DISTRIBUTING REPAST SIMULATIONS USING ACTORS

F. Cicirelli, A. Furfaro, A. Giordano, L. Nigro
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy
E-mail: {f.cicirelli,a.furfaro,a.giordano}@deis.unical.it, l.nigro@unical.it

KEYWORDS
Modelling and simulation, multi-agent systems, RePast, actors, distributed simulation, HLA/RTI, Java.

ABSTRACT

RePast is a well-known agent-based toolkit for modelling and simulation of complex systems. The toolkit is normally used on a single workstation, where modelling, execution and visualization aspects are dealt with. This paper describes an approach aimed to distributing RePast models, with minimal changes, over a networked context so as to address very large and reconfigurable models whose computational needs (in space and time) can be difficult to satisfy on a single machine. Novel in the approach is an exploitation of a lean actor infrastructure implemented in Java. Actors bring to RePast agents migration, location-transparent naming, efficient communications, and a control-centric framework. Actors can be orchestrated by an in-the-large custom control structure which can ensure the necessary message precedence constraints. Preliminary experience is being carried out using HLA/RTI as middleware. However, the realization can also work with other standard transport layers such as Java Socket and Java RMI. The paper introduces the design rationale behind mapping RePast on to actors and discusses a distributed example.

INTRODUCTION

Multi-agent systems (MAS) (Wooldridge, 2002) simulations have proved their usefulness in such diverse application domains as physics, anthropology, biology, sociology, artificial intelligence etc. However, the complexity of some MAS scalable models can be so highly demanding in computational resources to prohibit their execution on a standalone sequential computer. Therefore, the problem exists of porting a MAS on a distributed context so as to take advantage of the processing capabilities furnished by a collection of low-cost networked computers.

RePast (RePast, on-line)(Collier, on-line)(Macal & North, 2006) is a state-of-art collection of tools and libraries useful for modelling and simulation in Java of multi-agent systems. Its effectiveness has been assessed in (Tobias & Hofmann, 2004). RePast borrows much from the design of the Swarm simulation toolkit (Swarm, on-line). A similar toolkit is SIM_AGENT (SIM_AGENT, on-line).

An experience of distributing sequential RePast over the High Level Architecture/Runtime Infrastructure (HLA/RTI) middleware (Kuhl *et al*., 2000) is described in a recent paper (Minson & Theodoropolous, 2008). HLA was chosen because it eases interoperability with existing simulation systems, and promotes model reuse. The implementation directly integrates the RePast mechanisms within the HLA/RTI infrastructure. A distributed RePast simulator is an HLA federation of multiple interacting instances of RePast models. Distributed synchronization depends on conservative synchronization (Fujimoto, 2000) which favours trasparency and backward compatibility with sequential RePast model. The RePast scheduling algorithm was replaced with a new one which constrains local time advancement in a federate, in synchronization with the rest of the federation. Space/environment objects are mapped on the object architecture (Federation Object Model or FOM) and the publish/subscribe pattern supported by HLA. A critical problem concerns concurrent access/update to shared attributes, e.g. of the environment. Conflicts resolution is achieved by divesting attribute ownership to RTI. All of this can have performance penalties in the runtime. A similar realization was previously experimented with distributing SIM_AGENT toolkit on top of HLA (Lees *et al*., 2007).

This paper proposes an original approach to distributing RePast simulations which is based on actors (Agha, 1986) and in particular on the Theatre architecture (Cicirelli *et al*., 2007a-b)(Cicirelli *et al*., 2009). Adopted actors are reactive threadless agents which communicate to one another by asynchronous message-passing. Actors bring to RePast agents migration, location-transparent naming, efficient communications, and a control-centric framework. Actors can be orchestrated by an in-the-large custom control structure which can ensure the necessary message precedence constraints. Such standard transport layers can be used as Java Socket and Java RMI. Actors were ported also on top of the HLA/RTI middleware (Cicirelli *et al*., 2009) which provides, among others, transport layer and time management services. A key difference from (Minson & Theodoropolous, 2008) relates to conflict management on shared objects. In this work the recourse to attribute ownership mechanisms of RTI is

