Proceedings of the 2010 Winter Simulation Conference
B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, eds.

INTERFACING MULTI-AGENT MODELS TO DISTRIBUTED SIMULATION PLATFORMS: THE CASE OF PDES-MAS

B.G.W. Craenen

School of Computer Science
The University of Birmingham
Edgbaston, Birmingham, B15 2TT, UK

G.K. Theodoropoulos

School of Computer Science
The University of Birmingham
Edgbaston, Birmingham, B15 2TT, UK

ABSTRACT

Multi-Agent Systems (MAS) are increasingly used to solve larger and more complex problems. To provide the computational resources needed to do this, MAS are increasingly distributed over multiple computational platforms. Different approaches for distributing MAS have been proposed over the years. One problem remains central whichever approach is used: how to translate MAS behaviour into a format suitable for the distribution approach used. In this paper we describe the Agent Distributed Shared Memory Interface (ADSMI), an interface between a general MAS description and the PDES-MAS platform as an implementation of a Distributed Shared Memory (DSM) system for distributed MAS simulations. The ADSMI provides a translation of MAS behaviour into event-interactions with PDES-MAS, as well as functionality for handling time progressing in the MAS, and a messaging mechanism between agents.

1 INTRODUCTION

A Multi-Agent System (MAS) is a system composed of multiple interacting intelligent agents. MAS are commonly used to model and solve problems that are difficult or impossible for an individual agent, or monolithic system, to solve. Examples of problems appropriate for MAS research include online trading, disaster response, and modelling social structures.

MAS are increasingly used to solve larger and more complex problems. The computational resources the MAS require for solving these problems has increased to such an extend that they now often surpass the resources available on a single computer platform. This has lead to research into distributing MAS over multiple computer platforms.

Various approaches for distributing MAS over multiple computer platforms can be used (Theodoropoulos et al. 2007). One such approach focuses on partitioning the simulation topology into several semi-autonomous regions and then distributing these partitions over the computer platforms available. Another approach is to partition the shared state space of the simulation using a Distributed Shared Memory (DSM) system over the distributed computer platform available; an example of this approach is the PDES-MAS framework (Logan and Theodoropoulos 2001a). These, and other approaches all have specific advantages and limitations, making them particularly appropriate for certain MAS and situations but less so for others. All however share the problem of having to translate MAS behaviour or topology into a format suitable for the distribution approach used.

This paper will investigate a solution for this problem where the general MAS is distributed using the DSM approach. We will describe a translation method for converting general MAS behaviour into event-interactions with a DSM using an interface called the Agent Distributed Shared Memory Interface (ADSMI). The high-level ADSMI description may be applicable for different DSM systems but a specific use-case for the PDES-MAS platform is described here. Other DSM the ADSMI could use include a distributed database, a shared memory interface on a multi-core computer, or Java's Terracotta platform using distributed shared objects (Terracotta Inc. 2010).

The paper is organised as follows. Sections 2 and 3 will provide a brief overview of the requirements and characteristics of MAS and DSM respectively. Section 4 will then provide a high-level description of how the ADMSI can bridge these

requirements and translate from the MAS to the DSM and the other way round. The PDES-MAS platform will be used as an implementation of a DSM, and details about this platform will be given in 5. Section 6 will then describe how the ADSMI is implemented using the PDES-MAS platform. Finally, section 7 will draw some overall conclusions and discuss future work.

2 MULTI-AGENT SYSTEM (MAS)

A MAS simulation defines a number of agents that are assumed to be intelligent and that interact with each other and with their environment. The agents' interactions with other agents and its environment are hard to predict in advance, indeed discover at all. Moreover, how the agents interact with each other and the environment is often a primary goal of the simulation. A defining characteristic of agents is their autonomy (Wooldridge and Jennings 1995).

In the conventional MAS the agents go through what is called the sense-think-act cycle. Over the life-time of the simulation, time progression is defined by requiring all agents to go through many of these cycles. During the sense-part of the cycle, information is gathered about both the agent itself and its environment. During the think-part of the cycle the gathered information is used to determine a (pre-described) behaviour which in turn is translated into actions in the act-part of the cycle.

