An Architecture for Modular Distributed Simulation with Agent-Based Models

David Scerri[1]
Alexis Drogoul[2]

[1] School of CS and IT
RMIT University
Melbourne, Australia
firstname.lastname@rmit.edu.au

Sarah Hickmott[1]
Lin Padgham[1]

[2]UMI 209 UMMISCO IRD/UPMC
Institut de Recherche pour le Développement
Hanoi, Vietnam
alexis.drogoul@gmail.com

ABSTRACT

Agent-based simulations are an increasingly popular means of exploring and understanding complex social systems. In order to be useful, these simulations must capture a range of aspects of the modeled situation, each possibly requiring distinct expertise. Moreover, different paradigms may be useful in modelling, ranging from those that use many lightweight reactive agents, to those that use cognitive agents, to those that focus on agent teams and organisational structures. There is need for an architecture which supports the development of a large simulation, through the integration of separately developed modules. This paper describes a framework and architecture which facilitates the integration of multiple agent-based simulations into a single global simulation. This architecture naturally supports distributed simulation and incremental development, which are ways of addressing the computational and conceptual complexity of such systems. In this paper we focus particularly on how to ensure proper management of simulation data that is affected by agents in different modules, at the same logical time. We also provide some preliminary performance evaluation addressing scalability, as well as a comparison of how other available systems handle the issue of shared data.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]

General Terms

Design, Experimentation

Keywords

Agent-based Modeling, Simulation, Modular Architecture, Shared Variables, Conflict Resolution

1. INTRODUCTION

Agent-Based Modelling is emerging as a popular technique for simulating complex social systems in order to inform areas such as policy development and enhance theoretical understanding of social phenomena (see e.g. [5, 8, 2]). However, in order to be useful, such simulations must typically capture many different aspects of the relevant area. This makes them time consuming and complex to develop, as well as requiring various areas of expertise. In addition, different paradigms may be more appropriate for different aspects, ranging from paradigms that use many lightweight reactive agents, to those that use cognitive agents, to those that focus on agent teams and organisational structures. Finally, there is some criticism that current usage of agent based models lacks transparency and requires over-simplification to the point they can not provide insightful results [4].

In our work on agent-based simulation for exploring approaches to city-based adaptation to climate change, we aim to build up a complex simulation by incrementally adding new agent-based models created by members of a large distributed community, interested in the application area. We have thus developed a platform which allows individual modules, possibly pre-existing and implemented under different paradigms, to be integrated in a common environment. The architecture aids the creation of more complex global simulations, whilst at the same time improving the transparency of single modules by reducing their domain of interest. It also naturally supports distributed simulation, which is one way of addressing the computational complexity of such systems.

Transparency of the global simulation requires that the execution framework integrates the modules in a principled and fair manner, that does not lead to anomalies with respect to the domain being modelled. One particular issue which arises is how to appropriately manage the conceptually parallel execution of all modules, at a single logical timestep of the simulation. This is one of the key issues explored in this paper.

The High Level Architecture (HLA) standard [1, 14] developed by the US Department of Defense is a widely used specification for distributed simulation which supports interoperability of separate heterogeneous modules. However, it does not adequately address the issues that arise in the kind of modularisation we are targeting. In particular difficulties arise when multiple modules affect the same variable at the same logical time. HLA assumes that any variable (an object attribute in HLA) is owned by a particular module [9]. While that ownership can be transferred between modules, allowing multiple modules to update the variable, it does

Cite as: An Architecture for Modular Distributed Simulation with Agent-Based Models, D. Scerri, S. Hickmott, A. Drogoul and L. Padgham, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

not provide any control nor support for managing logically concurrent access to a variable. The type of modelling done in many HLA applications avoids the need for this. For example if different spatial/location areas of the simulation are covered by different modules, then objects will not need to be modified by two modules at the same logical timestep (as they cannot be in two places at once). However we explicitly aim to model multiple processes potentially affecting the same aspect of an environment. For example both a disease module and an immigration module would affect population in a particular location.

Work such as [10] provides some basis for the interoperation of separate agent-based models via HLA-Repast, a generic middleware layer which integrates HLA and Repast (a popular multi-agent simulation toolkit [3]). However this work has a different focus to ours, seeking to modularise an existing simulation for the purpose of distributed execution (as opposed to building a simulation through the incremental addition of new modules). Consequently their approach assumes a level of homogeneity among modules which we want to avoid. In particular, with respect to their management of atomic actions involving multiple variables, HLA-Repast requires updates to be logically linked, so that multiple variable changes can be inferred from a single variable update. This is not practical when incrementally integrating independently developed modules, and is also not transparent.

The architecture presented here is based on the idea that conflicts over shared data should be resolved at a more global level than the modules involved, as otherwise modules are (potentially) required to have information about each other. At the same time, for the purpose of modularity, conflicts involving specialist knowledge of the module should be resolved locally within that module.

