STATECHART-BASED ACTORS FOR MODELLING AND DISTRIBUTED SIMULATION OF COMPLEX MULTI-AGENT SYSTEMS

F. Cicirelli, A. Furfaro, A. Giordano, L. Nigro

Laboratorio di Ingegneria del Software
(http://www.lis.deis.unical.it)
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy

KEYWORDS
Multi-agent systems, statecharts, modelling and simulation, distributed simulation.

ABSTRACT

This paper discusses the use of Theatre, a multi-agent simulation architecture, for the distributed simulation of discrete event systems (DESs) whose entities express complex behaviours. Complexity is dealt with by exploiting statechart-based actors which constitute the basic building blocks of a model. Actors are lightweight reactive autonomous agents that communicate to one another by asynchronous message passing. The threadless character of actors saves memory space and fosters efficient execution. The behaviour of actors is specified through "distilled statecharts" that enable hierarchical and modular specifications. Distributed simulation is accomplished by partitioning the system model among a set of logical processes (theatres). Timing management and inter-theatre communications rest on High Level Architecture (HLA) services. The paper highlights the current implementation status and demonstrates the practical application of the approach through a manufacturing system model.

INTRODUCTION

The size and the complexity of systems which are usually modelled as discrete event systems (DESs) (e.g. communication networks, biological systems, weather forecasting, etc.) is ever increasing. Modelling and simulation of such systems is challenging in that it requires suitable specification languages and efficient simulation tools.
Multi-agent architectures (Wooldridge, 2002) have proven to be effective in exploiting parallel and distributed computing environments for the simulation of DESs involving a large number of interacting entities (Cicirelli *et al.*, 2009)(Cicirelli *et al.*, 2007) (Jang *et al.*, 2003). In this context, state-based formalism have been successfully used for specifying agent behaviours. However, as each single agent may express a complex behaviour such languages have to face the well-known state-explosion phenomenon which is typically addressed by resorting to hierarchical and modular constructs.

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)

Statecharts (Harel, 1987)(Booch *et al.*, 2000) are an extension of classical state transition diagrams which have such type of features. The basic mechanism consists in the possibility of nesting a sub automaton within a (macro) state thus encouraging step-wise refinement of complex behaviour. In addition, a macro state can be *and*-decomposed for supporting a notion of concurrent sub automata. Statecharts have been successfully applied to the design of reactive event-driven real-time systems (Harel & Polity, 1998)(Selic & Rumbaugh, 1998)(Furfaro *et al.*, 2006), as well as to modelling and performance analysis, e.g. (Vijaykumar *et al.*, 2002-2006).
This paper discusses the use of a Java framework which supports the execution of agents whose behaviour is modelled by means of statecharts. The approach improves previous authors' work (Cicirelli *et al.*, 2008) by extending it to the context of a distributed-simulation architecture named Theatre (Cicirelli *et al.*, 2009). A particular notion of agents is adopted which rests on actors (Agha, 1986). Actors are lightweight, threadless reactive components which communicate to one another by asynchronous message passing. An actor is characterized by its message interface, a set of hidden data variables and a behaviour for responding to messages (events). A multi-actor subsystem (theatre) is orchestrated by a control machine which provides basic timing, scheduling and dispatching message services to actors. In this paper the behaviour of an actor is specified through a "distilled" statechart, where only the *or*-decomposition of states is admitted. Concurrent sub states are avoided. The approach is demonstrated by applying statechart actors to modelling and simulation of a manufacturing system. Results from simulation experiments are reported. Finally, conclusions are presented together with indications of future work

THEATRE

The architecture of Theatre consists of two main parts:

- *Execution platforms*, i.e. theatres, which provide 'in-the large' features, i.e. the environmental services supporting actor execution, migration and interactions. Services are made available to actors through suitable Application Programming Interface (APIs).

- *Actor components*, i.e. the basic building blocks 'in the-small', which capture the application logic.

While Theatre can be hosted by different object-oriented programming languages, the implementation considered in this work refers to Java.

