

Distributing RePast Agent-Based Simulations with HLA

Rob Minson

Georgios Theodoropoulos

School of Computer Science

University of Birmingham

Birmingham, B15 2TT, UK

{R.Minson|G.K.Theodoropoulos}@cs.bham.ac.uk

ABSTRACT. *Large, experimental Multi-Agent System (MAS) simulations are highly demanding tasks, both computationally and developmentally. Agent-toolkits provide reliable templates for the design of even the largest MAS simulations, without offering a solution to computational limitations. Conversely, Distributed Simulation architectures offer performance benefits but the introduction of parallel logic can complicate the design process significantly. The motivations of distribution are not limited to this question of processing power. True interoperation of sequential agent-simulation platforms would allow agents designed using different toolkits to transparently interact in common abstract domains. This paper discusses the design and implementation of a system capable of harnessing the computational power of a distributed simulation infrastructure with the design efficiency of an agent-toolkit. The system permits integration, through an HLA federation, of multiple instances of the Java-based lightweight-agent simulation toolkit RePast. Comparative results of the performance of the new system vs. sequential RePast are presented and conclusions are drawn on the challenges faced in creating such mappings in general.*

1 Introduction

Multi-Agent Systems (MAS) are a useful paradigm for the decomposition of many complex problems, in both the analytical and the practical domains. MAS simulations have been put to use in the physical sciences [8, 4], anthropological and biological experiments [13], AI research [11], and proprietary MAS design [3]. The vast diversity of agent-design and simulation systems is therefore unsurprising. Some systems focus on providing complex structures for the modelling of the agent’s internal cognitive functions. Other systems focus more on providing an open framework, limited to the executive and some environmental features, such systems are often used in models with a large number of lightweight agents, such as [4]. Often different implementations of the same model use heterogeneous executives, agent-design paradigms and environment implementation toolkits, leaving little consistency in the implementation of abstract domain semantics between one implementation and another.

Further to this, the complexity of MAS models can quickly take on a computational profile which prohibits their execution by sequential processing systems, AI or anthropological experiments for example, where the agent itself is a large, complex system such as [7] may also need to be able to scale to a larger number of agents and a larger interaction domain quickly outstripping the storage and processing ca-

pabilities of a single machine.

The High Level Architecture (HLA) interoperability protocol [9, 1] provides the simulation executive designer with a mechanism for solving both the problem of scalability and of interoperation. The HLA is a protocol based around the notion of the ‘federation’. An individual federate in a given federation is an instance of some simulation executive which is currently modelling a portion of the larger simulation. The federates in a federation communicate through a central ‘Runtime Infrastructure’ or RTI, and synchronise their local schedules with the global schedule through one of the RTI’s time management services. Each federate shares in the global model through a common semantic understanding of the data delivered to it by the RTI, the structure of this data is defined in a ‘Federation Object Model’ (FOM), while the actual interpretation of this data is the responsibility of the federate itself. This semantic independence of data in the HLA, provides the basis for model interoperation. The HLA, however, is not an agent-design framework and as such the modeller loses the developmental boon of such frameworks in adopting this approach.

This paper introduces a middleware layer between the sequential simulation toolkit RePast [2] and the HLA, this layer we term HLA_REPAST. In section 2 we discuss the RePast toolkit, describing its nature as a sequential executive, in section 3 we describe the HLA_REPAST system itself and how the mapping from events in RePast to events in

the HLA is realised. In Section 4 we then go on to give an example of an implementation of a Tileworld MAS model which exploits this system. In Section 5 we discuss the performance profile of the finished system. Finally, in Section 6 we present some conclusions on the design of such systems in general and discuss proposed future work.

2 The RePast Simulation Toolkit

The RePast system [2] is a Java-based toolkit for the development of lightweight agents and agent models. It was developed at the University of Chicago’s Social Science Research Computing division and is derived from the Swarm simulation toolkit. It has become a popular and influential toolkit, assessed by [12] as the most effective development platform currently available for large-scale simulations of social phenomena.

Unlike large simulation infrastructures, such as HLA itself, RePast can properly be termed a ‘toolkit’. The system provides an inter-dependent collection of tools and structures which are generally useful for the simulation of agents, but does not require that the modeller use these structures. It is therefore incorrect to think of RePast models as having any describable generic structure, since this is wholly left to the discretion of the modeller. Despite this it would also be wrong to imagine RePast models as offering no common interface to the executive (this being a vital first-step in the development of middleware such as that with which we are here concerned). This interface is described below.

2.1 Scheduling

All RePast models are implementations of the interface `SimModel`. This interface requires that the model make available to the executive an instance of the `Schedule` class, this being an implementation of a discrete event scheduling engine. In RePast discrete events are modelled as instances of the `BasicAction` class, this class defines a single method; `execute()`, the invocation of which represents the occurrence of the event in question. The `BasicAction` instances extant in the `Schedule` at any point represent events which have been scheduled but are yet to occur in the system.

An important observation at this point is the distinction between the classical interpretation of a ‘Discrete Event’ and the one that can be derived from RePast’s implementation. Generally, and vitally in the case of HLA, a discrete event is taken to be a single, atomic, state transition in one of the model’s member entities. An agent moves one square to the left, for example. While an event can cause the scheduling of future events, it is not generally accepted that the

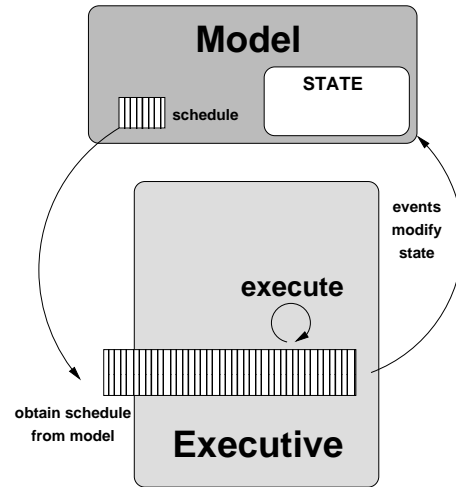


Figure 1: The Standard RePast Model-Executive Interface

execution of a discrete event will itself involve more than one discrete state transition in the model. By contrast, the definition of ‘Discrete Event’ one infers from the RePast implementation, is a potentially infinite and unconstrained sequence of state transitions, whose order and magnitude is undecidable. We discuss the general implication of this semantic distinction further in section 3.1.

