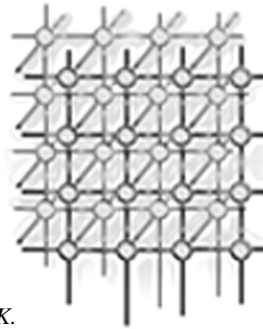


Distributing RePast Agent-Based Simulations with HLA

R. Minson^{1,*} and G. K. Theodoropoulos¹

¹*School of Computer Science, University of Birmingham, Birmingham B15 2TT, U.K.*



SUMMARY

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. Our main contribution is in abstractly defining the engineering process necessary in creating such middleware, and in reporting on our experience in the specific case of the RePast toolkit.

KEY WORDS: Distributed Simulation, Multi-Agent Simulation

1. INTRODUCTION

Multi-Agent Systems (MAS) are a useful paradigm for the decomposition of many complex problems [13]. MAS simulations have been put to use in the physical sciences [18, 12], anthropological and biological experiments [26] and AI research [24] to name but a few areas. The vast diversity of agent design and simulation systems is therefore unsurprising [23, 9, 20, 4, 3, 22]. Some systems focus on providing complex structures for the modelling of the agent's internal cognitive functions. Other systems, often used in models with a large number of lightweight agents, focus more on providing an open framework, limited to the executive and some environmental features. 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. However, no one testbed can be appropriate to all agents and environments. There is therefore a strong incentive to reuse existing toolkits and models and somehow combine them to build larger and more complex scenarios [16].

Further to this, the complexity of MAS models can quickly take on a computational profile which prohibits their execution by sequential processing systems. In AI or anthropological experiments for example, the agent itself is a large, complex system (e.g. [17]) and the need to be able to scale to a larger number of agents and a larger interaction domain quickly outstrips the storage and processing capabilities of a single machine [22, 3, 11, 21].

*Correspondence to: R. Minson, School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK

†E-mail: R.Minson@cs.bham.ac.uk



Large scale, distributed simulation can offer a solution to both problems of toolkit interoperability and model scalability. Distributed simulation has received an explosion of interest in the last decade, as a strategic technology for both speeding up simulations as well as linking simulation components of various types at multiple locations to create a common virtual environment. The culmination of this activity (which originated in military applications where battle scenarios were formed by connecting geographically distributed simulators via protocols such as the Distributed Interactive Simulation protocol (DIS)), has been the development of the High Level Architecture (HLA) [14]. HLA, which is now an IEEE standard, facilitates interoperability among simulations and promotes reuse of simulation models. Using HLA, a large-scale distributed simulation can be constructed by linking together a number of (geographically) distributed simulation components (or federates) into an aggregate simulation (or federation).

This paper investigates the problem of integrating an agent toolkit into HLA in order to achieve both interoperability and scalability. The paper introduces HLA_REPAST, a middleware layer between the HLA and the sequential MAS simulation toolkit RePast [7], assessed by [25] as the most effective development platform currently available for large-scale simulations of social phenomena.

The rest of the paper is organized as follows: the next section provides a short overview of HLA and section 3 discusses the RePast toolkit, describing its nature as a sequential simulation platform; section 4 discusses the fundamental issues that need to be addressed to achieve a mapping between sequential and distributed simulation platforms; section 5 describes the HLA_REPAST system itself; section 6 illustrates the application of the system in an implementation of a classic MAS benchmark model; section 7 discusses the performance profile of the finished system; finally, section 8 presents some conclusions on the design of such systems in general and discusses proposed future work.

2. THE HIGH LEVEL ARCHITECTURE

The High Level Architecture (HLA) interoperability protocol [19, 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 may be written in different languages and may run on different machines. 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. Within a given FOM the classes of objects, which are to be used in a specific federation, are defined by a name (unique within the hierarchy) and a set of un-typed attributes.

At run-time the federate interfaces with the RTI through the use of an `RTIAmbassador` instance. This object provides access to the remote invocation services provided by the HLA specification. In a similar way, the federate itself must provide an implementation of the `FederateAmbassador` interface, which accepts callbacks from the RTI to notify of events pertinent to this federate.

Communication between the RTI and the federates in a simulation takes place in the language of the HLA, using the specific semantics defined by that federation’s FOM. State changes in general are communicated from federate to RTI via invocations of the `UPDATE_ATTRIBUTE_VALUES` service. In response to this invocation, the RTI will invoke the `REFLECT_ATTRIBUTE_VALUES` callback on the `FederateAmbassadors` of other federates. This call contains a set of $\langle \textit{attribute_handle}, \textit{value} \rangle$ tuples, which the federates are expected to use to update their local copies of the simulation state.

* The HLA also provides support for real-time simulations which do not use time-management, these will not be discussed here.

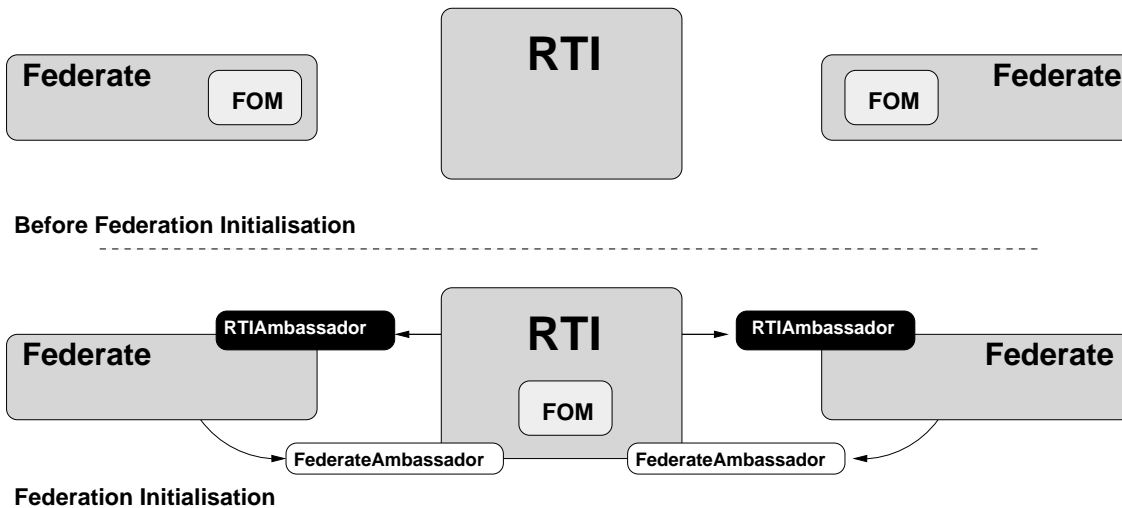


Figure 1. The 'Ambassador' abstraction used by the HLA

In logical-time synchronised simulations, all `UPDATE_ATTRIBUTE_VALUES` invocations are time-stamped. Under conservative synchronisation schemes* `REFLECT_ATTRIBUTE_VALUES` callbacks in response to updates do not happen at arbitrary points but only when the federate requests to advance to or beyond the logical time at which the event occurred. In this way the local schedule of the individual federate can be harnessed to act as the engine to drive forward the tightly-coupled activities of state-transition and time-advance of the simulation at large.

3. THE RePast SIMULATION TOOLKIT

The RePast system [7] 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, providing the development platform for several large multi-agent simulation experiments, particularly in the field of social phenomena [2].

Unlike some large simulation infrastructures, such as HLA/RTI 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 simulations as executing in a wholly unconstrained manner (this being a vital first-step in the development of middleware such as that with which we are here concerned).