Proceedings 23rd European Conference on Modelling and Simulation ©ECMS Javier Otamendi, Andrzej Bargiela, José Luis Montes, Luis Miguel Doncel Pedrera (Editors)
ISBN: 978-0-9553018-8-9 / ISBN: 978-0-9553018-9-6 (CD)

avoided. Local environment and space objects are accessed in the normal way. Consistency and conflict resolution in the access to shared and remote environment variables are mediated by environment actors and messages, and harnessing features of the simulation control engine.
The achieved mapping of RePast over actors is named ACTOR_REPAST. In this paper ACTOR_REPAST design and prototype implementation is discussed in terms of preliminary experience with HLA. Other distributed scenarios are possible.
The paper is structured as follows. First RePast modelling and simulation concepts are summarized. Then the adopted actor model is presented together with a conservative distributed simulation engine. The paper goes on by discussing ACTOR_REPAST design and prototype implementation, and an adaptation of the actor-based conservative simulation engine to distributed RePast simulations. After that a model example and its execution are demonstrated. Finally, conclusions are drawn with an indication of on-going and future work

REPAST CONCEPTS

The RePast toolkit includes a runtime executive (*scheduler* and *controller* components) which furnishes an event-driven simulation engine, and a user interaction interface through which a simulation experiment can be controlled.
A system (see Fig. 1) typically consists of a collection of *agents*, a collection of *spaces* modelling the physical environment within which the agents are situated (have coordinates) and operate, and a *model* object which contains information (e.g. for configuration) about the entire system. The *state* of system is scattered among model, agent and space objects. An agent-based simulation normally proceeds in two stages. The first one is a setup stage that prepares the simulation for execution. The second stage is the actual running of the simulation. During the setup phase, the model object is created as an instance of a Model class