The RePast executive in its sequential form interfaces with the model by extracting and loading the schedule into a `Controller` object, which incrementally executes events with increasing timestamps from the schedule. This continues until the executive experiences a `SimulationStop` event. This structure is demonstrated in figure 1.

2.2 State Composition

Unlike the case of the RePast scheduling system which is defined in a constrained (although not altogether constraining) way, the actual state of the simulation is wholly unspecified. While the toolkit provides collections of classes which can be used to construct the state, it does not constrain the model in the use of instances of these classes. In much the same way, then, that the number and magnitude of state transitions during one ‘discrete’ event is unknown, so the composition of the state at any given time in the model is also not accessible by the executive, and therefore indeterminable without prior knowledge of the composition of the model.

2.3 Agent-State Interaction

Just as RePast defines no constraints on the composition of the model’s state, it also defines no mechanism for interac-

tion between the environment and the agents that make up that state. The implication of this is that there is no protocol for state transition (beyond that already defined by the `Schedule`) to which the model must conform. Again, therefore, access to and modification of the agent's environment cannot be observed by the executive.

3 The HLA_REPAST system

A critical aspect of designing a mapping from events in RePast to 'Events' in the HLA is the ability of the middleware to *detect* the occurrence of changes in the RePast model. In order to detect an occurrence, one must first *define* the conditions of an occurrence. In the current RePast system, as explained in section 2 above, there is no imperative definition of state - or agent-state interaction - such that the executive can reliably observe such events. This being so some further constraint *must* be placed on the modes of interaction between the RePast model and its executive (which hereafter is considered to include both the traditional RePast `Controller` class and the middleware itself), to allow the executive to observe events within the model. This said, transparency is a critical aspect of the system design, the essential flexibility of the sequential toolkit should not become a casualty in the name of integration. The eventual system is an extension (and constriction) of sequential RePast, which provides a level of abstraction low-enough to retain the toolkit's flexibility without over-complicating the development process. The following section discusses the essential elements of the middleware and the process of its design. The only element of the middleware visible to the model is an instance of the `LocalManager` class¹, which provides a very minimal number of basic functions to the model though generally direct accesses to the `LocalManager` by the model code itself will be very infrequent².

3.1 Scheduling

In integrating the RePast and HLA scheduling systems the critical step was to identify the time-advance paradigm most appropriate, both to agent-based modelling and to the existing RePast scheduling system. The first choice, between optimistic and conservative scheduling was dictated largely by the demands of transparency. Optimism requires a rollback facility at each node, this in turn requires some form of state-saving mechanism. The two possibilities for im-

plementing this mechanism were, firstly, to obtain references from the model of all mutable elements of the system. While this was conceivable (as is clear from section 3.2) it would also have reduced transparency by demanding the constant registration of new objects and notification of the removal of old ones. The alternative to this system was to allow state-saving to be done in-situ by the modeller herself. Unfortunately the viability of an application-level saving mechanism is very much dictated by the complexity of the model itself; for example, models based on entirely reactive agents really only need to save the state-variables of the environment (a minimal set of data). Models which contain largely intelligent agents, by comparison, must save potentially vast amounts of very complex cognitive data, reducing not only the overhead of the save but, more importantly, the complexity of the development process. Since RePast models have no clear uniform complexity, it was impossible to decide the viability of this second optimistic scheme.

Conservative time-advance (the HLA implementation of which being through the `TIME_ADVANCE_REQUEST` or `NEXT_EVENT_REQUEST` operations) was therefore the preferable paradigm. However the traditional method of using a lookahead value to ensure against deadlock is not appropriate when a node contains an agent process. In conventional distributed simulations, even non-deterministic processes will produce events in response to a stimulus (i.e. incoming events) with some deterministic delay, specifiable by a *lookahead* value (the minimum amount of time in to the simulated future before which the process will produce another event relevant to the system at large). Agent processes are inherently non-deterministic, but further than this, and as discussed in [5] and [14], they are also capable of producing events on a non-reciprocal basis - an agent can cause an event in the system apropos of nothing but its own internal motivations, the *de facto* lookahead value of such a system is therefore 0.

To ensure a maximum of transparency in scheduling it was desirable to simply create a subclass of the RePast `Schedule` class and override the algorithm by which this class advances through its sequence of events. The kernel of the new algorithm - as with all conservative algorithms - was to not execute a `BasicAction` scheduled for time T until it is possible to ensure that all events incoming from the RTI with timestamp $< T$ had been received. This is achieved by first ascertaining this T then requesting advance to $T - 1$. The incoming events are then stored in an external event buffer, once advance to $T - 1$ is granted, this buffer is flushed resulting in a number of modifications to the model all with timestamps $\leq T$. After this point the set of local events with timestamp T can be executed and the process repeated.

The only further requirement was that linked to I/O and to

¹This architecture can be seen as an extension of the general 'ambassador' scheme employed by the HLA (see [9] for more detail)

²Throughout this section the term 'middleware' refers to the HLA_REPAST system in general, but where it is used in reference to a service accessed by the model it can be considered synonymous with this `LocalManager` class

the accurate representation of the current simulation time (known in RePast as the ‘tick’). In standard RePast, display updating and data collection are simply performed through `BasicAction` instances, inserted in to the schedule. In the distributed situation this is unacceptable, as a local I/O action would be executed at a point at which the model was accurate for local events to some time T while only being accurate for external events $< T$. The solution is to provide a ‘safe’ slot for the execution of a single `BasicAction` to perform ‘end-of-tick’ actions such as these. The correct position for this slot in the algorithm described above is immediately after the flushing of the external event buffer but before the execution of the proceeding local events. Executing here ensures that the model is correct for all local *and* external events up to time $T - 1$ and no further. The scheme here described ensures transparent interaction between the model and executive vis-a-vis scheduling to achieve synchronisation in a federation such as that depicted in figure 3. This is depicted in figure 2 which can be compared to figure 1 to demonstrate the transparency of the architecture.