Agents encapsulate a number of variables, some visible or shared with other agents in the simulation, others private and not shared with other agents but only available to the agent itself. The values of all private and shared variables at a certain point in time during the simulation then describes the state of a simulation at that time. The collective state of the shared variables can be seen as similar to space-time memory (Ghosh and Fujimoto 1991, Mehl and Hammes 1993). In this context shared variables offer a natural representation of simulated context when interactions between agents are described as interactions on these variables. In other words; agents alter the state of the simulation by interacting with either their own, or other agent's variables. It is this context that will be used as a basis to define the functionality of the ADSMI.

3 DSM

A DSM system is commonly used in distributed computing for making space-time memory available across distributed computer platforms, often without exposing the exact way in which the memory is accessed or organised. Examples of DSM systems include the mechanisms used for making shared memory available on multi-core computer architectures, distributed database platforms, general DSM platforms like Terracotta (Terracotta Inc. 2010), or more specific MAS oriented DSM platforms like PDES-MAS (see section 5).

Access to the shared memory in a DSM is provided through read and write operations on the shared data. These two events provide the ability to read the value stored at a certain memory-address (also called an ID-query), and write a value to a certain memory-address. Memory-addresses are allocated by the DSM and are specific, although indexing over memory-addresses is often offered as a feature of a DSM system. Range-queries are an extension of the read-event and are used to retrieve sets of memory values that satisfy a condition (associative memory). In a MAS, range-queries are often used to retrieve sense data from the agent's surroundings, for example, the location of all agents in viewing range (Suryanarayanan et al. 2009).

An important part of the functionality of a DSM system is to maintain data consistency. Data consistency is maintained using a synchronisation mechanism, allowing distributed access to the data simultaneously. For distributed simulation, two overall synchronisation techniques are commonly used that should be supported by the underlying DSM system: pessimistic synchronisation where the DSM system should disallow conflicting access to the data through prediction and strict access-rules; and optimistic synchronisation where the DSM system could allow free access to the data and repairs conflicting accesses afterwards through a roll-back mechanism (Lees et al. 2003).

4 ADSMI

The main functionality the ADSMI provides is to translate state variable accesses in the MAS into events for the DSM to handle. The ADSMI does this by encapsulating the state variables of the MAS so as to intercept usage of the variables on the atomic level. Thus intercepted the ADSMI translates the interactions in the MAS to events for the DSM to handle. The declaration of a state variable in the MAS is then translated into the declaration/addition of a new memory variable in the DSM, while the instantiation or assignment of a (new) value in the MAS is translated into a write event on a memory value in the DSM. When a state variable in the MAS is used (for example during the sense-cycle of an agent) the ADSMI translates this into a read-event for the DSM. Note, however, that the translation works on the atomic level. Where state

variable accesses are combined in the MAS, these are translated into a sequence of atomic events for the DSM. For example, when a variable is assigned from another variable in the MAS, this assignment is translated into two atomic events for the DSM: one read-event to retrieve the value of the latter variable, and one write-event to assign this value to the first variable.

Although the translation of state variable interaction in the MAS into event generation for the DSM is the main function of the ADSMI, additional functionality can and must be provided as well. A MAS simulation contains a temporal aspect whereby the simulation progresses in logical time from the defined start through to the end of the simulation. The ADSMI translates this into a discrete number of time-steps from the start time-step to the final time-step. The same definition of logical time is used for the DSM system to determine memory access conflicts, for example when memory is read at one time-step but has not yet been written at an earlier time-step (straggler write in an optimistic synchronisation setting). The ADSMI should therefore also provide a mechanism – essentially a scheduler – that maintains logical time. This scheduler is then used to call the agents in the MAS to go through their sense-think-act cycles at the appropriate time-step while at the same time providing time-stamps for the DSM it provides access to the state variables stored there.

Specific, or ID-query, DSM access requires memory access by address (either through an index or directly), which requires the ADSMI to allocate identifiers to the agents. The ADSMI therefore needs to maintain an index of identifiers to both the variables and the agents in the MAS. The ADSMI needs to ensure that these identifiers, or a combination of these identifiers, are unique throughout the distributed platform.

Many MAS also use an additional mechanism of agent-interaction through the use of messaging between agents. This is often used to simulate additional ways to do sensing or agent-communication. The ADSMI should therefore be able to handle messaging between agents as well. Messaging is handled through the provision of a special type of memory access by encapsulating messages into a standard format and then reading from and writing to a special message-box variable exposed to the agent. The message-box variable is available in the MAS through special access methods which the ADSMI translates to specific memory access in the DSM.

