Distributed Simulation of MAS

Michael Lees[1], Brian Logan[1], Rob Minson[2], Ton Oguara[2], and Georgios Theodoropoulos[2]

[1] School of Computer Science and IT
University of Nottingham, UK
{mhl|bsl}@cs.nott.ac.uk

[2] School of Computer Science
University of Birmingham, UK
{txo|rzm|gkt}@cs.bham.ac.uk

Abstract. The efficient simulation of multi-agent systems presents particular challenges which are not addressed by current parallel discrete event simulation (PDES) models and techniques. While the modelling and simulation of agents, at least at a coarse grain, is relatively straightforward, it is harder to apply PDES approaches to the simulation of the agents' environment. In conventional PDES approaches a system is modelled as a set of logical processes (LPs). Each LP maintains its own portion of the state of the simulation and interacts with a small number of other LPs. The interaction between the LPs is assumed to be known in advance and does not change during the simulation. In contrast, the environment of a MAS is read and updated by agent and environment LPs in ways which depend on the evolution of the simulation. As a result, MAS simulations typically have a large *shared* state which is not associated with any particular agent or environment LP. In [1] we proposed a new approach to the distributed simulation of MAS in which the shared state is maintained by a tree of additional logical processes called Communication Logical Processes (CLP). In this paper we refine this model by giving precise definitions of a set of operations which allow agent and environment LPs to interact with the shared state and briefly outline how these operations could be implemented by a CLP.

1 Introduction

Simulation has traditionally played an important role in multi-agent system (MAS) research and development. It 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, freeing the agent designer or researcher to focus on key aspects of a system. As researchers have attempted to simulate larger and more complex MAS, distributed approaches to simulation have become more attractive [2–4]. Such approaches simplify the integration of heterogeneous agents and exploit the natural parallelism of a MAS, allowing simulation components to be distributed so as to make best use of the available computational resources.

However the efficient simulation of a multi-agent system presents particular challenges which are not addressed by current parallel discrete event simulation (PDES) models and techniques. While the modelling and simulation of agents, at least at a coarse grain, is relatively straightforward, it is harder to apply conventional PDES approaches to the simulation of the agents' environment. Parallel discrete event simulation approaches based on the logical process paradigm assume a fixed decomposition into processes, each of which maintains its own portion of the state of the simulation. The interaction between the processes is assumed to be known in advance and does not change during the simulation. In contrast, simulations of MAS typically have a large *shared* state, the agents' environment, which is only loosely associated with any particular process. The efficient simulation of systems with a large shared state is therefore a key problem in the distributed simulation of MAS.

In [1] we proposed a new approach to the distributed simulation of MAS in which the shared state is maintained by a tree of additional logical processes called Communication Logical Processes (CLP). In this paper we refine this model by giving precise definitions of a set of operations which allow agent and environment logical processes to interact with the shared state and briefly outline how these operations could be implemented by a CLP. In section 2, we present a model of a MAS as a set of logical processes and argue that MAS simulations naturally result in systems with a large shared state. In section 3 we briefly describe our approach to the efficient distribution of the shared state across a tree of CLPs and define a set of operations which allow agent and environment logical processes to access and update the shared state maintained by the CLPs. We then sketch how these operations could be implemented by a CLP, paying particular attention to the problems of efficient sensing, parallel actions and action conflicts. In section 4 we discuss related work and in section 5 we conclude with a brief outline of future work.

2 Modelling a MAS

We adopt a standard parallel discrete event approach with optimistic synchronisation [5, 6]. Decentralised, event-driven distributed simulation is particularly suitable for modelling systems with inherent asynchronous parallelism, such as agent-based systems. This approach seeks to divide the simulation model into a network of concurrent *Logical Processes* (LPs), each maintaining and processing a disjoint portion of the state space of the system. State changes are modelled as timestamped events. Internal events have a causal impact only on the state variables of the LP, whereas external events may also have an impact on the states of other LPs. External events are typically realised as timestamped messages exchanged between the LPs involved.