3.2 Simulation Classes and their Attributes

Ensuring that the federates execute in global synchronisation is only the first constraint that must be made to ‘federatise’ RePast. A far more significant task was ensuring that local events could be expressed to the RTI as HLA events. Realising this required two mechanisms; firstly a way of ensuring the middleware receives notification of state changes in the model, and secondly a way of translating from the Java expression to an HLA `UPDATE_ATTRIBUTE_VALUES` invocation.

3.2.1 The PublicObject scheme

The second requirement above was in fact a more simplistic one and required simply some formal scheme for the translation. The mechanism to ensure complete notification, whilst retaining the highest possible level of transparency was far more problematic. A scheme of object registration was adopted which relies very much on constraining the model to conform to a minimal standard in the way simulation objects of federation-wide significance are expressed. The modeller registers new instances of such objects with the local instance of the HLA_REPAST executive. These objects must provide a set of variables which are considered its ‘public interface’, viewable by the entire federation. These variables are wrappers around primitive types which can only be accessed through an ‘access’ and ‘update’ protocol. Upon updating of a variable, the code in the update wrapper executes, notifying the RTI of this update.

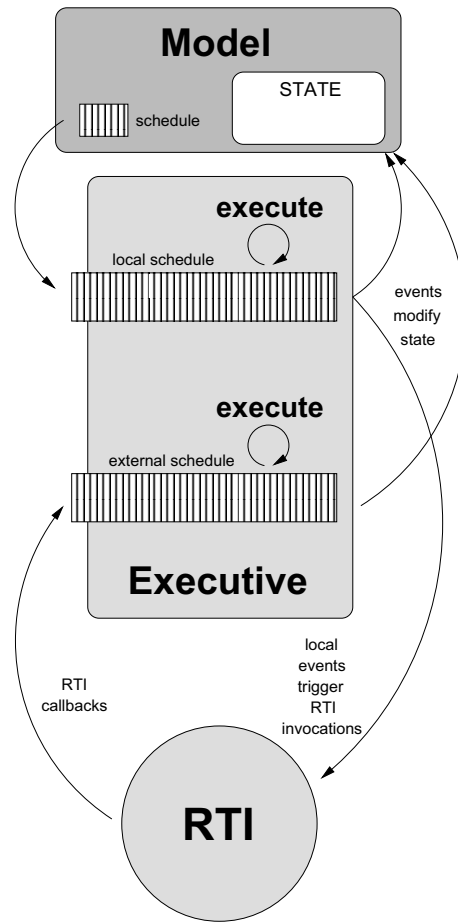


Figure 2: The HLA_REPAST Model-Executive Interface (A Single HLA_REPAST Federate)

To more precisely specify the implementation of the scheme defined above, the model can register instances of any subclass of the `PublicObject` class. This class defines the mechanisms for the middleware to access the public interface of the object. The interface is defined by the model as a set of instances of the `PublicVariable` class. This class (and its subclasses) define the wrappers around primitives as mentioned above. Upon registration with the middleware, these instances are passed a reference to the `LocalManager`, which they then access upon an invocation of `update()`, in order to ensure the update is reflected throughout the federation. This mechanism is depicted in figure 4.

From the perspective of the middleware, this scheme results in simulation entities which are expressed in a very similar manner to those found in other HLA-interfacing executives, such as [6]. By defining this structural protocol to which ‘public’ objects must conform, this approach brings RePast slightly closer to an HLA-like modelling architec-

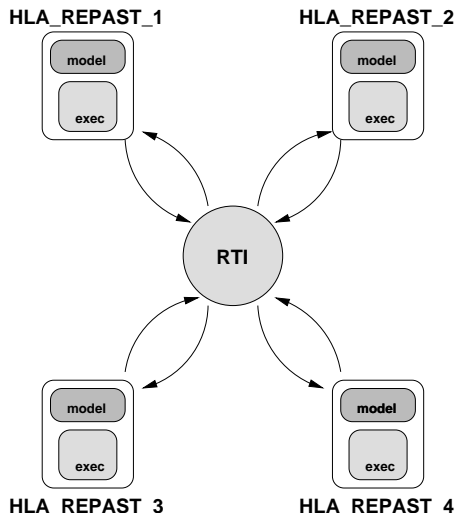


Figure 3: An HLA_REPAST Federation

ture, wherein models are defined strictly in terms of objects and their attributes.

At the other federates, update of attribute values is received by the REFLECT_ATTRIBUTE_VALUES RTI callback³. These events are translated by the middleware in to modifications of some extant PublicVariable instance. These instances will be member variables of either a PublicObject or a RemoteObject instance (see below), both these classes provide the same interface to the middleware, this being a ‘public’ interface of a set of PublicVariable objects. Updates to variable values are therefore handled by accessing the appropriate PublicObject or RemoteObject instance, obtaining the correct PublicVariable from its public interface, and updating the underlying value.

PublicObject instances are instantiated by the local RePast model, while RemoteObject instances are instantiated by the middleware itself, on receipt of a DISCOVER_OBJECT_INSTANCE callback. The objects resulting from this process are then maintained by the middleware and stored in a datastructure termed a ReflectedList, one instance of this class exists for each class in the local FOM (see section 3.2.2). The ReflectedList for a class can be retrieved by the model through a call on the LocalManager, the model then keeps a reference to the object and observe insertions and deletions from it as new instances are created or deleted elsewhere. Because DISCOVER_OBJECT_INSTANCE callbacks do not contain any information about the object’s state (i.e. the value of its

³Note that REFLECT_ATTRIBUTE_VALUES and DISCOVER_OBJECT_INSTANCE are events and, as such, are only received between a federate call to NEXT_EVENT_REQUEST and an RTI callback of TIME_ADVANCE_GRANT