Where the DSM uses optimistic synchronisation, the ADSMI should also provide a way to do conflict handling where this is exposed to the MAS. Conflict handling in optimistic synchronisation relies on the DSM issuing roll-backs. A roll-back is an event where for an agent or a number of agents in the MAS their time (maintained asynchronously in a distributed MAS) is reset to an earlier value (i.e., the last calculated Global Virtual Time (Fujimoto 2000)). Since the ADSMI already maintains time for the MAS, this can be handled in a way that requires little user handling. In essence, when a roll-back event is thrown by the DSM, the ADSMI resets the time for the agents involved to the time-step indicated by the DSM. If a sense-think-act cycle is in progress during the roll-back, this will need to be interrupted. The DSM handles resetting the values of the shared variables of involved agents to the correct values. The ADSMI is responsible for resetting the private variables to the correct values. After the roll-back event has been handled, the ADSMI will then resume the agent from the time provided by the roll-back. At this point, all state variables of the agents will have been reset to the state the agent was in before the conflict occurred. The MAS should only be aware of the possibility of a roll-back event and provide an elegant way of handling the sense-thing-act cycle interruption that may occur.

The ADSMI described above then provides mechanisms that can be used to design and implement a MAS so that the underlying use of the DSM is opaque to the user. It provides methods to define, instantiate, initialise, and assign both shared and private (not-shared) state variables. In addition a mechanism to control time during the simulation is provided, as is an index method to be used to identify both state variables used by the agents and the agents themselves. The ADSMI also supports MAS using messaging as an additional way for agents to interact, again using the DSM in a way opaque to the user. Where DSMs can throw roll-back events, the ADSMI provides a mechanism to reset private variables and a way of the scheduler to restart the sense-think-act cycle when appropriate.

5 PDES-MAS

In this paper we will use the PDES-MAS platform as an implementation of a DSM-based simulation infrastructure for MAS to illustrate the functionality and architecture of ADSMI.

PDES-MAS is a framework for the distributed simulation of multi-agent systems (Logan and Theodoropoulos 2001b). It is based on a DSM structure which is used to represent the shared state (the public variables, including publicly accessible attributes of agents) of the simulation. In PDES-MAS, DSM variables are represented by Shared-State Variable (SSV) data-structures which store the history of values taken by a particular variable over time (Lees et al. 2003). Following the PDES paradigm, agents modelled in the MAS are assigned to Logical Processes, known as an Agent-Logical-Process (ALP), with a single ALP potentially modelling more than one agent.

The primary philosophy of PDES-MAS is to provide multiple ALPs concurrent access to the set of SSVs in a scalable manner by a balanced distribution of SSVs around a tree-like network of server processes known as Communication Logical Processes (CLPs) as depicted in Figure 1.

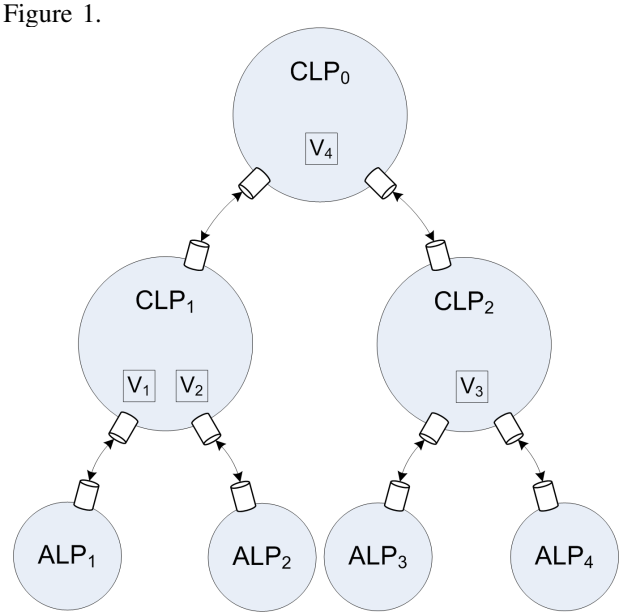

Figure 1: The PDES-MAS framework.

