MODELLING AND SIMULATION USING STATECHART-BASED ACTORS

Franco Cicirelli[(a)], **Angelo Furfaro**[(b)], **Libero Nigro**[(c)]

[(a)(b)(c)]Laboratorio di Ingegneria del Software
(www.lis.deis.unical.it)
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy

[(a)]f.cicirelli@deis.unical.it, [(b)]a.furfaro@deis.unical.it, [(c)]l.nigro@unical.it

ABSTRACT
This paper describes an agent infrastructure centred on statechart-based actors for modelling and simulation of complex systems. Actors are lightweight reactive autonomous agents which communicate to one another by asynchronous message passing. Actor dynamism is specified through a "distilled statechart" which simplifies the expression of complex behaviour. The threadless character of actors conserves memory space and ensures efficient execution. The paper highlights current implementation status of statechart actors and demonstrates their practical use and programming style in Java through a manufacturing system model. Simulation experiments are reported. Finally, on-going and future work are summarized in the conclusions.

Keywords: Multi agent systems, statecharts, modelling and simulation, Java.

1. INTRODUCTION
Statecharts (Harel and Politi, 1998)(Booch *et al*., 2000) are a well-known extension to classical state transition diagrams with constructs to control state explosion during development of complex systems. 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 and Polity, 1998; Selic and Rumbaugh, 1998; Fortino *et al*., 2001; Furfaro *et al*., 2006), as well as to modelling and performance analysis, e.g. (Vijaykumar *et al*., 2002-2006).

In the work described in this paper an original framework in Java is developed which enables modelling and simulation of agent-based systems (Wooldridge, 2002; Jang *et al*., 2003) using statecharts. The proposal is an evolution of previous work of authors described in (Fortino *et al*., 2001; Furfaro *et al*., 2006) specifically to support M&S activities. The agent infrastructure is part of a more general distributed architecture named Theatre (Cicirelli *et al*., 2007a). A particular notion of agents is adopted which rests on actors (Agha, 1986; Cicirelli *et al*., 2007b). 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. 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 are difficult to exploit, for space/time constraints, for 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.

In this paper statechart-based actors are demonstrated by applying them to modelling and simulation of a manufacturing system. Results from simulation experiments are reported. Finally, conclusions are presented together with indications of future work.

2. A MODELLING EXAMPLE
In the following a system model is described concerning a manufacturing system. The example is adapted from (Vijaykumar *et al*., 2002-2006) where it was handled by statecharts with *and*-decomposition and event broadcasting (Harel and 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 more general exploration of system behaviour properties.

In the system (see Fig. 1), jobs are first processed by Machine A and then by Machine B. Job movements are mediated by a bounded capacity Inventory. A Robot is actually in charge of loading/unloading machines with jobs, possibly using the Inventory for temporary job buffering. Machines and the robot are subject to failure. Repairing from a failure is responsibility of an Operator.

Machines, Robot, Operator and the Inventory are modelled as actors. Event broadcasting is simulated in Fig. 1 by having a shared environment (passive) object Env which is updated by robot and machines and inspected by the robot and the operator to decide next action.