Agents are *autonomous*. The actions performed by an agent are not solely a function of events in its environment: in the absence of input events, an agent can still produce output events in response to autonomous processes within the agent. As a result, agent simulations have zero lookahead [7]. We therefore adopt an optimistic synchronisation strategy as this theoretically gives the greatest speedup and avoids the problem of lookahead. With optimistic synchronisation, LPs run asynchronously and each has its own local notion of time within the simulation, referred to as its *Local Virtual Time* (LVT). In distributing the simulation across multiple processes a key problem is ensuring that there are no causality violations. An LP is said adhere to the *local causality constraint* (LCC) if it processes all events in nondecreasing time stamp order. If a message arrives in an LP's past (as determined by its LVT) it must rollback its state to the timestamp of the straggler event, and resume processing from that point. It must also cancel any messages it sent with timestamps greater than that of the straggler event, which may in turn initiate rollbacks on other LPs.

We model agents and their environment as Logical Processes. Each agent in the system is modelled as a single *Agent Logical Process* (ALP). Similarly, the properties and behaviour of the objects comprising the agents' environment, e.g., walls, doors, light switches, clocks, etc. and processes not associated with any particular object in the environment, e.g., weather, are modelled as one or more *Environment Logical Processes* (ELP). For example, in a simple Tileworld simulation [8], each Tileworld agent would be simulated by an ALP and the objects in the Tileworld environment (tiles, holes, obstacles etc.) by one or more ELPs.[1] In addition to creating the objects in the environment at simulation startup, the ELP(s) would also be responsible for the creation and deletion of tiles and holes during the simulation. ALPs and ELPs are typically wrappers around existing simulation components. They map to and from the sensor and action interfaces of the agent and environment models to a common representation of the environment expressed in terms of entities and attributes, and also provide support for rollback processing. In what follows we shall use the generic term 'LP' to refer to both ALPs and ELPs, since, unless otherwise noted, their behaviour is very similar.

Each LP has both public data and private data. Private data is data which is not accessible to other LPs in the simulation, e.g., an agent's model of the environment, its current goals, plans etc. or the internal state of a complex object. Public data is data which can, in principle, be accessed or updated by other LPs in the simulation, e.g., the colour, size, shape, position etc. of an agent or object. Public data is held in globally accessible locations or *state variables*, while private data is local to a particular LP. Access to public data and/or the ability to update it may be restricted particular groups of LPs. For example, it may be impossible for any LP to change the size or colour of objects in the environment or for ALPs to update the position of some objects such as obstacles. We model the public data of the LPs in terms of *entities* and *attributes*. We assume each entity in the simulation (agent or object) has a type, and each entity type is associated with a number of attributes. For example, a Tileworld simulation might contain entity types such as *agent*, *tile*, *hole* and *obstacle* and attributes such as *x-position*, *y-position* etc. The shared state of the simulation would therefore consist of a variable number of entities (agents, tiles, holes obstacles etc.) whose properties are defined by the value of their attributes.

In a conventional decentralised event-driven distributed simulation each LP maintains its own portion of the simulation state and LPs interact with each other in a small number of well defined ways. The topology of the simulation is determined by the topology of the simulated system and its decomposition into LPs, and is largely static. In contrast, with multi-agent systems, public data is updated by many LPs and is not logically associated with any of them. Different kinds of agent and environment pro-

[1] Tileworld is a well established testbed for agents. It consists of an environment consisting of tiles, holes and obstacles, and one or more agents 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

cesses have differing degrees of access to different parts of the environment at different times. In the case of agents, the degree of access is dependent on the range of the agent's sensors (read access) and the actions it can perform (write access). Moreover, in many cases, an agent can effectively change the topology of the simulation, for example, by moving from one part of the environment to another. It is therefore difficult to determine an appropriate topology for a MAS simulation *a priori*, and such simulations typically require a (very) large set of shared variables which could, in principle, be accessed or updated by the ALPs and ELPs.

3 Distributing the shared state

We therefore propose an approach in which the shared state is loosely associated with a group of special LPs, namely Communication Logical Processes (CLPs), and the distribution of state (i.e., its allocation to CLPs) changes at run-time, in response to the events generated by the ALPs and ELPs during the simulation. Both the allocation of state to CLPs and the synchronisation window are driven by an underlying characteristic of the agent simulation, which we call the sphere of influence [1]. In the Tileworld example above, public data such as the positions of the agents and objects in the environment (tiles, holes and obstacles), the height of the tilestacks, depth of the holes, etc. would be maintained by the CLPs.