In the following section we use an example to explore the challenges of simulating the conceptually parallel execution of multiple modules which are dependent on both shared and local data. We then describe our architecture and implementation, which addresses these and other issues and provides a sound basis for integrating multiple modules into a common simulation environment. We finish with a preliminary performance evaluation that considers the case of both computationally intensive and lightweight agents. We also describe in detail the behaviour of our system as compared to other systems in the literature, with respect to actions requiring shared data.

2. EXAMPLE

In this section we incrementally develop an example involving a module which requires the use of both local and shared variables. We show some of the problems that can arise when the module executes an action involving one or multiple shared variables. This then sets the scene for some aspects of our architecture.

Consider an agent-based model (ABM) module which captures the spread and treatment of the H1N1 virus. The availability of H1N1 antivirals is modeled by a variable which is local to this module. Shared variables capture the availability of a hospital bed, and public funding for the issue of treatment, both of which may be required by other modules. A successful treatment for a person infected with H1N1 requires the availability of an antiviral, a bed and sufficient public funding. The amount of public funding required to treat an infected agent depends on their type of health insurance.

Module Name : H1N1

Local Variables : `Antivirals`, `Agent.infectedStatus`, `Agent.insurance`, `InfectedAgents`*, `HealthyAgents`*. *These are virtual lists which return the subset of agents such that `Agent.infectedStatus` is set to `true` or `false` respectively.

Shared Variables : `BedsAvailable`, `PublicFunding`

Description : Each agent represents a single person. At each timestep, agents interact and with some probability a member of `HealthyAgents` may contract H1N1 from a member of `InfectedAgents`. All members of `InfectedAgents` progress through a period of infection and eventually execute a `Request-Treatment` action. This atomic action succeeds if the agent can occupy one of the `BedsAvailable`, consume one of the `Antivirals`, and use some amount of `PublicFunding` which is dependent on their type of insurance (`Agent.insurance`). In this case, the agent quickly recovers and `Agent.infectedStatus=false`. If there are insufficient `BedsAvailable`, `Antivirals`, or `PublicFunding`, then the action fails in its entirety, in which case the agent dies or recovers naturally with some probability.

2.1 An Independent Module

Let us gradually work up to this problem. First consider the case where the only condition for `Request-Treatment` to succeed is that it is possible to consume one of the `Antivirals`. If there are multiple requests at the same logical time, then the module may randomly select and succeed these requests until such point that all the `Antivirals` are consumed. This is indeed how this single H1N1 module may be simulated, e.g. in Repast. It seems fair to say that the outcome of this sequential execution would model the reality of multiple infected persons seeking treatment at the same time, with only a subset succeeding if there are insufficient resources for all.

2.2 A Single Shared Variable

Now let us introduce one of the shared variables, `PublicFunding`. Suppose now that the conditions for `Request-Treatment` to succeed are that it is possible to consume one of the `Antivirals` *and* there is sufficient `PublicFunding` to pay for the treatment.

If the H1N1 module had a number of `InfectedAgents` it may wish to `Request-Treatment` for all of them at a single logical timestep. If another module also needed to access `PublicFunding` then it could be considered *unfair* if the H1N1 module consumed all of this resource by obtaining ownership of the object (as would be done in HLA), and not releasing it until all its actions were completed. We would prefer a mechanism that only allowed modules to perform a single action affecting shared variable(s) at a time. This could potentially be accomplished using HLA if ownership release was enforced after every update of a variable. There could be a similar issue if, for example, a single action in one module consumed a large amount of a shared resource. Randomisation of the order in which modules are allowed to act addresses this to some extent, similarly to the way the independent module discussed above was fair in its random distribution of `Antivirals`. If it makes sense for the domain, the modeller can potentially split the action into smaller

ones to enable more even distribution between modules. It is important to aim for a framework where this option remains in the modeller's control.

2.3 Multiple Shared Variables

We finally introduce the second shared variable, `BedsAvailable`. Now the conditions for `Request-Treatment` to succeed are that it is possible to consume one of the `Antivirals`, there is sufficient `PublicFunding` to pay for the treatment, *and* `BedsAvailable` is at least one.

The previous approach of obtaining exclusive local ownership of shared variables can now suffer from deadlock. For example, suppose the H1N1 module obtains ownership of the `BedsAvailable` variable; meanwhile another hospital-related module is holding on to `PublicFunding` and awaiting ownership of `BedsAvailable`: deadlock. Observe that this problem does not usually occur in applications using the HLA because modules do not generally modify the same variables at the same logical time; however, we aim to model situations where this is needed.

HLA-Repast avoids these issues by defining middleware which transparently obtains ownership whenever a shared variable is updated within the module. In the case where an action needs to modify multiple variables, an event-handler can be used to infer other changes from a single variable modification, i.e. updating one shared variable triggers the automatic update of some related variable(s). This approach suffices for the example in [10] because it assumes homogeneous modules and updates which are clearly and logically linked. However in a system like the one we are developing their approach becomes quickly impractical if not impossible.

