Accepted Manuscript

Modeling and Simulation of Complex Manufacturing Systems using Statechart-based Actors

Franco Cicirelli, Angelo Furfaro, Libero Nigro

PII: S1569-190X(10)00222-4
DOI: 10.1016/j.simpat.2010.10.010
Reference: SIMPAT 1025

To appear in: Simulation Modeling Practices and Theory

Received Date: 3 July 2010
Revised Date: 1 October 2010
Accepted Date: 18 October 2010

Please cite this article as: F. Cicirelli, A. Furfaro, L. Nigro, Modeling and Simulation of Complex Manufacturing Systems using Statechart-based Actors, Simulation Modeling Practices and Theory (2010), doi: 10.1016/j.simpat.2010.10.010

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.
Modeling and Simulation of Complex Manufacturing Systems using Statechart-based Actors

Franco Cicirelli, Angelo Furfaro, Libero 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, actors, statecharts, modelling and simulation, distributed simulation, HLA, Java.

ABSTRACT
This paper describes a multi-agent architecture based on the actors computational model, for the distributed simulation of discrete event systems whose entities have a complex dynamic behaviour. 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 thread-less 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 achieved by partitioning a system model among a set of logical processes (theatres). Timing management and inter-theatre communications rest in a case on the High Level Architecture services. The paper illustrates the practical application of the proposed modelling and simulation methodology by specifying and analysing a complex manufacturing system.

1. INTRODUCTION
The size and complexity of systems which are usually modelled as discrete event systems (DESs) (e.g. communication networks, biological systems, weather forecasting, manufacturing systems, 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., 2009b)(Cicirelli et al., 2007b)(Jang et al., 2003). In this context, state-based formalisms 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.

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 (Vijaykumar et al., 2002-2006). Other uses of statecharts concern the behavioural specification of multi-agent systems.

In (Guer et al., 2001) a combination of Object-Z and statecharts is proposed for formally specifying urban transportation systems. A specification is translated into Statemate working environment (Harel et al., 1999) for rapid prototyping and system simulation. In (Obst, 2002) a method for specifying multi-agent systems using statecharts is proposed with the goal of making agents more adaptive in the context of soccer simulation. Here statecharts models are translated into plan scripts which agents select according to preferences and probabilities to react and adapt to uncertainty environment situations. In (Arai & Stolzenburg, 2002) UML statecharts are used for modelling and analysis of multi-agent systems for robot soccer, network applications etc. A specification is made executable by transforming it into Prolog.

An integrated approach –eUDEVS- for modelling and simulation is described in (Risco-Martin et al., 2009). It characterizes for the use of the DEVS formalism (Zeigler et al., 2000) for turning an UML specification executable, and in particular analyzable through simulation. eUDEVS permits the modeller to start with defacto standard UML modelling diagrams when abstracting a system structure and behaviour. UML diagrams, including statechart diagrams,
are then mapped into the specific class of DEVS models termed Finite and Deterministic DEVS (FD-DEVS) which are represented into an XML format conforming to the XFD-DEVS schema. A transformed specification can then be simulated for quantitative property evaluation using such tools as DEVSIJAVA (DEVSIJAVA, on-line) and Microsim/Java (Mittal, 2008).

Other approaches for transforming specifically UML statecharts directly into Java code for the purpose of execution/simulation are reported in (Niaz & Tanaka, 2004)(Tiella et al., 2007).

The approach proposed in (Vijaykumar et al., 2002-2006) depends in particular on statecharts with and-decomposition and event broadcasting (Harel & Politi, 1998). It allows one to analytically study a statechart model preliminarily transformed into a continuous time Markov chain (CTMC).

This paper proposes the use of an original Java framework which supports simulation of agents whose behaviour is modelled by means of statecharts. A key difference from the work e.g. of Vijaykumar et al. (Vijaykumar et al., 2002-2006)(Frances et al., 2005) relates to the fact that statecharts are used both for modelling and simulation. The use of discrete-event simulation opens to the possibility of choosing probability distribution functions for event occurrences beyond the exponential one which is normally a prerequisite for building a CTMC as in (Vijaykumar et al., 2002-2006)(Frances et al., 2005). In addition, simulation can be flexibly directed to investigation of general quantitative properties of system behaviour.