ALPs and ELPs interact with the shared state maintained by the CLPs via events, implemented as timestamped messages. The purpose of this interaction is to exchange information regarding the values of those shared state variables which can be accessed or updated by the agent's sensors and actions or by environment processes. Different types of events will typically have different effects on the shared state, and, in general, events of a given type will affect only certain types of state variables (all other things being equal). The 'sphere of influence' of an event is the set of state variables read or updated as a consequence of the event. We can use the spheres of influence of the events generated by each LP to derive an idealised decomposition of the shared state into logical processes (see [1] for details).

3.1 CLPs

The CLPs form a tree with the ALPs and ELPs as the leaves and each CLP maintains a subset of the shared state which is associated with the ALPs/ELPs which are below it in the tree (see Figure 1). CLPs also hold partial information on attributes of entities maintained by other CLPs in the tree, to allow routing of events to the appropriate CLP.

ALPs and ELPs interact with CLPs by exchanging messages. There are 5 message types:

add one or more attributes (and their initial values) at a given timestamp;
remove one or more attributes from a given timestamp;
read the value of one or more attributes at a given timestamp;
write the value of one or more attributes at a given timestamp; and
rollback and resume processing from a given timestamp.

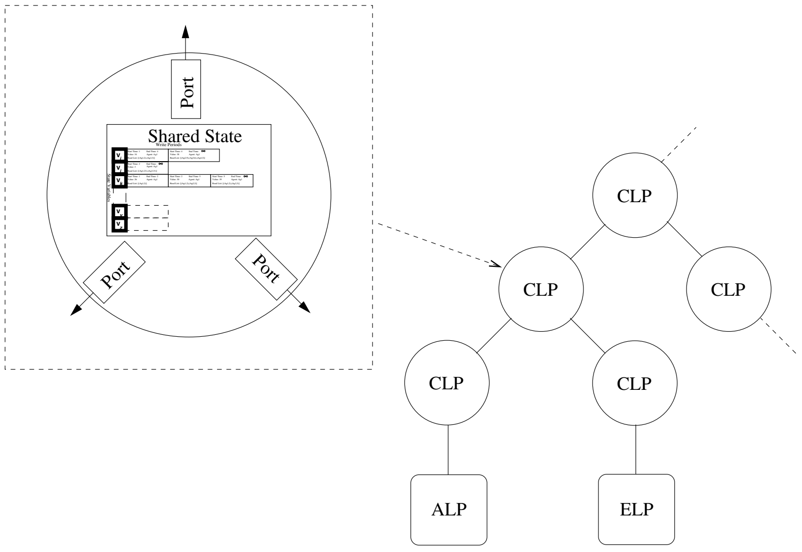

Fig. 1. The tree of CLPs

add, *remove*, *write* and *rollback* are non-blocking. A *read* blocks until the requested values are returned. Add, remove, read and write messages originate with an ALP or ELP, while rollbacks are initiated by a CLP. All operations on the shared state occur asynchronously and at the specified simulation time. We assume that the operations are atomic and may be arbitrarily interleaved.[2]

In the remainder of this subsection, we consider each message type in turn and briefly describe their arguments, possible reply messages and any side-effects on the shared state and the state of other LPs. We consider first the case in which the message argument(s) are maintained by the ALP's/ELP's parent CLP. In section 3.2 we describe how messages which can't be handled by the parent CLP are propagated through the tree.

Add messages When an ALP or ELP creates a new entity in the simulation its parent CLP adds a new variable to the shared state to hold the value of the attribute. The timestamp indicates the simulation time at which the attribute of the new entity acquired the specified value. Adding the first attribute to an entity instance implicitly creates the entity in the shared state. For simplicity, we assume that entities are only ever created in their entirety, i.e., we cannot create an entity without specifying all values for all its attributes.

[2] The distinction between read and write operations is similar to the query event tagging proposed in [9] and should have similar advantages in reducing both the frequency and depth of rollback and the state saving overhead.

Remove messages Removing an attribute of an entity in effect deletes the attribute from the specified time forward. Subsequent read and write operations on the attribute with timestamps prior to the specified timestamp proceed as normal. Reads with timestamps later than the specified timestamp give rise an empty list of values. Attempting to add a new attribute with a timestamp greater than the specified timestamp has no effect (i.e., it is not possible to recreate an attribute after it has been removed from the simulation). As with creation, we assume that entities are only ever deleted in their entirety.