Actor modelling with Statecharts

Actors are reactive 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 specified by means of a state machine, which, in the context of this work, is a statechart-based hierarchical state machine.

A state of a hierarchical state machine can recursively be decomposed into a set of sub states, in which case it is said to be a *macro* state. A state that is not decomposed is said to be a *leaf* state. The root state of the decomposition tree is the only one having no parent and it is referred to as the *top* state.

Statecharts admit two types of state decomposition: *or*-decomposition and *and* decomposition (Harel and Politi, 1998). In the former case a state is split into a set of sub states which are in an "exclusive-or" relation, i.e. if at a given time the state machine is in a macro state it is also in exactly one of its sub states. In the latter case sub states are related by logical "and", i.e. if the state machine is in a macro state it is also in *all* of its direct sub states, each of which acts as an independent concurrent component. The type of statecharts used for modeling actor behavior are "distilled", in the sense that they permit only the *or*-decomposition and thus the single actor is the unit-of-concurrency. All of this complies with the basic assumptions of the adopted actor computational model where concurrency exists at the actor level but not within actors. In other words, concurrency stems from reacting to messages and not from the use of heavyweight multi-threaded agents which can have space/time constraints in the M&S of large systems. A message reaction represents an atomic action which can modify the actor internal data, generate messages to known actors (*acquaintances*) including itself for pro-activity, create new actors, change the current state of the actor.

At a given point in time, an actor finds itself simultaneously in a set of states that constitutes a path leading from one of the leaf states to the top state. Such a set of states is called a *configuration* (Harel and Naamad, 1996). A configuration is uniquely characterized by the only leaf state which it contains.

Each macro state *S* specifies which of its sub states must be considered its *initial state*. This sub state is indicated by means of a curve originating from a small solid circle and ending on its border. This curve, although technically is not a transition, is referred to as the *default transition* of *S*. State transitions are represented by edges with arrows. Each transition is labelled by *ev[guard]/action* where *ev* is the trigger (event causing the transition), *guard* a logical condition which enables the transition when it evaluates to true, and *action* the action "à la Mealy" associated with the transition.

Both source and destination of a transition can be states at any level of the hierarchy. Whereas a transition always originates from the border of a state, it can reach its destination state either on its border or ending on a particular element called *history connector* or *H*-connector. Such a connector is depicted as a small circle containing an *H* (*shallow history*) or an *H** (*deep history*), and it is always inside the boundary of a compound state.Firing a transition leads the actor to switch from one configuration to another. When a configuration is left, each of its macro states keeps memory of its direct sub state that is also part of the configuration. This sub state is referred to as the *history* of the macro state. The first time a state is entered, its history coincides with its initial state.

Let *S* be the destination state of a transition *tr*. The configuration which is assumed as consequence of firing *tr* depends on the way *tr* reaches *S*:

- If *S* is a leaf state the new configuration is the only one that contains *S*.
- If *S* is a macro state and *tr* ends on its border, the next configuration corresponds to the destination state being the initial state of *S*.
- If *S* is a macro state and *tr* ends on a shallow history connector, the next configuration corresponds to the destination state being the state that is the history of *S*.
- Finally, if *S* is a macro state and *tr* ends on a deep history connector, the configuration depends on the nature of the state *D* which is currently history of *S*. If *D* is a leaf state, the configuration will be the only one that contains *D*, otherwise the configuration corresponds to the case *tr* would end on a deep history connector of *D*.

Each state can have entry/exit actions which are respectively executed each time the state is entered or exited. Moreover, within a state can be present "internal transitions" which denote incoming events which are processed *without* leaving/re-entering the state. Therefore, an internal transition never triggers exit/entry actions into execution. The concept differs from self-loop transitions which imply execution of exit/entry actions of the state.

Theatre architecture

Fig. 1 depicts the architecture of Theatre. Each theatre node has the following components:

- an instance of the Java Virtual Machine (JVM)
- a Control Machine (CM)
- a Transport Layer (TL)
- a Local Actor Table (LAT) and
- a Network Class Loader (NCL).

