Modelling Environments for Distributed Simulation

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. Decentralised, event-driven distributed simulation is particularly suitable for modelling systems with inherent asynchronous parallelism, such as agent-based systems. However the efficient simulation of multi-agent systems presents particular challenges which are not addressed by standard parallel discrete event simulation (PDES) models and techniques. PDES 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 fixed 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. In this paper, we present a model of the shared state of a distributed MAS simulation of situated agents. We consider the problems of efficient sensing, parallel actions and action conflicts, and present preliminary work on an approach to the simulation of the environment which addresses these issues.

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 [1–3]. Such approaches simplify the integration of heterogeneous agent simulators 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 multi-agent systems presents particular challenges which are not addressed by standard parallel discrete event simulation (PDES) models and techniques [4, 5]. 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 fixed 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. At different times, different agents can access and update different parts of the shared state. The efficient simulation of the environment of a multi-agent system is therefore a key problem (perhaps even *the* key problem) in the distributed simulation of MAS.

In this paper, we present a model of the shared state of a distributed MAS simulation. We consider the problems of efficient sensing, parallel actions and action conflicts, and present preliminary work on an approach to the simulation of the environment which addresses these issues. The remainder of the paper is organised as follows. In section 2, we briefly outline a model of a MAS as a set of logical processes and explain why MAS simulations naturally result in a large shared state. In section 3 we present a model of the shared state as a global tuple space and describe how the agents' sensing and actions in the environment can be modelled as operations on the tuple space. In section 4 we describe an approach to the efficient distribution of the shared state and briefly describe a prototype implementation of this approach based on Communication Logical Processes. In section 5 we discuss related work and in section 6 we conclude with some remarks on the relationship between our approach and the requirements of distributed environments for MAS in general.

2 Modelling Multi-Agent Systems

In this section, we outline our model of the agents and their environment.

We are primarily concerned with the simulation of *situated agents* [6], e.g., simulations of agents such as robots situated in a physical environment, or characters in a computer game or interactive entertainment situated in a virtual environment. The systems of interest typically involve large numbers (thousands or tens of thousands) of agents in complex environments, e.g., individual-based ecological modelling or simulations of massively multi-player online games, and the "agents" that we wish to simulate may be models of agents (e.g., DEVS models [7]), or they may be actual implemented agents in a simulated environment, or a mixture of the two. We therefore view the agents as 'black boxes' and focus on the interaction of the agents through the medium of their shared environment.

We adopt a standard parallel discrete event approach with optimistic synchronisation [4, 5]. 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. The LPs run asynchronously and each has its own local notion of time within the simulation, referred to as its *Local Virtual Time* (LVT). State changes are modelled as timestamped events. From an LP's point of view, two types of events are distinguished: internal events which have a causal impact only on the state variables of the LP, and external events which may also have an impact on the states of other LPs. External events are typically modelled as timestamped messages exchanged between the LPs involved. In distributing the simulation across multiple processes, a key

problem is ensuring that there are no causality violations. An LP is said to 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) and objects and processes within the agents' environment are modelled as one or more *Environment Logical Processes* (ELP)[3]. 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 objects and attributes, and also provide support for rollback processing.

In general, the agents' environment can be decomposed into ELPs in a number of different ways. For example, the blocks in a simple 'blocks world' environment could each be modelled as a separate ELP, as could the physics of stacking blocks etc. Alternatively, all the blocks could form part of a single 'blocks system' ELP. The appropriate 'grain size' of the simulation will depend both on the application and on practical considerations, such as the availability of existing simulation code. While there are obvious advantages in reusing part or all of an existing simulation, this can result in an inappropriate grain size which makes it difficult to parallelise the model. For example, modelling the environment as a single logical process can create a bottleneck in the simulation which degrades its performance.[4] The approach presented below is neutral with respect to the decomposition of the environment into processes.

Each ALP and ELP 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. 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 object or agent. Public data is held in globally accessible locations or *state variables*, while private data is local to a particular LP. ALPs and ELPs interact via events, modelled as timestamped messages. The purpose of this interaction is to exchange information regarding the values of those shared state variables which define the agent's manifest environment and the interfaces between the ELPs.[5]

There are several ways in which this interaction could be managed. One approach would be to adopt a subscription-based approach to sensing, where the agent, via its sensors, implicitly indicates the kind of data it is interested in, and data which matches the subscription is sent to the agent whenever the environment changes. However there are a number of problems with this approach. If the agent senses less frequently than the

[3] For simplicity, we do not consider fine-grained modelling of processes within an agent, i.e., distributing the agent model across multiple LPs.

[4] Existing attempts to build distributed simulations of agent based systems have often adopted such a centralised approach in which the agents' environment forms part of a central time-driven simulation engine [8, 9, 1].