Read messages To sense the environment an ALP or ELP it must issue a state query. A *state query* is either a range query (query by attribute value) or an id query (query by attribute id). A *range query* is a list of 4-tuples of the form:

$$< \textit{entity-type}, \textit{attribute-type}, \textit{value-range}, \textit{timestamp} >$$

where the *value-range* indicates the attribute values which are of interest (i.e., that match the query). Range queries allow sensing such as 'all tile x-positions within 5 squares'. For example, in a Tileworld simulation, an ALP simulating an agent may issue a range query to discover which tiles are within the sensor range of the agent. Similarly, an ELP responsible for the creation of tiles within a particular region of the Tileworld may issue a range query to check that the cell in which a new tile is to be placed is not currently occupied by an agent (or by a tile created by another ELP and pushed into this region of the Tileworld by an agent).

An id query is a list of 2-tuples of the form:

$$< \textit{attribute-id}, \textit{timestamp} >$$

Id queries allow query by reference, for example, it allows an ALP or ELP to obtain the current value of one of its own public attributes or the current value of an attribute returned by a range query. They are provided as an optimisation for those cases where the attribute in question is guaranteed to persist until after the timestamp of the query.

Reads give rise to a read-response message containing a (possibly empty) set of values (in the case of range queries), or, in the case of an attribute query, a single value. The values returned are those which were valid at the time denoted by the query timestamp. If there is no value with a timestamp equal to that of the query, for example, if the query timestamp lies between the timestamps of two values or is greater than the timestamp of any matching attribute, the read returns the value with the greatest timestamp prior to the timestamp of the query.

Write messages When an ALP or ELP updates an attribute of an existing entity, it sends a write message to its parent CLP with a new value and timestamp, indicating the simulation time at which the attribute acquired the specified value. Attribute values are stored in write periods of the appropriate state variable. A *write period* is a logical time interval during which an attribute maintains a particular value. Each write period stores its start and end time, the value of the attribute over that time period, the LP which performed the write and the timestamp of the most recent read by each LP which read the attribute over the time period.[3] New write periods are created when an LP updates the value of an attribute. This splits an existing write period, and triggers a rollback on any LPs which read the previous value of the variable at a logical time between the start and end times of the new write period (see below).

In general, there will be a delay between an agent's sensing and action. It is therefore impossible for an LP to know that the state of the environment it sensed before performing an action still holds when the write is performed. We therefore allow write operations to be guarded. A *guard* is a predicate on the shared state in the form of a list of attribute values which must evaluate to true (i.e., the attributes must have the specified values at the timestamp of the write) for the operation to be performed. A guard functions as the precondition for the successful execution of an action in the environment. If the guard evaluates to false, the write is not performed (with the exception that we ignore violations of the precondition due to writes performed by the same agent at the same timestamp). For example, to prevent two (or more) agents pushing the same tile at the same time in Tileworld, we can require that the tile is still where the agent sensed it (e.g., directly in front of the agent) before allowing the agent to update the position of the tile. All writes also have an additional implicit guard, namely that the attribute being updated has not been removed at a timestamp prior to the write.

We distinguish different categories of attributes depending on the types of updates they admit [10]. *Static attributes* are set once, e.g., when an entity is created, and can't be changed during the simulation. Attributes which can be updated at most once at a given timestamp are termed *mutually exclusive attributes*. For example, in Tileworld, we may wish to prohibit two agents picking up a tile at the same time. *Cumulative attributes* can be updated at most n times by different LPs at the same timestamp. For example, in the Tileworld, several agents may be able to drop a tile into a hole at the 'same' time, with each operation decreasing the depth of the hole by one. All updates of static attributes are ignored. If two or more LPs attempt to perform conflicting updates, e.g., attempt to specify different values for a mutually exclusive attribute at the same timestamp or attempt to drop a tile into a hole that has already been filled by other agents at the same timestamp, we apply the update of the LP with the highest rank. The *rank* of an LP determines it's priority when attribute updates conflict. Ranks may reflect some property of the LP which is relevant to the simulation, but in general are simply a way of ensuring repeatability. If both LPs have the same rank then we choose an update arbitrarily (saving the random seed to preserve repeatability). If the attribute has already been updated at this timestamp by an LP with lower rank, this value is over-written and any LPs which read the previous value are rolled back (see below).