3.1. The Model/Executive Paradigm

At the highest level, each RePast simulation has two components. The first of these is the model, specified by the simulation designer and composed usually of a mix of bespoke components and RePast library

*Optimistic synchronisation schemes will not be discussed at this point, a discussion of the propriety for the HLA_REPAST system of each of the time-advance schemes is presented in 5.2.

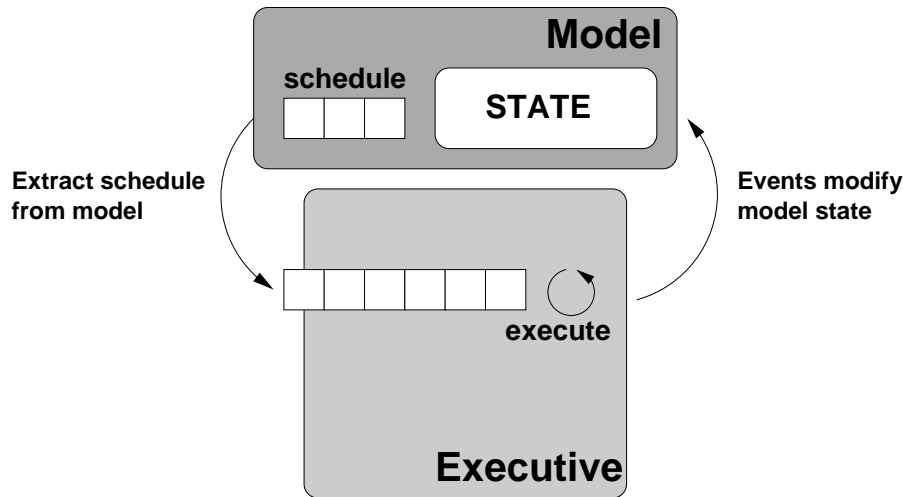


Figure 2. The Standard RePast Model-Executive Interface

components (such as grids to represent the environment, or graphical display components). The second component is the executive object (an instance of the `RePast Controller` class), this interacts with a model to affect a single simulation execution. These two components interact initially by a model being loaded in to the controller, the state-transitions in the model are then controlled (monotonically) through the executive - i.e. the model can be ‘stepped’ forward to the next scheduled event by invocations of ‘step’ upon the executive. For this control to be achieved, there exists a well defined interface, which a model must expose to an executive, this interface is exploited in the `HLA_REPAST` system and is described below.

3.2. Scheduling

All models are first required to make available to the executive an instance of the `Schedule` class, which is an implementation of a discrete event scheduling engine. In RePast, events are modelled as instances of the `BasicAction` class, which defines a single method, `execute()`, whose invocation 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 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 5.2.

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 2.



3.3. 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.

3.4. Agent-State Interaction

Just as RePast defines no constraints concerning the composition of the model’s state, it also defines no mechanism for interaction 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.

4. A General Approach

Having given an overview of one agent toolkit, it is now worth presenting some observations on what is generally needed to achieve this type of mapping between sequential and distributed simulators, before we discuss the specifics of the current implementation. In general such a mapping involves three intellectual tasks:

- **Mapping the Scheduling System.** This task involves first determining what type of time-advance paradigm best suits the sequential system being mapped (e.g. conservative, optimistic, etc.). From this point the task is then to determine how best to integrate the sequential scheduler itself in to the parallel scheduling algorithm of the distributed kernel. A critical first step in general in this process, is understanding how the notion of *event* in the sequential system relates to the notion held to by the distributed system.
- **Mapping the State-Representation.** This task is concerned with determining how to communicate state-transitions in the sequential simulator through the distributed simulator to other sequential instances. Again this involves two ideas. Firstly there must exist some way for the model to express the distinction between ‘public’ data (transitions in which should be communicated to the global system) and ‘private’ data (which are of concern only to the local model and should not be communicated). Once this partition is define-able, the next task is to determine some way of detecting the transitions in public state when they occur, thus enabling their communication to the global system. This is likely to be the aspect of the mapping that is most critical in terms of retaining transparency of the distribution, as a non-transparent mechanism for recognising updates when they occur will mean that large sections of modelling code must be re-written.
- **Conflict Resolution for Shared Variables.** This task is not immediately as evident as the previous two. Most simulation models, particularly in the case of models of agent-systems, involve large numbers of shared variables. These variables (usually modelling some element of the agent’s environment) can be the subject of frequent concurrent updates and, in many cases, these updates may be attempting to place the variable in logically conflicting states. In a sequential system these conflicts can be dealt with using a sequential logic, with some operations being allowed and some being rejected. In the distributed case however, this ‘conflict resolution’ is complicated by the lack of a centralised, sequential logic. Firstly some mechanisms must be determined within the distributed kernel to achieve reliable resolution for conflicts and, secondly, the sequential model must be presented with an interface on to these mechanisms. This interface must be as simple and as transparent as possible.

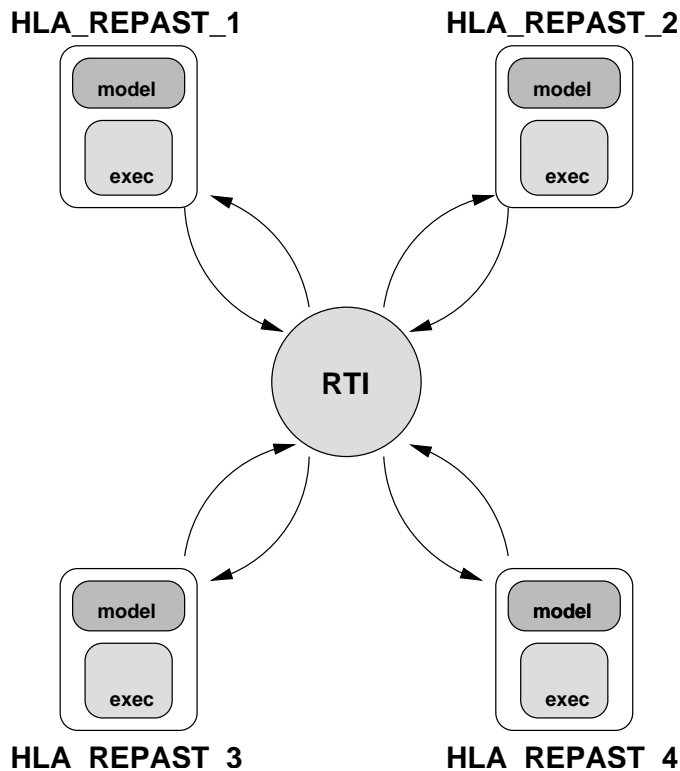


Figure 3. An HLA_REPAST Federation

In general, the process described above is interested in two elements of the design: firstly, determining mechanisms for achieving distributed execution of a given sequential simulator, and secondly, devising transparent interfaces for accessing these mechanisms. Having described the tasks one must complete in the general case, section 5 presents the design process for the specific case of HLA_REPAST and demonstrates how these tasks were approached.

5. THE HLA_REPAST SYSTEM

HLA_REPAST is a middleware layer which enables the execution of a federation of multiple interacting instances of RePast models within HLA as depicted in figure 3.

In general distributed simulation systems start from this notion of independent but communicating processes. In order to achieve parallel execution of a *single* model in such a system, the model must be partitioned in to several sub-models, each modelling a subset of the components of the system. In the case of a MAS simulation examples of possible partitions are:

- For an N agent model partitioned across $N + 1$ federates: one agent at each federate and one federate to model the environment.
- The environment divided in to N physical regions across N federates, with agents moving from federate to federate as it moves through the environment.
- For an agent architecture with $N - 1$ discrete cognitive components: one federate for the environment and one federate for each component. Here each of the cognitive federates houses A separate instances of its given component, where A is the number of agents in the model.