More specifically, this paper proposes a notion of statechart-based actors (Agha, 1986)(Cicirelli et al., 2008a-2009a) which can be configured to run on a standalone machine or on a networked context according to the Theatre architecture (Cicirelli et al., 2009b). Adopted actors are lightweight, thread-less 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) which is expressed by a “distilled” statechart where only the or-decomposition of states is admitted. Concurrent sub states are avoided. The actor infrastructure is control-centric, i.e, it rests on the possibility of customizing the control engine (i.e, the scheduling/dispatching message services) regulating the runtime evolution of actors. Basic actor infrastructure was successfully applied as an efficient agent middleware for distributing RePast models for high-performance simulations over HLA (Cicirelli et al., 2010), for supporting Parallel DEVS M&S (Cicirelli et al., 2008b) and as a viable net-centric interoperable architecture for general DEVS based systems (Cicirelli et al., 2008c).

The rest of this paper is structured as follows. First a flexible manufacturing system (FMS) model is described in section 2, which is used as a running example throughout the paper. The system exploits the modelling features of statechart actors. The paper then clarifies, in section 3, the semantics underlying hierarchical actors. After that a developed Java API is outlined in section 4 which supports statechart actors which can be easily operated under a discrete/dense time model. In addition the resulting programming style in Java is shown. The paper goes on by summarising in section 5 the Theatre architecture which permits an actor model to be split and simulated over a networked context, e.g. abstracted by HLA/RTI services. The practical aspects of statechart-based actors are demonstrated by simulating and reporting, in section 6, the collected results of the FMS model. Starting from the simulation experiments, the FMS model is then extended and scaled, in section 7, so as to improve its dynamic behaviour. Subsequently, the extended FMS model is further scaled so as to demonstrate, in section 8, the achievable simulation performance. Finally, the conclusions are presented along with an indication of further work.

2. AN FMS MODELLING EXAMPLE

The following describes a modelling example based on statechart actors, related to a flexible manufacturing system (FMS). The example was adapted from (Vijaykumar et al., 2002-2006) where such a model was developed using statecharts with and-decomposition and event broadcasting (Harel & Politi, 1998), and analytically studied by preliminarily covertting it into a continuous time Markov chain (CTMC). In this paper the FMS model is instead simulated. Later in this paper, the FMS model will be extended and scaled for demonstrating the modelling and simulation capabilities of the proposed approach.

A functional unit is considered (see Fig. 1) which is composed of two machines, an inventory and a robot. The two machines, respectively referred as MA and MB, operate in series and process the submitted jobs for producing a single product. Jobs are first processed by MA and then by MB. The inventory is used for provisionally storing jobs already processed by MA which cannot immediately be handled by machine MB. The inventory has a bounded capacity and allows to decouple the operation of the machines thus reducing their wait times. The robot is actually in charge of loading/unloading the two machines with jobs, possibly using the inventory for temporary job buffering. Each of these entities is modelled by a software component (actor) whose behaviour is specified by means of a statechart. Both machines and the robot may be subject to failures in which case they need the assistance of a human operator for being fixed and able to restart their work. Whenever a failure is detected a corresponding reparation request is issued to a
software system, named FacilityManager, which is in charge of queuing and dispatching such requests to the available operators.

![Diagram of FunctionalUnit, Inventory, Robot, Operate, and FacilityManager](image)

**Figure 1.** System architecture

As in eUDEVS (Risco-Martín *et al.*, 2009) the system structure is abstracted through an UML component diagram which clarifies components, their interconnections and the associated contracts, i.e. the message interfaces which specify requested/offered message types. Inter-component connectors are of the “socket/ball” type and typically use implicit ports. Delegation connectors (dashed lines in Fig. 1) realize external/internal connections and specify the internal components devoted to serving externally originated messages or providing requests to external components. Coupled components like FunctionalUnit are supposed to be unfolded when transforming the component diagram into the terms of Java actor design and implementation (see later in this paper).

It can be seen that the FunctionalUnit component is actually made of four interconnected subcomponents: two instances of Machine, one of Robot and one of Inventory. The component named FacilityManager is able to manage a certain number of operators (modelled as instances of the Operator component) by routing to them repair requests coming from more functional units. An Operator, in order to satisfy a repair request, needs to be connected to the component exporting the Repairable interface which issued the request. Obviously, such a binding is achieved dynamically through the mediation of the FacilityManager component which transmits to the chosen operator the proper reference of the component that needs to be fixed.

