a large multiple of the minimum time between events in the simulation. JAMES uses a centralised server architecture to simplify the tracking and administration of moving agents. This introduces a centralised bottleneck which can affect performance when there are large numbers of agents in the simulation. The main objective of the JAMES work is to facilitate small and large scale testing of multi-agent systems, and the system has been used to implement a version of Tileworld for testing simple planning agents. Interoperability with other simulations is not an aim.

DGensim (Distributed Gensim) [1], is also an extension of a non-distributed agent simulator, Gensim [2]. In the original Gensim, as in SIM_AGENT, actions performed by agents are processed on an agent-by-agent basis. This has the undesired effect that the results of some agent's actions become apparent before others. Although using small time cycles reduces the problem it does not completely eliminate it. DGensim divides the simulation onto n node processors, $n-1$ of which execute the internals of an agent and an *agent manager*. The remaining processor executes the *environment manager*. In DGensim agents send their decisions (which are timestamped) asynchronously to an action monitoring agent inside the environment manager. Although agents make their decisions asynchronously, the environment manager is a time-driven simulation. As a result, agent actions aren't processed by the environment until the environment's simulation time reaches the timestamp associated with the action. The central aim of DGensim was to improve fidelity of the original Gensim system, with increased speed of simulation execution viewed as a useful side-effect. However, DGensim also has a centralised bottle neck in the environment manager. As with JAMES, interoperability with other types of simulation was not a design goal.

MACE-3J is a Java-based MAS simulation, integration and development testbed [9]. It is an extension of the original MACE testbed [8], which aims to fulfil the need for tools to support distributed collaborative scientific research in large scale, large-grain MAS. As with HLA_AGENT, agents are distributed via proxies. This allows 'real' agents to interact with simulated agents as all interaction is performed via the proxy. MACE3J has been used in a number of experimental applications, including the distribution of legacy a agent application written in Java.

7 Summary

In this paper, we have presented an approach to distributing simulations of agent-based systems using the SIM_AGENT toolkit and the HLA. 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. The resulting library, which we call HLA_AGENT, incorporates a simple solution to the problem of action conflicts in a distributed environment which exploits the existing Ownership Management services of the HLA. The integration of SIM_AGENT and HLA is transparent in the sense that an existing SIM_AGENT user simulation runs unmodified and the agents are unaware that other parts of the simulation are running remotely, and symmetric in the sense that no additional management federates are required. The allocation of agents to federates can be easily configured to make best use of available computing resources.

We are currently investigating the performance of HLA_AGENT in a number of SIM_AGENT applications, and preliminary results for Tileworld example described above show that we can obtain speedup by distributing agents and federates across multiple nodes in a cluster. However further work is required to characterise the performance of HLA_AGENT with different kinds of agents and environments. One key problem is the efficient propagation of updates to the shared environment. Our approach currently makes no use of the Data Distribution Management services provided by the RTI. This is an area of current work [12]. As part of this work we are implementing additional tools for data collection and debugging. Existing low-level tools such as the Federation Management Tool, which allows control and monitoring of a federation execution, do not present a user-level view of the simulation. As a first step we have implemented a monitoring federate for the Tileworld federation which simply subscribes to a given set of classes and attributes and graphically displays updates to proxies of objects and agents simulated on other federates. It should also be relatively straightforward to implement a simple form of code migration to support coarse grain load balancing. One advantage of using heavyweight proxies is that it is easy to move the execution of a local object to its proxy on another federate, by transferring the values of its unpublished slots and the contents of its database and resetting the proxy flags.

Another 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 inter-operability guidelines for SIM_AGENT simulations.

Acknowledgements

We would like to thank Bora Kumova, Aaron Sloman, Ron Chrisley and other members of the Modelling and Analysis of Systems and Agents groups 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[6] and is supported by EPSRC research grant No. GR/R45338/01.

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, and J. W. Rozenblit, editors, *Proceedings of the Tenth Conference on AI, Simulation and Planning, AIS-2000*, pages 36–44. Society for Computer Simulation International, March 2000.

[2] J. Anderson and M. Evans. A generic simulation system for intelligent agent designs. *Applied Artificial Intelligence*, 9(5):527–562, October 1995.

[3] S. M. Atkin, D. L. Westbrook, P. R. Cohen, and G. D. Jorstad. AFS and HAC: Domain general agent simulation and control. In J. Baxter and B. Logan, editors, *Software Tools for Developing Agents: Papers from the 1998 Workshop*, pages 89–96. AAAI Press, July 1998. Technical Report WS–98–10.

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

[5] High level architecture interface specification, version 1.3, 1998.

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

[6] E. H. Durfee and T. A. Montgomery. MICE: A flexible testbed for intelligent coordination experiements. In *Proceedings of the Ninth Distributed Artificial Intelligence Workshop*, pages 25–40, September 1989.

[7] E. Ephrati, M. Pollack, and S. Ur. Deriving multi-agent coordination through filtering strategies. In C. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 679–685, San Francisco, 1995. Morgan Kaufmann.

[8] L. Gasser, C. Braganza, and N. Herman. Mace: A flexible testbed for distributed ai research. In M. N. Huhns, editor, *Distributed Artificial Intelligence*, pages 119–152. Pitman Publishers, 1987.

[9] L. Gasser and K. Kakugawa. Mace3j: Fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proceedings of AAMAS-2002*, Bologna, July 2002.

[10] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, 1999.

[11] B. Logan, M. Fraser, D. Fielding, S. Benford, C. Greenhalgh, and P. Herrero. Keeping in touch: Agents reporting from collaborative virtual environments. In K. Forbus and M. S. El-Nasr, editors, *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Symposium*, pages 62–68, Menlo Park, CA, March 2002. AAAI Press. Technical Report SS–02–01.

[12] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. *Proceedings of the IEEE*, 89(2):174–186, February 2001.

[13] M. E. Pollack, D. Joslin, A. Nunes, S. Ur, and E. Ephrati. Experimental investigation of an agent commitment strategy. Technical Report TR 94–31, University of Pittsburgh, Pittsburgh, PA 15260, 1994.

[14] M. E. Pollack and M. Ringuette. Introducing the tileworld: Experimentally evaluating agent architecture. In *National Conference on Artificial Intelligence*, pages 183–189, 1990.

[15] B. Schattenberg and A. M. Uhrmacher. Planning agents in JAMES. *Proceedings of the IEEE*, 89(2):158–173, Feb. 2001.

[16] M. Scheutz and B. Logan. Affective vs. deliberative agent control. In *Proceedings of the AISB'01 Symposium on Emotion, Cognition and Affective Computing*, pages 1–10. AISB, The Society for the Study of Artificial Intelligence and the Simulation of Behaviour, March 2001.

[17] A. Sloman and B. Logan. Building cognitively rich agents using the SIM_AGENT toolkit. *Communications of the ACM*, 42(3):71–77, March 1999.

[18] A. Sloman and R. Poli. SIM_AGENT: A toolkit for exploring agent designs. In M. Wooldridge, J. Mueller, and M. Tambe, editors, *Intelligent Agents II: Agent Theories Architectures and Languages (ATAL-95)*, pages 392–407. Springer–Verlag, 1996.

[19] A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(2):206–232, 2001.

[20] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.