[5] 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.

environment changes, this needlessly propagates information to the agent. Moreover, with optimistic synchronisation, environmental updates propagated to the agent may be in its past or future. To receive only data with the "correct" timestamp, the agent's subscription must include the agent's LVT and the subscription must be continuously updated as the agent advances in time or rolls back. We therefore adopt a query based approach to sensing, and use other techniques (see below) to reduce the cost of querying the shared state.

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. Even if the interactions are stochastic, the type of interaction and its possible outcomes are known in advance. 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, the interaction of agents in a multi-agent system is often hard to predict in advance. Different kinds of agent have differing degrees of access to different parts of the environment at different times. 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 environment, for example, by moving from one part of the environment to another.[6] For example, if an agent is "mobile", then what it can sense at different times is a function of the actions it performed in the past which is in turn a function of what it sensed in the past. As a result, it is difficult to predict which state variables it can or will access without running the simulation.

It is therefore difficult to determine an appropriate topology for a MAS simulation *a priori*. As a result, MAS simulations typically require a (very) large set of shared variables which could, in principle, be accessed or updated by the agents (if they were in the right position at the right time etc.).

3 Modelling the Shared State

We model the state of the simulation in terms of objects and attributes. We assume each object in the simulation has a type, and each object type is associated with a number of attributes. For example, a simple Tileworld [10] simulation might contain object types such as *tile* and *hole* and attributes *x-position*, *y-position* etc. The simulation consists of a variable number of objects whose state is defined by the value of their attributes. Events generated by LPs read and write attribute values. Each attribute has a timestamp which indicates the time at which the attribute acquired the value. The values of attributes can be set independently of each other and at different times, i.e., updates to the environment do not have to specify values for all the attributes of an object. The global state of the simulation is split into the shared state: i.e., those attributes which

[6] It may be the case that, at any particular time, there are parts of the environment that are not accessible to any agent. However, if there is no sequence of actions that any agent can perform from the initial state which makes some data accessible, then this data does not form part of the shared state as defined here.

are accessible to more than one LP, and the local state of each LP (which for ease of exposition we assume to be also modelled in terms of objects and attributes).

We represent the simulation state as a set of tuple spaces. All LPs can access a global tuple space containing the shared state of the simulation. The global tuple space consists of a set of 6-tuples:

$$< \mathit{object\text{-}type}, \mathit{object\text{-}id}, \mathit{attribute\text{-}type}, \mathit{attribute\text{-}id}, \mathit{value}, \mathit{timestamp} > .$$

For example, the fact that a tile in the Tileworld has an x-position of 5 at time 25 might be represented

$$< \mathit{tile}, \text{tile101}, \mathit{x\text{-}position}, 101001, 5, 25 > .$$

As the simulation progresses, new tuples are added to the shared state, either because a new object (and its corresponding attributes and values) has been created, or because one of the LPs comprising the simulation has changed the value of an attribute of an object. Note that the shared state may contain different values for the same attribute so long as these have different timestamps. In addition, each LP has its own private tuple space containing the private state of the LP.

LPs can perform a number of operations on the global tuple space:

request the value(s) of one or more attributes with a given timestamp;
add the value(s) of one or more attributes at a given timestamp; and
remove one or more attributes from a given timestamp.

add and *remove* operations are non-blocking. A *request* blocks until the requested tuples are returned. Operations can also give rise to 'exceptions' which indicate that it was impossible to complete the requested operation on the shared state. All operations occur asynchronously and at the specified simulation time. However problems can arise when an operation is performed in real time after another operation on the same attribute which has a later timestamp, resulting in further processing of the global tuple space and the private tuple space of one or more LPs. Such causality violations are a standard problem with optimistic synchronisation approaches and our solution is discussed in more detail below. The operations are atomic and may be arbitrarily interleaved. As a convenience, the operations accept multiple arguments, but the processing of arguments may be interleaved with other operations. As we will show, so long as the processing of each argument is atomic, correct behaviour is guaranteed.

In the remainder of this section, we consider each operation in turn and briefly describe their arguments, return values, exceptions and any side-effects on the shared state and the state of other LPs.

3.1 Requests

For an LP to sense the world it firstly constructs a *state query*. The state query consists of a query id and a set of query tuples. A query tuple 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:

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

The *value-range* indicates the attribute values which are of interest (i.e., that match the query). Range queries allow sensing of the environment. For example, to find the x-positions of all tiles within 5 squares of an agent at time 50, we could use the range query

$$< \textit{tile}, \textit{x-position}, a_x - 5 \leq x \leq a_x + 5, 50 > ,$$

where a_x is the *x-position* of the agent at time 50.

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 LP 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.