Event broadcasting, assumed in (Vijaykumar *et al.*, 2002-2006), is simulated 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.

Figures from 2 to 4 depict the statecharts modelling respectively the behaviours of Machine, 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.

![Machine Top Statechart](image)

**Figure 2.** Machine behaviour

After being initialized, a machine (see Fig. 2) goes into state W where it waits for a job to be processed. 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 facility manager through a RepairReq message. After being repaired (arrival of the external message EndRepair coming from an 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.

![Inventory behaviour](image)

The behaviour of the Inventory is portrayed in Fig. 3. 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. 3. 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. 4 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. 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 condP is specified as the logical disjunction of all of them (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).

![Inventory behaviour](image)

Table 1: Robot conditions

| cond1 | MB is in WU; |
| cond2 | MA is in WU & MB is in W; |
| cond3 | MB is in W & ~inv is empty; |
| cond4 | MA is in WU & ~inv is Full; |
| cond5 | MA is in W; |
condP = cond1 ∨ cond2 ∨ cond3 ∨ cond4 ∨ cond5;

The time spent by the robot in any operating state depends on the particular operation (see Table 2) that it is accomplishing. All the density probability distribution functions in Table 2 are assumed to be negative exponential. 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.

### Table 2: Timing attributes

<table>
<thead>
<tr>
<th></th>
<th>Machine A</th>
<th>Machine B</th>
<th>Robot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production rate</td>
<td>βp=8, 10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Failure rate</td>
<td>λA=1</td>
<td>λB=0.5</td>
<td>λR=1</td>
</tr>
<tr>
<td>Repairing rate</td>
<td>μA=10</td>
<td>μB=15</td>
<td>μR=10</td>
</tr>
<tr>
<td>Loss probability</td>
<td>πA=0.5</td>
<td>πB=0.3</td>
<td></td>
</tr>
<tr>
<td>Loading rate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unloading rate</td>
<td>δU1=100</td>
<td>δU2=100</td>
<td></td>
</tr>
<tr>
<td>Loading rate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unloading rate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moving rate from m. A to m. B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image.png)

**Figure 4. Robot behaviour**

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 issues a repair request to the relevant facility manager.
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 intervention of an operator is asked. 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.

![FacilityManager behaviour](image)

Fig. 5 reports the behaviour of a FacilityManager. The Active state corresponds to a situation where there is at least one available operator. In this state the arrival of a repair request is handled by forwarding it to an operator. Each request carries inside it the identity of the device which needs to be repaired. If the assigned operator were the only one available, the FacilityManager switches to the FullBusy state, otherwise it remains in Active. Once an operator becomes available again it sends to the FacilityManager an IdleOperator message. If such a message is received in the Active state, the manager simply increases the count of available operators.

The FullBusy state corresponds to a situation where there are some pending repair requests but no available operators. In such a case, the arrival of a new repair request is queued for being successively accounted for. The arrival of an IdleOperator allows the manager to handle the first queued request. If the request queue becomes empty the Active state is entered.

![Operator behaviour](image)

The behaviour of the Operator is depicted in Fig. 6. After being initialized, it waits in state W for a repair 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. The completion of the repairing process is achieved with an internal message EndProcess to which the actor reacts by sending both an EndRepair message to the relevant actor and an IdleOperator message to the facility manager. The role of the state M and of the Transfer and BackHome messages will be clarified successively when a more complex scenario is considered.

3. CONCEPTS OF HIERARCHICAL ACTORS

The following clarifies and elaborates basic concepts and semantics of statechart actors which were intuitively used in the FMS model described in the previous section.
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 and represents the unit of scheduling and dispatching in a subsystem of actors. The dynamic behavior of an actor is specified by means of a statechart 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 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 and 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 (*proactive* behaviour), create new actors, change the current state of the actor. From this point of view, the runtime behaviour of a statechart actor based system follows the “UML semantics” (Eshuis, 2009)(UML, on-line). The interpretation proceeds according to a *step* notion which comprises one single event (message) whose processing always runs to completion. Events generated during a step are buffered so as to be subsequently selected, dispatched and processed, one at a time. When a system is split into multiple subsystems executed on distinct processors, the step semantics makes it possible that during a step in a subsystem, external messages can be received which get queued in the normal way.