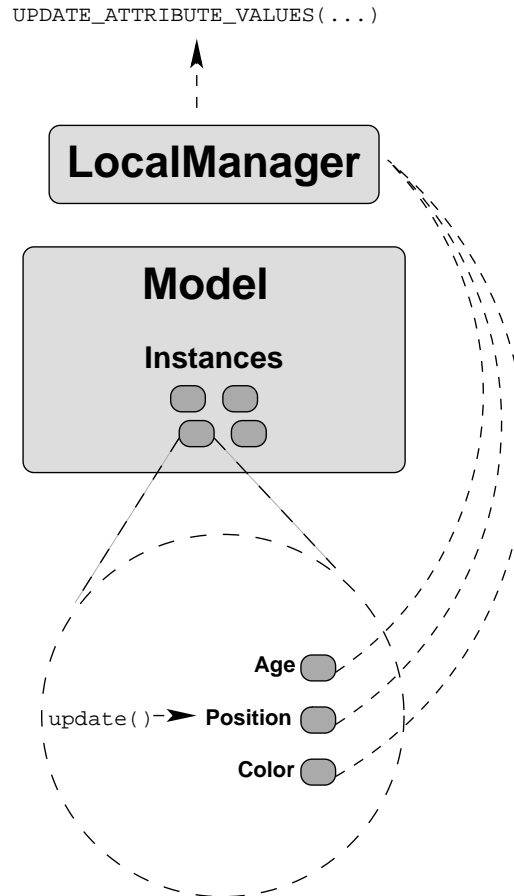


Figure 4: PublicObject/PublicVariable Paradigm

attributes), there always exists some initial ‘limbo’ period during which a federate knows of the existence of the object but not its state. During this time, the instance is not inserted in to the list but is stored in another internal datastructure in the middleware. It is only safe to allow the model access to an object once values have been received for all the variables in that object’s public interface. Each variable therefore keeps a flag which is set when the variable is first updated, once all variable flags are set the object is shifted to the ReflectedList. While this system is a secure way of ensuring RemoteObject instances are always reflected correctly, it can create problems for models with objects whose value is set at instantiation time and never updated.

Note, with reference to section 3.1, that the integration of the RePast scheduling system is not in fact performed at the event-scheduling level. Since an event is generated by the local federate in response to an operation performed on a PublicVariable instance, there no longer exists any canonical mapping between events extant in the RePast scheduler and events that are eventually dispatched to the

RTI. The removal of this canonical mapping solves the issues resulting from the distinction between the HLA concept of a ‘discrete event’ and that derived from RePast’s scheduling system.

3.2.2 FOM creation

Before the mechanism described in 3.2.1 can operate, the simulation classes defined by the model must be expressed at initialisation time in the FOM. This demands either that the modeller create a ‘.fed’ file⁴, or that it be created for them. Given that the existing scheme of `PublicObject` and `PublicVariable` provides the middleware with a natural mapping from the Java types present in the model at run-time to an HLA expression of these types, it is logical to create the FOM automatically, further increasing the transparency of the system.

At initialisation time all federates (i.e. instances of RePast) pass a set of Java `Class` objects to their `LocalManager`. During instantiation all federates first establish contact with the RTI executive, once this is complete they all attempt to create a federation execution by the same name, for $n - 1$ federates this operation will result in an exception, whilst for one federate the operation will be successful, this federate is declared the ‘master’ and the rest declare themselves ‘slaves’⁵. This initial federation is a utility federation, which uses its own FOM, defining two object classes: `LocalFom` and `GlobalFom`. Each slave federate sends a copy of its `LocalFom` to the master, these are then merged at the master in to a `GlobalFom` instance and this is broadcast to the slaves. The master then retires from this initial federation and creates the actual user-federation. Each slave also copies this operation upon receiving its instance of the `GlobalFom`, until all federates are joined to the user-federation, at which point the execution of the user model can begin.

3.2.3 Variable Types and Conflict Resolution

The scheme thus far described ensures that any object registered to the `LocalManager` will have its local updates sent to the RTI and will remain synchronised with remote updates occurring on `RemoteObject` proxies of itself. This system does, therefore, realise the fundamentals of communication between RePast instances which is the basis of the architecture. It does, however, lack the sophistication

⁴The .fed format is the file format for the expression of a federation’s FOM. It is a hierarchically structured class definition schema. The RTI1.3 version used for the HLA_REPAST system employs a bespoke nested markup language, HLA 1516 - the recently established IEEE standard - employs an xml encoding in its place, see [9] and [1] for further details

⁵Note that this master-slave relationship only has relevance during this FOM-creation phase and is ignored thereafter

necessary for modelling complex interaction between simulation entities, where the particular *semantic* of an attribute update may require that the number and provenance of such updates are constrained in some way. An object, for example, cannot be picked up by two separate entities during the same timestep, as this very act by the first entity invalidates the pre-conditions of the act by the second⁶.

The HLA’s ownership management services provide general solutions for consistency problems such as this. The simple principle is that only the federate which owns a given variable may invoke `UPDATE_ATTRIBUTE_VALUES` services upon for that variable. Ownership can be ‘divested’ to the RTI, from which it can be reclaimed by another federate⁷.

In HLA_REPAST, ownership management is used to implement a set of variable-types which provide the various semantic behaviours required by agent models, these semantics are described in table 1. Using the ownership management services, the implementation of these semantics is relatively straightforward. The general mechanism is to default all ownership of variables to the RTI itself, ownership is then acquired on a ‘pull-only’ basis. The details of implementation are also given in table 1.

A major weakness with the given implementation of exclusive variables is that the HLA ownership management services are based currently on real-time exclusion, rather than logical-time exclusion. To clarify: two federates cannot both own one attribute concurrently, even if they are at different logical times; similarly, two federates can both end up owning the same attribute at the same logical time, provided the first federate releases the attribute in real time before the second federate’s request reaches the RTI.

Given the scheduling system described in section 3.1, it is currently possible for two federates to be at different logical times at the same real time. In the context of the current discussion we infer that it is therefore possible for the middleware to erroneously exclude Fed_1 at T_x from an attribute held by Fed_2 at T_y , provided the request from Fed_1 is received while Fed_2 holds the attribute in real time. Exclusive variables, once gained, are not divested again until it and all other federates are granted past the current logical time (see table 1). We further infer therefore that it is impossible for the middleware to erroneously grant ownership of the same variable to two federates at the same logical time.

⁶Note that were both entities in this example being modelled at the same federate the event would automatically take effect and hence the model’s own logic would prevent the inconsistency. The conflict-resolution system described simply ensures that potentially conflicting updates only ever occur between entities at the same federate.