2.4 Central Coordination

It is apparent that it may be useful to have a central agent that owns all the shared variables within a certain scope and takes full responsibility for coordinating and applying interrelated updates to these variables. Modules will not need to acquire ownership of shared variables themselves, and thus there will be no problem with deadlock when actions involve multiple shared variables. Furthermore we will not require every module to know the variable dependencies entailed by the actions of every other module. A means of coordination which is external to the modules would also have other benefits, such as allowing for more reasoning power with respect to the distribution of a shared resource. For example multiple requests can be taken, prior to deciding which ones will and will not succeed.

There are some anomalies which can arise through this approach however, and must be accounted for. Firstly, consider that we will be allowing a module to request transactions involving multiple shared variables. This requires some means of grouping related modifications so they are applied (or rejected) in their entirety. For example, if the H1N1 module's request to decrement the number of `BedsAvailable` is rejected, then its related request to use some amount of `PublicFunding` for treatment should also be rejected.

Secondly, consider that the success of an action depends on internal and external constraints, i.e. constraints involving both local and shared variables. Once an update to a shared variable is committed, it should not be the case that the update needs to be rolled back due to a constraint internal to a module being violated. Thus a request to update shared variables should not be made unless local conditions for success are met. At the local level the module can choose to either suspend until the result of a possible external conflict is known, or save the state at the point where the request is made, rolling a module back to this state if the action fails. Rolling a module back may require a significant amount of information to be stored every time an action involving shared variables is executed. Either way, careful consideration must be given to ensure the final output implies no commitment to the rejected action. To see this, recall that the particular amount of `PublicFunding` needed to succeed a `Request-Treatment` action will depend on `Agent.insurance`, e.g. private or public. Now suppose that `Antivirals==1`, and two agents initiate `Request-Treatment` actions at the same logical time. Only one of these agents has private health insurance. At the same logical time, the value of `PublicFunding` is only sufficient for a patient with private health insurance to be treated. Clearly, resolution of the internal and external conflicts associated with this request should not result in both agents being denied treatment. That is, if two actions are internally conflicting, and one of these experiences conflict externally, then the other action should have the opportunity to succeed.

3. ARCHITECTURE FOR MODULARITY

We describe here an architecture to realise the vision of a platform which allows the integration of multiple modules in a common simulation environment, where each module captures the representation and reasoning about some specialised aspects of the domain (e.g. disease spread, employment, economic aspects, etc.)

We have implemented a modified version of Civilization IV as the user interface for the system. It serves as a 3D environment for showing the state of the overall simulation as well as allowing the user to interact with a given city in ways which can affect the operation of the modules. This popular game platform was chosen for its openly available SDK and potential for modification. There is also an established community of modifiers of the game which supports reuse and collaboration.[1]

The architecture provides some centralised services which integrate separate modules into a common simulated environment. These services support reasoning within a module over local data, but controls the access to shared data in a manner that models logically concurrent access to shared variables where appropriate. The resolution of conflicts at the module and system level is managed to avoid the possible anomalous action failures described in the previous section. This approach facilitates both distributed execution and reuse of existing models.

Our architecture currently provides two main services to the modules in the system. The first is that of *Time Manager* (TM), which ensures that all modules advance time in a consistent way. This is implemented in much the same way as the conservative algorithm of the HLA [6]. The second service is that of the *Conflict Resolver* (CR), which manages module interaction with shared data to avoid the potential problems identified in the previous section. This service implements the global co-ordination

[1]For a list of open source modifications based on Civilization IV see e.g. http://sourceforge.net/projects/civ4mods/.

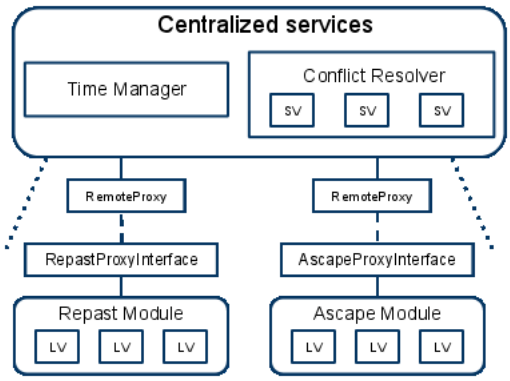

Figure 1: The system architecture, where LV = Local Variable and SV = Shared Variable.

approach described in the previous section, by encapsulating all shared variables and regulating access to them in a fair and principled way.

In the following, we define the world state as a set of value assignments to shared variables. The local variables of a module are those variables which can only be modified by that module. Shared variables are those that may be modified by multiple modules. Conflict is the potential for the violation of a variable constraint. This can be internal conflict, i.e. involving local variable(s), or external conflict involving shared variable(s).