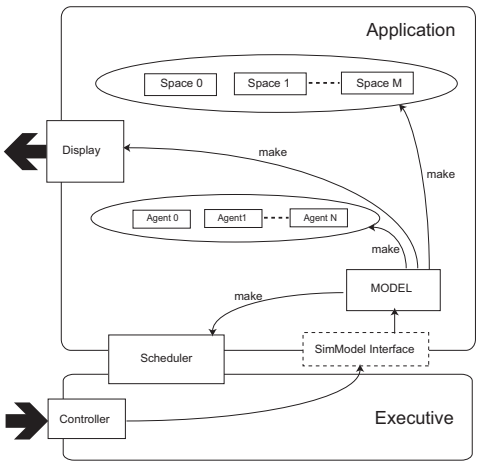

Figure 1: Starup scenario

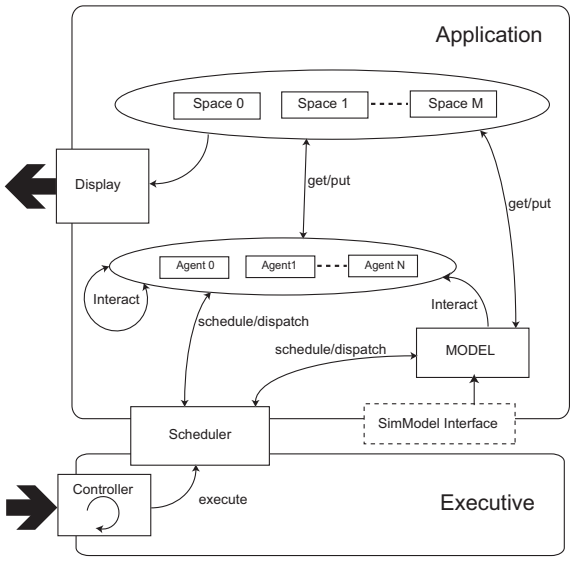

Figure 2: Runtime scenario

(implementing the SimModel interface) which makes instances of agents and spaces, display and scheduler (the latter is an instance of the Schedule class). It is the executive which actually asks the model object to execute the setup phase. After that, when the simulation is started, the executive achieves from the model object the scheduler object used to control the simulation. Fig. 1 summarizes the operations of model bootstrap, whereas Fig. 2 depicts model execution.
RePast events are instances of the BasicAction class. An event occurrence is mirrored by an invocation of the execute() method of a basic action object. Actions are scheduled to occur at certain simulation times (*ticks*). Ticks can be equally spaced or, more in general, not equally-spaced or event-driven. All pending events existing at a given moment are buffered, ranked by ascending timestamps, within the scheduler object. At each iteration of its control loop, the controller asks scheduler to extract the (or one) most imminent pending action and to dispatch it to its destination object, i.e. model or agent. The consequence of an event occurrence is in general a chain of method invocations, which can cause state changes in agents, in model or in space objects.
Space objects can be *data* (or *diffusive*) *spaces* or *object* (*agent*) *spaces*. An application based on diffusive spaces typically has the model which repeatedly executes a cycle made up of three basic phases: *diffuse- modify-update*. During the diffuse phase, the environment is asked to synchronously update itself according to a diffusive logic. Then, in the modify phase agents are allowed to introduce further changes to the data space. For consistency, though, these changes are stored in a temporary copy of the environment. Finally, in the update phase, the temporary copy is restored on the actual environment. An object space, on the other hand, behaves more asynchronously. Agents can issue get/put operations to

spaces, which affect immediately the environment. At each tick, the Model object causes the environment changes to be displayed by invoking the Display *redraw* method.

ACTORS

The following gives an overview of a variant of the Actor model (Agha, 1986) which is used for distributing RePast. The same architecture is being experimented for supporting Parallel DEVS (Zeigler *et al*., 2000)(Cicirelli *et al*., 2008). Actors (Cicirelli *et al*., 2009) are reactive thread-less objects which encapsulate a data state and communicate to one another by asynchronous message passing. Messages are typed objects. Actors are at rest until a message arrives. Message processing is atomic: it cannot be suspended nor preempted. Message processing represents the unit of scheduling and dispatching for a theatre. The dynamic behavior of an actor is modeled as a finite state machine which is realized in the *handler(message)* method which receives the message to process as a parameter. Responding to a message consists in general of the following actions:

- new actors are (possibly) created
- some messages are sent to known actors (*acquaintances*). For proactive behavior, an actor can send to itself one or more messages
- the actor migrates to another execution locus (theatre, see below)
- current state of the actor is changed (become operation).

Actor mechanisms are supported in Java by a minimal API. Actor classes derive directly or indirectly from the Actor base class. Similarly, message classes derive from the Message base class. A subsystem of actors allocated for execution on a computing node of a networked context is treated as a unit-in-the-large and is termed a *theatre*. A theatre furnishes the communication (including migration), scheduling and dispatching message-based services to actors. A fundamental component of a theatre is the *control machine* which interacts with peers in a distributed context through a suitable transport layer.

A migrating actor leaves on the originating theatre a *forwarder* (proxy) version of itself which will route incoming messages to the destination theatre where the real version of the actor exists. To avoid multi-hop communications, the address information of a moving actor is updated on the source theatre so as to mirror current actor location.

CONSERVATIVE CONTROL ENGINE

Control machines of a distributed system are required to coordinate each other according to a specific control structure in order to guarantee messages are ultimately executed in timestamp *and* causal order during a distributed simulation (Fujimoto, 2000). The actor model was implemented in a case on top of IEEE 1516 standard HLA/RTI middleware (Cicirelli *et al*., 2008)(Kuhl *et al*., 2000)(Pitch, on-line) whose in-built time management services can be exploited to achieve, e.g., a conservative distributed simulation engine like the one outlined in the following.

A theatre naturally maps on to an HLA federate. It is assumed that the theatres/federates of an HLA federation are both *time constrained* and *time regulating*. Each theatre asks RTI for a time advancement through Next-Event-Request-Available (NERA) calls (zero lookahead) (Kuhl *et al*., 2000). Simulation (or virtual) time grows according to model time. However, at a given model time, simultaneous or contemporary events are handled by a tie-breaking mechanism which ranks events on the basis of their *generation*. The generation concept is useful to ensure *cause-effect* relationships among messages. Of course, the effect has always to be processed after its cause. For example, contemporary messages created by an actor in the course of responding to an incoming message m, can be managed in the next generation with respect to the generation of m. The logical time presented to HLA/RTI through a NERA is actually a triple <virtual−time, generation, step>. virtual-time is the model time. The (optional) step field acts as a least significant bit of generation. The control structure can use the step field to further discriminate some concurrent messages. For instance, a migration request can purposely be delayed at the *end* of current generation to ensure pending messages destined to the migrating actor are processed in the source theatre *before* migration. All of this can be achieved by scheduling the migration to occur at step=1 of current generation, and waiting from RTI the corresponding release of Time-Advance-Grant.