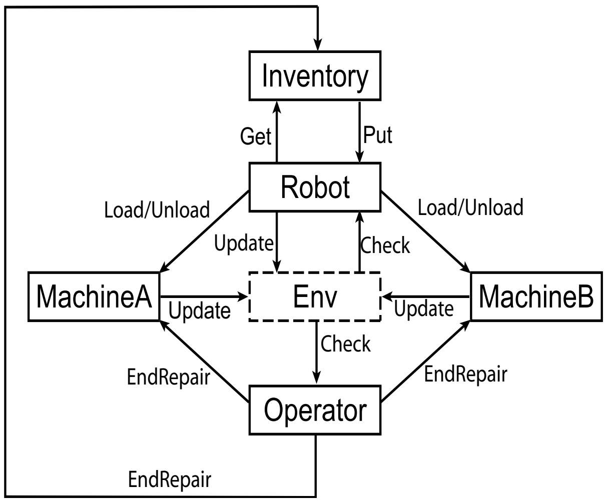

Figure 1: A manufacturing system model

Figg. from 2 to 5 depict respectively the behaviour of Machine, Operator, Inventory and Robot actors. The default state of each statechart is represented by a New leaf state from which the actors move into operation when receiving an Init message carrying the initialization parameters. Or-decomposition of states means that an actor occupying a macro state can find itself in one sub state only at each time. However, due to state hierarchy, for example, when the Robot is in the U2 sub state, it effectively is simultaneously in the U2, P, and Robot Top states (such a state chain is said a configuration). For Robot, Robot Top and P are examples of macro states. Remaining states are leaf states (they do not admit further decomposition). 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 (not shown in the figures for the sake of simplicity). In Fig. 5 the transition with trigger Failure 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. The transition with trigger EndRepair transmitted by the Operator, 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 it was last left off at the time of Failure. Another form of history connector (not used in the example) is H* (deep history) which would case recursive state re-assumption within P and its current sub macro state down to a leaf state. In this case H and H* would be equivalent because all the internal sub states of P are leaf states.

The transition labelled by Tr[condP] and connecting state W of Robot to the boundary of state P asks for the default state of P (leaf state D) to be entered.

After being initialized, a machine (see Fig. 2) goes into the W state where it waits for a job to be processed. From W it moves into P state when loaded by the robot with a job. In the P state the loaded job is processed. At processing end, the machine moves into WU waiting for unloading. The message EndProcess is an internal message which the machine sends to itself for simulating the processing time (dwell time in P). During its stay in P the machine can have a failure in which case it passes to B state. Failure is another internal message which the machine sends to itself according to the next time to failure defined by a corresponding probability distribution function.