More complex environment models can be implemented using combinations of these features. For example, with an appropriate choice of guard on a cumulative attribute, we can allow several agents to push a tile at the same time to give motion which is, e.g., the vector sum of the motion imparted by each agent. Alternatively, an entity's motion can be computed by the ALP or ELP responsible for maintaining the entity in

[3] In practice, not all write periods need to be stored in state variables, e.g., if a write period has a timestamp lower the LVT of any LP it is inaccessible within the simulation and can be fossil collected.

the simulation, with each agent and object updating an input force vector represented as a cumulative attribute.

Rollback messages Some sequences of operations by the LPs give rise to further processing of the shared state and the private state of one or more LPs.

An add, remove or write operation with timestamp t which is processed in real time after a read with timestamp t_r, where $t < t_r$, invalidates the read, and triggers a rollback on all LPs which read the previous (interpolated) value of the attribute. A *rollback* indicates that the set of values returned in response to the read was incorrect, and that the LP should rollback its processing to the timestamp of the read and restart.[4] Rolling back an LP undoes all the updates to the LP's private state which have a timestamp $> t_r$ and resets the LP's LVT to t_r. The effect is as if the LP had just returned from the original read (at timestamp t_r), but this time with the 'correct' values of the attributes. (A subsequent add, remove or write with timestamp t', where $t' < t < t_r$ can of course cause further rollbacks on the LP.) Rolling back an LP also cancels any add, remove or write operations on the shared state performed by the LP which have a timestamp $> t_r$. This may in turn invalidate reads by other LPs, requiring them to rollback too.

Note that the presence of rollback obviates the need for coarse-grain atomic operations, i.e., each attribute update can be processed independently of any others and may be arbitrarily interleaved with other operations such as read operations. It is therefore possible for an LP to 'see' an inconsistent version of the shared state or for the guard conditions of a write to evaluate to true for some orderings of operations on the shared state and false for others. When all the updates are finally made, the inconsistency will be detected and any affected LPs rolled back.

3.2 Ports

Each CLP holds only part of the shared state. Read and write operations on shared state variables not maintained by a CLP are forwarded through the tree to the relevant CLP(s).

CLPs communicate with their neighbours in the tree via ports. Each *port* holds information about the ranges of attribute values maintained by CLPs beyond the port in the form of 4-tuples:

$$< \textit{entity-type}, \textit{attribute-type}, \textit{value-range timestamp-range} >$$

For example, in a Tileworld simulation, a port tagged with *entity-type tile*, *attribute-type x-position value-range* 10–20 and *timestamp-range* 50–100 would indicate that state variables holding x positions of tiles with values in the range 10 to 20 and timestamps between 50 and 100 are held in CLPs beyond this port. (Where the port leads to an ALP

[4] Note that a write with timestamp t_w which arrives in real time after a write with timestamp t'_w, where $t_w < t'_w$ and there are no intervening reads, does not trigger a rollback. In contrast to standard optimistic synchronisation approaches which rollback on every straggler event, or which only avoid rollbacks on straggler reads [9], this optimisation results in a significant reduction in the number of rollbacks [11].

or an ELP, the port information is empty, since all public information in the simulation is held in the CLPs).

In the case of range queries, the query is compared against the range information for each port. If the ranges overlap the CLP forwards the query to the CLP beyond the port. This process proceeds recursively until a CLP with no ports (as opposed to maintained state variables) that match the query is reached. Each CLP waits until it receives replies from all CLPs to which it forwarded the query, appends the value of any state variables it manages that match the query and sends a reply to the originating CLP. When the replies reach the root CLP for this query, the sensing is complete and the values matching the query can be returned to the requesting ALP/ELP.

Initially, the *value-range* for each entity and attribute type at each port is "all values" for all timestamp ranges and all queries are forwarded to all neighbouring CLPs. By analysing the responses to range queries by the neighbouring CLPs, a CLP acquires information about the kinds of attributes (and their ids) that lie beyond each port. This provides a simple form of 'lazy' interest management, and avoids repeated traversal the whole tree when sensing the environment. In addition, each port also holds information about the attribute instances maintained by other CLPs that can be reached via the port. This routing information allows a CLP to forward reads and writes of particular attributes that it does not maintain to the appropriate CLP.