A specialization of the above control strategy was implemented to support Parallel DEVS models (Cicirelli *et al*., 2008). Another adaptation is discussed later in this paper as part of distributing RePast.

ACTOR_REPAST

A RePast model is partitioned into a collection of Logical Processes (LPs) which are allocated for the execution on different computing nodes of a networked system. The structure of nodes/LPs is summarized in Fig. 3. Each LP hosts a portion (*region*) of the environment and a subset of agents (mapped as actors). Agents can migrate from an LP to another at runtime. The following points out basic problems arising from partitioning and the particular software engineering solutions which were adopted to cope with them. Mapping issues essentially fall in three main areas: state space representation, scheduling system, conflict resolution on shared variables.

Partitioning and Region Boundaries

Partitioning mainly splits environmental spaces into regions assigned to distinct LPs. As suggested by Fig. 3, the RePast model is “instantiated” on *each* participating

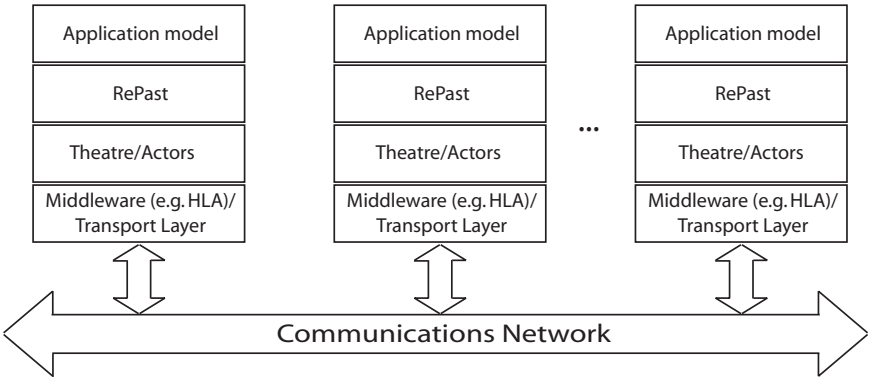

Figure 3: Architecture of a networked RePast model

node. In reality, only the environment portion relevant to each particular node gets effectively instantiated at configuration time. Configuration will assign real or proxy versions of the agents (actors) to the various nodes which comprise the distributed system. Of course, the display facilities into each node are relative to the environment portion managed by the node. As a consequence of instantiating the "entire" model on every node, the Model object gets replicated on each node. However, only the Model object in the *master* node (designated at configuration time) is allowed to start-up the simulation by scheduling the initial actions.

In the following it is assumed that a RePast agent perceives only a part of its environment identifiable as its neighbourhood (or "sphere of influence" (Logan & Theodoropoulos, 2001)). Environmental region boundaries introduce shared variables for the agents belonging to adjacent LPs. A detailed look at region boundaries is exposed in Figures 4 and 5 where the border of two neighbouring LPs is shown. More complex boundary scenarios can occur in practice. In particular, Fig. 5) clarifies that the border region of an LP is in reality composed of two parts: a *local* border of the LP and a *mirror* border of the local border of the adjacent LP. The mirror parts are replicas of boundary space cells. A first problem raised by boundary regions is the requirement of keeping consistent local and mirrored parts. An update to a boundary local cell must be reflected also in the mirror part. A second problem concerns conflict resolution during concurrent accesses to shared space cells by agents executing in the two LPs. In the approach described in this paper, conflicts are resolved by harnessing mechanisms of the distributed simulation engine (see later in this paper).