The actor model adopted in this work differs from systems modelling as parallel composition of multiple statecharts like in (Vijaykumar et al., 2002-2006)(Frances et al., 2005) for the system structure dynamism. In the actor model the system structure can vary dynamically and also the acquaintance relationships among actors can be changed at runtime and communicated by messages. A static system structure is instead implicitly assumed in (Vijaykumar et al., 2002-2006)(Frances et al., 2005) for statecharts modelling.

At a given point in time, an actor finds itself simultaneously in a set of states that constitute 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 (see e.g. Fig. 4). 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 or message 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. When omitted, the guard is implicitly assumed to be true. For the sake of simplicity, i.e. to avoid picture cluttering, the action part of transitions in the FMS example is often left unspecified.

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 a 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 (optional) 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 and 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, if there are any, of the state.

4. JAVA FRAMEWORK FOR STATECHART-BASED ACTORS

The following outlines the development of a minimal Java framework supporting hierarchical actors (see Fig. 7), by commenting the most important API classes and interfaces. For simplicity, distribution concerns are omitted.

**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.

![Diagram](https://example.com/diagram.png)

**Figure 7. Simplified UML class diagram of Actors framework**

**SimulationDiscreteTimeClock, SimulationDenseTimeClock.** 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, handle( 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 (or 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
counter 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() or exitAction() to be executed. Message Notify in Fig. 4 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 a 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 another concrete class providing common methods for random variate generators, e.g. uniform,
exponential, normal etc.

4.1 Programming Style
To give an idea of the resultant Java programming style, Fig. 8 shows an excerpt from the Robot actor of the FSM
model example. The setupBehaviour() method of the Robot class is in charge of creating the state hierarchy of the

```java
public setupBehaviour() { //method invoked by the constructor
    //build the state hierarchy
    MacroState Top=new MacroState(null);
    State heurpex LeafState(Top);
```

With respect to the case of plain actors (Cicirelli et al., 2009b), there is no need to redefine the handler() method
inherited from the Actor base class. All of this simplifies programming and makes it possible to automating translation
from a visual design to Java code.
5. THEATRE: THE SYSTEM VIEW OF ACTORS

The general architecture of a distributed system based on actors is named Theatre (see Fig. 9) and 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.

Each theatre node is equipped of 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 as 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. As a consequence, the operation of the control machine naturally adheres to the step semantics discussed in section 3.
The Transport Layer furnishes the services for sending/receiving network messages and migrating agents. When moving to a different theatre, an actor leaves on the originating theatre a forwarder of itself (i.e. a proxy) which has no status and serves only to redirect incoming messages to current known location of the actor. For efficient communications, the address information of a moving actor is updated in source theatres so as to minimize the number of hops of a transmitted message. More details about the lightweight migration mechanism are described in (Cicirelli et al., 2009b). 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. In this paper HLA is used as a “transport layer” and its timing services exploited.

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., 2007a) 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., 2009b) 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 handling contemporary or simultaneous messages, i.e. occurring a same simulation time, and required to fulfill precedence constraint relationships, is described in (Cicirelli et al., 2008b) 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., 2009b) is assumed for the distributed simulation experiments based on statechart actors.

![Theatre architecture](image)

**Figure 9: Theatre architecture**

### 6. SIMULATING THE FMS MODEL

The same setting and time attributes (see Table 2) of (Vijaykumar et al., 2002-2006) were assumed for making some simulation experiments on the FMS model previously described in this paper. In particular, the facility manager handles one single functional unit and only one available operator. Table 2 gives the rates (number of events per time unit) used for simulation. Loading/unloading rate of machines are shown as dwell times in the corresponding operation state of the robot.

The manufacturing model was simulated using dense time. Each experiment lasts after a time limit of $t_{\text{end}}=5\times10^4$.

A Monitor actor for collecting useful information about the simulation was added to the model. The 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 $t_{\text{end}}$. Following the arrival of such a message, the actor displays collected statistical information.

The system model was thoroughly 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. 10. Observed system productivity vs. inventory capacity.

Fig. 10 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.