3.1 Time Management and Synchronization

The Time Manager service dictates the current logical time of the simulation, based on indications from modules regarding their next desired time of execution. Within these constraints, the modules must manage their own execution.

The TM must support both: (1) time-driven modules operating at different levels of granularity, and (2) event-driven modules. To do this we use a scheme similar to the conservative scheduling of HLA [6], which addresses a number of the synchronisation subtleties discussed in [7]. Modules must submit a request to the TM, to advance to a new logical time. The TM then sorts these and picks the earliest time in the queue, say T_1. All modules are notified that the time will be advanced to T_1, and at the same time receive an update of the world state (from the CR).[2] The updated state information may trigger execution of event-driven modules. The new logical time may trigger execution of time-driven modules.

The modules to run are initially included in an *active list*. Each module in this list runs, executing actions as appropriate, until they indicate that they are done, at which point they are removed from the active list. For time driven modules this will be via a time advance request. An active module may intermittently stop to await a response from the CR regarding a request to execute an action in the environment (see Section 3.2). When the active list is empty (i.e. all modules have finished executing for the current timestep), the TM service advises the CR that the time will be progressed. This causes it to send out state updates to each module for any variables they have registered an interest in.[3] The TM then advances the logical time to the minimum of all the requests, and the process repeats.

We note that no agent can access changes to shared variables *during* a logical time step. Thus agents are not able to change their intended actions based on something which occurred at conceptually the same time, which we consider an important characteristic for appropriate modelling.

3.2 Action Execution and Conflict Resolution

The key features which we have aimed for, and believe are important in our Conflict Resolver are:

1. Effects of atomic actions are applied consistently, i.e. as a single transaction.
2. The interaction of internal and external conflict resolution must not result in anomalous failure of actions.

We define an *action*, as is usual, by a set of *preconditions* which are necessary for the action to be successfully executed, and a set of *effects* which are the relevant changes to the world state that result from the action. We define an *external action* as consisting of the subset of preconditions and effects which refer to shared variables (which we will call *pre* and *eff* respectively). When a module wishes to execute an action involving shared variables, it must send a request to the CR containing the external action. The CR will then inform the module whether or not the action has executed successfully (in which case any local effects can also be applied).

The CR service is responsible for allowing or denying actions requested by the modules based on the concurrency semantics and constraints of the variables involved. Suppose that the current logical time has just been advanced to T_1, and the world state at this time is S_{T_1}. Furthermore, $M = \{m_1...m_n\}$ is a set of modules wishing to execute actions at logical time T_1; $A = \{a_1...a_n\}$ is a set of pending actions, with a maximum of one action per module; and, C is the set of all actions approved for execution at logical time T_1. The CR fails an action $a_x \in A$ for one of two reasons. Firstly, a_x fails if one of its effects will violate a variable constraint, based on the variable's value in the state $S^C_{T_1}$, where $S^C_{T_1}$ is the state S_{T_1} updated with the combined effects of actions currently in C. Secondly, a_x fails if it is deemed to be semantically mutually exclusive with any of the actions in C. For example the standard notion of mutually exclusive actions employed by the AI planning community could be adopted [13], which would entail e.g. if the preconditions of a_x are clobbered by the effects of an action $a_y \in C$ then a_x is mutually exclusive with a_y. If a_x is failed, then an ACTION_FAIL reply is sent to the appropriate module, m_x, and the action is removed from the pending set A. Otherwise an ACTION_SUCCESS reply is sent, and a_x is removed from A and added to C. The CR continues to take actions from A and process them in this manner. Once the TM indicates that time will be advanced to T_2 (at this, point the set A will necessarily be empty), the CR executes the set of actions C to obtain S_{T_2}, and sends this information to all modules.

To ensure that no module has an unfair advantage in accessing shared variables, it may be that the CR will delay processing a module's action request until other modules have had the opportunity to make requests. Currently we disallow modules getting ahead by more than one action, but this number can be varied. It is also possible, if desired,

[2]We could easily exclude some modules if we have knowledge that they will not run at this time.

[3]Modules may be interested in updates to a larger set of variables than those they affect. Some variables may provide input to the module, without ever being affected by it.

to wait until having received an action from a number of modules in order to potentially do some further processing regarding a conflict. However this is likely to have some performance costs.

In our system we actually have a hierarchy of CR services for different levels of granularity in the system. For example there is a CR service for variables shared within a single city, but not beyond, a CR service for each grouping of cities, such as a country, and a CR for the entire simulation. Variables are declared and managed at the lowest level possible. It could also be possible to partition variables within a level and have multiple, distributed CRs, each with a separate partition of variables.

3.3 Execution within Modules

Modules should not execute beyond the end of the current logical timestep, at which point they receive a world state update from the CR service, and can execute (if they wish) within the next logical timestep. Modules must manage their own execution within a timestep; this can be done in an optimistic or conservative manner.