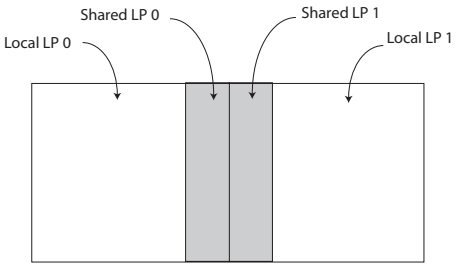

Figure 4: Boundary regions of two adjacent LPs

Mapping Design Rationale

Mapping design was guided by the goal of maintaining a distributed model almost transparent with respect to the sequential model. In the proposed mapping, local action executions behave exactly as in the non distributed version of the RePast model. However, remote action requests, i.e. when the target agent is located on a remote LP, are captured in actor messages and sent across the network. To achieve this effect, agents and model object of a RePast system are turned into actors. Since agents have no special constraints in RePast, a (minimal) requirement was added to develop agent classes and model classes respectively as heir of base classes ACTOR_AGENT and ACTOR_MODEL, which derive from Actor and provide relevant common behaviour. Action objects are transformed into messages and scheduled in the theatre's control machine of the LP. A key point of ACTOR_REPAST is a transfer of control responsibilities from RePast to theatre control machine. Basically, RePast controller remains in charge of interactive events only. Simulation and time management services are instead provided by theatre control machines.

To figure out common behaviour of actor-agents, the handler(incoming_message) method of ACTOR_AGENT is actually responsible of invoking a specific method (whose name is stored in the incoming message) of the corresponding RePast agent, with the help of Java reflection. As another example, when a message is up to be dispatched to a forwarder, i.e. a

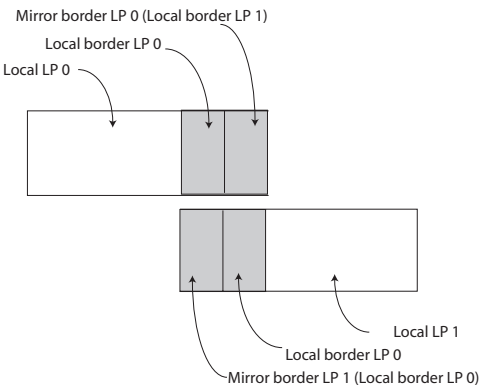

Figure 5: Detailed view of border regions

proxy of a moved actor, the behaviour in ACTOR_AGENT manages to route the message to the corresponding remote actor. Similar behaviour is provided by ACTOR_MODEL. It is worth noting that a model object does not require migration.

A more general task is accomplished by *interceptors*, i.e. entities which intercept method calls and superimpose to them suitable behaviour, with the goal of minimizing "code intrusion" in the original RePast model. Interceptors are realized as *cross-cutting concerns* or *aspects* (Kiczales *et al*., 2001) of AspectJ (AspectJ, on-line).

An aspect controls the execution of the chain of method calls triggered by dispatching an action. When the target agent of a method invocation is local, the aspect lets the agent to execute the method call in the normal way. In the case the receiver is remote, the aspect replaces the standard behaviour of method call by building a message, filling it with information about the requested method, and sending the message over the network. A second aspect is concerned specifically with capturing action scheduling requests in the RePast model and redirecting them to the theatre control machine. A third aspect is used during the start-up phase of a model to inhibit initial action scheduling in non master LPs. Therefore, the Model object really executes only in the master LP.

An important actor in every LP, created at start-up time, is the "environment actor" (EnvActor) which knows about configuration information (taken from a configuration file) of the entire RePast model, and offers a common interface to the model for accessing spaces and environments. For example, the idea that every node/LP is an instantiation of the "entire" model implies that into each sub-model assigned to an LP be present the "global view" to the environment. The transformation of positional coordinates for a situated agent from the global-view to the local-view corresponding to the portion of the environment effectively managed by the LP is accomplished by the EnvActor. A second responsibility of EnvActor is that of propagating to its peer(s) in neighbouring LP(s) the updates of agents in the local border of the belonging LP. Similarly, when an agent moves to the border region, the EnvActor is in charge to migrate the agent, if required, to a neighbour LP. EnvActor is also a key for the diffuse/update/redraw processes. In particular, when the model in the master node raises a diffuse/update/redraw operation, the request is intercepted and a corresponding message is created and sent to all the LPs, which is heard by the EnvActors. An EnvActor then acts so as to actuate the diffuse/update/redraw operation in the local environment.