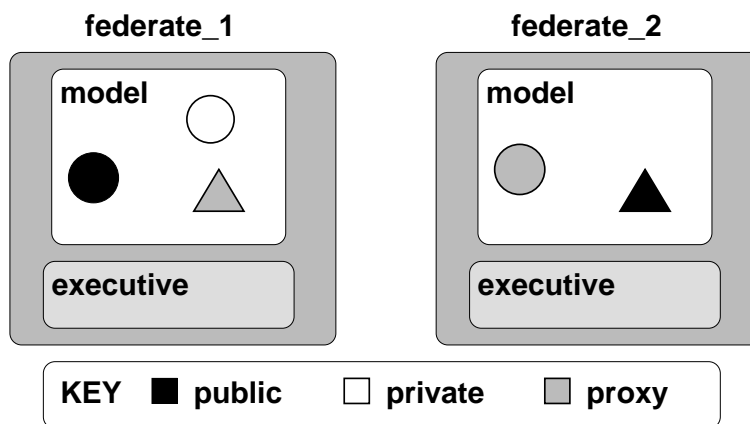


Figure 4. The HLA_REPAST system of model-distribution

Given a partition of the model into a set of components and the distribution of the components across federates, there must exist some mechanism by which a component can interact with the rest of the model. In the case of HLA_REPAST this is achieved through the use of proxy objects which reflect the state of remotely modelled objects. Each federate in an HLA_REPAST federation sees the local model as a set of objects. Some of these objects will be locally modelled and shared with the rest of the federation (public objects), some will be locally modelled and not shared (private objects) and some will be proxies for public objects at other federates (proxy objects). This architecture is depicted in figure 4.

This approach to the distribution of the model was derived from an analysis of the imperatives found when considering the RePast system and the HLA. This derivation came through the process of first studying both systems and then attempting to address the three design tasks identified in section 4. The remainder of this section describes this process.

5.1. Fundamental Issues

The critical task of the middleware is to detect the occurrence of changes in the RePast model and be capable of reliably communicating them to the HLA. Given that the HLA is based on the principle of discrete events, the solution that seems immediately to present itself, is to map from `BasicAction` instances to HLA events. Recall from section 3, however, that there is no clear mapping, or at least mapping constraint, between a RePast `BasicAction` and a discrete state-transition. A `BasicAction` may contain 0, 1 or an infinite number of individual attribute value modifications, which may be all on a single object, or to multiple objects. In order to obtain a reliable mapping between state-transitions in RePast and `UPDATE_ATTRIBUTE_VALUES` invocations in the HLA, the system must be capable of detecting individual variable accesses and generating events for each one.

This approach leads us to review the general case discussed in section 4. The RePast scheduler itself is now only used to ensure a constrained progression through logical time. It is the state-representation itself that must provide the middleware with the information it needs to accurately communicate state-updates. Due to this, the mapping of the scheduling system is a relatively minimal activity, with the majority of the architecture being defined by the approach to the mapping of the state-representation.

This being so, some further constraint *must* be placed on the modes of interaction that may occur within a RePast model to allow the executive to observe state-transitions. This said, transparency is a critical aspect of the system design, the essential flexibility of the sequential toolkit should not become a casualty to pragmatism. 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` class*, which provides a very minimal number of basic functions to the model, thoughl, generally, direct accesses to the `LocalManager` by the model code itself will be very infrequent[†].

5.2. Mapping the Scheduling System

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.

Optimistic synchronisation [8] requires a rollback facility at each node, which in turn requires some form of state-saving mechanism. The two possibilities for implementing 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 5.3) 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. This would most likely have taken the form of some model-defined function, accessible by the executive. 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 tend to have less permanent cognitive state - memory and planning is not an issue here - and hence the effort to save this state is insignificant (and simple to implement). 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 synchronisation [10, 6, 5] was therefore the preferable paradigm. 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). However the traditional method of using this lookahead value to ensure against deadlock is not appropriate when a node contains an agent process. Agent processes are inherently non-deterministic, but further than this, and as discussed in [15] and [27], 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, the normal RePast algorithm for stepping through the schedule was replaced with a new algorithm that constrained local advance in synchronisation with the rest of the federation. 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 (the time of the earliest event in the local schedule), and then requesting advance to T . The RTI will then send a set of events to the federate, all with timestamps $\leq T$, these events are stored in an external event buffer. Once advance to T is granted the buffer is flushed for all events with timestamp $< T$, resulting in a number of modifications to the model. After this point the set of local events with timestamp T can be executed and the process repeated. Figures 5 and 6 shows this algorithm in a more detailed format. From a modelling perspective the only modifications that need to be made for use of this distributed schedule is to obtain a reference to an instance of the HLA_REPAST class `DistributedSchedule` which can then be populated transparently as a standard `Schedule` instance.

The only further requirement was that linked to I/O and to the accurate representation of the current simulation time (known in RePast as the 'tick'). In standard RePast, display updating and data collection

* This architecture can be seen as an extension of the general 'ambassador' scheme employed by the HLA (see [19] for more detail).

[†] Throughout 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.

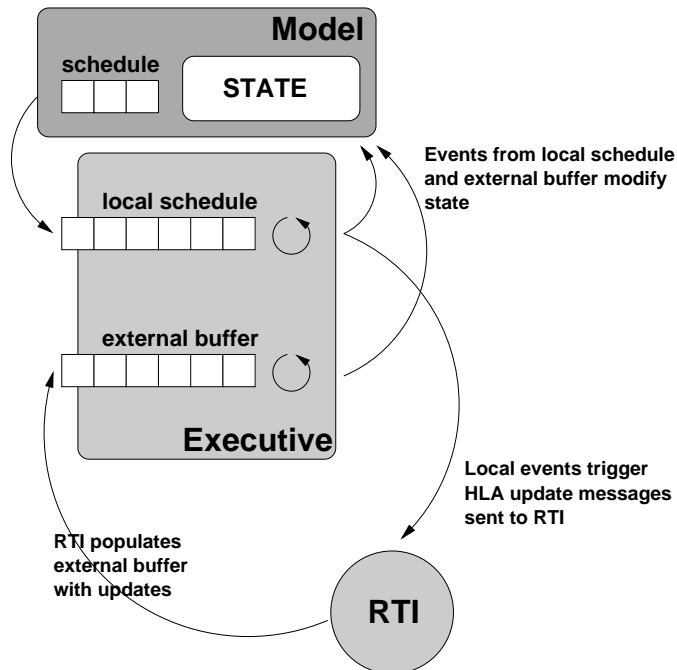


Figure 5. The HLA_REPAST Model-Executive Interface (A Single HLA_REPAST Federate)

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. Comparing figures 2 and 5 it can be seen how the flow of events in and out of an individual federate appears the same from the RePast perspective but from a global perspective has been integrated with the rest of the federation.

5.3. Mapping the State-Representations

As mentioned above, the structure of RePast itself dictates that the middleware, as a system for communicating state changes to the HLA, cannot operate solely at the level of the scheduling system. It was determined that the state-representation itself would have to take responsibility to ensure the communication of events when necessary and do so in a consistent and reliable way. Realising this required two mechanisms: firstly, a way of ensuring that 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.