Requests can give rise to a (possibly empty) set of tuples (in the case of range queries), or, in the case of an attribute query, a single tuple or a “no such attribute” exception. The tuple(s) contain the value(s) of the requested state variable(s) which were valid at the time denoted by the request timestamp. If there is no tuple with a timestamp equal to that of the request, for example, if the request timestamp lies between the timestamps of two tuples or the query timestamp is greater than the timestamp of any matching tuple, the request returns the tuple with the greatest timestamp prior to the timestamp of the request. For example, if agent 1 has an *x-position* of 10 at time 50, evaluating the range query above against the tuples

$$< \textit{tile}, \text{tile101}, \textit{x-position}, 101001, 6, 40 >$$
$$< \textit{tile}, \text{tile101}, \textit{x-position}, 101001, 7, 52 >$$

would return the tuple

$$< \textit{tile}, \text{tile101}, \textit{x-position}, 101001, 6, 40 > .$$

3.2 Add

When an LP creates a new object in the simulation or updates an attribute of an existing object, it adds a new tuple to the shared state with the new value and timestamp, indicating the simulation time at which the object was created or the attribute acquired the specified value. Add operations are non blocking and do not return a result. However they may give rise to an exception if objects of the specified *object-type* can’t have attributes of the specified *attribute-type*. For simplicity, we assume that objects are only ever created or deleted in their entirety, i.e., we cannot create an object without specifying all values for all its attributes. Adding the first attribute to an object instance implicitly creates the object in the shared state.

Assuming agents execute a sense, think, act cycle, an update will occur after the range query *request*(s) generated by sensing. The agent (or ELP) will therefore have a list of the sensed attributes and their ids, which can be used to construct the new tuple.

We assume that there is a delay between an agent's sensing and action. In general, it is impossible for an LP to know that the state of the environment that led to an *add* operation still holds when the operation is performed. We therefore allow *add* operations to be guarded. A *guard* is a predicate on the shared state in the form of a list of tuples which must be true (i.e., the attributes must have the specified values at the timestamp of the *add*) for the operation to be performed. A guard is effectively the precondition for the successful execution of an action in the environment. If the guard evaluates to false, the *add* operation is not performed (with the exception that we ignore violations of the precondition due to add operations 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) at the time of a push action before allowing the agent to update the position of the tile.

We distinguish different categories of attributes depending on the types of updates they admit. *Static attributes* are set once, e.g., when an object 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 multiple 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, i.e., attempt to specify different values for a mutually exclusive attribute at a given 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).

3.3 Remove

Removing an attribute of an object in effect deletes the attribute from the specified time forward. Subsequent request and add operations on the attribute with timestamps prior to the specified timestamp proceed as normal. Range queries with timestamps later than the specified timestamp give rise an empty list of result tuples. 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 object id after it has been removed from the simulation). As with creation, we assume that objects are only ever deleted in their entirety, with all attributes being deleted at the same timestamp.

3.4 Rollbacks

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 or remove operation with timestamp t which is processed in real time after a request with timestamp t', where $t' > t$ invalidates the request operation, and triggers a rollback on all LPs which read the previous (interpolated) value of the attribute. A *rollback* indicates that the set of tuples returned in response to the request was incorrect, and that the LP should rollback its processing to the timestamp of the request and restart. Rolling back an LP removes all tuples from the LP's private tuple space which have a timestamp $> t'$ and resets the LP's LVT to t'. The effect is as if the LP had just returned from the original request operation (at timestamp t'), but this time with the 'correct' value of the attribute. For example, if agent 2 moves tile101 at time 47 so that it's *x-position* is now 5 but the tuple recording the update is not added to the global tuple space until after the range query by agent 1 in section 3.1 has been performed, then agent 1 must be rolled back to time 50 and restarted, returning from the *request* with the tuple

$$< \mathit{tile}, \text{tile101}, \mathit{x\text{-}position}, 101001, 5, 47 > .$$

A subsequent add operation with timestamp t'', where $t'' < t < t'$ can of course cause further rollbacks on the LP. Rolling back an LP also cancels any add operations on the shared state performed by the LP which have a timestamp $> t'$. This may in turn invalidate requests made 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 tuple argument to an add operation can be processed independently of any others and may be arbitrarily interleaved with other operations such as request operations.[7] It is therefore possible for an LP to "see" an inconsistent version of the shared state or for the guard conditions of an add operation to evaluate to true for some orderings of operations on the shared state and false for others. When the updates are finally made, the inconsistency will be detected and any affected LPs rolled back.

4 Distributing the Shared State

A naive implementation of the shared state, e.g., in which the shared state is maintained by a single process, is a potential bottleneck in a MAS simulation. In this section we present an approach to the efficient distribution of the shared state.

The shared state of the simulation is stored in *state variables*. Each state variable corresponds to a set of tuples, namely those that have the same *object-type*, *object-id*, *attribute-type* and *attribute-id*.[8] We assume that each LP is capable of generating and responding to a finite number of event types, and a specification of the possible input and output event types forms the interface between the LPs. 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). For example, in [11], we showed that for a simple predator and prey simulation, the probability of a given state variable being accessed by more than 3 agents was fairly small, and agents tend to access the same state over time.

[7] While this isn't a correctness issue, it may be an efficiency issue.