The Control Machine hosts the runtime executive of the theatre, i.e. it offers basic services of message scheduling/dispatching which regulate local actors. The Control Machine can be made time-sensitive by

managing a time notion ("real" or simulated). Actually, the Control Machine organizes all pending (i.e. scheduled) messages in one or multiple message queues. Instead of having one mail queue per actor (Agha, 1986), the Control Machine buffers all sent messages and superimposes them an application-dependent control strategy.

During the basic control loop, a pending message is selected (e.g. the (or one) most imminent in time) and dispatched to its destination actor to whom is given the possibility of executing a relevant state transition. After the transition code is executed, the control loop is re-entered, all messages sent by the last activated agent are scheduled and, finally, the next cycle is started.

The Transport Layer furnishes the services for sending/receiving network messages and migrating agents. Details about the lightweight migration mechanism are described in (Cicirelli *et al*., 2009).

As an example, Java Sockets or Java Remote Method Invocation (RMI), based on reliable Transmission Control Protocol (TCP) FIFO channels, can be used as a concretization of the Transport Layer. They are part of existing implementations of Theatre.

The Local Actor Table contains references to local agents of the theatre. The Network Class Loader is responsible for getting dynamically and automatically the class of an object (e.g. a migrated agent) by retrieving it from a network Code Server acting as a code repository for the distributed application.

Control machines in a Theatre system have to coordinate each other in order to ensure a global synchronization strategy. Different global synchronization algorithms were implemented on top of Theatre. In (Cicirelli *et al*., 2007) a time warp based simulation control engine (Fujimoto, 2000) is described which uses state-saving and rollback mechanisms to implement an optimistic distributed simulation framework. In (Cicirelli *et al*., 2009) a conservative synchronization structure (Fujimoto, 2000) is proposed which is based on HLA and is capable of handling actor migrations. A specialization of this global control strategy with a tie-breaking mechanism for managing contemporary or simultaneous messages, i.e. occurring a same simulation time, and required to fulfil precedence constraint relationships, is described in (Cicirelli *et al*., 2008) for supporting specifically the execution of Parallel DEVS (Zeigler et al., 2000) systems realized over theatres and actors. In this paper, the conservative simulation algorithm proposed in (Cicirelli *et al*., 2009) is assumed for the distributed simulation experiments based on statechart actors.

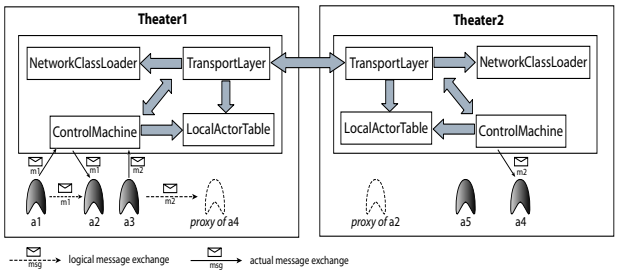

Figure 1: Theatre architecture

A MODELLING EXAMPLE

This section illustrates an application of the approach by considering a model of a manufacturing system. The example is adapted from (Vijaykumar *et al*., 2002-2006) where it has been handled by statecharts with *and*-decomposition and event broadcasting (Harel & Politi, 1998), and analytically studied by preliminarily transforming the model into a continuous time Markov chain (CTMC). In this paper the model is simulated. Obviously, simulation opens to the possibility of using probability distribution functions for event occurrences beyond the exponential one which is normally a prerequisite for building a CTMC. In addition, simulation can be exploited for investigation of more general properties about system behaviour.

In the considered manufacturing systems (Fig. 2), two machines, respectively referred as Machine A and Machine B, operate in series processing the submitted jobs for producing a single product. Jobs are first processed by Machine A and then by Machine B. A bounded capacity Inventory is used for decoupling the operation of the machines thus reducing their wait times. A Robot is actually in charge of loading/unloading the two machines with jobs, possibly using the Inventory for temporary job buffering. Both machines and the robot may be subject to failures. Repairing from a failure is responsibility of an Operator.