As in a DSM, the sensing and acting phases of the agents sense-think-act cycle in PDES-MAS are realised in terms of timestamped operations on the shared state. The ALPs interact with the shared state by reading and writing SSVs with sensing giving rise to read-events, and acting giving rise to write-events. In this context, as in the conventional DSM systems described earlier, SSVs are similar to space-time memory. ALPs link to the leaf CLP nodes in the tree. An ALP issues requests to access shared state variables through its parent CLP. If the required SSV is not held locally, the parent CLP passes the request up through the tree until it reaches its destination. The return data and other control messages are also conveyed to the ALP via its parent CLP. Conceptually an ALP in PDES-MAS then has the same functionality as the implementation of an ADSMI.

A CLP is responsible for synchronising the read/write events it receives from ALPs. PDES-MAS uses optimistic synchronisation. Synchronisation algorithms that have developed for PDES-MAS are reported in (Lees et al. (2005), Lees et al. (2006a), Lees et al. (2006b), Lees et al. (2003), Lees et al. (2004), Lees et al. (2008), Lees et al. (2009)). In general, each SSV is associated with a list of Write Periods representing the values taken by the variable at different logical times through the simulation. If a write period is subsequently invalidated by a straggler write, any ALPs which read that period must by rolled back.

In PDES-MAS, the CLP tree is reconfigured dynamically and automatically to reflect the interaction patterns between the agents and their environment (the access patterns to SSVs) so that SSVs which are accessed most frequently by a given ALP are as close as possible to that ALP in the tree. The aim is to concurrently minimise the average hops an access would take to reach an SSV and the load imbalance between CLPs. Reconfiguration of the CLP tree can be achieved by creating/deleting CLP, migrating ALP through the tree, or by migrating SSV between CLP. In the implementation of PDES-MAS used in this paper, the CLPs are configured in a fixed tree-structure with the SSVs migrated through the tree to achieve the redistribution (Oguara et al. 2005).

6 ADSMI IMPLEMENTATION

Thus far we have given a general description of a MAS simulation, and how a DSM system can be used to support the MAS. We have also given a high-level description of how an ADSMI provides the connection between these two systems. With the details on the PDES-MAS platform described in the previous section, we can now describe how the ADSMI is

implemented using the PDES-MAS platform as an implementation of a DSM system. First we give a conceptual description of the ADSMI architecture and then provide a detailed description of the implemented concepts.

6.1 ADSMI Architecture

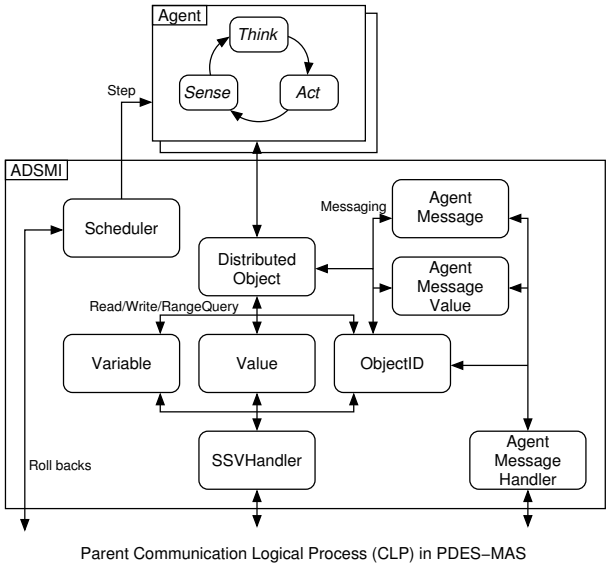

Figure 2: The ADSMI architecture.

Figure 2 depicts the implemented ADSMI architecture. Above the ADSMI itself are depicted the agents incorporating the sense-think-act cycle. The *step* method of the agents is called by the *Scheduler* and it interacts with the ADSMI through the *DistributedObject*. The *DistributedObject* uses *ObjectID*, *Variable*, and *Value* to interact with the *SSVHandler*. Interaction with the *AgentMessageHandler* requires the use of *ObjectID* again, as well as *AgentMessage*, and *AgentMessageValue*. Conceptually the ADSMI represents an ALP in PDES-MAS, and both *SSVHandler* and *AgentMessageHandler* interact directly with the parent CLP in PDES-MAS. A depiction of the ALP to CLP relationship has been given earlier in figure 1.

6.2 DistributedObject

All agents in the MAS are implemented in classes derived from *DistributedObject*. *DistributedObject* provides methods for adding, assigning, and retrieving SSVs, performing range queries over SSVs, as well as sending and receiving messages to/from other agents. These methods provide the functionality for the agents to separate their encapsulated SSVs from from other agent logic so that these can be handled by PDES-MAS. Both private as well as shared variables are handled in this way, although shared variables can be accessed through the use of a *ObjectID* identifier of an agent, where as private variables can only use the agent's own identifier.