[8] In practice, not all tuples need to be stored in state variables, e.g., if a tuple has a timestamp lower the LVT of any LP it is inaccessible within the simulation and can be garbage collected.

Another way of expressing this is to say that different types of event have different *spheres of influence* within the shared state. 'Sphere' is used here metaphorically, to indicate those parts of the shared state immediately affected by an instance of an event of a particular type with a given timestamp. More precisely, we define the 'sphere of influence' of an event as 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 [11] for details). We define the sphere of influence of an LP p_i over the time interval $[t_1, t_2]$, $s(p_i)$, as the union of the spheres of influence of the events generated by the LP over the interval. Intersecting the spheres of influence for each event generated by the LP gives a partial order over sets of state variables for the LP over the interval $[t_1, t_2]$, in which those sets of variables which have been accessed by the largest number of events come first, followed by those less frequently accessed, and so on. The rank of a variable v_j for LP p_i over the interval $[t_1, t_2]$, $r(v_j, p_i)$ is the number of events in whose sphere of influence v_j lies.

Intersecting the spheres of influence for each LP gives a partial order over sets of state variables, the least elements of which are those sets of state variables which have been accessed by the largest groups of LPs over the interval $[t_1, t_2]$. This partial order can be seen as a measure of the difficulty of associating variables with a particular ALP or ELP: the state variables which are members of the sets which are first in the order are accessed by the largest number of ALPs and/or ELPs, whereas those sets of state variables which come last are accessed by only a single LP. (Assuming that all variables have the same rank.)

For example, suppose there are three ALPs, a_1, a_2 and a_3, and five variables, $v_1, \ldots, v_5$. The first ALP generates events which read and update only the variables v_1 and v_2; its sphere of influence therefore is $\{v_1, v_2\}$. Similarly, let the sphere of influence of a_2 be $\{v_2, v_3\}$ and the sphere of influence of a_3 be $\{v_4, v_5\}$. The variable v_2 is accessed by two agents, hence $\{v_2\}$ is the least in the ordering, followed by $\{v_1\}, \{v_3\}$ and $\{v_4, v_5\}$.

To minimise the computational and communication loads, any approach to the decomposition of the shared state into logical processes should, insofar as is possible, reflect this ordering and grouping of variables. In the example above, we would expect the state to be partitioned into $\{v_1\}, \{v_3\}, \{v_4, v_5\}$ and $\{v_2\}$. $\{v_1\}, \{v_3\}$ and $\{v_4, v_5\}$ can be located close (in a computational sense) to the ALPs in whose sphere of influence the variables lie, i.e., a_1, a_2 and a_3 respectively. $\{v_2\}$ is shared by a_1 and a_2 and should be allocated to an LP which is equidistant (in a computational sense) from a_1 and a_2. However, any implementation can only approximate this idealised decomposition, since calculating it requires information about the global environment, and obtaining this information in a distributed environment is costly. Moreover, this ordering will change with time, as the state of the environment and the relative number of events of each type produced by the LPs changes.

In the remainder of this section, we describe a prototype implementation of these ideas which distributes the state according to the spheres of influence of the LPs in the

simulation. The partitioning of the shared state is performed dynamically, in response to the events generated by the ALPs and ELPs during the simulation.

4.1 CLPs

The decomposition of the state is achieved by means of an additional set of Logical Processes, namely *Communication Logical Processes* (CLPs). The CLPs form a complete binary tree with the ALPs and ELPs as the leaves and each CLP maintains a portion of the state which is associated with the ALPs/ELPs which are below it in the tree (see Figure 1).

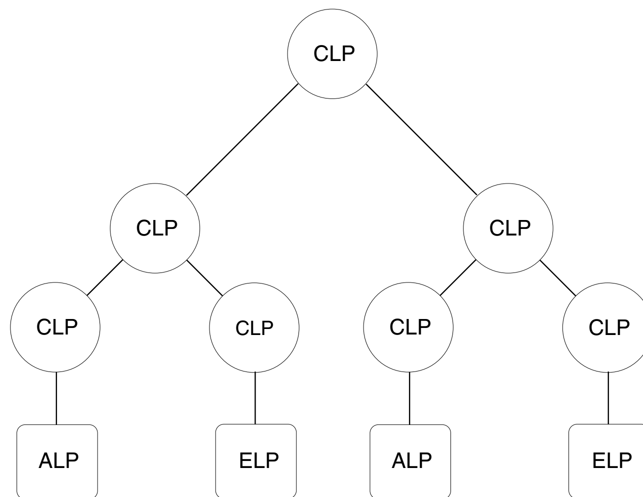

Fig. 1. The tree of CLPs

At any given point in the simulation, each CLP maintains a disjoint subset of the state variables and the interaction of ALPs and ELPs is via the variables maintained by the CLPs. In general, different CLPs will maintain different numbers of state variables. The aim is to minimise both computational and communication loads. Frequently accessed data should therefore be maintained by CLPs close to the ALPs which access it. The tree of CLPs provides a large number of local caches for data accessed by a single agent small groups of agents. Less frequently accessed data can be stored further away. In the limit, the root node may hold, e.g., 90% of the shared state (swapped out) — so long as this is never or very infrequently accessed, the load on the root node may be similar to that on (leaf) CLPs with small numbers of frequently accessed variables.