The conservative approach is to halt the module, each time an action request is sent to the Conflict Resolver, until a response is received. The optimistic approach is to assume the request succeeds, and continue processing. If the action is subsequently failed, then the module must roll back to the state at which the request was sent, and resume processing with the knowledge the action actually failed. Note however that a module can not send a second request to the CR, until the result of the first request is received. To accommodate this, additional action requests could be buffered, with failure of the first action resulting in the buffer being emptied and the state being rolled back appropriately.

The conservative approach is clearly much simpler, however it also means that there is likely to be some wasted processing time while modules wait for a response. In scenarios where the wait time for a response is high, and the processing required by a model between each action being determined is also high, it may be preferable to adopt the optimistic approach. The modules we have implemented so far all adopt the conservative approach.

4. BUILDING OR ADAPTING MODULES

4.1 Module interface requirements

ABMs may be developed in many different languages and programming paradigms, although as our framework is developed using Java, each module will need to provide a Java interface to communicate with the rest of the system. For Java based modules, including ABM models developed in Repast[11, 3] or Ascape[12], this will simply involve implementing an appropriate subclass. For modules developed in other languages, a Java wrapper will need to be developed specific to the module, although this should be trivial. At present, we have implemented two basic interfaces `RepastProxyInterface` and `AscapeProxyInterface` which provide the basic functionality required to connect Repast and Ascape models respectively. These interfaces take the model settings as arguments, and initialise the model along with a specialised implementation of the Runner class to control the execution of the model. When the ProxyInterface receives a message to advance time from the TM service, the interface calls `step` on the Runner. When the model makes a call to `performAction`, the interface will send the request to the CR service, then block until a response is received, thus implementing a conservative approach transparently to the model.

4.2 Adapting Existing Models

Aside from providing the interface, there are some changes which need to be made to existing ABMs to allow them to conform to our framework requirements. Firstly, we must identify each of the variables used in the model, and determine whether they are local or shared. Naturally, this may change depending on the configuration of modules in the framework. Secondly, for each of the shared variables, we must develop a mapping between their names and the names of those available externally. Currently we are doing this manually at each `performAction` call, but we are working on an approach which will implement an automated mapping, based on an interface declaration. Thirdly, we must identify which areas of the code involving shared variables should be performed atomically, and consider these as actions. In these areas, we must make some changes in order to avoid anomalous behaviour and ensure consistent updates. The new code should follow the pattern:

1 Check action preconditions.
    If preconditions are not met, end.
2 Build external action and call `performAction`.
    If external action fails, end.
3 Apply local effects.

Algorithm 1 shows psuedocode for the `Request-Treatment` action in the H1N1 model, as designed to be run as an independent simulation; Algorithm 2 shows how the action would need to be adapted for our system, assuming the shared variables are `BedsAvailable` and `PublicFunding`. Although the rewriting looks substantial, it is well defined, and happens only at points where shared variables are involved.

Algorithm 1 H1N1 `Request-Treatment` action designed for independent simulation

```
fundsReqd = (Agent.insurance==Private ? 2000 : 6000);
if (Antivirals > 0 && BedsAvailable > 0
                   && PublicFunding > fundsReqd)
  BedsAvailable–;
  PublicFunding -= fundsReqd;
  Antivirals–;
  Agent.infectedStatus = false;
endif
```

5. EVALUATION

We evaluate our system firstly by providing some experimental data regarding the execution time of our model, and secondly by comparing our approach to shared data management with other approaches in the literature.

In general, the communications cost of using multiple machines needs to be balanced by a gain in efficiency due to the use of multiple processors, if there is not to be an overall loss of efficiency. As is shown in [10], systems with computationally intensive agents will benefit most from the addition of extra CPU power, while those with lightweight agents will be most affected by the extra cost associated with communications.

Algorithm 2 H1N1 `Request-Treatment` action adapted for modular simulation

```
fundsReqd = (Agent.insurance==Private ? 2000 : 6000);
if (Antivirals > 0)
   effects.add("BedsAvailable", -1);
   effects.add("PublicFunding", fundsReqd * -1);
   externalAction = new Action(preconditions=null, effects);
   externalResult = interface.performAction(externalAction);
   if (externalResult)
      Antivirals–;
   Agent.infectedStatus = false;
   endif
endif
```

In this example, from the perspective of the module there are no preconditions; all conditions for success are determined by the external constraints on `BedsAvailable` *and* `PublicFunding`.

For our experiments with computationally intensive agents we used a network of 30 machines connected by a gigabit ethernet. Each machine had 2 x Quad Core, AMD 2356, a 2.3GHz CPU (i.e. a total of 8 cores), 32Gb RAM and was running CentOS 5.3. In each run we had 2400 agents, with ten per module. Modules were distributed as evenly as possible across the available machines. Each agent was programmed to take between 9 and 12 milliseconds to compute each action, and 10% of the actions of each agent involved shared variables. As a comparison we ran the same number of agents sequentially within a single module, on a single machine, with no communications (as an integrated rather than modularised ABM). This resulted in an average time of 26080ms per timestep, more than six times as long as the distributed version on a single machine. This is omitted from Figure 2 as the scale required would reduce clarity. Each execution ran for 100 timesteps and an average was taken. Each configuration was also run 5 times, but there was little variance so we simply averaged these. All the modules were coded using Repast, with the sequential model being pure Repast, and the distributed models being Repast connected to our system.