Distributed Simulation Engine and Conflict Management

An HLA distributed RePast model is governed by an adaptation of the conservative simulation engine previously reported in this paper. The adaptation concerns a particular use of both the *generation* and *step* fields of the time notion presented to RTI, with the goal of controlling contemporary messages occurring at a given simulation time. More precisely, the generation field is used to resolve conflicts on shared environment border objects by constraining adjacent LPs to make requests to conflicting objects at *different* real times. All of this is achieved by assigning to conflicting agents a different occurrence generation. The step field, instead, is devoted, at each generation of current time or to next simulation time, to ensure that update operations to border cells are definitely propagated to adjacent LPs *before* such updates are visible to the model. Toward this, all updates to border cells are captured by an EnvActor and collected into a bag message, one for each distinct adjacent LP. Each bag message is sent across the network to be processed at step=1 of current generation. Therefore, when bags are eventually received and processed by corresponding EnvActors, their effect will only be heard at the next generation or next tick. Also at step=1 are realized agent migrations.

At the beginning of any simulation time, a method of the EnvAct is launched which analyzes the border area and assigns to potentially conflicting agents different generation numbers. These numbers are stored and will be used for scheduling, at current simulation time, action messages or method invocation messages triggered by an action, directed to border agents. It is worth noting that agent population of the border regions of two adjacent LPs is identical. Moreover, the algorithm is run at both sides of the adjacent LPs and makes an identical assignment to the same agents of the border area. In the following, the operation of the algorithm will be sketched in the hypothesis of having agents which can move to a next position with unit distance. Fig. 6) shows an example of border (potentially) conflicting agents. Recall that a border region (see also Figg. 4 and 5) is composed of two sub regions, one corresponding to a local area of the LP, the other to a mirror of the local area of adjacent LP. In addition, in the hypothesis of unit-step movements, the two sub-regions are concretely two columns wide. Fig. 7) depicts a worst case situation of conflicts where 7

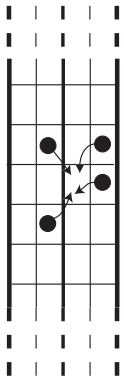

Figure 6: Example of (potentially) conflicting agents

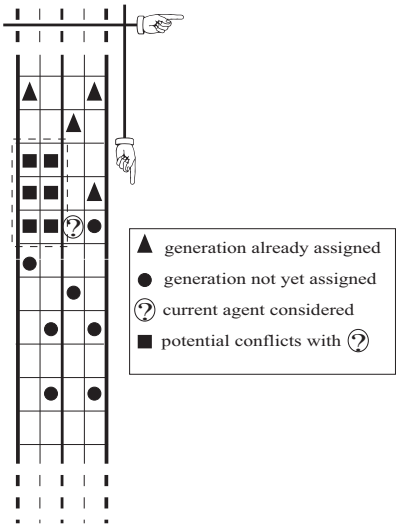

Figure 7: A worst case conflict situation

generation numbers are sufficient for resolving conflicts.

The algorithm proceeds by scanning the complete border area of the LP starting from the upper left corner down to (in row order) the right corner in last row. For each agent (denoted ? in Fig 7)), its relevant neighbourhood is examined. For the agent ? in Fig. 7) the relevant neighbourhood is represented by the dashed rectangle. In fact, the lower part of the neighbourhood is still not assigned, and the upper right part is composed of agents belonging to the same adjacent LP. In the worst case scenario all previously assigned agents in the relevant neighbourhood have received distinct generation numbers. Generations of assigned agents in the relevant neighbourhood are marked as unavailable for the current agent. Then a generation, randomly chosen in the set of available generations, is assigned to current agent. The algorithm naturally reuses generations and this is important both to keep low the number of exploited generations but also to increase the concurrency degree of border agents. Determinacy of the algorithm (at both sides of adjacent LPs) relies on a pseudo-random generator whose seed, at each choice, is defined in terms of current simulation time and the agent environmental position. Moreover, determinacy is also ensured by the scanning order of the border area.

It should be noted that border agents without conflicts are always assigned the first generation (numbered 0) of a simulation time. In addition, both local agents and border local agents without conflicts are handled exactly in the same way.