Read and write operations on state variables are effectively mapped into *request* and *add* operations on tuples in the global tuple space. Each tuple corresponds to a *write period* of the appropriate state variable (see Figure 2). A write period is an interval during which an attribute maintains a particular value. Each write period stores its start and end

time, the value of the state variable over that time period, the LP which performed the write and a list of LPs which read the state variable over the time period, together with the logical times at which they read the variable. New write periods are created when an LP performs an *add* operation, i.e., when the state variable concerned is written to. This splits an existing write period, and triggers a rollback on any LPs which read the previous version of the variable at a logical time later than the start of the new write period (see [12] for details).

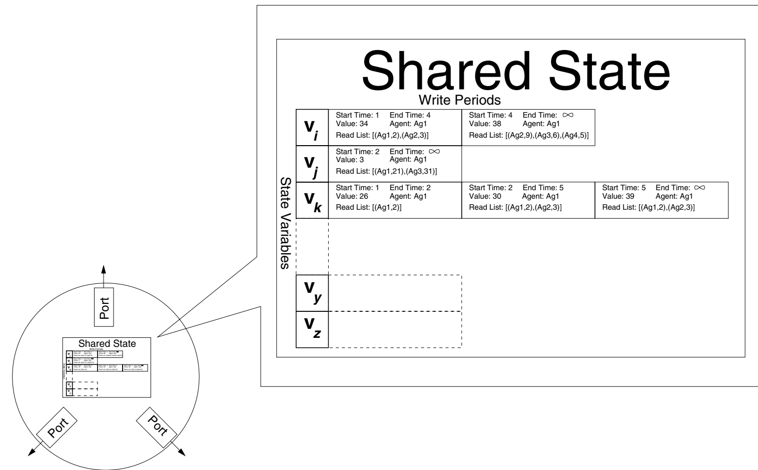

Fig. 2. The structure of a CLP

4.2 Ports

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:

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

For example, in a Tileworld simulation, a port tagged with *object-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. Initially, the *value-range* for each object and attribute type at each port is “all values” for all timestamp ranges. As range queries are processed (initially by forwarding the query to all CLPs in the tree), a CLP acquires information about the kinds of attributes that lie beyond each port by analysing the responses to the range query by the neighbouring CLPs. This provides a simple form of ‘lazy’ interest management, which avoids repeated traversal the whole

tree when processing *request*s, e.g., when an agent repeatedly senses 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 is cached during the processing of range queries, and allows a CLP to forward reads and writes of state variables that it does not maintain to the appropriate CLP. Where the port leads to an ALP or an ELP, the port information is empty (since all public information in the simulation is held in the CLPs).

Updating the value of a state variable may involve updating the range information of the ports leading to the CLP which manages the variable. Each CLP keeps a record of all queries it has received since the last GVT computation together with the port through which the query arrived at the CLP. All add operations are checked against this query history, and, if the tuple 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 tuples.[9] Conversely, if a tuple 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 tuple beyond the current CLP.

4.3 Load Balancing

As well as storing state variables and enabling communication via ports, the CLPs also facilitate load balancing. As the total number and distribution of instances of each event type generated by an ALP/ELP varies, so the partial order over the spheres of influence changes, and the contents of the CLPs must change accordingly to reflect the ALPs/ELPs' current behaviour and keep the computational and communication loads balanced. This may be achieved in two ways, namely by changing the position of the ALPs/ELPs, and by moving the state variables from one CLP to another. In general, it is easier to migrate the shared state than the agents, and our strategy is to bring the environment close to the agents (in a computational sense).

To achieve this we have developed a load balancing scheme in which the cost of accessing state variables maintained at each CLP is used in making load management decisions. We define the cost of accessing a state variable v_j by an ALP/ELP p_i as the number of times v_j was accessed by p_i times the number of CLPs that must be traversed to reach v_j during the time interval $[t_1, t_2]$. The access cost for a CLP is therefore:

$$\sum_j \sum_i (r(v_j, p_i) \times l(v_j, p_i))$$

where $r(v_j, p_i)$ is the number of accesses by each ALP/ELP p_i to each variable v_j maintained by the CLP, and $l(v_j, p_i)$ is the number of CLPs traversed to reach v_j from p_i. Periodically, each CLP chooses a set of state variables it maintains, which, if migrated to a neighbouring CLP, would reduce the total access cost at the originating CLP. This simple load shedding scheme is complemented by a "reverse migration" phase which ensures that the load on any single CLP does not exceed a maximum value.