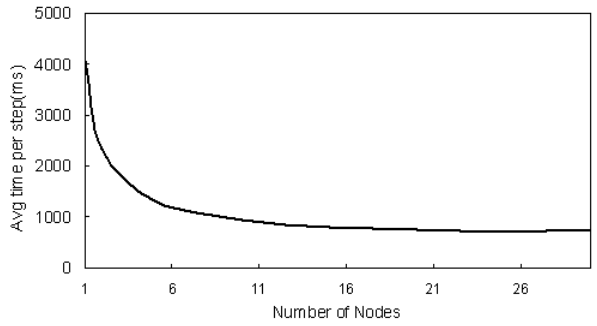

Figure 2: Average time per step for 240 Computationally Intensive agents with varying resource availability

The results show that for these computationally intensive agents our model is always preferable to running all agents in a single module.

To assess the impact of our model in situations where we have very lightweight agents using negligible processing time per action, we ran an experiment with 1000 agents in each module, varying the number of modules from one to ten, on a single machine. The machine used was a Windows XP, 3GB RAM, Intel Core Duo at 3.16Ghz. We used two variations, one where we assumed that all agent actions involved shared variables (and therefore required interaction with the CR), and another where we made the assumption that 10% of agent actions would involve a shared variable. As a baseline we ran the same number of agents in a single module(also on a single machine), but with no communication with a CR. The results can be seen in Figure 3. As expected there is a cost for communications which increases with the number of agents (i.e. the number of communications). However, we note that without such communication, either there can be no data which is affected by multiple modules at the same time, or all modules must be integrated into a single piece of software which internally resolves conflicts between agent actions at a given time step. With a cost of approximately 4 seconds for each time step in a system with 10,000 agents with 10% actions involving shared variables, we would argue that this approach is scalable, although of course this is affected by the granularity of the simulation one wishes to run.

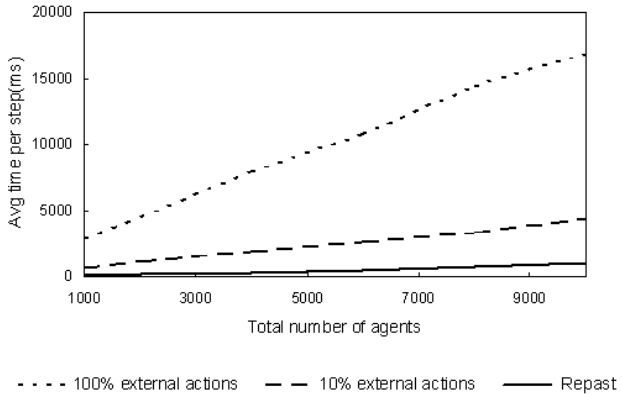

Figure 3: Execution time for varying numbers of lightweight agents

We now proceed to evaluate our approach with respect to the issue of how to allow data to be modified by agents from different modules, at the same logical time-step. We have found two other pieces of work that recognise, and attempt to address some of the limitations that arise with HLA regarding maintaining logical consistency when multiple modules affect the same object/variable during a single time step. In HLA, modules must first request ownership of the object to be modified, and only then can they affect it. HLA ensures that only one module can have ownership of an object at any one *real* time. However this can lead to inconsistencies that either wrongly allow joint ownership at the same logical time, because real-time differs, or alternatively wrongly disallow ownership at different logical times, because the real time happens to coincide (see [10] for detailed analysis of these issues).

Wang et al [15] describe a system where they use a middleware component called an *Interaction Resolver* (IR) which is provided at each module to reconcile inconsistencies when updates from other modules are received. Each IR is implemented identically, so that they conclude the same result without requiring extra communication. However their approach leads to the kind of problem we mentioned briefly at the end of section 2.4, where unwarranted failures

can occur due to the interaction between shared and local variables. Considering the earlier example, let us assume we have two modules M1 and M2. M1 has the following (partial) internal state:

`InfectedAgents(Agent.insurance==public):A1, A2;`
`InfectedAgents(Agent.insurance==private):A3;`
`Antivirals=2.`

The (partial) world state is:

`BedsAvailable=6; PublicFunding=$15,000.`

Suppose now that in a particular logical timestep all 3 of our `InfectedAgents` request treatment, requiring for each agent an antiviral, a hospital bed, and either $2,000 or $6,000 of public funds, depending on whether or not they have private insurance. Assume also that in the same timestep an agent `A4` from M2 requests an action that uses $4000 of public funds.