Machines, Robot, Operator and the Inventory are modelled as actors. Event broadcasting, which was present in the original model, is replaced by direct communications. In particular, because the robot bases its decisions on the operation state of the two machines and of the inventory, it gets directly notified by these components about relevant state changes. Analogously, when the robot, or one of the machines, goes in a failure state it directly asks the operator for being fixed.

Statechart Models

Figures from 3 to 6 depict the statecharts modelling respectively the behaviours of Machine, Operator, Inventory and Robot actors. The top states of all these statecharts have a default state named New, which is a leaf state, where each actor waits for the arrival of an Init message carrying initialization information.

After being initialized, a machine (see Fig. 3) goes into state W where it waits for a job to be processed.

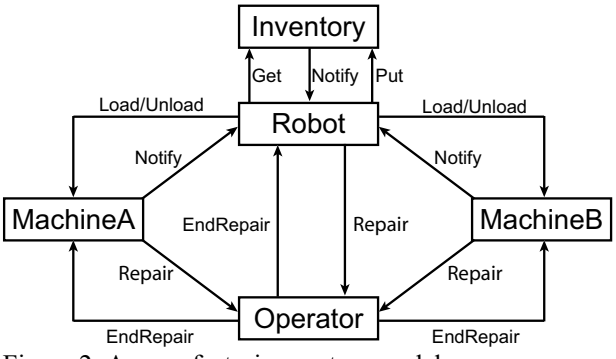

Figure 2: A manufacturing system model

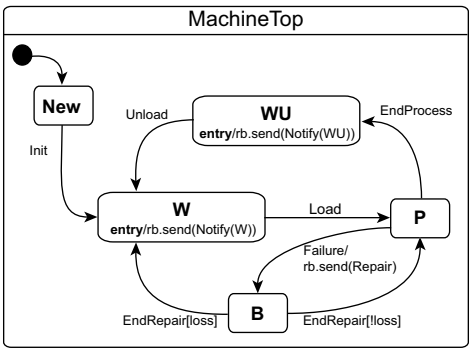

Figure 3: Machine behaviour

From W, it moves into state P when it has been loaded by the robot with a job. While residing in state P the machine processes the loaded job. At processing end, the machine moves into WU waiting for being unloaded. Both states W and WU have an entry action that consists in sending a Notify message to the robot for letting it know that the sending machine needs to be loaded or unloaded.

The message EndProcess is an internal message which a machine sends to itself for simulating the processing time (dwell time in P). During its stay in P, a machine can be subject to a failure in which case it moves to state B. As for processing end, failure is also modelled by another internal message which the machine sends to itself according to the next time to failure defined by a corresponding probability distribution function. When a Failure message is received, a request for being fixed is sent to the operator through a Repair message. After being repaired (arrival of the external message EndRepair coming from the Operator), the machine can return into state P for continuing processing of the interrupted job, or it can go back to W in the case the partial processed job is lost. The two possibilities are controlled by the boolean variable loss whose value is determined accordingly to the specified loss probability.

The behaviour of the Operator is depicted in Fig. 4. After being initialized, it waits in state W for a repairing request. As soon as such a request is received it moves into state R. The Operator resides in R for a dwell time that models the repairing time whose mean duration changes depending on the actor (a machine or the robot) that has made the request. While in R, other repair requests may arrive. They are handled by an internal transition whose action consists in storing them into suitable internal variables, thus postponing their processing. The completion of the repairing process is achieved with an internal message EndProcess to which the actor reacts by sending an EndRepair message to the relevant actor and a Check message to itself. Upon receiving a Check message, the operator inspects its internal variables for checking whether there are pending repair requests. When faced with the decision of which entity to repair first, the operator chooses according to the following priority order: first machine Mb, then machine Ma and lastly the Robot.