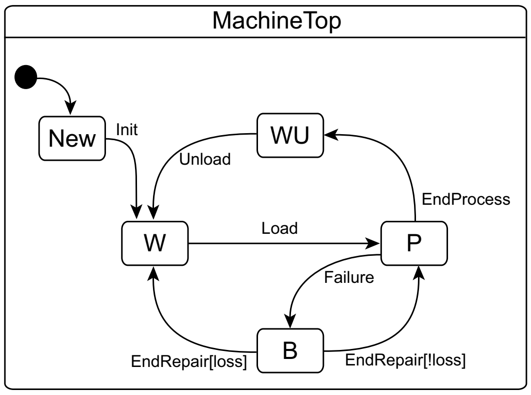

Figure 2: Machine behaviour

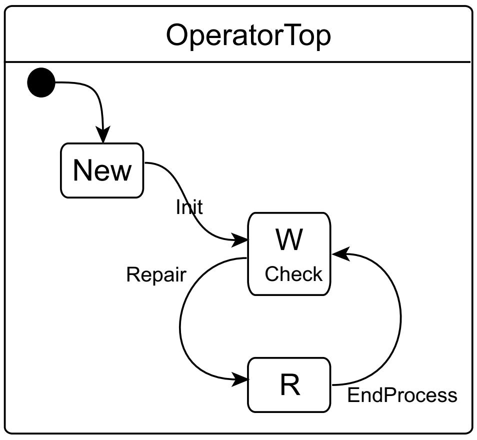

Figure 3: Operator behaviour

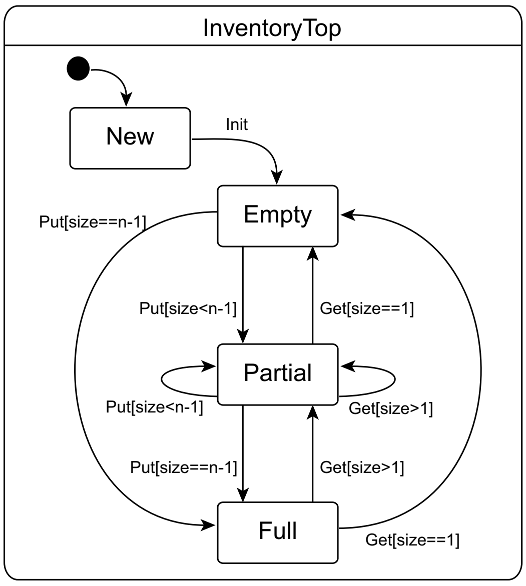

Figure 4: Inventory behaviour

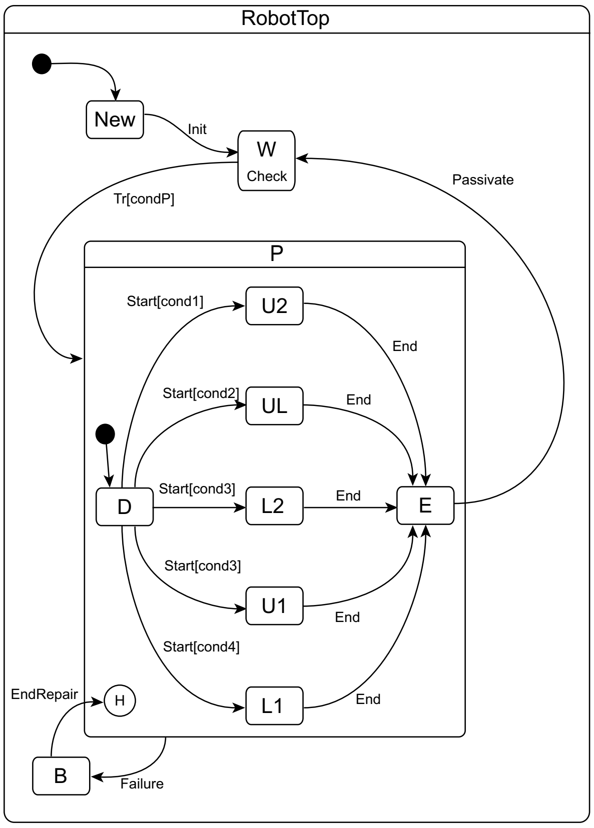

Figure 5: Robot behaviour

While staying in B the machine is repaired by the Operator. After repairing (arrival of the external originated message EndRepair from the Operator), the machine can return into P for continuing processing of interrupted job, or it can go back to W if the job is lost. The two possibilities are controlled by the loss guard which obeys to loss probability.

The Operator (Fig. 3) waits for a repairing request in the W state. The Check message is an internal event to W. It is received from the environment and alerts the operator to consult the environment for checking if a machine or the robot are in the B state requiring repair. If a repair request exists, the operator moves to the R state where the dwell time models the repairing time, which obviously is different from Machine A, Machine B and the Robot. The dwell time is tied to the internal message EndRepair whose action sends to a machine or robot a similar EndRepair message.

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, then the Robot.

Fig. 4 portrays the behaviour of the Inventory. It is a bounded buffer of capacity n, which can be 0 or a positive value. At each moment the initialized Inventory can be in one state among Empty, Partial or Full. Depending on current inventory size a Get/Put message can switch the inventory between Empty, Partial or Full as shown in Fig. 4.

When in the state W the robot (Fig. 5) waits for an operation to be exercised on the machines. Table 1 shows some logical conditions which influence the order in which the robot decides its next move.

Table 1: Environmental conditions affecting robot operation

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;

As one case see from Fig. 5 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). Robot effectively passes from state W to state P (and then to default state D) when condP which is the logical or of the various conditions is true. From D an internal immediate message Start is sent to itself for moving conditionally to a particular processing state among U2 to L1. The time spent by the robot in any operating state depends on the particular operation (see Table 2). Internal message End is self-sent for witnessing the end of operation, in which case the robot moves first into the E state and sends itself a Passivate message whose arrival takes the robot from state E to state W where the behaviour repeats again.

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

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.

Table 2: Simulation parameter values

	Machine A	Machine B	Robot
Production rate	β=8,10 or 12	β=10	
Failure rate	λ=1	λ=0.5	λ=1
Repairing rate	μ=10	μ=15	μ=10
Loss probability	p=0.5	p=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

3. OUTLINE OF JAVA FRAMEWORK FOR STATECHART-BASED ACTORS

The following recapitulates the most important Java classes which were developed for supporting hierarchical actors.

Time, *AbsoluteTime*, *RelativeTime*. Are interfaces specifying a time notion. An absolute time is an instant in time when something can happen. A relative time is a duration, e.g. an amount of time measured from now. The interfaces have methods for adding/subtracting times as meaningful.