⁷Although HLA does have in-built support for negotiated divestiture between federates where the RTI is never involved, these services are not used in this system and will therefore not be discussed further.

Type	Semantic	Implementation
Exclusive	May only be updated by one federate at a given logical time. This prevents situations occurring where an initial update invalidates a precondition for a second event occurring, but (because the second event occurs at the same logical time) the invalidation is not reported to other federates in time to prevent them allowing the second event to occur.	request ownership from RTI if (acquisition successful) update value else throw exception wait for end of T_{now} divest ownership to RTI
Cumulative	May be updated cumulatively by one federate at a time, updates are only ever permissible as adjustments relative to the current value as opposed to assignments of an absolute value. This would be necessary in situations where many entities at disparate federates need to modify the value of some variable at the same logical time.	while (attribute not owned) request ownership from RTI accumulate value divest ownership to RTI
Viewable	May only ever be updated by one federate during the entire federation execution. This semantic is required for attributes which can only be modified by the entity to which they belong.	if (attribute of local object) update value else throw exception

Table 1: Variable Semantics and their Implementations

At present this weakness is prevented from manifesting itself by simply ensuring a simple condition amongst all schedules in the system. If one considers the set of schedules in the federation as sequences, then this set must be capable of being ordered in a strict hierarchy. In this hierarchy there must exist the condition that, for any sequence, all other sequences are either equal to that sequence or a strict super-sequence, or a strict sub-sequence of that sequence. With this condition it is impossible for two schedules to be at different logical times but request ownership of the same attribute at the same real time.

3.3 Deletion of Objects

The final building block of modelling that must be accounted for is deletion of objects. In HLA the removal of an object from the simulation is an explicit event with a well defined semantic: the DELETE_OBJECT_INSTANCE service. In RePast the situation is far more ambiguous, as with many other things, deletion is a matter for the model itself, rather than the executive. As with value updates there is no ‘hook’ for object deletion on which to hang RTI notification code. Further than this even, unlike value updates in which the low-level semantic is clear - a change to a variable value represents a state-transition in the model - object removal is an event for which the semantic is far harder to define. Some models, for example may remove objects from some master list, which is then reflected on the environment each tick, others may remove objects from some environmental

data structure, others still may simply set some variable belonging to the object itself to denote the end of participation in the model. It is therefore a far more taxing problem than the other issues of this kind so far addressed by HLA_REPAST.

An attractive starting point is to observe that although Java contains no explicit object destructor, it does include an internal memory reclamation system. Through interaction with this system one can embed code to execute upon the reclamation of an object. This approach of reference listening is taken by the HLA_AGENT system [6]. While this seems an attractive (and most importantly, wholly transparent) solution to the issue of object deletion, it is based on the assumption that the model view of an object lifecycle will be identical to the Java view. Examples of an object which is extant in the model code despite being semantically removed from the model are references in listeners, caches in data loggers, ‘self’ references in object code, or (perhaps most perniciously of all) references to objects in the environment held in the data structures modelling the cognitive processes of agents. In these situations the model’s view of the object’s lifecycle is different from the Java view, and reference tracking alone will not accurately communicate deletion events to other federates.

The general problem of resolving the semantics of object deletion is an ongoing research issue in this project and is anticipated to contribute substantially to interoperability work between HLA_REPAST and other HLA-interfacing platforms (see section 6).

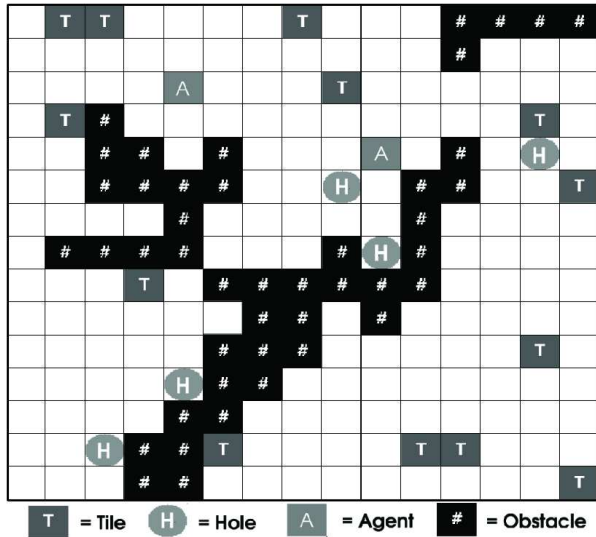


Figure 5: A 20x20 Tileworld

4 An Example Model - The Tileworld

In order to evaluate the system an implementation of popular agent simulation-testbed Tileworld was developed. A sequential model was first implemented, using standard RePast. This model was then distributed using the middleware and modification of the model's code⁸.

The Tileworld, introduced in [10], is a grid-based domain containing obstacles (which agents must navigate), tiles (which agents can pick-up), and holes (which agents can drop tiles in to). The goal of the domain is for the agent to score as many points as possible by filling holes with tiles. The greater the initial depth of the hole, the greater the score available for filling it. Figure 5 shows an example of Tileworld, with two agents in a 20x20 environment. There are a number of reasons why Tileworld is a pertinent testbed for this system, these are largely the same reasons for Tileworld's general popularity in the modelling community. The inherent dynamism of the model in terms of rapid (and adjustable) object creation and deletion make it useful for analysing the registration and removal mechanisms of HLA_REPAST. The parameterisability of the domain give it flexibility when analysing the effects on performance of such parameters as model size, agent architecture, modes of interaction, etc. Finally, Tileworld is a domain rich in conflict, with a high frequency of conflicting tile-pickups and hole-fills between agents.

⁸The sequential model was developed prior to the development of the middleware itself and uses modelling paradigms similar to a number of existing independent RePast models, these measures were taken to evaluate the system in as credible a usage scenario as possible.