DistributedObject has a one-to-one association with *Statebase* where private variables are stored and where access to SSVs and messages is handled. *DistributedObject* also handles creating and maintaining the agent's unique *ObjectID* identifiers and this too is stored in *Statebase*. Because *Statebase* stores the private variables, it also provides functionality to restore the private state of *DistributedObject* (and consequently the agent in the MAS) when a roll-back call is passed from PDES-MAS. To this end, *Statebase* maintains a time-stamped list of private variables.

Determining whether a variable is private or shared is done though annotating the variables and reflection in *Statebase*. Access calls to SSVs pass from *DistributedObject* and through *Statebase* to the *SSVHandler* which interfaces directly with PDES-MAS. *Statebase* also contains a message-queue implementation of a message-box, available through *DistributedObject*. *AgentMessageHandler* maintains the message-box through a direct interface with PDES-MAS and also handles sending messages.

Every time-step the agent as a sub-class from *DistributedObject* is called by the *Scheduler* to perform its sense-think-act cycle. The logic for this cycle is implemented in the *step* method, which is an overriden method from *DistributedObject*. It

is the logic implemented in the *step* method that uses the functionality thus far described. The contract with the *Scheduler* assures that each agent's *step* method is called once for each time-step of the MAS.

The *DistributedObject* can thus be seen as providing the interface between the MAS and the ADSMI.

6.3 ObjectID

Each agent in the MAS is identified through a unique identifier and this identifier is implemented in the *ObjectID* class. *ObjectID* is instantiated upon creation of *DistributedObject* and stored in *Statebase*. Uniqueness of *ObjectID* is guaranteed in a distributed environment by using a combination of the node identifier and a unique running counter of all created *DistributedObjects* on that node.

6.4 Variable and Value

Each variable type in the MAS needs to be uniquely identifiable across the distributed platform, with different *DistributedObject* able to share SSV types by combining them with unique *ObjectID*. A MAS therefore defines a static set of variable types. These variable types are exposed through a java `enum` (for enumerate) implementing the *Variable* interface. Each variable type is associated with a value type (integer, double, String, etc.) and reflection is used to verify the correct value type is used whenever a variable is accessed. The *Variable* interface also associates a unique variable identifier to each variable type. The variable identifier is used in PDES-MAS and is guaranteed to be unique. The combination of a unique *ObjectID*, and a unique *Variable* identifier, allows unique access to SSVs in PDES-MAS. *Variables* are annotated to be private or shared using the *PrivateVariable* and *PublicVariable* annotations. These annotations are used in *Statebase* so that it can handle shared and private variables differently.

Variables that can be used in range-queries are required to implement the *RangeQueryable* interface. Currently only the *Location* variable type can be used in range-queries. An example of this would be a *Variable* called *LOCATION* of variable type *Location*. As *Location* implements the *RangeQueryable* interface, an agent in the MAS can issue a range-query over the *LOCATION* variable of all agents in the MAS that has such a variable. The response would be a map of *ObjectID* with *Location Value* wrappers indicating which agents are located where within the range of the range-query. Range-query responses can be filtered through the use of the *CLASS* variable. This allows the range-query to focus on one type or class of agents in the MAS only. The *CLASS* variable is the only variable required by the ADSMI. The *CLASS* variable is however not different from other variables the MAS might define, and is a normal SSV and that has the *PublicVariable* annotation.

The value associated with each *Variable* in the ADSMI is wrapped in a *Value* object. The *Value* wrapper is used for all variable access, i.e., retrieving a *Variable* will return a *Value* object that encapsulates the retrieved value, while assigning a *Value* to a *Variable* requires wrapping the value into *Value* object. The *Value* object provides methods to retrieve the value type of the encapsulated value so that it can be verified with the value type of the *Variable* accessed. The *Value* class uses defensive copying to prevent pointer tampering with encapsulated values and this requires the construction of a new wrapper if a new value is required.