AbsoluteDiscreteTime, *AbsoluteDenseTime*, *RelativeDiscreteTime*, *RelativeDenseTime*. Are concrete classes implementing basic time interfaces. A discrete time is a long. A dense time is a double. The classes have a value() method which returns discrete/dense value of a time instance.

Clock. A basic interface for clocks. A clock has a method for checking current time, and methods for advancing the clock according to an absolute or relative time.

SimulationDiscreteTimeClock, *SimulationDense-TmeClock*. Are concrete classes implementing a clock for simulation, based respectively on discrete time or dense time.

Actor. Is the base abstract class for actors. An application actor derives directly or indirectly from Actor. Methods of Actor include send(Message) for sending a message to an acquaintance, handler(Message) which triggers an actor into operation for processing an arrived message (making a state transition), now() which returns an absolute time indicating the current time. The ultimate meaning of now() depends on a control machine.

Message. Is the base class for messages. A message carries the receiver information.

Timer. Is an heir of Message. A timer is a triple: <Message timeout, Actor receiver, RelativeTime firetime>. At fire time the timeout message is consigned to its receiver actor. A created timer can be set and reset. Moreover its remaining time to firing and the elapsed time from its set time can be checked. The firetime is expected as a relative time which added to current time establishes the absolute fire time.

ControlMachine. Is the base class for simulation control engines. Its methods allow to schedule/unscheduled a message. A fundamental method is controller() which starts the control-loop of the engine.

Simulation. Is a concrete class deriving from ControlMachine. Its constructor receives the simulation time limit (an absolute time) and a clock to be used for managing the simulation time. The application passes a SimulationDenseTimeClock when it wants a dense time model to be used. Otherwise it has to pass a SimulationDiscreteTimeClock. Simulation uses a PriorityQueue for timers and a LinkedList for immediate concurrent messages, which are to be processed at current time.

State. Is the base abstract class for states. A state can be entered in a DEFAULT, THROUGH, HISTORY and DEEP_HISTORY mode. The THROUGH mode occurs when a transition reaches its destination state by crossing a state hierarchy. A state has a parent when it is nested into a macro state. To each state are associated the two basic methods entryAction() and exitAction() which have a default void implementation. They are executed respectively at each enter and exit from the state.

MacroState, *LeafState*. Are concrete classes extending State, respectively modelling a macro (o super) state which has inner nested states, and a LeafState which is a leaf in the state hierarchy tree.

Transition. Is a base abstract class modelling the concept of a transition in a statechart. A transition object carries its source state and the trigger message. Methods void action(Message) and boolean guard(Message) can be redefined in concrete transition objects for programming respectively the action and guard components of the transition.

InternalTransition. Is a concrete class extending Transition. It models internal events to a state, which do not cause entryAction()/exitAction() to be executed. Message Check in Figg. 3 and 5 is modelled by an internal transition.

Become. Is a concrete class extending Transition in the more general case of transferring from a source state to a destination state according to given enter mode. Following a become requires the exit-path (configuration) of source state and the enter-path (configuration) of destination state to be determined. Such paths extend from source or destination up to and excluding their common ancestor state. Doing a become will imply exiting the states of the exit-path and entering the states of the enter-path.

DefaultPolicy. Is a class realizing the *Policy* interface. The class provides the default strategy for choosing among candidate transitions outgoing from a state (also considering hierarchy), corresponding to a given trigger. Selection always gives preference to a transition exiting from an inner state with respect to a transition exiting from an enclosing state and for a given state the choice is non-deterministic.

RandomGenerator. Is a concrete class providing common methods for random variate generators, e.g. uniform, exponential, normal etc.

4. PROGRAMMING STYLE

To figure out the resultant Java programming style, Listing 1 shows an excerpt from the Robot actor. The constructor of the Robot class is in charge of creating the state hierarchy of the corresponding statechart. Both states and transition objects must be defined. tR and tF are two timers for timing robot operation and next failure occurrence. eval() is an helper method checking environmental conditions as in Table 1.

Operating states U2, UL etc. redefine the entryAction() so as to set both tF and tR timers. This is important because the entry action gets executed every time the state is entered, e.g. by history.

As one can see the programming style is essentially declarative. In addition there is no need to redefine the handler() method coming from the Actor base class. All of this simplifies programming and makes it possible to automating translation from visual design to Java code.