An Example

ACTOR_REPAST was preliminary tested by using some sequential models available in the software distribution of RePast version 3. The following demonstrates ACTOR_REPAST operation by distributing the Swarm-based HeatBug multi-agent model, which is representative of a class of models

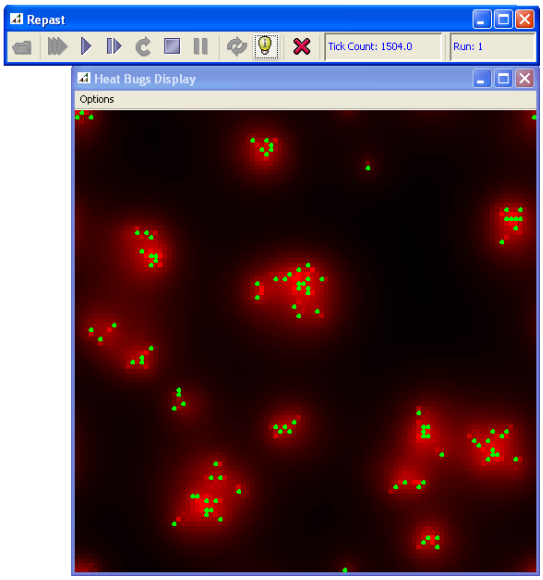

Figure 8: A screenshot from sequential RePast simulation of the HeatBug model

where the environment component admits both diffusive and agent object spaces. The system model logically consists of two parallel grids: the first one is populated by heat bug agents. The other grid is employed for diffusing from cell to cell the heat owned by bugs. The diffusion logic is already available in the implementation of RePast diffusive spaces. Heat bugs are simple agents which absorb and expel heat. Heat bugs have an ideal temperature and will move in an attempt to achieve this temperature.

Fig. 8 shows a screenshot taken from RePast sequential model execution. As one can see, bugs tends definitely to group so as to reach their ideal temperature.

For demonstration purposes, the sequential HeatBug model was partitioned into two LPs and executed, under ACTOR_REPAST, on two theatres/federates on top of HLA pRTI 1516 (Pitch, on-line). The two federate are allocated to two distinct physical processors consisting of WinXP platforms, Pentium IV 3.4GHz, 1GB RAM,

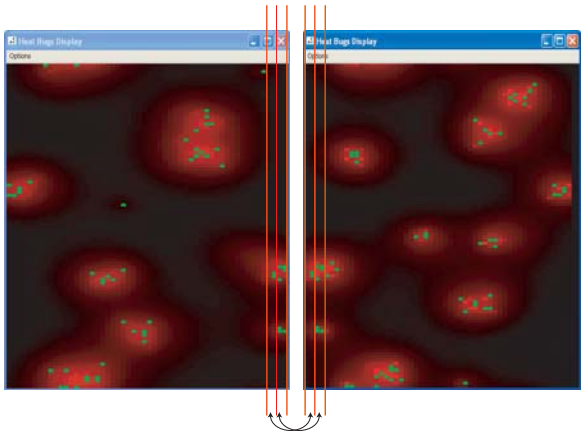

Figure 9: A screenshot of the two display taken from ACTOR_REPAST

interconnected by a 1Gb Ethernet switch. The distributed system naturally exhibits agent migration from an LP to the other. The system was configured (in an XML configuration file, one per LP) with such information as: (a) the list of LPs, each one specified by <IP address, port>, (b) the master LP identification, (c) the topology of the distributed model, with definition of LP adjacency and size of the border regions. Fig. 9 shows the display produced by the two LPs, which confirm the basic behaviour of the bug agents.

Other architectural scenarios are possible specifically for the visualization purposes. For example, as in (Low *et al*., 2007) one single federate (time constrained but not time regulating) could be dedicated only to visualization.

Conclusions

This paper describes an approach and prototype implementation to distributing RePast (Collier, on-line) simulations. The realization, ACTOR_REPAST, owes to a lean and lightweight actor framework in Java which allows customization of the simulation control engine. Preliminary experiments are being carried out using the HLA/RTI 1516 IEEE standard (Kuhl *et al*., 2000)(Pitch, on-line). Other distributed and transport layers, though, are possible.