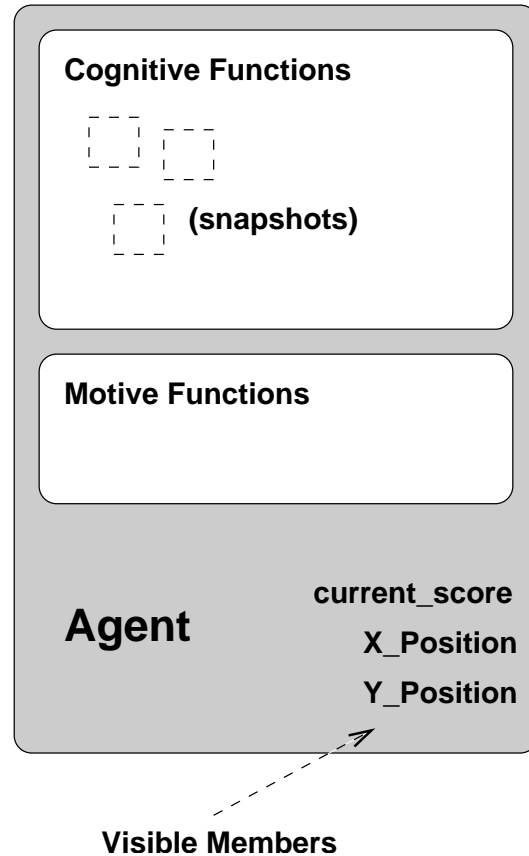


Figure 6: A Tileworld Agent's Internal Structure

4.1 Sequential Tileworld

The key classes which comprise the implementation of the sequential Tileworld are detailed below, the specifics of their attributes and types are given in table 2, the distributed versions of these details are also given for comparison. The three environmental object types; `Terrain`, `Tile` and `Hole` are very simple objects whose state (position, age, depth, etc) is maintained by a single instance of the `TileWorld` class. The `Agent` class holds a reference to this instance and invokes methods on it in order to move through the environment and interact with its contents. The `Agent` object itself is internally highly complex, the structure is divided in to motive and cognitive components, both of which are encapsulated in the agent's 'physical' representation (as demonstrated in figure 6). Both the motive and cognitive components are themselves compositions of many interacting objects, such as long- and short-term planners, deliberators and, importantly, snapshots of the environment, gathered during the continuous process of perception. These snapshots themselves are copies of the environment at some point in the past and, as such, are composed of patterns of `Tile` and `Hole` objects.

Class	Attributes (RePast model)	Attributes (HLA_REPAST model)
Terrain	int X_Position int Y_Position	ViewablePoint position
Tile	int X_Position int Y_Position int age	ExclusivePoint position ExclusiveBoolean is_held
Hole	int X_Position int Y_Position int initial_depth int current_depth int age	ViewablePoint position ViewableInteger init_depth CumulativeInteger curr_depth ExclusiveBoolean is_full ViewableInteger age
Agent	int X_Position int Y_Position int current_score	ViewablePoint position ViewableInteger currentScore

Table 2: Sequential Tileworld - Classes and Attributes

4.2 Distributing Tileworld

When assessing the level of transparency provided by the middleware, the key concern was the extent to which the logic of the model had to be modified. The actual units of transaction (i.e. whether a variable is a variable object or an integer) is of less consequence as these modifications will be relatively trivial. By comparison, having to modify the algorithms and interactions which drive the model is a major task, and one which scales in proportion to the complexity of the model. The reimplementaion detailed in table 2 should therefore be considered in the context of this measurement.

4.3 Conflict Resolution

The most obvious area in which a model’s logic may have to be altered to achieve distribution is that of conflict resolution. In the case of Tileworld, conflict resolution was needed to handle the possibility of simultaneous tile-pickup or hole-fill operations. It was generally assumed during the development of the middleware that conflict resolution would be one of the most arduous development tasks during re-implementation of an existing model. However, in the case of our implementation of Tileworld, there already existed a protocol for interaction between agent and environment. Conflict detection was simply integrated in to this existing protocol. To give an example, where the *pickupTile(Tile)* method of the *TileWorld* class previously threw an exception if the agent was not in the correct position, in the distributed version the same exception is thrown in response to notification that the federate has been

excluded from ownership by the middleware. Although it is therefore necessary to modify the internal mechanisms of individual object classes, the interface these classes still present to the model at large is essentially identical.

4.4 Object Deletion

The general theme for the modifications so far discussed was to ‘internalise’ the representation of object-state as far as possible. To clarify, where previously the ‘held’ status of a tile was represented purely by its external situation, it is now represented explicitly by the internal state values of the object. Externally defined state is not something that the variables system described in section 3.2 can easily communicate, hence any externally represented state must be internalised.

A similar case is encountered for object deletion where, as outlined in section 3.3, only certain representations of deletion are feasibly communicable and/or detectable by the middleware. Using the reference tracking mechanism, models which allow program-extant references to objects which are conceptually deleted from the model, must arrange some other mechanism for detecting the remote ‘deletion’ of these objects. The sequential Tileworld model did allow such reference-states to occur in the form of the ‘snapshots’ extant in the memory of the agents, as mentioned in section 4.1. This fact made it impossible, without major modifications to the cognitive functions of the agent, to use the reference tracking mechanism and forced the implementation of deletion-tracking at the model-level. The general point is whether other, or even the majority, of models will allow references to enter such a state, and whether

- 1: check for obstacle at (b_x, b_y)
- 2: remove agent from grid (a_x, a_y)
- 3: place agent in grid (b_x, b_y)
- 4: update agent.position to (b_x, b_y)

Figure 7: Semantically Dependant Steps in a RePast Event

the scheme is generally infeasible for modelling languages which rely heavily on references as an internal mechanism, these questions are considered open topics of investigation at this time.

4.5 Contextualising External Events

In sequential Tileworld events of global significance (e.g. an object moving) occur as single elements of larger, semantically related procedures. An agent moving from (a_x, a_y) to (b_x, b_y) involves the steps shown in figure 7.

In a sequential model these steps can be atomized to prevent inconsistency, simply by placing all steps in a single commit/abort style procedure. In the distributed version, events such as these occur disjoint from their semantic context, arriving simply as object additions/deletions or as attribute value changes.

The approach adopted for re-establishing model consistency is to provide some mechanism for model-specific event-handlers for discovery, removal and update events. In this manner all events can be contextualised with maximum efficiency. In the example of figure 7 an event handler would be registered to the positional attribute of each agent which executed steps 2 and 3 when an update occurred.