Updating the value of an attribute may involve updating the range information of the ports leading to the CLP which manages the associated state variable. Each CLP keeps a record of all queries it has received together with the port through which the query arrived at the CLP. All add operations are checked against this query history, and, if the new attribute value matches a previously evaluated query, the add is propagated back along the path of the query to update the port information. When the traversal reaches the ALP that initiated the query this triggers a rollback, as the first time the query was evaluated, it returned too few values. Conversely, if an attribute value matches no query in the query record, then no ALP has ever queried this attribute value at this timestamp, and there is no need to propagate the value beyond the current CLP.

3.3 Load balancing

As well as storing state variables and enabling communication via ports, CLPs also facilitate load balancing. As the number of instances of each event type generated by an ALP or ELP varies, so the partial order over the spheres of influence changes, and the contents of the CLPs must change accordingly to reflect the LPs' current behaviour and keep the communication and computational load balanced. This may be achieved in two ways, namely by swapping pairs of ALPs/ELPs, and by moving subsets of state variables from one CLP to another. In general, it is easier to move state than LPs, and our strategy is to bring the environment close (in a computational sense) to the LPs within whose sphere of influence the corresponding portion of the shared state lies. For example, in a Tileworld simulation, the state associated with entities currently being sensed or manipulated by an agent would ideally be located on the parent CLP of the ALP responsible for simulating the agent. As the agent moves around the Tileworld, the state maintained by the ALP's parent CLP (and its parent CLPs in turn) should change to reflect the agent's changing sphere of influence.

Periodically, the CLPs offer to swap state variables with their neighbours. A CLP will offer to swap a state variable if doing so will reduce the total cost of access. In order to calculate the cost, each query carries with it the ‘distance’ it travelled through the tree before reaching the CLP. The hop counts for queries arriving through each of the CLP’s ports are totalled for each variable maintained by the CLP, and this information is used to determine which port (i.e., neighbouring CLP) to swap with. For example, if the majority of accesses to a state variable arrive through a particular port, a CLP may offer to swap the variable with the CLP which can be reached via the port.

4 Related work

There is a considerable amount of work in the simulation literature on the efficient distribution of updates, particularly in the context of large scale real-time simulations where it is termed *Interest Management*. Interest Management techniques utilise filtering mechanisms based on *interest expressions* (IEs) to provide the processes in the simulation with only that subset of information which is relevant to them (e.g., based on their location or other application-specific attributes). Special entities in the simulation, referred to as *Interest Managers*, are responsible for filtering generated data and forwarding it to the interested processes based on their IEs [12].

In most existing systems, Interest Management is realised via the use of IP multicast addressing, whereby data is sent to a selected subnet of all potential receivers. A multicast group is defined for each message type, grid cell (spatial location) or region in a multidimensional parameter space in the simulation. Typically, the definition of the multicast groups of receivers is static, based on a priori knowledge of communication patterns between the processes in the simulation [13]. For example, the High Level Architecture (HLA) utilises the *routing space* construct, a multi-dimensional coordinate system whereby simulation federates express their interest in receiving data (subscription regions) or declare their responsibility for publishing data (update regions) [14]. In existing HLA implementations, the routing space is subdivided into a predefined array of fixed size cells and each grid cell is assigned a multicast group which remains fixed throughout the simulation; a process joins those multicast groups whose associated grid cells overlap the process subscription region.

Static, grid-based Interest Management schemes have the disadvantage that they do not adapt to the dynamic changes in the communication patterns between the processes during the simulation and are therefore incapable of balancing the communication and computational load when the communication patterns change, with the result that performance is often poor. Furthermore, in order to filter out all irrelevant data, grid-based filtering requires a small cell size, which in turn implies an increase in the number of multicast groups, a limited resource with high management overhead.

In contrast, our approach is not confined to grids and rectangular regions of multidimensional parameter space and does not rely on the support provided by the TCP/IP protocols. Rather, the shared state is distributed dynamically based on the spheres of influence of the ALPs and ELPs in the simulation. In addition, our approach exploits this decomposition in order to perform load balancing.

