A COMPONENT-BASED ARCHITECTURE FOR MODELLING AND SIMULATION OF ADAPTIVE COMPLEX SYSTEMS

Franco Cicirelli, Angelo Furfaro, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, 87036 Rende (CS) – Italy
E-mail: {f.cicirelli,a.furfaro}@deis.unical.it, {l.nigro,f.pupo}@unical.it

KEYWORDS
Software architecture, variable structure, components, composition, agents, modelling and simulation, server relocation, Java.

ABSTRACT

This paper proposes a component-based software architecture (Theatre) hosted by Java, which enables modelling and discrete-event simulation of complex and dynamically reconfigurable systems, possibly on top of a distributed computing context. At the "programming in-the-small" level, Theatre rests on light-weight reactive components (actors or agents) which interact to one another by asynchronous message-passing. Actor behaviour is modelled by a finite state machine. Actors can be easily composed to create new reusable components. At the "programming in-the-large" level a subsystem of actors can be assigned to an execution locus (theatre). A theatre provides to local agents the basic message scheduling, dispatching, communication and mobility services. The paper describes component-based M&S support of Theatre and demonstrates its practical use through examples.

INTRODUCTION

The work described in this paper aims at the development of language structures and software tools for modelling and simulation of complex systems which are component-based, timed, mobile and whose structure can change during runtime (Hu *et al.*, 2005)(Jang *et al.*, 2003)(Jang & Agha, 2006)(Posse & Vangheluwe, 2007)(Cicirelli *et al.*, 2007b). Such systems are not adequately supported by conventional M&S tools where structure is often assumed to be static and dynamism only relates to state changes caused by the occurrence of events. However, many systems exist (e.g. predator/prey models in biology, adaptive networks in telecommunication systems accommodating for the presence of mobile users, and so forth) which require structure dynamism for them to be effectively modelled and analyzed.

In the context of DEVS (Zeigler *et al.*, 2000) –Discrete Event System Specification- and particularly in the DEVSJAVA environment (DEVSJAVA) some extensions were defined (Hu *et al.*, 2005) which allow variable structure models to be dealt with. All of this relies on adding/removing component models, adding/removing couplings among models and adding/removing input/output ports to models. Changing the interface of a component is a critical aspect because it may require modifications to the component behaviour.

A modelling language directly founded on the specification of adaptive, dynamic structure discrete systems is Kiltera (Posse & Vangheluwe, 2007). Kiltera is formally based on a process algebra with two-way communications and timing constructs, which is useful to specify systems whose structure can change dynamically through the concept of *link mobility*, i.e. the possibility of altering the channel interconnection infrastructure among system components (processes). At current time, though, Kiltera is not assisted by concrete tools for making simulation of complex modelled systems, e.g. on a distributed context.

This work argues that mobile agent systems offer a natural yet challenging computing infrastructure where to build and simulate dynamic structure systems.

Jang *et al*. in (Jang *et al.*, 2003)(Jang & Agha, 2006) propose a distributed agent architecture based on the Actors Model (Agha, 1986) especially designed and implemented for modelling and simulation of large adaptive systems. The approach is characterized by the techniques it uses for ensuring efficient communications despite agent mobility, and the provisions e.g. for co-locating highly interacting agents thus conserving bandwitdth during distributed simulation. The agent architecture was applied to modelling and distributed simulation of unmanned aerial vehicles which cooperate to one another in order to fire moving targets. In the application, agent discovery as well as patterns of interaction and coordination are intrinsically dynamic and challenge for the availability of suitable runtime infrastructures.

Theatre (Cicirelli *et al.*, 2007b) is a software architecture (Shaw & Garlan, 1996) which belongs to the family of actor (agent) computational models and

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

rests on asynchronous message-passing. Key features of Theatre are: (i) the adoption of a lightweight notion of actors, which does not introduce internal threads and thus favours time predictability in real-time applications and the achievement of good performance in distributed simulations; (ii) the use of a runtime executive which can reason upon "real" or virtual time, and which can be customized through programming in order to fulfil specific application needs; (iii) a direct embedding in Java through a minimal API, which can exploit common transport layers like Java Socket, Java RMI and recently HLA/RTI infrastructure (iv) the use of an efficient mobility mechanism which is a key for model adaptivity.

This paper focuses on the component-based modelling and simulation capabilities of Theatre. The compositional mechanism is illustrated which facilitates the construction of component off-the-shelf reusability units which are not distinguishable from elementary components. The paper demonstrates the practical use of Theatre and shows a variable structure system concerning a distributed adaptive relocation server model.

AN OVERVIEW OF THEATRE

A system consists of a collection of interacting theatres. Each theatre offers the runtime executive to a collection of application actors. In particular, a theatre furnishes to local actors the basic services of message scheduling/dispatching and timing, as well as mobility and communication mechanisms. Communication is based on one-way asynchronous message passing: the send operation is non-blocking.

An actor (see Fig. 1) is characterized by its *message interface*, *hidden data variables* and *behaviour* which is modeled as a finite state machine.