* This problem of establishing I/O safe-points is similar to the problem of establishing such safe-points in optimistic simulation where it is more frequently referred to as the problem of establishing Global-Virtual-Time (GVT). An implementation of a GVT control system is the TimeWarp kernel [8].



```

S ← new DistributedSchedule
E ← external event queue
populate S with local events
while simulation running do
  Tmin ← lowest timestamp in S
  L ← {e ∈ S : ets = Tmin}
  while not granted to TMin - 1 do
    invoke NEXT_EVENT_REQUEST (Tmin - 1)
    //This will populate E with external events with ts ≤ Tmin - 1
    while E ≠ ∅ do
      deque_and_execute(E)
    end while
  end while
  //no more events with ts ≤ Tmin can now be recieved
  execute I/O action with ts = Tmin - 1
  localTime ← Tmin
  while L ≠ ∅ do
    deque_and_execute(L)
    //If I/O occurs now model is only correct for local events, not remote ones
  end while
end while

```

Figure 6. Integration of the RePast schedule

5.3.1. The PublicObject scheme

The second requirement above was in fact a more trivial one and required simply some formal scheme for the translation. This was based largely on the system described in section 5.4. The mechanism to ensure complete notification, whilst retaining the highest possible level of transparency, was far more complicated. Three general schemes were investigated:

- **Libraries.** The first scheme provides library services, which are manually accessed by the model. The modeller is responsible for keeping track of the handles of variables and for passing $\langle handle, value \rangle$ tuples to these services as updates occur. This scheme effectively presents a simplified version of the HLA interface to the model and it is flexible, but it is wholly opaque from the modeller's perspective.
- **Tracking.** This scheme tracks the values of variables manually in the middleware. This would require some initial registration of all variables in the model, a means of accessing the value periodically and some way for the model to notify the executive of the appearance of new attributes or the disappearance of old ones. With these mechanisms in place, it is simply a question of copying the values at the beginning of each tick, noting any external updates during the tick, then checking the values against the copy at the end of the tick. Any values that are inconsistent with their copies and have not experienced an external update will then have updates sent for them. This is the most transparent scheme in principle, but it has no way of constraining the updating of attribute values. It is therefore impossible to implement the attribute synchronisation schemes critical to the run-time transparency of the system at large (see section 5.4).
- **Registration.** The final scheme is a compromise between the first two and relies very much on constraining the model to behave in as HLA-like a way as possible. The modeller registers new objects with the middleware, which provide access to a set of variables, which in turn 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

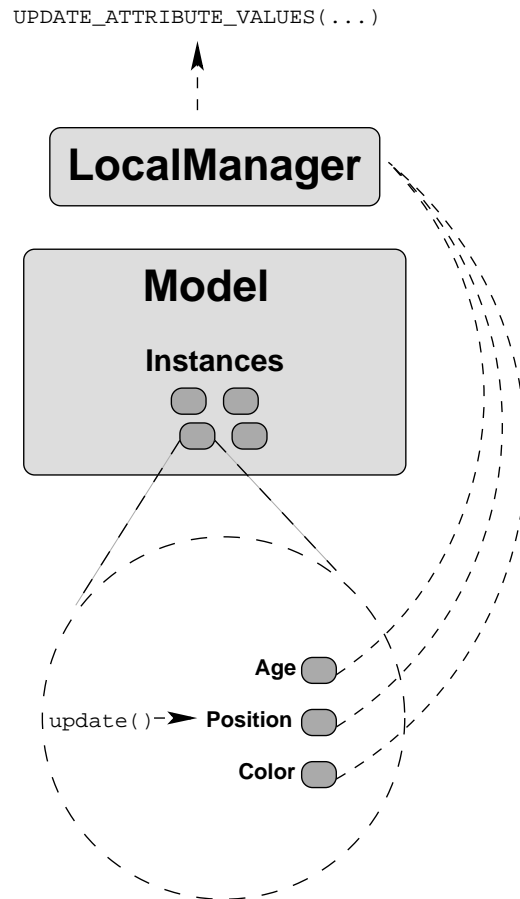


Figure 7. PublicObject/PublicVariable Paradigm

updating of an variable, the code in the update wrapper executes, notifying the RTI of this update. This is the scheme that was implemented in the system.

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 7.

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 [16]. By defining this structural protocol to which 'public' objects must conform, this approach brings RePast slightly closer to an HLA-like modelling architecture, 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 callback*. These events are translated by the middleware into 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 5.3.2). The population of this list changes in response to the receipt of callbacks to discover or remove remote instances. The `ReflectedList` for a specific class of remote object can be retrieved by the local model through a call on the `LocalManager`; the model then keeps a reference to the object and observes insertions and deletions from it, as new instances are created or deleted elsewhere.

Note, with reference to section 5.2, 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.

5.3.2. FOM creation

Before the mechanism described in 5.3.1 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’ file[†], 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, each federate presents a set of object classes to the executive. The inheritance relationships within this set are determined using Java’s reflection facilities and are parsed in to a local view of the FOM (FOM^{local}). The goal after this point is to integrate these local FOMs in to a single global FOM, which can be used to instantiate the actual federation. To achieve this the federates first participate in a bootstrapping federation (*proto-fed*). Within the bootstrapping federation the federates first elect a leader, which, once elected, collects all local FOMs and merges them in to a single global FOM (FOM^{global}), returning this to all other federates. The full algorithm is given in figure 8.

5.3.3. Deletion of Objects

The final building block of state-modelling that must be accounted for is the deletion of objects. In the 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 (i.e. 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

* Note 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`.

[†] The .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 mark-up language, HLA 1516 - the recently established IEEE standard - employs an xml encoding in its place, see [19] and [1] for further details.



```
FOMlocal  $\leftarrow$  model.classes
fedname  $\leftarrow$  model.name
num_federates  $\leftarrow$  model.num_federates
CREATE_FEDERATION_EXECUTION(proto_fed)
if creation successful then
    status  $\leftarrow$  master
else
    status  $\leftarrow$  slave
end if
JOIN_FEDERATION_EXECUTION(proto_fed)
if status = slave then
    REGISTER_OBJECT_INSTANCE(FOMlocal)
    UPDATE_ATTRIBUTE_VALUES (FOMlocal)
    wait for receipt of FOMglobal
else
    total_recieved  $\leftarrow$  0
    while total_recieved < num_federates do
        wait for receipt of FOMremotei
        total_recieved  $\leftarrow$  total_recieved + 1
    end while
    FOMglobal  $\leftarrow$  FOMremote0  $\cup$  ... FOMremotei
    REGISTER_OBJECT_INSTANCE(FOMglobal)
    UPDATE_ATTRIBUTE_VALUES (FOMglobal)
end if
RESIGN_FEDERATION_EXECUTION(proto_fed)
if status = master then
    CREATE_FEDERATION_EXECUTION(fedname)
end if
JOIN_FEDERATION_EXECUTION(fedname)
```

Figure 8. Algorithm for automatic FOM generation

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 datastructure; 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 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. 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. The allusion here is to objects, which are extant in model code despite being *semantically* removed from the model. Examples of such objects are references in event-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 datastructures modelling the cognitive processes of agents. 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 8).

There are two viable alternatives to exploiting this internal system. The first is to equip the `PublicObject` and `RemoteObject` classes with the explicit destructors, which Java lacks. This option is both viable and would effectively solve all object-deletion issues internal to the middleware, there is,



Federate_A	Federate_B
updates attribute a of instance n (n_a) to x at T_n	updates n_a to y at T_n
Variable object n_a at Federate_A (n_a^A) invokes UPDATE_ATTRIBUTE_VALUES (x)	n_a^B invokes UPDATE_ATTRIBUTE_VALUES (y)
Federate_A receives REFLECT_ATTRIBUTE_VALUES callback for n_a^A with new value of y	Federate_B receives REFLECT_ATTRIBUTE_VALUES for n_a^B with new value of x
$n_a = y$ despite being updated to x	$n_a = x$ despite being updated to y

Table I. The Dangers of Unconstrained Attribute Updates

however, the serious problem of transparency, which this solution raises. The other alternative is to simply ignore the issue all together, stipulating that *all* inter-model communication should be undertaken using the variable system provided. None of these solutions is seen as optimal but the current implementation of HLA_REPASt has used explicit destructors as a stopgap solution pending further investigation of likely modelling paradigms and the correlation of this with the most appropriate scheme for deletion-detection.