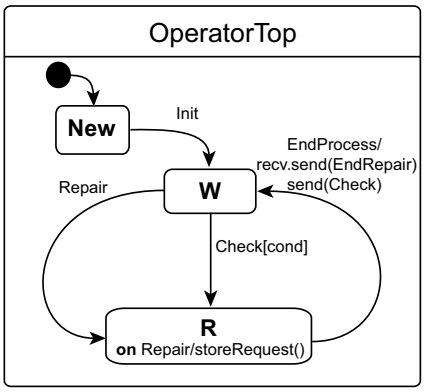

Figure 4: Operator behaviour

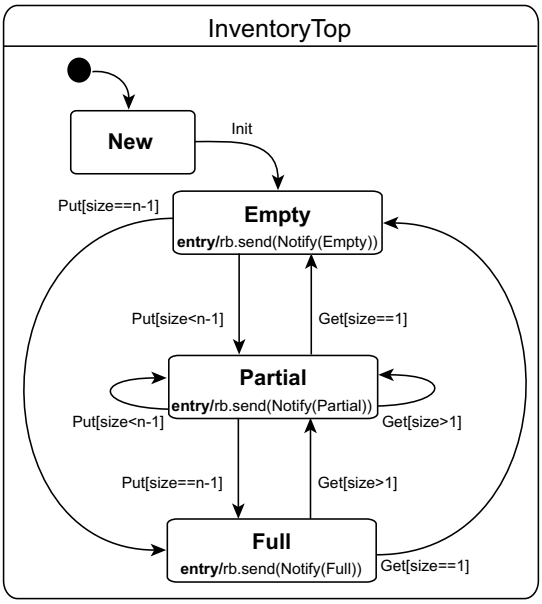

Figure 5: Inventory behaviour

The behaviour of the Inventory is portrayed in Fig. 5. It is a bounded buffer of capacity n, which can be 0 or a positive value. After initialization, the Inventory can be in one of the states among Empty, Partial or Full. Depending on the current available space, a Get/Put message can switch the inventory between Empty, Partial or Full as shown in Fig. 5. All of these three states have an entry action which consists in notifying the robot about the current inventory state.

The robot is the most complex entity as can be seen from the statechart of Fig. 6 which models its behaviour. While in state W, the robot waits for an operation to be exercised on the machines. Whenever the robot receives a Notify message it always updates its internal variables according to the received information. This is mirrored by the internal transitions of states W, P and B. In particular, if the robot gets notified when it is in state W, it proactively sends to itself a Check message. On the basis of the information about the state of the other components, if it is able to do some operation when the Check message is received, it switches to macro state P and sends to itself a Start message.