In the system of Wang et al this situation will result in M1 making some decision regarding which of the 3 agents will receive the antivirals, of which there are only 2. Let us assume that it is decided that A1 and A2 will receive antivirals. At the end of the timestep, each Interaction Resolver will receive the updates from the other module and integrate their own updates. As the combined updates of A1, A2 and A4 would violate a constraint on `PublicFunding`, the IR must disallow one or more of these. Let us assume now that the IR decides in favour of A1 and A4, in which case the effects of A2 will be reversed. Consequently, overall, both the treatment requests of A2 and A3 have failed. However this is not necessary. There is an antiviral left, and sufficient public funds to treat A3 (who has private insurance and requires less public funds). We refer to this situation as anomalous behaviour.

HLA-Repast would avoid the anomalous behaviour described, but there are other issues. Firstly HLA-Repast's approach to inter-related modifications to shared variables is not appropriate for our context. Let us consider the above example from the perspective of HLA-Repast: when A1 executes a `Request-Treatment` action M1 will initiate the action effects by requesting the value of *one* shared variable, e.g. `BedsAvailable`, to be modified accordingly. The provided middleware will then decrement the value of the `BedsAvailable` variable and notify all interested modules of these changes. This will require an event handler associated with each module to infer that a decrement in `BedsAvailable` is associated with a decrement in `PublicFunding`, and subsequently update the module's knowledge of `PublicFunding` also. This solution is not appropriate for systems like ours, as it would require M2 - which may otherwise have no interest in `BedsAvailable` - to monitor this variable and make inferences based on any changes. Moreover, quantifying the decrease in `PublicFunding` will require M2 to know whether the particular agent(s) occupying one of the `BedsAvailable` has public or private health insurance. Furthermore, there may be other modules using hospital beds which subsequently affect different shared variables. Defining `PublicFunding` as the primary variable has similar problems: any modification via M1 would infer a change to `BedsAvailable` whereas a modification via M2 does not.

Another issue that arises in HLA-Repast which is addressed in our architecture, is the problem that can arise in HLA (and which HLA-Repast does not provide a solution for), where unfortunate real time ordering can still result in failures, even though there is not a logical time problem. In our example, imagine that instead of the 4 agents wanting to act at the same time-step, we sequentialise them so that A1 acts first, then A4. (A2 and A3 are not needed in this illustration). The sequence that *could* happen in HLA-Repast is as follows. We represent logical time as T_i and real-time as t_i.[4]

`T1 t1 A1 requests ownership of PublicFunding.`
Request succeeds, and therefore action succeeds.
`T2 t2 A4 requests ownership of PublicFunding.`
Request fails as A1's release has not yet arrived.
`T2 t3 A1 releases ownership of PublicFunding.`

Use of conservative time management ensures that mostly this issue will not arise. However, because things do not happen instantaneously, and some sequentialisation is employed, the above anomaly can occur in HLA-Repast, as noted by the authors. In our architecture there is no use of ownership management (or rather the CR owns all shared variables in a particular scope), and therefore this issue does not arise.

Pure HLA is not able to link modifications to `PublicFunding` and `BedsAvailable`, without running the risk of deadlock. If these modifications are not linked one may assign a bed, but not the funds, leaving the system in an inconsistent state.

Our system will ensure that the action is handled atomically, i.e. either all the resources needed for a treatment are utilised, and it succeeds, or none, and it fails. It will also ensure that there are no anomalous or unnecessary failures. The first step in our system will be to process an action request for A1 to be treated. The M1 module will ascertain that the local preconditions (availability of an antiviral) are met, and then send the external-action request to the CR. Our CR will process, and succeed, the treatment request for A1, creating a commitment of $6,000 against the `PublicFunding`, (as well as the commitment of one of the `BedsAvailable`). On receipt of the success from the CR, M1 will execute the action locally, including decrementing the value of `Antivirals`. The action request from M2 regarding A4 will be similarly processed by the CR, resulting in a total commitment of $10,000 against the $12000 of `PublicFunding`. M1 will then start a similar process for A2, but when this is processed by the CR, it will realise that only $5,000 uncommitted funds are available, and the CR will notify M1 that its external action request failed. M1 will therefore not execute this action locally (and no antiviral will be used). Now M1 processes the `Request-Treatment` action for A3 (for which there is still an antiviral available), sending the external-action request to the CR which will succeed the request as it requires only $2,000 of the remaining $5,000 of public funds. We note that in our system it is important that the relevant local and shared data that interact are part of the same action. This however seems a natural requirement. We also require that local preconditions are assessed prior to external-action requests involving shared data. In this way if external-action requests are succeeded, the shared data is guaranteed to be affected.

[4]We also assume here that `PublicFunding` is defined as an exclusive variable for HLA-Repast, meaning it can only be used once in a logical time-step, as they do have a solution for what they call cumulative variables. Public funds would intuitively be cumulative, but it can nevertheless illustrate the issue.