5.4. Conflict Resolution for Shared Variables

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 necessary for modelling complex interactions 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.

To demonstrate why this is true consider the sequence of interactions between two federates described in table I.

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 second*.

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 federate[†].

In HLA_REPASt, ownership management is used to implement a set of variable-types, which provide the various semantic behaviours required by agent models, and which are described in table II. 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. Three different types of variables are distinguished[‡], namely exclusive, cumulative and viewable, as described in table II.

In this discussion of conflict resolution we refer to concurrent updates with the notion of *logical-time* concurrency. The HLA's ownership management services, which we use to implement our conflict resolution algorithms, use instead the notion of *real-time* concurrency - they prevent concurrent real-time ownership of a single variable by more than one federate. Because of this notional inconsistency, two

*Note 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.

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

[‡]Note that the given taxonomy for variable types is not necessarily complete. There may exist a hybrid type variable. Consider, for example, a cake. Initially everyone can modify the size of the cake concurrently without violating any preconditions - the **size** variable is therefore semantically cumulative. However, at the point at which the cake is all gone the last piece cannot be taken by two people, so it suddenly acquires an exclusive semantic. To satisfy this form of variable semantic it must be possible to modify the global type of a variable in full synchronisation with accesses to that variable, this is considered ongoing research.



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. (e.g. a door object with a variable open , this must be modelled as an exclusive variable as the precondition for opening (or closing) a door is that it is closed (or open), this precondition being violated by the action itself)	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. (e.g. a stack of pennies with a variable height , this variable must be modelled as a cumulative variable as placing a penny on a stack is not an operation which invalidates its own precondition)	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 that can only be modified by the entity to which they belong. (e.g. a chameleon with a variable color , note that this type of variable could be modelled as any of the three types, as it requires no special constraint, however the executive can minimise the communication and synchronisation necessary for a variable if it can guarantee it will only be modified at the local model.)	if (attribute of local object) update value else throw exception

Table II. Variable Semantics and their Implementations

potential problems exist for logical-time conflict resolution systems based on the ownership management services of the HLA *:

1. Two federates at different points in real time may make requests for ownership with the same logical times. In this situation both federates will be granted ownership, even though a system respectful of logical-time concurrency should detect this conflict and grant ownership to only one of the two requests.
2. Two federates at different logical times may request ownership of the same variable at the same real-time. In this situation one of the federates will experience an erroneous conflict detection.

* Note that these are not *general* problems of scheduling or state mapping. They are problems which only arise due to our interest in providing conflict resolution at the middleware layer. If logical-time mutual exclusion could be guaranteed at the application layer then the mechanisms already described for schedule- and state-mapping would be sufficient.



At present in HLA_REPAST we avoid the first problem through the architecture of the scheduling system itself. To demonstrate this we will consider the example of the algorithm used to implement the exclusive semantic. In the current algorithm a federate, upon receiving ownership of a variable at T_x , will not release it until it is granted time advance to a time $> T_x$. This being true, as long as we can ensure the property that no other federate will execute events at T_x *after* the first federate is granted to $> T_x$ in real-time, then we can ensure that only one federate will actually be granted ownership of the variable at T_x throughout simulation execution. From the time-advance algorithm given in figure 6 and from the guarantees given by the HLA's time-management services, we can observe that our current scheduling algorithm does, indeed have this property. To put this more formally: consider a federate F_a , assume that F_a will modify some variable v at (logical) time T_x . At some point in real time, F_a requests advance to T_x . Because we use conservative time advance with a 0 lookahead, the RTI will not send a time advance grant of T_x to F_a until all other federates have requested advance to $\geq T_x$. At this point two possible scenarios exist:

1. There is no other federate in the federation with an event scheduled for T_x
2. There is some number of federates with an event at T_x . Zero, one, or more of these events may modify v .

In the first scenario F_a will receive a grant to T_x *only* once all other federates have requested advance to a time $> T_x$. Because our time advance system is strictly monotonic, F_a will be the only federate which can ever request ownership of v at T_x during this execution. In the second scenario F_a will only be granted to T_x once (in real-time) all other federates have requested advance to a time $\geq T_x$. After this point assuming F_a is the first federate (in real time) to request ownership of v and is therefore successful, it will then not divest ownership until receiving a time advance grant to a time $> T_x$. Following a similar logic to above we can show that this will not occur until all other federates have requested advance to some time $> T_x$ and can therefore not legally request ownership of v with timestamp T_x .

At present there is no reliable and transparent system in HLA_REPAST for avoiding occurrences of the second problem stated above, that of erroneous conflicts due to requests from federates at different logical times at the same real time. In general the problem will not manifest itself, simply because no federate can execute events at a time $< T_x$ if some other federate is at T_x . The only events that can legally be executed in this situation are events at a time $\geq T_x$, and no federate will be permitted to advance beyond T_x while there exists some federate which has not requested advance beyond T_x . However, following on from the scenario envisaged above, when the RTI sends time advance grants to all federates to time $> T_x$, F_a will divest ownership of v , however this will not happen instantaneously and there will be a period during which the other federates may issue ownership requests for v with timestamp $> T_x$ which will be denied because F_a has not yet divested ownership. Mechanisms for guaranteeing this second problem cannot manifest itself in this way are issues for future research.

6. An Example Model - The Tileworld

In order to evaluate HLA_REPAST, an implementation of the 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 code*.

The Tileworld, introduced in [20], 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 6 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 agents modelling community. The inherent dynamism of the model in terms of rapid (and adjustable) object creation and

*The 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.

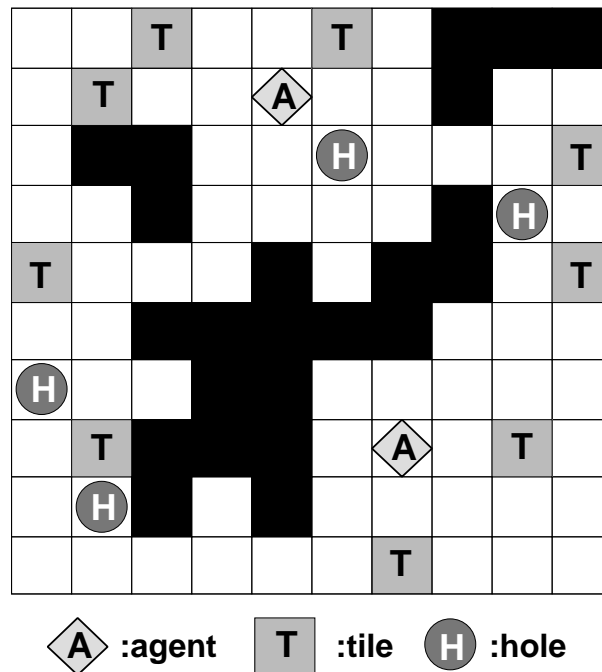


Figure 9. A 20x20 Tileworld

deletion make it useful for analysing the registration and removal mechanisms of HLA_REPAST. The parameterisability of the domain gives 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.

6.1. Sequential Tileworld

The key classes, which comprise the implementation of the sequential Tileworld are detailed below, while the specifics of their attributes and types are given in table III; 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. Its structure is divided in to motive and cognitive components, as demonstrated in figure 10. Both these 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.