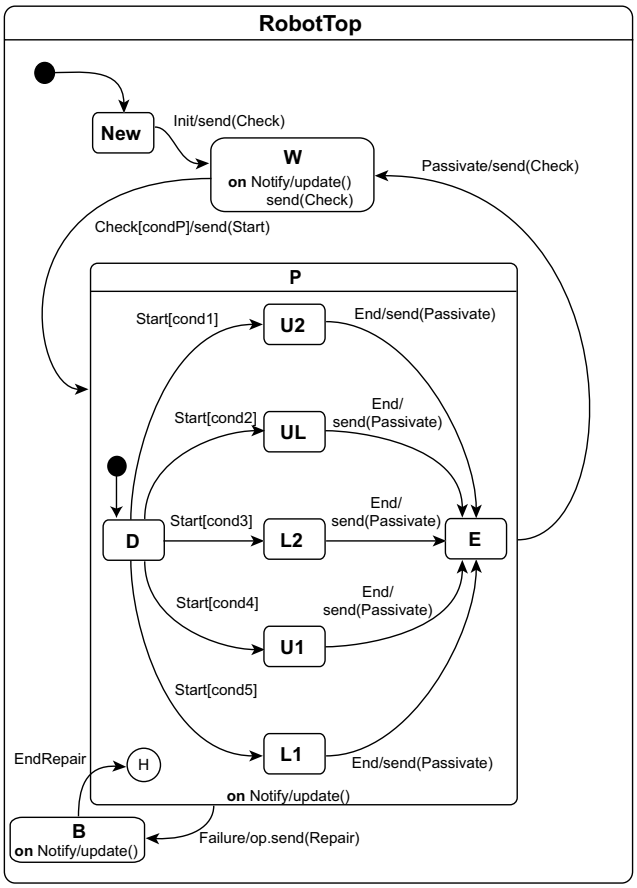

Figure 6: Robot behaviour

This condition is reflected by the value of the boolean variable condP which is the logical or of various conditions as summarized in Table 1.
State D is the default sub state of P which is left as soon as the Start message is delivered. At least one of the guards of the transitions leaving D is satisfied because each of them implies condP (see Table 1). The robot gives priority to unloading machine Mb (cond1 and sub state U2), then to simultaneously unloading machine Ma and loading machine Mb (cond2 and sub state UL), then to loading machine Mb (cond3 and sub state L2), then to unloading machine Ma (cond4 and sub state U1) and, finally, to loading machine Ma (cond5 and sub state L1).

The time spent by the robot in any operating state depends on the particular operation (see Table 2) that it is accomplishing. The internal message End is self-sent for witnessing the end of the operation, in which case the robot moves first into the E state and then sends itself a Passivate message whose arrival takes the robot from state P to state W where the behaviour repeats again.

The transition having Failure as the trigger message is an example of a group transition. It means that whatever is the internal sub state of P, the arrival of the Failure message causes the state P to be exited and state B to be entered, where the Robot requires to be repaired by the Operator.

Table 1: Environmental conditions for the robot

cond1 = Mb is in WU;
cond2 = Ma is in WU && Mb is in W;
cond3 = Mb is W && ¬(Inv is empty);
cond4 = Ma is in WU && ¬(Inv is Full);
cond5 = Ma is in W;
condP = cond1 || cond2 || cond3 || cond4 || cond5;

Table 2: Simulation parameters

	Machine A	Machine B	Robot
Production rate	β_A=8,10 or 12	β_B=10	
Failure rate	λ_A=1	λ_B=0.5	λ_R=1
Repairing rate	μ_A=10	μ_B=15	μ_R=10
Loss probability	p_A=0.5	p_B=0.3	
Loading rate machine A			γ_{L1}=100
Unloading rate machine A			δ_{U1}=100
Loading rate machine B			γ_{L2}=100
Unloading rate machine B			δ_{U2}=100
Moving rate from m. A to m. B			α_{UL}=70

While the Robot is in an operating state, it can fail (internal message Failure received). In this case the on-going operation is interrupted and the operator is asked for intervention. Which event arrives first between End and Failure depends on the next time respectively for completing the operation and for failing.

The transition triggered by EndRepair causes the Robot to return into macro state P with history (see the shallow connector history H). This way, the actor returns exactly into the internal sub state of P which was current when P was last left off at the time of Failure.

For the purpose of experimentation, all the timed events in the system are assumed to be exponentially distributed. Table 2 summarizes the rates (number of events per time unit) used for simulation (as in (Vijaykumar et al., 2002-2006)). Loading/unloading rate of machines are shown as dwell times in the corresponding operation state of the robot.

Simulation results

The manufacturing model was simulated using the parameter values in Table 2, with the aim of validating the distributed runtime infrastructure of statechart-based actors. Simulation uses dense time. Each experiment lasts after a time limit of $tEnd=5\times10^5$. The model was split in two Logical Processes (LPs)/federates assigned to two distinct processors (Pentium IV 3.4GHz, 1GB RAM) interconnected by a 1Gb Ethernet switch in the presence of HLA (pRTI 1516). One LP was assigned the machines and the Operator, the other LP the Robot and the Inventory. Into each LP is also present a Monitor actor for

collecting useful information about the simulation. Each monitor has methods for capturing such data about system productivity, utilization of machines, robot and operator, losses in machines, average inventory size etc. In addition, every monitor sends to itself, at the beginning of the simulation, a timed message to be received at tEnd. Following the arrival of such a message, the actor displays collected statistical information.

The system model was studied in three cases: when machine A has respectively a lower/equal/greater production rate than B (see Table 2). System properties were analyzed vs. the inventory bounded capacity which was varied from 0 to 20. Experimental results comply with those reported in (Vijaykumar *et al*., 2002-2006), but furnish more detailed information about system behaviour.