5. SIMULATION EXPERIMENTS

The manufacturing model was simulated using the parameter values in Table 2, with the aim of validating the runtime infrastructure of the achieved implementation of statechart-based actors. Simulation uses dense time. Each experiment lasts after a time limit of tEnd=5×10^5. The environment object in Fig. 1 was extended in order to collect statistical information about system productivity, utilization of machines, robot and operator, losses in machines, average inventory size etc. 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 and then unbounded. Experimental results comply with those reported in (Vijaykumar *et al.*, 2002-2006), but furnish more detailed information about system behaviour.

Fig. 6 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.

```
...
public Robot(){ //constructor
      super( new DefaultPolicy() );
      //make state hierarchy
      MacroState Top=new MacroState(null);
      State New=new LeafState(Top);
      State W=new LeafState(Top);
      State B=new LeafState(Top);
      MacroState P=new MacroState(Top);
      Top.setDefaultChild(New);
      State D=new LeafState(P);
      State E=new LeafState(P);
      State U2=new LeafState(P){
        public void entryAction(){
              tR.set( new End(), Robot.this,
                     new RelativeDenseTime(rg.exponential(deltaU2)) );
              tF.set( new Failure(), Robot.this,
                     new RelativeDenseTime(rg.exponential(lambdar)) );
        }
      };
      State UL=new LeafState(P){
        public void entryAction(){
              tR.set( new End(), Robot.this,
                     new RelativeDenseTime(rg.exponential(alfaUL)) );
              tF.set( new Failure(), Robot.this,
                     new RelativeDenseTime(rg.exponential(lambdar)) );
        }
      };
      ...//similar definitions for L2, U1 and L1 sub states
      //make state transitions
      new Become( New, W, State.Mode.DEFAULT, Init.class ){
        public void action( Message m ){ ...  }
      };
      new InternalTransition( W, Check.class ){
        public void action( Message m ){ send( new Tr() ); }
      };
      new Become( W, P, State.Mode.DEFAULT, Tr.class ){
        public void action( Message m ){ send( new Start() ); }
        public boolean guard( Message m ){ eval(); return condP; }
      };
      new Become( B, P, State.Mode.HISTORY, EndRepair.class ){
        public void action( Message m ){ env.setRbInB(false); }
      };
      ...//other transitions
      init( Top ); //initializes actor top state
}//end constructor
```

Listing 1: A fragment from the Robot constructor

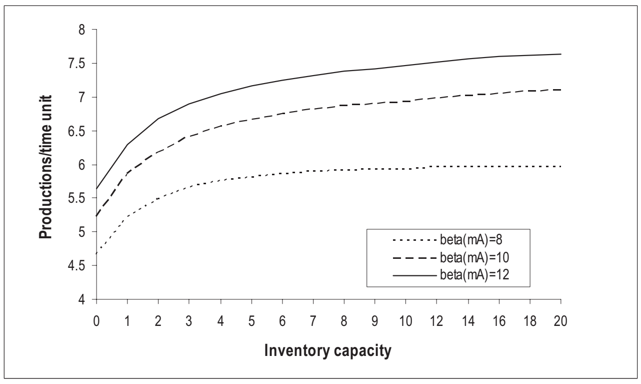

Figure 6: Observed system productivity vs. inventory capacity.

The positive effect of using a not zero inventory size can be checked in Fig. 7 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.