6.2. 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

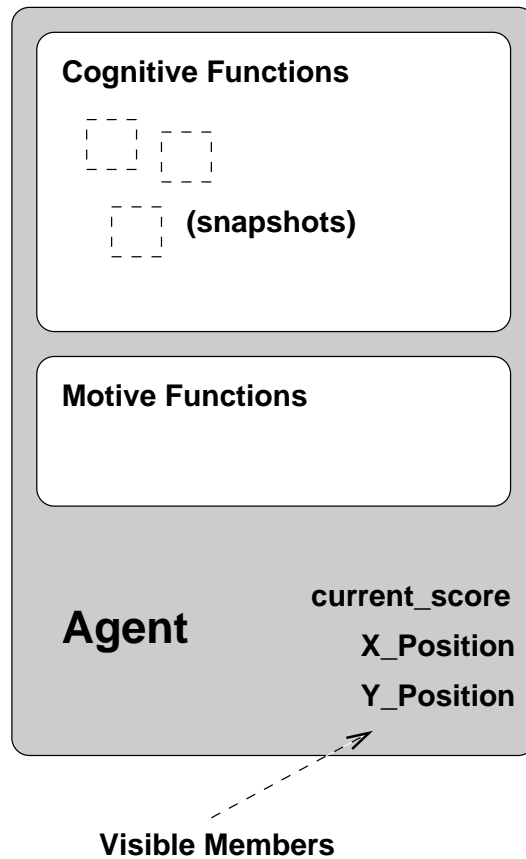


Figure 10. A Tileworld Agent's Internal Structure

one which scales in proportion to the complexity of the model. The reimplementation detailed in table III should therefore be considered in the context of this measurement.

6.2.1. Conflict Resolution

As explained in section 4, one of the areas 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 of the existing protocol and how it was modified, we consider the exclusive-semantic applied to the action of picking up a Tile object. 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.

In general, for HLA_REPAST it has been hypothesised that there exists a correlation (for any reasonably well-behaved model) between the complexity of the sequential model, and the likelihood that it will already



Class	Public Interface (RePast model)	Public Interface (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 Tile held_tile	ViewablePoint position ViewableInteger currentScore

Table III. Sequential Tileworld - Classes and Attributes

employ such a protocol for interaction between agent and environment. In simplistic models, where such a protocol is less likely to exist, the simplicity of the model will minimise the hardship of modifying it for conflict resolution. It is therefore anticipated that this coincident fact will generally minimise the impact of conflict resolution on transparency, although an empirical survey of modelling paradigms is required to justify this assumption.

6.2.2. Object Deletion and the Internalisation of State

The general theme for the modifications so far discussed was to ‘internalise’ the representation of object-state as far as possible. Where previously the ‘held’ status of a tile was represented purely by its external situation (i.e. whether it was held in some slot in the environment datastructure or whether it was held in the slot of some Agent entity), it is now represented explicitly by the internal state values of the object. To clarify this, the reader should refer back to table III, noting how the tile’s ‘held’ status is represented externally in the left-hand column (as the Agent’s held_tile variable), and internally in the right-hand column (as the Tile’s is_held variable). Externally defined state is not something that the variables system described in section 5.3 can easily communicate, hence any externally represented state with global significance must be internalised.

A similar case is encountered for object deletion where, as outlined in section 5.3.3, 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 that 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 6.1. 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 the scheme is generally infeasible for modelling languages that rely heavily on references as an internal mechanism. These questions are considered open topics of investigation at this time.

Generally the implemented system can only report model-state that it can observe. Programmatically this is defined as anything provided to it by the public-interface of the object-classes. It is therefore the



- 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 11. Semantically Dependant Steps in a RePast Event

responsibility of the modeller to ensure that state-changes of global significance are always explicitly represented internally, rather than implied by more subtle relationships between objects.

6.2.3. Contextualising External Events

In sequential Tileworld events of global significance (e.g. an object moving) occur as single elements of larger, semantically related procedures. For instance, an agent moving from (a_x, a_y) to (b_x, b_y) involves the steps shown in figure 11.

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. To refer back to the example of figure 11, a remote federate observing objects of the class `Agent` would see only step 4 of the operation and would need some way of understanding that steps 2 and 3 are implied by step 4 and therefore must be 'filled-in' at the local model.

Two general measures can be used to re-establish model-consistency. One option, similar to the notion of an I/O event (see section 5.2), is to have a general 'consolidation' event, which executes after the middleware has flushed the external event-queue. Once all remote events for a particular time have been processed, the model is scanned for any resultant inconsistencies (for example the agent in figure 11 remains in at grid (a_x, a_y)) and realigns the model to reflect the global state. This approach, although reliable, is extremely inefficient, particularly for models where the number of potential inconsistencies (and hence the magnitude of the scan*) is large in comparison to the number of actual inconsistencies caused by external events. The preferable approach 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 11 an event handler would be registered to the positional attribute of each agent, which executed steps 2 and 3 when an update occurred. Listeners can also be registered to `ReflectedList` instances in order to listen for object discoveries and deletions - for example, to place a new object in the grid at the correct position.

Given certain design patterns this approach can produce inconsistencies. Events in this system can be viewed as a cascade, with each node in the cascade representing one of three types of event, as shown in figure 12. Generally, the purpose of consolidation event-handlers is to re-structure the local model in response to external events. In this sense the cascade below the root event will usually consist only of local-type nodes. As is shown in the example, instability is introduced where the cascade contains external output-type nodes. The given scenario actually shows the pathological case that is clearly an incorrect usage. Here the system is responding to an incoming event by generating a semantically identical event. Because neither the HLA nor the HLA_REPAST system has any way to determine that this second event is a duplicate, it will be treated exactly the same as the first, creating an infinite loop between the two federates. This scenario is plainly a logic error. A more pernicious scenario is presented in figure 13, where a consolidation cascade that, from the local perspective, seems rational, is revealed as unstable

* Note that this magnitude is also determined by the complexity of the relationships between entities. Where there is only a single-dimension of consistency (i.e. as in this example of object-position) the scan's size is proportional to the number of entities. Where there are multiple dimensions of consistency the scan's size increases exponentially, an example of this is an n -dimensional graph where the consistency of v vertices must be checked against all other vertices through n dimensions, giving a scan size of $v^2 n$.

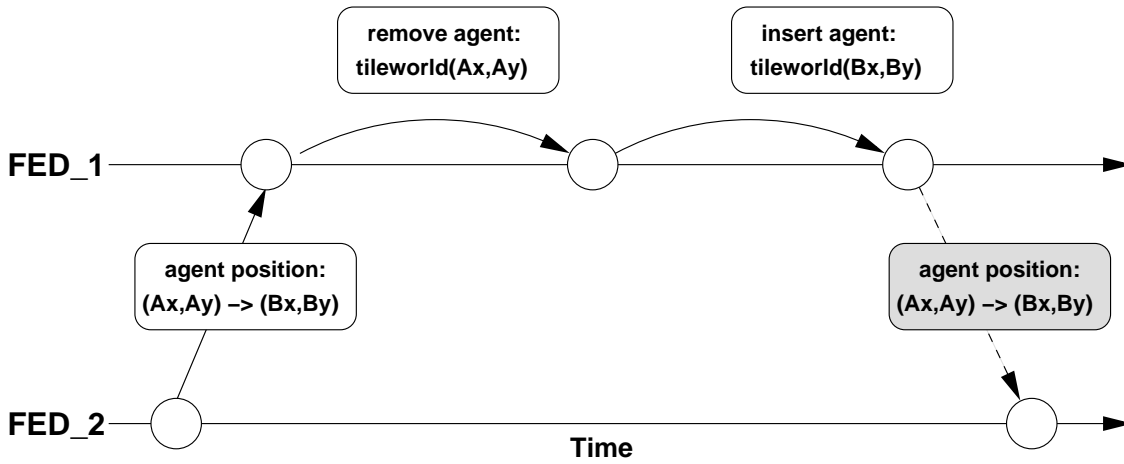


Figure 12. Example of an unstable event cascade. The second position update is erroneous.