Prosecution work is geared at:

- improving the transformation from sequential RePast models to corresponding distributed versions through a systematic exploitation of Java *text annotations* which would help tagging source classes and methods and hiding behind annotation management the necessary yet transparent code changes
- optimizing the algorithm which assigns generations to conflicting agents on border regions. More in general, a fundamental open issue concerns the evaluation of different organizations of the environment component of multi-agent systems (Logan & Theodoropoulos, 2001)
- using ACTOR_REPAST in very complex agent-based models with the goal of quantifying specifically the simulation performance towards a comparison with the implementation described in (Minson & Theodoropoulos, 2008).

REFERENCES

Agha G. 1986. *Actors: A model for concurrent computation in distributed systems*. The MIT Press.

AspectJ, http://www.eclipse.org/aspectj/

Cicirelli F., A. Furfaro, and L. Nigro. 2007b. “Exploiting agents for modelling and simulation of coverage control protocols in large sensor networks”. *The Journal of Systems and Software*, **80**(11):1817-1832, Elsevier.

Cicirelli F., A. Furfaro and L. Nigro. 2008. “Actor-based simulation of PDEVS systems over HLA”. In *Proceedings of 41st Annual Simulation Symposium*, Ottawa (Canada), 13-16 April, IEEE Computer Society, pp. 229-236.

Cicirelli F., A. Furfaro and L. Nigro. 2009. “An agent infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination”. *SIMULATION – Trans. of the Society for Modeling and Simulation International*, **85**(1):17-32, SAGE.

Cicirelli F., A. Furfaro, L. Nigro and F. Pupo. 2007a. “A component-based architecture for modelling and simulation of adaptive complex systems”. In *Proceedings of 21st European Conference on Modelling and Simulation (ECMS'07)*, pp. 156-163, 4-6 June, Prague, 2007.

Collier N. 2009. RePast: An extensible framework for agent simulation. http://www.econ.iastate.edu/tesfatsi/RepastTutorial.Collier.pdf

Fujimoto R. 2000. *Parallel and distributed simulation systems*. John Wiley & Sons, New York, NY, USA.

Kiczales G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold. 2001. “Getting started with AspectJ”. *Communications of the ACM*, **44**(10):59–65.

Kuhl F., J. Dahmann and R. Weatherly. 2000. *Creating computer simulation systems: An introduction to the High Level Architecture*. Prentice Hall, Upper Saddle River, NJ, USA.

Lees M., B. Logan and G. Theodoropoulos. 2007. “Distributed simulation of agent-based systems with HLA”. *ACM Trans. on Modeling and Computer Simulation*, **17**(3):1-25, July.

Logan B., and G. Theodoropoulos. 2001. “The distributed simulation ofmulti-agent systems”. *In Proceedings of the IEEE*, **89**(2):174–185.

Low H.M.Y., W. Cai, S. Zhou. 2007. “A federated agent-based crowd simulation architecture”. In *Proceedings of 21st European Conference on Modelling and Simulation (ECMS 2007)*, Prague, June 4-6, pp. 188-194.

Macal C. and M. North. 2006. “Tutorial on agent-based modeling and simulation, Part 2: How to model with agents”. In *Proc. of 2006 Winter Simulation Conference*, Monterey, CA, Dec. 3-6, pp. 73-83.

Minson R. and G. Theodoropoulos. 2008. “Distributing RePast agent-based simulation with HLA”. *Concurrency and Computation: Practice and Experience*, **20**:1225-1256.

Pitch Kunskapsutveckling AB. pRTI 1516. http://www.pitch.se/prti1516/ default.asp

RePast projects, http://repast.sourceforge.net/repast_3/index.html

SimAgent, http://www.cs.bham.ac.uk/research/projects/poplog/packages/simagent.html

Swarm, http://www.swarm.org

Tobias R. and C. Hofmann. 2004. “Evaluation of free Java libraries for social-scientific agent-based simulation”. *J. of Artificial Societies and Social Simulation*, **7**(1). http://ideas.repec.org/a/jas/jasssj/2003-45-2.html

Wooldridge M. 2002. *An introduction to multi-agent systems*. John Wiley & Sons, Ltd.

Zeigler B.P., H. Praehofer and T. Kim. 2000. *Theory of modeling and simulation*. Academic Press., New York, 2nd edition.