Fig. 11. Average wait time for unloading machine A

The positive effect of using a not zero inventory size can be checked in Fig. 11 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.

Fig. 12. Average wait time for unloading machine B

For completeness, Fig. 12 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 Figs. 13 to 15).
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. 16 which portrays the operator utilization.
Fig. 17 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 peak 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 peak value, with an unbounded inventory, of about 90. In the case beta(MB)=12, the inventory is more intensely occupied. As witnessed by Fig. 17, 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 peak value of about 277000.

7. EXTENDING THE FMS MODEL

From Fig. 16 it emerges that the operator utilization is quite low (about 12% in the best case). This resource may be better exploited by having one single operator that serves multiple functional units. The architecture of the model (Fig. 1) easily allows to evaluate such scenarios by simply increasing the number of functional units handled by the single facility manager. Fig. 18 reports the utilization factors of the various components vs. the number of functional units served by one operator. The adopted system parameter values are those that, in the previous simulation study, allowed to obtain the best results in terms of productivity, i.e. beta(MA)=12, beta(MB)=10 and a capacity of 20 items for the inventory. It can be seen from Fig. 18 that, as expected, the operator utilization drastically grows while the other components experience a moderate reduction of their utilization.

The utilization decrease is less than 5% when up to five functional units are considered whereas, in such case, the operator utilization reaches a value of 58%. Fig. 19 reports the effect of these changes on the (average) productivity of the various functional units. Since in the case of five functional units the productivity decrease is of about 6%, it seems that such a ratio is a reasonable trade-off between the increase in the operator utilization and the decrease of system productivity.

Considering that the operator is not full busy there is still room for improving the effect of its work. In the simplistic assumption that a facility may contain an arbitrary number of functional units and operators, the effect of scaling the
number of functional units, keeping a ratio of 5:1 with the operators, was studied. The results are reported in Fig. 20 and Fig. 21 and witness that this setting has a quite good effect because it is able to compensate the loss on the system productivity by increasing the utilization of the various components.

![Figure 20. Productivity vs. scale factor](image1)

![Figure 21. Utilization vs. scale factor](image2)

In a real manufacturing system, though, due to available physical room, there is a limit on the number of functional units that can be host inside one single facility. In a complex plant with multiple facilities, a solution overcoming the space problem needs to be devised. In the following it is assumed that a single facility can host five functional units and that the plant layout allows the transfer of operators from a facility to a near one within a reasonable amount of time. If the operator transfer time is low, with respect to the time needed for fixing a device, a strategy for sharing the operators may be exploited with the aim of approximating the positive effect shown in Fig. 20.

In order to study such a complex scenario, the system architecture was slightly changed as illustrated in Fig. 22. A new composite component (Facility) is introduced which models one facility. It is now possible to connect more facilities thus reproducing the fact that some of them, which are physically close to one another, are able to communicate and share the available operators. The interface and the behaviour of the FacilityManager were changed accordingly.

![Figure 22. Architecture of a Facility](image3)

Fig. 23 depicts the statechart modelling the behaviour of a FacilityManager component which is now aware of the presence of other facilities in its nearness and it is able to implement an operator-sharing strategy. Whenever the FacilityManager reaches the FullBusy state, it periodically sends OperatorReq messages to its neighbourhood peers asking for operators. The FacilityManager reacts to an operator request depending on its current state. If it finds itself in the FullyBusy state it simply ignores the message. Otherwise, it replies with a RequestAck message that acknowledges the request. The designed protocol requires that the FacilityManager that receives a RequestAck must reply with a DeclineReq or a Commit message. The first type of message indicates that the requesting manager no longer requires help, because in the meanwhile a local operator became idle or another RequestAck was accepted. A Commit message is instead used for committing the agreement for an operator transfer and it embeds a repair request. Upon receiving a Commit message, the FacilityManager extracts the request contained in the message and forwards it, through a Transfer message, to one of its operators.
Two different operator-sharing strategies were considered. The simplest one consists in asking help to all the neighbours without remembering committed operator loans, i.e. once an operator transfers to a facility it is viewed in exactly the same way as a local one. As a consequence, the operators freely move among the various facilities on the basis of the transfer requests. In a more sophisticated strategy each facility manager keeps track of the balance between satisfied requests (credits) and received help (debits). More in particular, whenever a request for help has to be issued, the manager asks only the neighbours with which it has no debits. In addition, while in the Active state, if an IdleOperator or a DeclineReq is received the manager tries to pays off a debit by sending a Transfer, holding a BackHome message, to one of its available operators. It is worth noting that the acquaintance relationship between an operator and a served facility is dynamically established and that the system exhibits a variable structure.