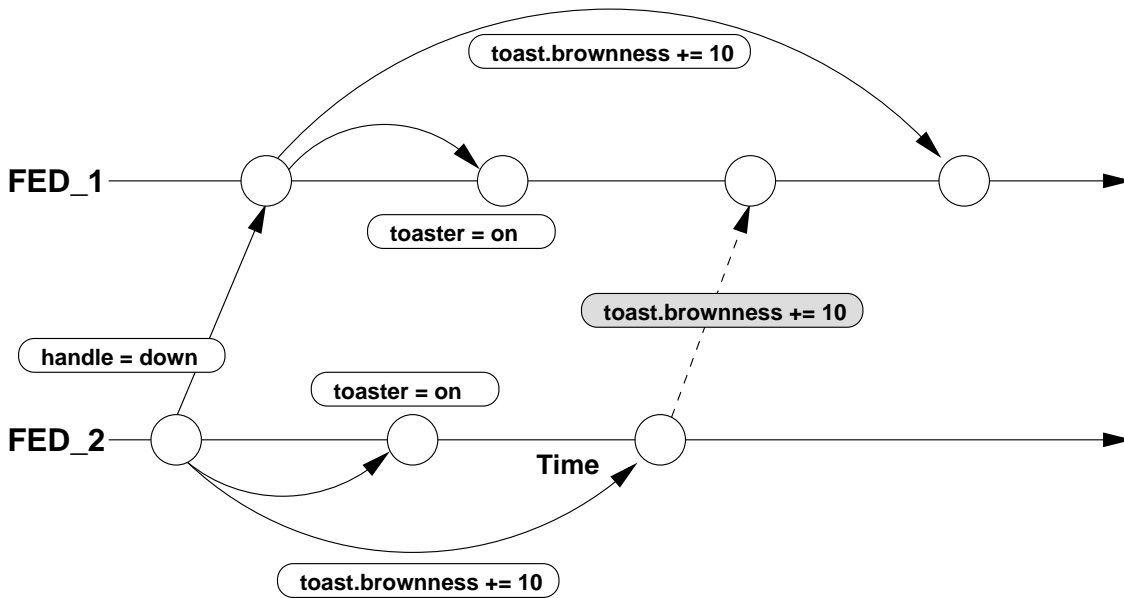


Figure 13. Example of an idempotence-violating cascade. Again the second update is erroneous.

when viewed from a global context. Here the 'brown-ness' of the toast experiences a second, erroneous modification. It should be noted that this scenario is only of danger to non-idempotent variables, such as cumulative variables, or toggle-like variables, for exclusive variables, idempotence is guaranteed by the semantics. The dangers of logic-errors in cascades presented here are part of the motivation for the creation of reliable design patterns for HLA_REPAST models discussed in section 8.



7. Performance

This paper has so far largely been concerned with the engineering challenge of mapping between a sequential and a distributed simulation platform. As section 1 explains, one of the key motivations for doing this was to enable the computation of large simulations on reasonable time scales. This section evaluates the system from a scalability perspective. This evaluation was undertaken by a direct comparison of the sequential and distributed implementations 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. Of the 42 processors available a maximum of 32 were used at any given time.

In all experiments we used two types of federate: the environment federate, which simulated the objects in the tileworld (holes, tiles and obstacles); and the agent federate, which simulated some number of agents, depending on the particular experimental setup.

The experiments varied two model parameters, namely the total number of agents - determining computational complexity - and the rate at which each agent interacted with the environment - determining the beauraucratic workload on the middleware. Other parameters remained static: a 50x50 grid with an obstacle density of 0.15*, a tile-generation probability of 0.05 and a hole-generation probability of 0.02†. All runs were executed over 1000 modelling cycles.

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

The first strategy, depicted in figures 14(a) and 14(b), 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 (i.e. one node).

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 and middleware beauracracy 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 the computation for N agents always being distributed across N nodes, the gradual rise that can be perceived in the distributed case is a manifestation of the central RTI component acting as a bottleneck.

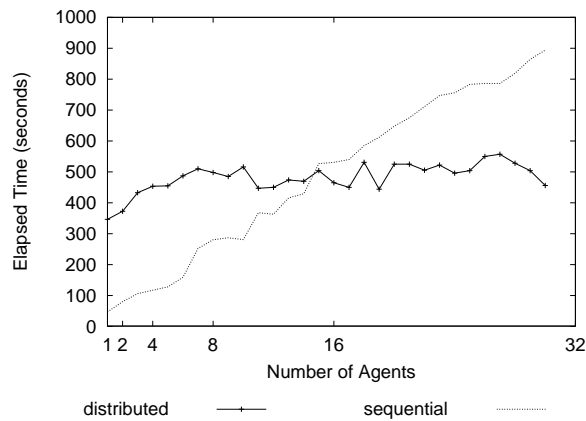
In each figure we observe a point of intersection of the two lines, denoting the level of complexity beyond which distribution becomes advantageous. Note that the precise location of this intersection on the complexity axis depends on the type of agent being modelled‡. As was speculated in [16], from 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 over a given real-time interval. We therefore observe that this intersection occurs early - around 14 agents - on the graph for a model using computationally intensive 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 modelled in each run was kept at a constant. Each run used a different number of federates to distribute this model, starting with one agent-

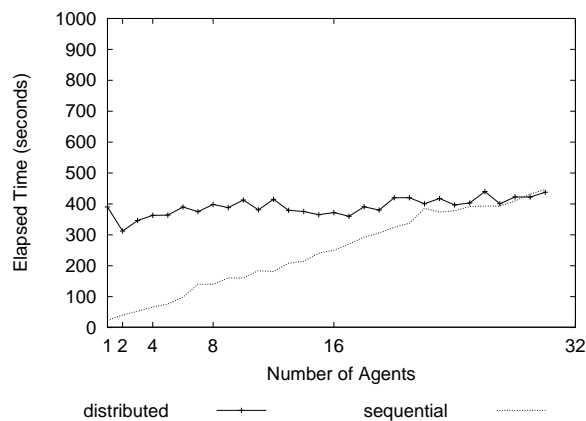
* Being the probability of a given square being occupied by an obstacle.

† These being the probability at any given time step that a new object will be created in the model. The ageing rate of objects was set to a corresponding level.

‡ Note 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



(a) Elapsed Time for 1-32 Computationally Intensive Agents with Scaling Resource Availability



(b) Elapsed Time for 1-32 Reactive Agents with Scaling Resource Availability

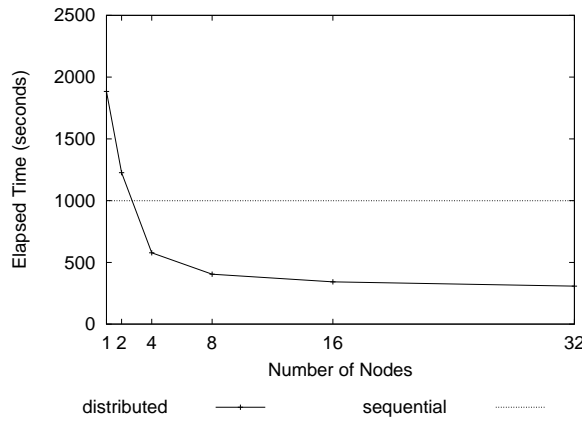
Figure 14. Tileworld with Scaling Resource Availability

federate, then scaling up incrementally to end with enough federates to provide one node per-agent*. The results for 32- and 64-agent models are given in figures 15(a) and 15(b), in both these figures the line representing sequential performance extends across the x axis as a reference.