6. DISCUSSION AND CONCLUSION

In this paper we have presented an architectural framework we have developed that allows us to start to realise the vision of building a complex simulation by combining specialised heterogeneous modules, some of which may well be pre-existing. We have described in detail some data management issues that can arise in this situation, when processing actions that logically happen in parallel, across different modules. In particular problems can arise when agents within modules have actions which affect both local variables, and multiple shared variables. Our architecture appropriately manages these issues. We are not aware of any other distributed agent based platform which does this.

Our approach allows for a transparent, changeable implementation of concurrency semantics, i.e. when is it semantically not appropriate for two external actions to be executed at the same logical time. This partially addresses the need for increased transparency in agent based simulations, regarding framework processing that can influence the outcome of simulations, but is not explicitly part of the model.

The approach is extensible to allow for more sophisticated conflict resolution managers, based on information about a particular shared variable/object. For example some variables may represent objects that can be used only once, or a limited number of times, within a logical time period, while others can potentially be, conceptually, simultaneously affected by as many modules as desired within a logical timestep. In HLA-Repast [10] these are referred to as exclusive (one use) and cumulative usage. Our approach can be extended to accommodate a more flexible version of this idea.

The approach could also be extended to allow for some actions to be partially successful. For instance an action request to water the crops with 1,000 litres of water may partially succeed, watering with 500 litres. This is something we are currently working on.

One issue which does arise in the current framework is that "fairness" is addressed between modules rather than between agents. We do not allow one module to execute (too many) additional actions before other modules have also been able to put forward action requests. However, this does mean that agents within a module with few agents will potentially be advantaged over those in a module with many agents. We are exploring possible approaches to address this issue, and we also aim to investigate it by integrating existing modules and examining the behaviour arising.

We are working to address the fact that there may be some variables which are only modified by a single module, but read by many. In our current implementation, this would require declaring the variable as shared, so that the other modules can receive any changes via the Conflict Resolver. We are developing a *Data Distribution* service to distribute updates for these variables, allowing them to be declared as local to a module, thus improving performance. It may be that the responsibility for a particular variable will change throughout the simulation. For this functionality we are developing an *Ownership Management* service similar to that of the HLA, which allows variables to be declared locally, but for their owner to change.

There will undoubtedly be new issues that arise as we pursue our goal of building a complex climate change adaptation simulation, using distributed heterogeneous modules developed by many different parties. However, we believe that the system presented here is a solid framework on which to further develop this goal.

7. ACKNOWLEDGMENTS

The authors would like to thank RMIT Global Cities Research Institute and in particular Peter Hayes for the seed funding that made this work possible. Thanks to the Australian Research Council for continuing support of this work under grant DP1093290. We also thank Fabio Zambetta for initial discussions regarding this work.

8. REFERENCES

[1] IEEE 1516 (Standard for Modelling and Simulation High Level Architecture Framework and Rules) , 2000.

[2] R. M. Axelrod. *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*. Princeton Univeristy Press, 1997.

[3] N. Collier. Repast: An extensible framework for agent simulation. Available online: http://www.econ.iastate.edu/tesfatsi/RepastTutorial.Collier.pdf, 2002.

[4] A. Drogoul, D. Vanbergue, and T. Meurisse. Multi-agent based simulation: Where are the agents? *Lecture Notes in Computer Science*, pages 1–15, 2003.

[5] J. Epstein. *Generative Social Science - Studies in Agent-Based Computational Modeling*. Princeton University Press, 2006.

[6] R. M. Fujimoto. Time Management in The High Level Architecture. *Simulation*, 71(6):388–400, 1998.

[7] R. M. Fujimoto. Parallel simulation: parallel and distributed simulation systems. In *33rd Winter Simulation Conference*, pages 147–157, 2001.

[8] G. Gilbert and K. Troitzsch. *Simulation for the social scientist*. Open University Press, 2005.

[9] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, NJ, USA, 1999.

[10] R. Minson and G. Theodoropoulos. Distributing RePast agent-based simulations with HLA. *Concurrency and Computation: Practice and Experience*, 20(10):1225 – 1256, 2008.

[11] M. North, T. Howe, N. Collier, and J. Vos. Repast simphony runtime system. In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, pages 13–15, 2005.

[12] M. Parker. What is Ascape and why should you care. *Journal of Artificial Societies and Social Simulation*, 4(1), 2001.

[13] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *16th International Joint Conference on Artificial Intelligence IJCAI'99*, pages 326–333, 1999.

[14] US Defence Modelling and Simulation Office. HLA Interface Specification, version 1.3, 1998.

[15] L. Wang, S. Turner, and F. Wang. Resolving mutually exclusive interactions in agent based distributed simulations. In *Proceedings of the 36th conference on Winter simulation*, pages 783–791. Winter Simulation Conference, 2004.