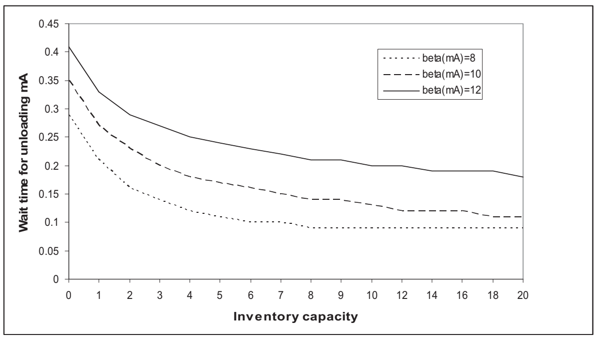

Figure 7: Average wait time for unloading machine A

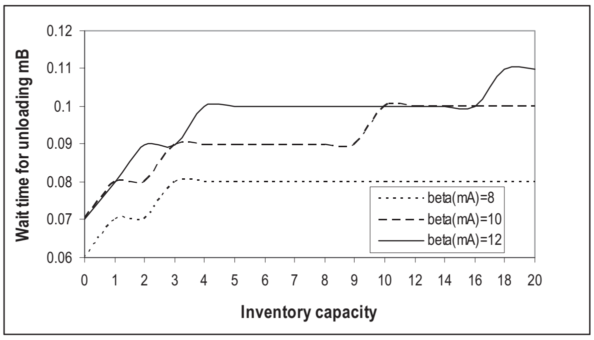

Figure 8: Average wait time for unloading machine B

For completeness, Fig. 8 illustrates the wait time for unloading machine B, which has priority with respect to unloading machine A. As expected, machine B has a lower wait time. The two wait times become similar in full-busy operation of the system.

System behaviour can also be studied by watching the utilization factor (cumulative service time divided by the simulation time limit) of the various components (see Figg. 9 to 12).

Machine B utilization grows as the production rate of machine A augments and the inventory capacity is increased so as to buffer products delivered by machine A.