[9] Note that remove operations do not require special processing: the removed tuple must have been read by the query and the reading ALP is recorded in the write period.

State variables are considered for migration when their access cost over the last period is greater than a preset threshold value. (With a cost threshold of zero, all variables maintained at the CLP are potential migration candidates.) Once the set of migration candidates has been determined, the CLP chooses to which of the neighbouring CLPs each migration candidate should be pushed (if any). For a variable to be pushed to a neighbouring CLP, the number of accesses to the state variable arriving through the port leading to the neighbouring CLP must exceed the total number of accesses to the variable arriving through other ports by a predetermined access threshold. To avoid the oscillation of a state variable between the two CLPs, the CLP initiating the load balancing process must check that the cost of accessing a state variable at its new location will be reduced (assuming that the pattern of accesses in the future is similar to that in the present), before pushing the load.

A state variable which satisfies both these conditions is migrated to the CLP responsible for the majority of the accesses to the variable over the last period. This simple strategy guarantees that the computational load on the "pushing" CLP and the overall communication load on the system will be lower following migration. However it can result in excessive computational loads on the "receiving" CLPs. To avoid this, we allow the receiving CLP to "swap" load with the pushing CLP. If the difference between the computational load of the pushing and receiving CLPs is greater than a predetermined load threshold, the receiving CLP may chose to swap some state variables with the pushing CLP. However, the receiving CLP will only chose to a swap state variable if doing so will reduce the cost of accessing that variable. A state variable is selected for swapping if and only if the majority of its accesses are through the port from which the pushed load was received. We use a swap load selection criterion in which the difference between the swap load (total access on the selected swappable variables) and the initial push load is less than or equal to the load threshold.

When load balancing is performed, the range information for the port through which the state is migrated must be updated to record the fact that attribute values held in the pushed variables are now accessible via the port. In addition, for the receiving CLP to correctly process new tuples in future, it must know what assumptions other CLPs would otherwise make about its contents based on previous queries. Any queries which match the state being pushed must therefore be copied from the pushing CLP to the receiving CLP.

5 Related Work

The model of the shared state presented above has some similarities with tuple space-based approaches [13]. For example, the *add* operator is similar to the *out* operator and the *request* and *remove* operators are similar to non-blocking *rdp* and *inp* operators. There has also been considerable work on distributed tuple spaces, for example systems such as LIME [14] and EgoSpaces [15], support distribution and the propagation of tuples from one tuple space to another.

However there are important differences. In Linda, matching is only on the position and type of a field (though some tuple space approaches, e.g., [16, 14, 15], support simple range matching in templates). Nor is it possible to guard an *out* operation, or

easily construct such an atomic operation from the existing primitives. The key difference, however, is the model of time. In PDES, operations occur asynchronously but at a specific virtual time. As a result, we have to deal with conflicting updates with the same timestamp. We also have to manage the relationship between virtual and real time, for example, recording which request operations have been performed so that we can detect straggler updates and rollback. In contrast, coordination languages don't have an explicit model of time built into the semantics. Some implementations, e.g., [16, 17], support leases and/or transactions, but these are insufficient to implement guarded updates and rollback. In [18] a framework is proposed which allows, e.g., timestamping, rollback, and atomic transactions (as user-defined operations), but as far as we are aware, no distributed implementation exists.

In addition to the differences in the operations supported by existing tuple space models, scalability is also an issue. For example, LIME uses a subscription-like model to implement query operations on (remote) tuple spaces, in which the middleware registers a 'weak reaction' on a remote tuple space which is triggered when a tuple matching a pattern is added to the remote tuple space. However this approach potentially requires a weak reaction to be registered with every remote tuple space for every query. In EgoSpaces, which in part builds on the work on LIME, 'network constraints' limit consideration to "nearby" tuple spaces, i.e., to a subnet of the network. While such a network metric has a natural interpretation in the ad hoc mobile environments for which EgoSpaces was developed, there is no obvious corresponding metric in a MAS simulation, and it is not clear that the LIME/EgoSpaces model would scale to the very large numbers of tuple spaces required for a large MAS simulation, where, in principle, any agent may sense and update any part of its environment.

The approach described above also has some similarities with the ant algorithm approach outlined in [19] and with the TOTA middleware described in [20]. In [19] templates and tuples are modelled as ants which search a landscape of tuple servers for matching tuples or templates respectively, leaving trails to the locations for successful matches. In TOTA, the propagation of tuples from tuple space to tuple space across a network is determined by propagation rules which form part of the tuple's definition. This approach is very flexible and can be used to implement some of the features described above. For example, in TOTA, tuples can be used to create a routing overlay structure in a way somewhat similar to the caching of port information by a CLP during the processing of a range query. However, in TOTA, routing has to be programmed at the user level using propagation rules, and TOTA provides no direct support for guarded updates, virtual time or rollback.

In the simulation community, the efficient distribution of updates has received more attention, 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 [21].