Fig. 7 portrays measured system productivity (number of unloads from machine B per time unit) vs. the inventory capacity. As one can see, starting from 0, an increase in the inventory capacity increases the system productivity until the system reaches full-busy condition. In this condition the system exhibits maximum parallelism among components, with the inventory which smooths out instantaneous differences in the production speed of the two machines. The smoothing effect is obviously greater when the production rate of machine A grows.

The positive effect of using a not zero inventory size can be checked in Fig. 8 which shows the waiting time for unloading machine A vs. the inventory capacity. This statistic was achieved by summing up the dwell time of machine A in state WU, waiting for the robot to unload the finished product, and then dividing the sum for the simulation time limit.

For completeness, Fig. 9 illustrates the wait time for unloading machine B, which has priority with respect to unloading machine A. As expected, machine B has a small wait time.

CONCLUSIONS

This paper describes and demonstrates, through an example, an approach to modelling and distributed simulation of multi-agent systems. Novel in the proposed approach is an exploitation of a lightweight actor computational model which permits the behaviour of each agent to be specified through a "distilled" statechart (Harel, 1987). All of this favours the expression of complex behaviour at the agent level. Complexity in the large is dealt with by the Theatre architecture (Cicirelli *et al*., 2009) which allows decomposition of a large system into sub-systems (theatres) each hosting a collection of application actors, allocated for execution on a physical processor. Actors are capable of migrating from a theatre to another, e.g. to cope with demands of dynamically reconfigurable systems (Cicirelli *et al*., 2009) or to respond to load-balancing requirements.

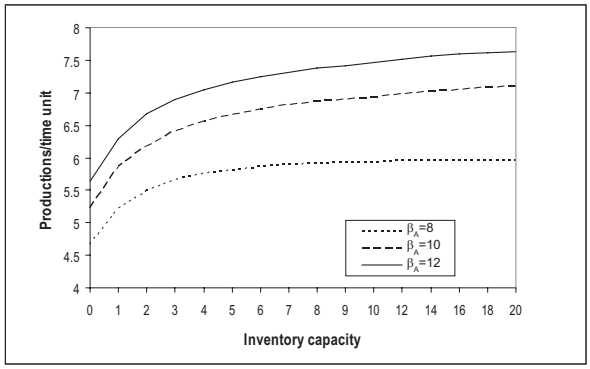

Figure 7: Observed system productivity vs. inventory capacity.

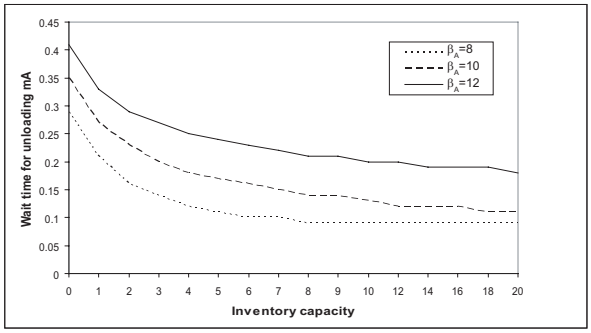

Figure 8: Average wait time for unloading machine A

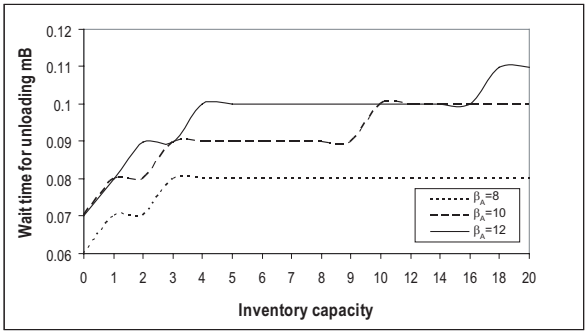

Figure 9: Average wait time for unloading machine B

Theatre and statechart actors, in a case, were implemented on top of HLA services (Kuhl *et al*., 2000)(Pitch, on-line) in the presence of conservative synchronization.