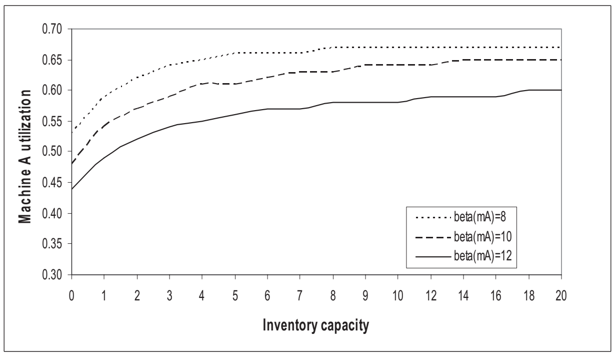

Figure 9: Machine A utilization vs. inventory capacity

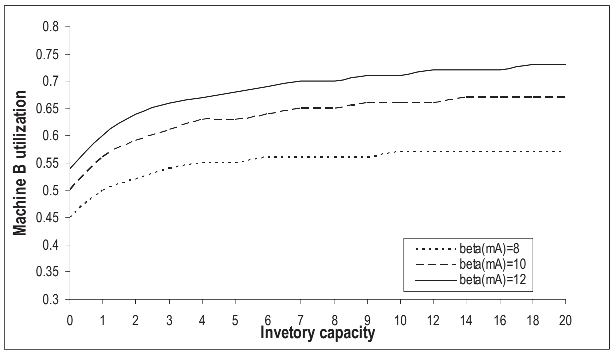

Figure 10: Machine B utilization vs. inventory capacity

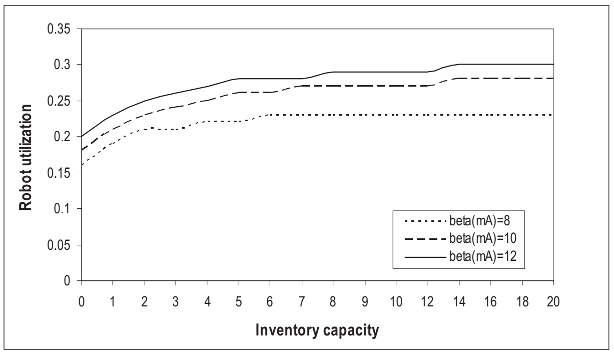

Figure 11: Robot utilization vs. inventory capacity

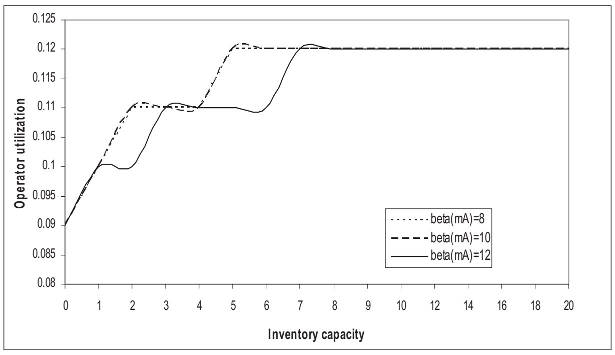

Figure 12: Operator utilization vs. inventory capacity

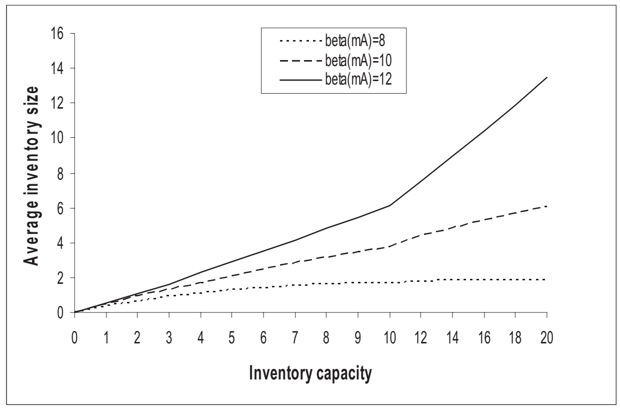

Figure 13: Average inventory size vs. inventory capacity

At best operating conditions, the utilization of machine B is about 73%. From the perspective of machine A, robot availability and synchronization concerns have the effect to diminish a little machine A utilization as its production rate augments from 8 to 12. However, system productivity and utilization of machine B are good in any case due to the inventory mediation. The incidence on the overall system behaviour of repairing failed components can be checked on Fig. 12 which portrays the operator utilization.

Fig. 13 depicts specifically the inventory usage, by showing the (temporal) average size of the inventory (collected through a path object) vs. the inventory capacity. When the production speed of machine A is 8, the average size of the inventory is definitely about 1.87 (even with unbounded capacity). With an unbounded inventory, though, it was found an instantaneous peek value of the inventory size of about 40.

By increasing machine A production speed, as expected, machine A tends to fill up the inventory as much as possible. In particular, for beta(mA)=10 (the two machines have identical speed), the average inventory size tends to be about 8 with a peek value, with an unbounded inventory, of about 90. In the case beta(mA)=12, the inventory is more intensely occupied. As witnessed by Fig. 13, the inventory average size continually increases, meaning that as long as there is a free slot in the inventory, machine A tends to fill it. The average inventory size, with capacity set to 20, was found to be about 13.5. Using unbounded capacity, it emerged that the average inventory size is about 139000 with a peek value of about 277000.

CONCLUSIONS

This paper argues that statechart-based actors (agents) have the potential to be really useful in the modelling and simulation of complex systems. A flexible and efficient (from both time and space) Java framework was developed which can directly be used through programming.

On-going and future work is geared at:

- porting the implementation in the Theatre architecture (Cicirelli *et al*., 2007b) for distributed simulation of very large systems, e.g. over HLA
- extending the realization toward design and implementation of embedded real-time systems
- experimenting with the use of statechart actors in the support of Parallel DEVS systems (Zeigler *et al*., 2000; Cicirelli *et al*., 2008)
- implementing a graphical tool for visual design of statechart actors and automatic generation of Java code.

REFERENCES

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

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

Cicirelli, F., Furfaro, A., Giordano, A., and Nigro, L., 2007a. An agent infrastructure for distributed simulations over HLA and a case study using unmanned aerial vehicles. *Proceedings of 40th Annual Simulation Symposium*, IEEE Computer Society Press, pp. 231-238. March 26-28, Norfolk (VA, USA).

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

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

Fortino, G., Nigro, L., Pupo, F. and Spezzano, D., 2001. Super actors for real-time. *Proceedings of Sixth Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS01)*, IEEE Computer Society, pp. 142-149. Rome (Italy).

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

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

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

Selic, B. and Rumbaugh, J., 1998. Using UML for modeling complex real-time systems. Available on-line at: http://www.ibm.com/developerworks/rational/library/content/03July/1000/1155/1155_umlmodeling.pdf [accessed April, 2008]

Vijaykumar, N.L., de Carvalho, S.V. and Abdurahiman, V., 2002. On proposing statecharts to specify performance models. I*nt. Trans. in Operational Research*, 9, 321-336.

Vijaykumar, N.L., de Carvalho, S.V., Andrade, V.M.B. and Abdurahiman, V., 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.

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