5 Performance

Evaluation of the system from a performance perspective was undertaken by a direct comparison of the sequential and distributed versions of Tileworld.

Experiments were run on a 42 processor cluster of 1.6GHz 1900+ AMD CPUs with 1GB of main memory, interconnected by a 100 Mbps ethernet switch. The NG-RTI version 1.3 for Linux RedHat 7.2 was used in conjunction with the attendant Java bindings.

The experiments were largely performed by varying the number of agents in the environment. All other parameters remained static, namely: a 50x50 grid; object creation probabilities of 0.03 for holes and 0.15 for tiles; average lifespan of 250 steps (both tiles and holes); and an obstacle density of 0.15⁹. All experiments adopted the approach (in

⁹Being the probability of a given square being occupied by an obstacle

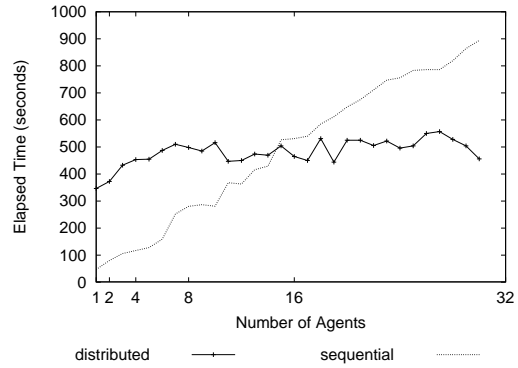


Figure 8: Elapsed Time for 1-32 Deliberative Agents with Scaling Resource Availability

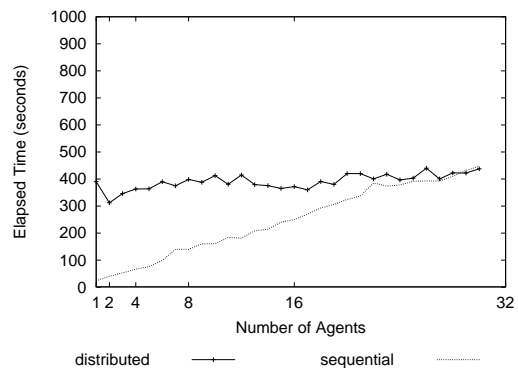


Figure 9: Elapsed Time for 1-32 Reactive Agents with Scaling Resource Availability

the distributed case) of a single environmental federate and some number of agent federates each with an equal number of agents. All runs were executed over 500 iterations.

Two general strategies for experimentation were followed, both focused purely on elapsed time as a performance metric.

The first strategy, depicted in figures 8 and 9, was to scale the computational power available (i.e. the number of nodes) in line with the computational complexity of the model (i.e. the number of agents to be modelled). The complexity of the sequential case is also scaled but in this case the access to computational resources is static.

This scenario was implemented by having all agent federates instantiate only a single agent, then varying the number of federates at each run. As can be seen, initially in the case of a single agent federate the overhead of network communication was distinct, with the distributed run executing approximately 10 times slower. As the computational load on the sequential model increases with the number of agents the execution times increase in a linear manner. The distributed version stays almost constant in comparison, with

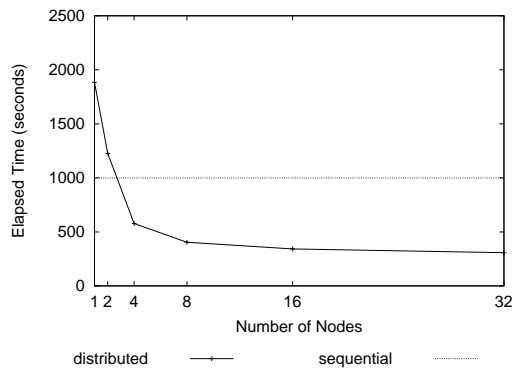


Figure 10: Elapsed Time for 32 Deliberative Agents with Varying Resource Availability

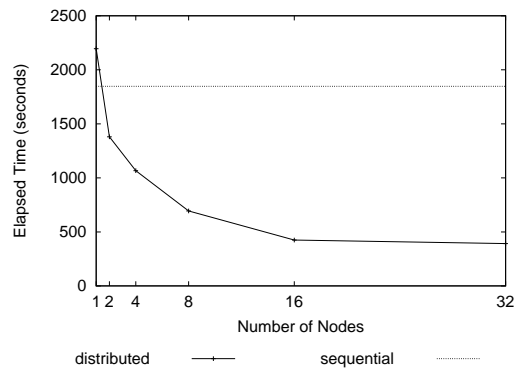


Figure 11: Elapsed Time for 64 Deliberative Agents with Varying Resource Availability

the slow increase being a manifestation of the RTI bottleneck. In each figure we observe a point of intersection of the two lines, denoting the level of complexity beyond which distribution becomes advantageous. Note that the precise location of this intersection on the complexity axis depends on the type of agent being modelled¹⁰. As was speculated in [6], from a distributed perspective computationally intensive agents are CPU-bound while reactive (or ‘lightweight’) agents are network-bound. This observation is based on the simple principle that, since computationally intensive agents spend more time ‘thinking’ and less time ‘acting’, they will cause less globally significant events (such as picking up a tile) and hence less network communication. We therefore observe that this intersection occurs early - 14 agents - on the graph for a model using complex, deliberative agents, while it occurs comparatively late - around 27 agents - on the graph for a model using simplistic, reactive agents.

The second experimental method was to keep a constant level of complexity while varying the computational resources available. In this vein the number of agents was kept at a constant. This model was then run on a system with only one agent-federate, and the system then scaled up incrementally to end with enough federates to provide one node per-agent¹¹. The results for 32- and 64-agent models are given in figures 10 and 11, in both these figures the line representing sequential performance extends across the nodes axis as a reference.