On-going and future work are geared to:

- modelling and simulation of large, scalable and reconfigurable systems in order to study the execution performance issues;
- extending the approach with optimistic synchronization, e.g. by implementing a Time Warp based distributed simulation algorithm (Fujimoto, 2000)(Cicirelli *et al*., 2007b);
- improving M&S capabilities by experimenting with environment-based multi-agent systems (Logan & Theodoropoulos, 2001).

The last point deserves some further comments. The example presented in this paper does not use the

environment concept at all. The interactions among actors simulate event-broadcasting (Harel & Politi, 1998) so as to reproduce cause-effect relationships. A different solution would have been introducing the *environment* component which is informed of significant changes in actors and propagates this information to interested actors. The environment can be realized according to different organizations.

Taking for example the viewpoint of HLA, a distributed system partitioned into N application theatres, each housing a given number of actors, could be really constructed along one of the following schemes:

- using N+1 theatres/federates, where the extra federate is dedicated to the centralized and shared environment component;
- partitioning the environment among the N federates so that each federate hosts the local portion of the environment accessed by local agents.

The first solution could be implemented into HLA by using the publish/subscribe architecture and by resorting to RTI ownership management for ensuring mutual exclusion on shared environment attributes. The solution, though, could suffer from computational degradation resulting both from centralization and ownership management.

The second solution (see also (Cicirelli *et al*., 2009)) can be preferable but introduces problems at the boundary of adjacent theatres, also considering that the solution must face agent migrations.

Environment organization is an open and important issue in general multi-agent systems. It will be investigated also in the light of the interesting work of "spheres of influence" described in (Logan & Theodoropoulos, 2001).

REFERENCES

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

Booch, G., J. Rumbaugh and I. Jacobson. 2000. *The Unified Modeling Language User Guide*. Reading, MA, Addison-Wesley.

Cicirelli, F., A. Furfaro and L. Nigro. 2008. "Modelling and Simulation Using Statechart-Based Actors". In *Proceeedings of International Workshop on Modeling & Applied Simulation (MAS'08)*, Campora San Giovanni, Italy, September 17-19, pp. 301-307.

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 - Transactions of the Society for Modeling and Simulation International*, 85(1):17-32, SAGE.

Cicirelli, F., A. Furfaro and L. Nigro. 2007a. "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, L. Nigro. 2007b. "Distributed simulation of modular time Petri nets: an approach and a case study exploiting temporal uncertainty". *Real-Time Systems*, 35(2):153-179, Springer.

Cicirelli, F., A. Furfaro, L. Nigro. 2008. "Actor-based Simulation of PDEVS Systems over HLA". In *Proceeedings of 41st Annual Simulation Symposium (ANSS'08)*, Ottawa, Canada, April 14 - 16, pp. 229-236.

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

Furfaro, A., L. Nigro and F. Pupo. 2006. "Modular design of real-time systems using hierarchical communicating real-time state machines". *Real-Time Systems*, 32(1-2), 105-123, Springer.

Harel D. 1987. "Statecharts: A visual formalism for complex systems". *Science of Computer Programming*, 8:231–274, July.

Harel, D. and M. Politi. 1998. *Modeling reactive systems with statecharts*. Mc Graw-Hill.

Harel, D. and Naamad, A. 1996. "The STATEMATE semantics of Statecharts". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4): 293–333.

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.

Jang, M.-W., S. Reddy, P. Tosic, L. Chen, and G. Agha. 2003. "An actor-based simulation for studying UAV coordination". In *Proceeedings of the 15th European Simulation Symposium (ESS 2003)*, pp. 593–601. Delft, The Netherlands.

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

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

Selic, B. and J. Rumbaugh. 1998. "Using UML for modeling complex real-time systems".

Vijaykumar, N.L., S.V. de Carvalho and V. Abdurahiman. 2002. "On proposing statecharts to specify performance models". *International Transactions in Operations Research*, 9(3):321-336.

Vijaykumar, N.L., S.V. de Carvalho, V.M.B. Andrade and V. Abdurahiman. 2006. "Introducing probabilities in statecharts to specify reactive systems for performance analysis". *Computer and Operations Research*, 33(8):2369-2386.

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

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