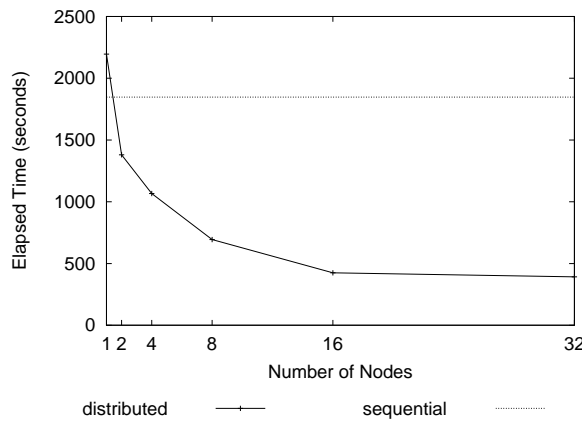
As expected the benefit of distribution is initially sharp, with this benefit levelling off as the magnitude of distribution increases and the RTI spends more time engaged in communication and synchronisation. The figures show that, even with small distributed resources (between 2 and 4 nodes), distribution will improve the performance of the very complex models such as these.

One aspect of figure 15 seems, at first, anomalous. Namely that the initial height of the distributed curve for figure 15(b) (64 agents) is far less than twice that of the distributed curve in figure 15(a) (32

*Clearly in the case of the 64-agent model our 42-node cluster could not provide for a 64-node federation, the 64-agent model therefore also uses just 32 nodes, with 2 agents at each agent federate.



(a) Elapsed Time for 32 Computationally Intensive Agents with Varying Resource Availability



(b) Elapsed Time for 64 Computationally Intensive Agents with Varying Resource Availability

Figure 15. Tileworld with Static Resource Availability

agents). This is further evidence of the extent to which HLA_REPAST is a communication-bound system. If (approximately) doubling the computation required only causes only a small ($\sim 18\%$) increase in elapsed time, we can deduce that in both cases the majority of time is spent performing actions common to both cases. The only common actions are the basic simulation infrastructure procedures of the HLA_REPAST layer: initialisation; time advance; and ownership request/divestiture.

Another possible factor falls out of the observation above, that ‘internal’ agent actions (those that modify only private variables of the agent) are less expensive per-operation in a distributed context than ‘external’ actions (those that modify one or more public variables, such as the agent’s position or the state of some object in the environment). 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 is very quickly 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 in a shorter time.

These factors explain why the elapsed times are similar throughout the experiments for both the 32 and the 64 agent cases.

8. Conclusions and Future Work

This paper has presented the interoperability middleware HLA_REPAST. This system constitutes an approach to the distribution and interoperation of agent-modelling platforms utilising the HLA interoperability framework. The paper has discussed the design issues involved in constructing such a middleware in general and has provided an example of the steps necessary to distribute an existing sequential model.

Our experiments confirm that significant speedup of large simulation models can be achieved by distribution using this system, this being particularly acute in the case of computation bound agents.

From the discussion of the modifications made to the RePast executive in section 5, one can derive some general conclusions. 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 section 5.2), 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.

A logical extension to the firm constraints placed on a model by HLA_REPAST is to define a set of looser modelling guidelines designed to ease the transition of a model from sequential to distributed. Such a set of guidelines may include such nebulous concerns as an adherence to a general object oriented paradigm, to specifics such as ensuring variable access occurs through a method-based protocol within the model. In the absence of a wholly transparent model-executive interface, such primer guidelines could prove vitally important in terms of the usability of the system from a modeller's perspective.

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.

Another important area, which calls for further research is conflict resolution. The Ownership Management services of HLA, based on real-time constraints, are not adequate to address logical-time conflicts. HLA_REPAST exploits the agent-environment interaction protocols already in place in RePast to achieve conflict resolution with a minimal impact on transparency. An empirical survey of other existing MAS modelling paradigms is required to assess the generality of this approach.

To our knowledge, the HLA_AGENT system, described in [16], 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



This work was funded by the School of Computer Science, University of Birmingham and the EPSRC research grant No. GR/R45338/01 (PDES-MAS project*).

We would like to thank Mike Lees at Nottingham University, for the benefit of his experience with HLA, agent 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. RePast projects page. World Wide Web <http://repast.sourceforge.net/papers/>, June 2006.
3. John Anderson. A generic distributed simulation system for intelligent agent design and evaluation. In Hessaam S. Sarjoughian, François E. Cellier, Michael M. Marefat, and Jerzy W. Rozenblit, editors, *Proceedings of the Tenth Conference on AI, Simulation and Planning, AIS-2000*, pages 36–44. Society for Computer Simulation International, March 2000.
4. S. M. Atkin, D. L. Westbrook, P. R. Cohen, and G. D. Jorstad. AFS and HAC: Domain general agent simulation and control. In Jeremy Baxter and Brian Logan, editors, *Software Tools for Developing Agents: Papers from the 1998 Workshop*, pages 89–96. AAAI Press, July 1998. Technical Report WS-98-10.
5. R. E. Bryant. Simulation of packet communication architecture computer systems. Computer Science Laboratory, Massachusetts Institute of Technology, 1977.
6. K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. In *IEEE Transactions on Software Engineering*, volume 5, pages 440–452, 1978.
7. Nick Collier. RePast: An Extensible Framework for Agent Simulation. <http://www.econ.iastate.edu/tesfatsi/RepastTutorial/Collier.pdf>.
8. S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A time warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference*, pages 1332–1339, December 1994.
9. Edmund H. Dufree and Thomas A. Montgomery. MICE: A flexible testbed for intelligent coordination experiments. In *Proceedings of the Ninth Distributed Artificial Intelligence Workshop*, pages 25–40, September 1989.
10. Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.
11. Les Gasser and Kelvin Kakugawa. MACE3J: Fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proceedings of AAMAS-2002*, Bologna, July 2002.
12. 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.
13. Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
14. Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, 1999.
15. 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.
16. 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 517–528. SISO, June 2003.
17. T. Lux and M. Marchesi. Scaling and Criticality in a Stochastic Multi-Agent Model of Financial Markets. *Nature*, 397:498–500, 1999.
18. 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.
19. US Defence Modelling and Simulation Office. HLA Interface Specification, version 1.3, 1998.
20. Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *National Conference on Artificial Intelligence*, pages 183–189, 1990.
21. Patrick Riley and George Riley. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, editors, *Winter Simulation Conference Proceedings*, 2003.
22. Bernd Schattberg and Adelinde M. Uhrmacher. Planning agents in JAMES. *Proceedings of the IEEE*, 89(2):158–173, February 2001.
23. Alexander Serenko and Brian Deltor. Agent toolkits: A general overview of the market and an assessment of instructor satisfaction with utilizing toolkits in the classroom. (Working Paper 455), McMaster University, Ontario, Canada, 2002.
24. Aaron Sloman and Ricardo Poli. A toolkit for exploring agent designs. In M. Wooldridge, J. Meuller, and M. Tambe, editors, *Intelligent Agents II: Agent Theories, Architectures and Languages (ATAL-95)*, pages 392–407. Springer-Verlag, 1996.
25. 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.
26. A. Urmacher and M. Röhl. The role of deliberative agents in analyzing crises in pre-modern towns. *Sozionik*, (3), 2001.
27. 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.

*<http://www.cs.bham.ac.uk/research/pdesmas/>