As it can be seen from Fig. 6, a Transfer message leads the receiving operator to the state M. As soon as M is entered, the operator extracts the payload contained within the Transfer (i.e. a RepairReq or a BackHome). Then the operator schedules the extracted message so as to be received by itself after a dwell time which models the time required for moving to the destination facility.

Fig. 24 shows the layout of facilities and their neighbourhoods used for experimenting with the two strategies (respectively referred to as “no debts” and “debts” in Fig. 25). The operator transfer time was modelled as a uniform random variable whose interval is \([7\times 10^{-3}, 13\times 10^{-3}]\) t.u. The timeout time for a help request was set to \(4\times 10^{-2}\) t.u.

The results reported in Fig. 25 show that both strategies are effective in approximating local operator-sharing (compare with Fig 20) and that the strategy accounting for debts/credits outperforms the naive one. Statistics about utilizations are not reported because they are very close to those reported in Fig. 21.

8. MODEL SCALING AND SIMULATION PERFORMANCE

The Theatre infrastructure allows the distributed simulation of huge complex models (Cicirelli et al., 2009b)(Cicirelli et al., 2007b). After having analyzed the manufacturing system behaviour, another set of simulation experiments were
conducted specifically addressing model scaling and simulation performance on a distributed context. The model of Fig. 24, along with the debits/credits strategy, was scaled by increasing the number of facilities and simulating it both on a single LP and on three LPs/theatres/processors. It is worth noting that, in the distributed setting, model partitioning implies that the agents corresponding to operators which have to move from a facility running on a LP to another facility residing on a different LP, need to migrate to the destination LP.

The simulation model was separately configured, validated and executed on a single machine and through a federation of three theatres using pRTI1516 (Pitch, on-line) as HLA implementation. A driver actor is in charge of model deployment, configuration and partitioning. In particular, the driver is equipped with an itinerary containing the addresses of the theatre nodes involved during simulation. Following such itinerary, the driver migrates from a node to another by instantiating the various components of the manufacturing system and by establishing acquaintance relations among facilities. Driver actor was also used to setup the system in the centralized scenario. In such a case, the itinerary only contains the address of one single LP.

The three theatres, among which the entire model was equally split, were allocated on three Pentium IV, 3.4 GHz, 1GB RAM, WinXP platforms interconnected by a Gigabit Ethernet switch. A grid size of 40x30 to 40x270 facilities was considered.

![Speedup vs. number of facilities](image)

**Figure 26. Simulation speedup vs. number of facilities (three processors)**

The wallclock time (WCT) required for completing the simulation respectively in the distributed context and on a centralized one (in this case the simulation model was run on a single machine of the distributed system, without HLA) was measured.

Fig. 26 portrays the observed simulation speedup, i.e. the ratio between centralized WCT and distributed WCT versus the number of facilities.

9. CONCLUSIONS

This paper describes 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.*, 2009b) 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 or to respond to load-balancing requirements.

Theatre and statechart actors were implemented on top of HLA services (Kuhl *et al.*, 2000)(Pitch, on-line) in the presence of conservative synchronization. Similar solutions, though, based on Java Sockets and Java RMI as the transport layer, and with a customized distributed time coordinator engine were also implemented.

On-going and future work are geared at:

- implementing a graphical tool for visual design of hierarchical actors and automatic generation of XML model representation and/or corresponding Java code
- improving M&S capabilities of statechart-based actors by experimenting with environment-based multi-agent systems (Logan & Theodoropoulos, 2001)
- extending the approach toward optimistic synchronization, by implementing a Time Warp based distributed simulation algorithm (Fujimoto, 2000)(Cicirelli *et al.*, 2007a).
The second point deserves some further comments. The example presented in this paper does not use the environment concept at all. The point-to-point 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., 2009b-2010)) 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” proposed in (Logan & Theodoropoulos, 2001).

REFERENCES