6.5 AgentMessage and AgentMessageValue

DistributedObject provides messaging functionality through the use of the *AgentMessage* and *AgentMessageValue* objects. The *AgentMessage* object is used as a type identifier of the message but also encapsulates fields for the source and destination *ObjectID* of the messages, and the time-stamp when the message was send. The actual message, or payload, is encapsulated in the *AgentMessageValue*, which like *Value* is a wrapper for different types of message values. Delivering messages encapsulated in the *AgentMessage* is handled by the *AgentMessageHandler* with the *DistributedObject* providing access to a message-box maintained in its *Statebase*.

6.6 Scheduler

The *Scheduler* maintains a list of all *DistributedObjects* handled on the local node. The combined lists of all *Schedulers* in the distributed environment then holds all *DistributedObjects*, i.e., agents, in the MAS. The main functionality of the *Scheduler* is to call the *step* method overridden by the agents in the MAS. The *DistributedObject* list is updated through addition and deletion buffers with updates occurring in between time-steps of the simulation. The *Scheduler* also contains functionality that allows polling and analysis of the agents in the MAS. This functionality can be used for logging and result-collection of

the state of the MAS. Access to the *DistributedObject* is gained through the *SchedulerListener* interface by using the *update* method provided by the *Scheduler*. Multiple listeners can be registered to each *Scheduler* on the distributed platform.

6.7 AgentMessageHandler

The *AgentMessageHandler* deals with sending and receiving messages between agents. The distribution of these messages is effected through an interface to PDES-MAS with messages effectively stored in specific SSVs in PDES-MAS itself. Messages send by a *DistributedObject* are send through PDES-MAS and stored in the mailbox SSV of the destination *DistributedObject*. A *DistributedObject* needs its *Statebase* to implement the *AgentMessageHandlerListener* interface to make use of the message-passing functionality. The interface provides hooks for the *AgentMessageHandler* and the *DistributedObject* that are used for polling, sending, and receiving messages.

6.8 SSVHandler

The *SSVHandler* provides an interface with PDES-MAS for retrieving, writing, and-range querying SSVs. These discrete events are translated into PDES-MAS calls with the use of the *ObjectID*, *Variable*, and *Value* objects. In addition, the *SSVHandler* provides unique *ObjectID* objects for *DistributedObject* to use and store in *Statebase*. Calls to *SSVHandler* are passed from the agents in the MAS through *DistributedObject* to *Statebase* where all relevant information is collected and method calls to *SSVHandler* are placed.

7 CONCLUSIONS

MAS are increasingly used to solve larger and more complex problems. Distributed simulation is emerging as the only viable approach to cope with the scale and complexity of MAS simulations. Different approaches for distributing MAS over multiple computational platforms have been proposed over the years. One such approach uses a DSM system to partition the shared state of the MAS over the distributed computational platform. Whatever the approach used to distribute the MAS, the problem remains on how to translate MAS behaviour into a format suitable for the distribution approach used.

This paper describes a way to distributed MAS simulations over a DSM infrastructure through an interface called the Agent Distributed Shared Memory Interface (ADSMI). The ADSMI provides a bridge for translating MAS behaviours into events usable for a DSM system. Events such as retrieval of, assignment to, and range-querying over SSVs in the DSM are supported as well as functionality for maintaining logical time for both the MAS and the DSM. Additionally, the ADSMI provides a mechanism for agents to send messages to each other.

Although the ADSMI is applicable as an interface to many DSM systems, this paper provides the description of an implementation of the ADSMI where PDES-MAS is used as a DSM implementation. The ADSMI implementation places minimal restrictions on the MAS however. This allows the ADSMI to be used for a large variety of MAS. As such, the ADSMI – PDES-MAS combination provides an adaptable, efficient, and light-weight solution for designing and implementing different types of MAS.

The ADSMI implementation for PDES-MAS described in this paper, is currently being deployed in a MAS simulation of medieval military logistics, as part of the MWGrid Project at the University of Birmingham (Murgatroyd et al. 2010). Future work includes providing a functionality and performance description of this use-case.

REFERENCES

Fujimoto, R. M. 2000. *Parallel and distributed simulation systems*. New York, NY, USA: John Wiley & Sons Inc.

Ghosh, K., and R. M. Fujimoto. 1991. Parallel discrete event simulation using space-time memory. In *Proceedings of the International Conference on Parallel Processing, Volume III, Algorithms & Applications*, 201–208.

Lees, M., B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos. 2005, Oct. Decision-theoretic throttling for optimistic simulations of multi-agent systems. In *Proceedings of the Ninth IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2005)*, ed. A. Boukerche, S. J. Turner, D. Roberts, and G. Theodoropoulos, 171–178. Montreal, Quebec, Canada: IEEE Press.