Various Interest Management schemes have been devised, utilising different communication models and filtering schemes. 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 [22–26]. 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) [27]. 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, with the result that performance is often poor. Furthermore, in order to filter out all irrelevant data, grid-based filtering requires a reduced cell size, which in turn implies an increase in the number of multicast groups, a limited resource with high management overhead. Some systems, such as JPSD [24] and STOW-E [28] allowed a degree of dynamism in their filtering schemes, and, more recently, there have been attempts to define alternative dynamic schemes for Interest Management concentrating mainly on the dynamic configuration of multicast groups within the context of HLA. For example, Berrached et al. [29] examine hierarchical grid implementations and a hybrid grid/clustering scheme of update regions to dynamically reconfigure multicast groups while Morse et al. [30] report on preliminary investigations on a dynamic algorithm for dynamic multicast grouping for HLA. The Joint MEASURE system [31–33] is implemented on top of HLA and utilises event distribution and predictive encounter controllers to efficiently manage interactions among entities. However, despite these efforts, the problem of dynamic interest management remains largely unsolved.

In contrast, our approach is not confined to grids and rectangular regions of multi-dimensional 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 aims to exploit this decomposition in order to perform load balancing. Although load balancing has been studied extensively in the context of conventional distributed simulations [34–39], it has received very little attention in relation to Interest Management, and work in this area to date is only preliminary [21, 40–42].

6 Conclusion and Further Work

In this paper, we have presented a model of the environment of a multi-agent system and an approach to the distributed simulation of MAS environments based on this model.

Our model addresses the problems of efficient sensing, parallel actions and action conflicts, and the efficient distribution of the resulting shared state of the simulation. 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 [43] and load balancing [44] 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 [12].

In the short term, our focus will be on completing the implementation and evaluating its performance relative to conventional PDES approaches. However in the longer term, it would be interesting to explore the application of the ideas described above to MAS environments in general. With the exception of our model of time (which is local to the agent), the concerns which we address, i.e., the tuple space-like model of the environment state, the operations we can perform on it (and the associated issues of simultaneity and guarded updates) and its efficient distribution, are relevant to environments for MAS in general. Much of the work on modelling of state and action and, at the implementation level, on distribution and routing could potentially carry over to the modelling and implementation of (non-simulated) environments.

Acknowledgements

This work is part of the PDES-MAS project[10] and is supported by EPSRC research grant No. GR/R45338/01.

References

1. 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
2. Schattenberg, B., Uhrmacher, A.M.: Planning agents in JAMES. Proceedings of the IEEE **89** (2001) 158–173
3. Gasser, L., Kakugawa, K.: MACE3J: Fast flexible distributed simulation of large, large-grain multi-agent systems. In: Proceedings of AAMAS-2002, Bologna (2002)
4. Ferscha, A., Tripathi, S.K.: Parallel and distributed simulation of discrete event systems. Technical Report CS.TR.3336, University of Maryland (1994)
5. Fujimoto, R.: Parallel discrete event simulation. Communications of the ACM **33** (1990) 31–53
6. Ferber, J.: Multi-Agent Systems. Addison Wesley Longman (1999)
7. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. 2nd edn. Academic Press (2000)
8. Baxter, J., Hepplewhite, R.T.: Broad agents for intelligent battlefield simulation. In: Proceedings of the 6th Computer Generated Forces and Behavioural Representation, Institute of Simulation and Training (1996)