As expected the benefit of distribution is initially sharp, with this benefit levelling off as the distribution increases

¹⁰Note that these two figures also vary in the ‘smoothness’ of the distributed graph. This is largely a function of the fact that the magnitude of the computation performed by an individual deliberative agent is not a constant (as with the reactive case) but varies according to more complex determinants

¹¹Clearly in the case of the 64-agent model our 44-node cluster could not provide for a 64-node federation, the 64-agent model therefore has a minimum of 2 agents per node.

and the system becomes more network-bound. The figures show that, even with small distributed resources (between 2 and 4 nodes), distribution will improve the performance of the very complex models such as these.

One aspect of figures 10 and 11 seems, at first, anomalous. Namely that the initial height of the distributed curve for 64 agents is far less than twice that of the distributed curve for 32 agents. To explain this anomaly we start, as above, from the observation that ‘internal’ agent actions are less expensive per-operation in a distributed context than ‘external’ actions. Analysis of the actual events occurring in executions of the models revealed that the number of tiles and holes in the 64-agent model was being reduced to 0 (as they were picked up/filled by the agents) at a far quicker pace than in the 32-agent model. In the context of performance it can be seen that a model which very quickly is stripped of all tiles and holes, and hence all means for producing an ‘external’ event will thereafter indulge in far less network communication and hence will execute more quickly.

6 Conclusions & Future Work

We have presented the interoperability middleware HLA_REPAST. The system is an approach to the distribution and interoperation of agent-modelling platforms. We have discussed the design issues involved in constructing such middleware in general and have given an example of the steps necessary to distribute an existing sequential model.

From the discussion of the modifications performed to the RePast executive 3, one can derive some general conclusions. Namely that the level of constraint present in the structure of a sequential executive determines the level of transparency that will remain between the model and modified executive. To give an example from HLA_REPAST, where the RePast executive demonstrated a high level of structural constraint (e.g. in the event scheduling system

described in 3.1), the modifications made were able to retain a wholly transparent interface between model and executive. Where the RePast executive demonstrated little or no structural constraint (e.g. in the definition of model representation), transparency was lost as some order had to be enforced over the model's internal expression of externally significant events.

Currently HLA_REPAST employs no interest management beyond the simple attribute subscription service. However, the HLA provides a far more sophisticated interest management scheme in the form of the Data Distribution Management system (DDM). Providing the model access to some useful abstraction of the DDM, or perhaps enabling the middleware to use this system transparently in some way, is a priority for future development.

To our knowledge, the HLA_AGENT system, described in [6], is the only other existing implementation of middleware for distributing MAS simulation executives with the HLA. Considering that the primary aim of the HLA is interoperation of simulations, the most intriguing possibility for future work is the federation of HLA_REPAST and HLA_AGENT to achieve the interoperation of RePast and SIM_AGENT models. Theoretically the middleware abstraction of the actual HLA data-exchange mechanisms in both cases will mean that all that is required is to agree on the abstract semantics of some domain to achieve such inter-operation.

Acknowledgements

We would like to thank Mike Lees and Brian Logan, both at Nottingham University, for the benefit of their experience with HLA, agents toolkits and for sound advice on agent-architectures in general.

References

- [1] IEEE 1516 (Standard for Modelling and Simulation High Level Architecture Framework and Rules), 2000.
- [2] Nick Collier. RePast: An Extensible Framework for Agent Simulation. http://repast.sourceforge.net/docs/repast_intro_final.doc, June 2003.
- [3] Agent Business Force Corporation. RIDL 2004. World Wide Web, <http://www.ab4s.com/products/RIDL>, 2004.
- [4] Dominique Groß and Barry McMullin. The Creation of Novelty in Artificial Chemistries. In *Proceedings of The 8th International Conference on the Simulation and Synthesis of Living Systems*, pages 400–409. MIT Press, 2002.
- [5] Brian Logan and Georgios Theodoropoulos. The Distributed Simulation of Multi-Agent Systems. In *Proceedings of the IEEE - Special Issue on Agent-Oriented Software Approaches in Distributed Modelling and Simulation*, 2000.

- [6] Brian Logan, Georgios Theodoropoulos, Michael Lees, and Tonworio Oguara. Simulating Agent-Based Systems with HLA: The Case of SIM_AGENT. PART II. In *2003 European Simulation Interoperability Workshop*, pages 517–128. SISO, June 2003.
- [7] T. Lux and M. Marchesi. Scaling and Criticality in a Stochastic Multi-Agent Model of Financial Markets. *Nature*, 397:498–500, 1999.
- [8] Christopher J. Mackie. Studying Political Identity Formation and Change: A Testframe for Autonomous-Agent-Based Simulations. In *Annual Meeting of the Midwest Political Science Association*, April 2003.
- [9] US Defence Modelling and Simulation Office. HLA Interface Specification, version 1.3, 1998.
- [10] Martha E. Pollack and Mark Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *National Conference on Artificial Intelligence*, pages 183–189, 1990.
- [11] Aaron Sloman and Ricardo Poli. A toolkit for exploring agent designs. In M. Woolridge, J. Meuller, and M. Tambe, editors, *Intelligent Agents II: Agent Theories, Architectures and Languages (ATAL-95)*, pages 392–407. Springer-Verlag, 1996.
- [12] Robert Tobias and Carole Hofmann. Evaluation of free Java libraries for social-scientific agent-based simulation. *Journal of Artificial Societies and Social Simulation*, 7(1), January 2004.
- [13] A. Urmacher and M. Röhl. The role of deliberative agents in analyzing crises in pre-modern towns. *Sozionik*, (3), 2001.
- [14] A. M. Urmacher and K. Gugler. Distributed, Parallel Simulation of Multiple, Deliberative Agents. In *14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, pages 101–108. IEEE Computer Society, 2000.

Author Biographies

ROB MINSON is a PhD student at the School of Computer Science, University of Birmingham, UK. He received an MSc in Computer Science from Birmingham in 2003. His thesis will be in the area of Interest Management Architectures for Distributed and Parallel Simulation Systems.

GEORGIOS THEODOROPOULOS is a Lecturer in the School of Computer Science, University of Birmingham, UK. His research interests include Parallel and Distributed Systems and Modelling and Computer and Network Architectures. He received MSc and PhD degrees in Computer Science from the University of Manchester, UK in 1991 and 1995 respectively. He is co-founder of the Midlands E-Science Center of Excellence in Modelling and Analysis of Large Complex Systems.