Lees, M., B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos. 2006a, Oct. Analysing probabilistically constrained optimism. In *Proceedings of the 10th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2006)*, ed. E. Alba, S. J. Turner, D. Roberts, and S. J. Taylor, 201–208. Malaga, Spain: IEEE Press.

Lees, M., B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos. 2006b. Analysing the performance of optimistic synchronisation algorithms in simulations of multi-agent systems. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS06)*, 37–44. Washington, DC, USA: IEEE Computer Society.

Lees, M., B. Logan, and G. Theodoropoulos. 2003. Adaptive optimistic synchronisation for multi-agent simulation. In *Proceedings of the 17th European Simulation Multiconference (ESM 2003)*, ed. D. Al-Dabass, 77–82. Delft: Society for Modelling and Simulation International and Arbeitsgemeinschaft Simulation, Society for Modelling and Simulation International.

Lees, M., B. Logan, and G. Theodoropoulos. 2004, Sep. Time windows in multi-agent distributed simulation. In *Proceedings of the 5th EUROSIM Congress on Modelling and Simulation (EuroSim04)*. Paris.

Lees, M., B. Logan, and G. Theodoropoulos. 2008. Using access patterns to analyze the performance of optimistic synchronization algorithms in simulations of mas. *Transactions of the Society for Computer Simulation International* 84:481–492.

Lees, M., B. Logan, and G. Theodoropoulos. 2009, Aug. Analysing probabilistically constrained optimism. *Concurrency and Computation: Practice and Experience Journal, special issue on Distributed Simulation, Virtual Environments and Real Time Applications* 21:1467–1482.

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

Logan, B., and G. Theodoropoulos. 2001b, Feb. The distributed simulation of multi-agent systems. In *Proceedings of the IEEE*, Volume 89, 174–186.

Mehl, H., and S. Hammes. 1993. Shared variables in distributed simulation. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS93)*, 68–75.

Murgatroyd, P., V. Gaffney, B. Craenen, G. Theodoropoulos, V. Suryanarayanan, and J. Haldon. 2010, Mar. Logistics: A case of distributed agent-based simulation. In *Proceedings of the Distributed Simulation & Online Gaming Conference (DISIO 2010)*. Torremolinos, Spain: ACM Digital Library.

Oguara, T., D. Chen, G. Theodoropoulos, B. Logan, and M. Lees. 2005, Oct. An adaptive load management mechanism for distributed simulation of multi-agent systems. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2005)*, ed. A. Boukerche, S. J. Turner, D. Roberts, and G. Theodoropoulos, 179–186. Montreal, Quebec, Canada: IEEE Press.

Suryanarayanan, V., R. Minson, and G. Theodoropoulos. 2009, Oct. Synchronised range queries in distributed simulations of multi-agent systems. In *13th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2009)*. Singapore.

Terracotta Inc. retrieved Jul 2010. Terracotta 3.3-beta documentation. Technical report, http://www.terracotta.org/documentation/.

Theodoropoulos, G., R. Minson, R. Ewald, and M. Lees. 2007. Simulation engines for multi-agent systems. *Multi-Agent Systems: Simulation and Applications*:77–108.

Wooldridge, M., and N. R. Jennings. 1995. Intelligent agents: Theory and practice. *Knowledge Engineering Review* 10:115–152.

AUTHOR BIOGRAPHIES

BART G.W. CRAENEN is currently a Research Fellow in the School of Computer Science at the University of Birmingham (UK). He received his B.Sc and M.Sc in Computer Science from Leiden University (NL) and his Ph.D in Computer Science and Artificial Intelligence from Vrije Universiteit Amsterdam (NL). His current research focuses on parallel and distributed simulation, distributed virtual environments and the use of Computational Intelligence in these systems. His email address is <B.G.W.Craenen@cs.bham.ac.uk>

GEORGIOS K. THEODOROPOULOS is currently a Reader in the School of Computer Science at the University of Birmingham (UK). He received a Diploma in Computer Engineering from the University of Patras (Greece) and an M.Sc and Ph.D. in Computer Science from the University of Manchester (UK). His current research is in the areas of parallel and distributed simulation, distributed virtual environments, Grid computing and Peer-to-Peer systems. His email address is <G.K.Theodoropoulos@cs.bham.ac.uk>