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

9. Vincent, R., Horling, B., Wagner, T., Lesser, V.: Survivability simulator for multi-agent adaptive coordination. In: Proceedings of the International Conference on Web-Based Modeling and Simulation 1998 (WMC'98). (1998)
10. Pollack, M.E., Ringuette, M.: Introducing the Tileworld: Experimentally evaluating agent architectures. In: National Conference on Artificial Intelligence. (1990) 183–189
11. Logan, B., Theodoropoulos, G.: The distributed simulation of multi-agent systems. Proceedings of the IEEE **89** (2001) 174–186
12. 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)
13. Carriero, N., Gelernter, D.: Linda in context. Communications of the ACM **32** (1989) 444–458
14. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the the 21st International Conference on Distributed Computing Systems (ICDCS 2001), IEEE Computer Society (2001) 524–533
15. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. SIGSOFT Softw. Eng. Notes **27** (2002) 21–30
16. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T Spaces. IBM Systems Journal **37** (1998) 454–474
17. Sun Microsystems: JavaSpaces service specification v1.1. http://www.sun.com/software/jini/specs/js1_1.pdf (2000) (verified 01/04/2004).
18. Merrick, I., Wood, A.: Coordination with scopes. In: Proceedings of the 2000 ACM Symposium on Applied Computing, ACM Press (2000) 210–217
19. Menezes, R., Tolksdorf, R.: A new approach to scalable lind-sysatems based on swarms. In: Proceedings of the 2003 ACM Symposium on Applied Computing, ACM Press (2003) 375–379
20. Mamei, M., Zambonelli, F., Leonardi, L.: Tuples on the air: A middleware for context-aware computing in dynamic networks (2003)
21. Morse, K.L.: Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27 (1996)
22. Smith, J., Russo, K., Schuette, L.: Prototype multicast IP implementation in ModSAF. In: Proceedings of the Twelfth Workshop on Standards for the Interoperability of Distributed Simulations. (1995) 175–178
23. Mastaglio, T.W., Callahan, R.: A large-scale complex virtual environment for team training. IEEE Computer **28** (1995) 49–56
24. Macedonia, M., Zyda, M., Pratt, D., Barham, P.: Exploiting reality with multicast groups: a network architecture for large-scale virtual environments. In: Virtual Reality Annual International Symposium. (1995) 2–10
25. Calvin, J.O., Chiang, C.J., Van Hook, D.J.: Data subscription. In: Proceedings of the Twelfth Workshop on Standards for the Interoperability of Distributed Simulations. (1995) 807–813
26. Steinman, J.S., Weiland, F.: Parallel proximity detection and the distribution list algorithm. In: Proceedings of the 1994 Workshop on Parallel and Distributed Simulation. (1994) 3–11
27. Defence Modeling and Simulation Office: High Level Architecture RTI Interface Specification, Version 1.3. (1998)
28. Van Hook, D., Calvin, J., Newton, M., Fusco, D.: An approach to DIS scaleability. In: Proceedings of the 11th Workshop on Standards for the Interoperability of Distributed Simulations. (1994) 347–356
29. Berrached, A., Beheshti, M., Sirisaengtaksin, O., de Korvin, A.: Alternative approaches to multicast group allocation in HLA data distribution. In: Proceedings of the 1998 Spring Simulation Interoperability Workshop. (1998)

30. Morse, K.L., Bic, L., Dillencourt, M., Tsai, K.: Multicast grouping for dynamic data distribution management. In: Proceedings of the 31st Society for Computer Simulation Conference (SCSC '99). (1999)
31. Hall, S.B., Zeigler, B.P., Sarjoughian, H.: Joint MEASURE: Distributed simulation issues in a mission effectiveness analytic simulator. In: Proceedings of the Simulation Interoperability Workshop, Orlando, FL (1999)
32. Hall, S.B.: Using Joint MEASURE to study tradeoffs between network traffic reduction and fidelity of HLA compliant pursuer/evader simulations. In: Proceedings of the Summer Simulation Conference, Vancouver, Canada, Society for Computer Simulation (2000)
33. Sarjoughian, H.S., Zeigler, B.P., Hall, S.B.: A layered modeling and simulation architecture for agent-based system development. Proceedings of the IEEE (2000)
34. Burdorf, C., Marti, J.: Load balancing strategies for Time Warp on multi-user workstations. The Computer Journal **36** (1993) 168–176
35. Glazer, D.W., Tropper, C.: On process migration and load balancing in Time-Warp. IEEE Transactions on Parallel and Distributed Systems **3** (1993) 318–327
36. Goldberg, A.: Virtual time synchronisation of replicated processes. In: Proceedings of 6th Workshop on Parallel and Distributed Simulation, Society for Computer Simulation, Society for Computer Simulation (1992) 107–116
37. Reiher, P.L., Jefferson, D.: Dynamic load management in the Time-Warp operating system. Transactions of the Society for Computer Simulation **7** (1990) 91–120
38. Schlagenhaft, R., Ruhwandl, M., Sporrer, C., Bauer, H.: Dynamic load balancing of a multi-cluster simulation on a network of workstations. In: Proceedings of 9th Workshop on Parallel and Distributed Simulation, Society for Computer Simulation, Society for Computer Simulation (1995) 175–180
39. Carothers, C., Fujimoto, R.: Background execution of Time-Warp programs. In: Proceedings of 10th Workshop on Parallel and Distributed Simulation, Society for Computer Simulation, Society for Computer Simulation (1996)
40. Messina, P., Davis, D., Brunette, S., Gottshock, T., Curkendall, D., Ekroot, L., Miller, C., Plesea, L., Craymer, L., Siegel, H., Lawson, C., Fusco, D., Owen, W.: Synthetic forces express: A new initiative in scalable computing for military simulation. In: Proceedings of the 1997 Spring Simulation Interoperability Workshop, IST (1997)
41. White, E., Myjak, M.: A conceptual model for simulation load balancing. In: Proceedings of the 1998 Spring Simulation Interoperability Workshop. (1998)
42. Myjak, M., Sharp, S., Shu, W., Riehl, J., Berkley, D., Nguyen, P., Camplin, S., Roche, M.: Implementing object transfer in the HLA. Technical report (1999)
43. 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
44. Oguara, T.: Load balancing in distributed simulation of agents. Thesis Report 5, School of Computer Science, University of Birmimgham (2004)