5 Conclusion and Further Work

In this paper we have argued that the efficient simulation of the environment of a multi-agent system is a key problem in the distributed simulation of MAS. Building on work in [1], we proposed an approach in which the shared state of a simulation is loosely associated with a group of special logical processes called Communication Logical Processes, and the distribution of state (i.e., its allocation to CLPs) is performed dynamically in response to the events generated by the agent and environment processes during the simulation. We defined a set of operations on the shared state which allow the interaction of agent and environment logical processes and sketched how these operations could be implemented by a CLP. Our approach addresses the problems of efficient sensing, parallel actions and action conflicts, and integrates an efficient approach to state saving which minimises the number of rollbacks with a simple load balancing scheme.

The work reported is still at a preliminary stage. To date, we have implemented the core of the CLPs including the rollback mechanism and calculation of virtual time [15] and load balancing [16] and are currently working on the implementation of interest management. Initial experiments with the rollback mechanism are encouraging, and show a reduction in the number of rollbacks compared to other approaches in the literature which rollback on every straggler event [17].

Acknowledgements

This work is part of the PDES-MAS project[5] and is supported by EPSRC research grant No. GR/R45338/01.

References

1. Logan, B., Theodoropoulos, G.: The distributed simulation of multi-agent systems. Proceedings of the IEEE **89** (2001) 174–186
2. Anderson, J.: A generic distributed simulation system for intelligent agent design and evaluation. In Sarjoughian, H.S., Cellier, F.E., Marefat, M.M., Rozenblit, J.W., eds.: Proceedings of the Tenth Conference on AI, Simulation and Planning, AIS-2000, Society for Computer Simulation International (2000) 36–44
3. Schattenberg, B., Uhrmacher, A.M.: Planning agents in JAMES. Proceedings of the IEEE **89** (2001) 158–173
4. Gasser, L., Kakugawa, K.: MACE3J: Fast flexible distributed simulation of large, large-grain multi-agent systems. In: Proceedings of AAMAS-2002, Bologna (2002)
5. Ferscha, A., Tripathi, S.K.: Parallel and distributed simulation of discrete event systems. Technical Report CS.TR.3336, University of Maryland (1994)
6. Fujimoto, R.: Parallel discrete event simulation. Communications of the ACM **33** (1990) 31–53
7. Uhrmacher, A., Gugler, K.: Distributed, parallel simulation of multiple, deliberative agents. In: Proceedings of Parallel and Distributed Simulation Conference (PADS'2000). (2000) 101–110

[5] http://www.cs.bham.ac.uk/research/pdesmas

8. Pollack, M.E., Ringuette, M.: Introducing the Tileworld: Experimentally evaluating agent architectures. In: National Conference on Artificial Intelligence. (1990) 183–189
9. Sokol, L.M., Briscoe, D.P., Wieland, A.P.: MTW: A strategy for scheduling discrete simulation events for concurrent simulation. In: Proceedings of the SCS Multiconference on Distributed Simulation. SCS Simulation Series, Society for Computer Simulation (1988) 34–42
10. Minson, R., Theodoropoulos, G.: Distributing RePast agent-based simulations with HLA. In: Proceedings of the 2004 European Simulation Interoperability Workshop, Edinburgh, Simulation Interoperability Standards Organisation and Society for Computer Simulation International (2004) (to appear).
11. Lees, M., Logan, B., Minson, R., Oguara, T., Theodoropoulos, G.: Distributed simulation of MAS. (In: Proceedings of the Joint Workshop on Multi-Agent and Multi-Agent-Based Simulation (MAMABS'04))
12. Morse, K.L.: Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27 (1996)
13. Morse, K.L.: An Adaptive, Distributed Algorithm for Interest Management. Ph.D. thesis, University of California, Irvine (2000)
14. Defence Modeling and Simulation Office: High Level Architecture RTI Interface Specification, Version 1.3. (1998)
15. Lees, M., Logan, B., Theodoropoulos, G.: Adaptive optimistic synchronisation for multi-agent simulation. In Al-Dabass, D., ed.: Proceedings of the 17th European Simulation Multiconference (ESM 2003), Delft, Society for Modelling and Simulation International and Arbeitsgemeinschaft Simulation, Society for Modelling and Simulation International (2003) 77–82
16. Oguara, T.: Load balancing in distributed simulation of agents. Thesis Report 5, School of Computer Science, University of Birmimgham (2004)
17. Lees, M., Logan, B., Theodoropoulos, G.: Time windows in multi-agent distributed simulation. In: Proceedings of the 5th EUROSIM Congress on Modelling and Simulation (EuroSim'04). (2004)