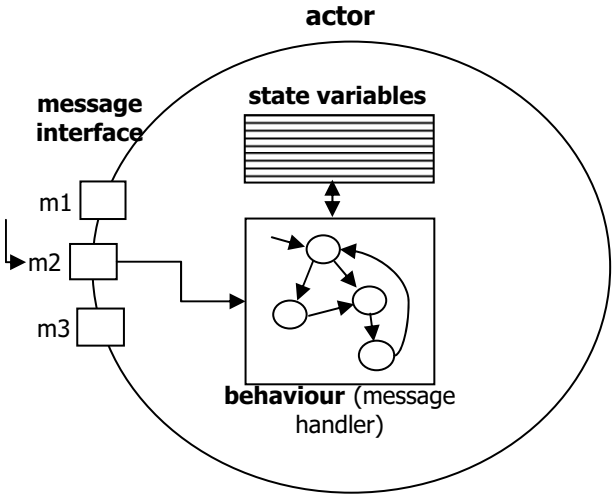

Fig. 1. Structure of an actor

Messages are first-class citizens: they can be sent and transparently buffered and managed according to different control disciplines. It is up to a theatre *control machine* to superimpose to messages the most apt control structure, tuned to the application needs.

The *controller* component of the control machine is in charge of repeating a basic loop. At each iteration, first the (or a) most imminent message is selected among pending messages, then the message is dispatched to its relevant destination actor. Message processing constitutes an atomic action and extends the control thread of the controller. At message processing termination, the controller loop is re-entered and continues with the next iteration.

An actor responds to an incoming message by executing basic actions as in the following:

- (**new**) creating new actors
- (**send**) sending messages to known actors (*acquaintances*) including itself (proactive behaviour)
- (**defer**) deferring a message to future when the message cannot be accepted in current state. Deferred messages are automatically re-sent as soon as the actor changes its state
- (**become**) making a state transition in the actor automaton. The next state depends on the arrived message and current status.

A lean Java framework (API) provides basic actor mechanisms. Actor classes derive directly or indirectly from the Actor abstract base class. Message classes are heirs of Message abstract base class which associates with a message its actor receiver. Main operation signatures in Actor are as follows:

```
public void send( Message m, long… at )
protected int currentStatus()
protected void become( int next_status )
protected void defer( Message m )
protected long now()
protected void handler( Message m )
```

Method send() can carry also zero, one or multiple time information. An instantaneous message does not have the at parameter. A typical timed message is accompanied by its occurrence time. Message temporal information is meaningful to a control machine which reasons upon time (e.g. a simulation machine). Current time is available to actors through the now() method whose exact implementation is responsibility of a control machine. Actor design purposely hides to actors the identity of a particular control machine.

The handler() method is activated by the controller with the incoming message as an argument. handler() codifies the actor finite state machine.

Message classes can directly be embedded in a user-defined actor class. In alternative, messages can be part of an interface which extends the MessageIF interface which defines the send() method according to the same signature as in Actor. An actor class then implements a message interface which acts as a contract with its peers. Actors normally have no need to override the send() method of MessageIF: they can rely on the version inherited from Actor. The send() method can be redefined in order to favour compositionality (see later in this paper).

An actor is always created as a local object of a theatre. After that, the actor can migrate to a different theatre. A theatre maintains information about local executing actors. After migration, on the original theatre a *forwarder* (proxy) version of the actor is kept. Would an actor come back to a theatre where a proxy version of itself exists, the actor status is copied upon the proxy which then switches to normal actor status. The approach ensures that Java actor references persist despite migration. Migration rests on a customization of Java serialization mechanism and minimal recourse to reflection for copying actor data statuses (Cicirelli *et al.*, 2007b). For communication efficiency, in the case a message experiments multiple hops before reaching its destination, the addressing information on the sender theatre will be automatically updated with current destination of the receiver theatre.

The control machines of a distributed simulation system based on Theatre cooperate to one another for time synchronization. Both conservative (Cicirelli *et al.*, 2006a)(Cicirelli *et al.*, 2007b) and optimistic (Cicirelli *et al.*, 2007a) synchronizations are possible.

ACTORS AS COMPONENTS

A component (Brown & Wallnau, 1998) is a "non trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces".

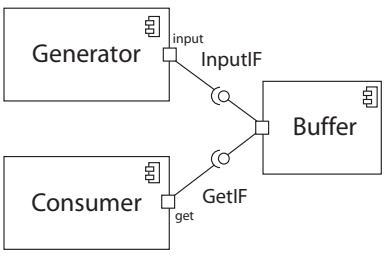

Fig. 2. Actors, ports and connectors

Actors naturally adher to the software component vision. The architecture of a subsystem of actors can be specified by an UML2 component diagram (Fig. 2) which shows ports, connectors and interfaces.

In Fig. 2 one Generator generates jobs towards a Buffer, and one Concumer gets jobs from the Buffer and consumes them. Generator has an output port input which is bound to the *required* interface InputIF (socket notation). Similarly, Consumer has an output port get which requires the GetIF interface. The Buffer has an input port which *provides* both InputIF and GetIF interfaces (ball or lollipop notation). InputIF and GetIF are respectively a contract for the Generator and the Consumer which can actually work with any actor which provides (implements) the required contract.

The interpretation of ports and connectors is straightforward. An output port corresponds to an acquaintance, i.e. an actor to which messages are sent asking for some services. An input port corresponds to the this actor, i.e. the actor who effectively provides the services (messages) specified in the exported interface. As a consequence, connectors between actors are simply Java references.

A port can be associated with a multiplicity factor to indicate the number of times the port is repeated in the component. For an output port, that is the number of required acquantainces (interacting partners). The realization of a given actor topology as in Fig. 2 is a matter of configuration and can occur, in a case, at system start up time when the main program creates the actor instances and links them by establishing the acquaintance network.

For client-server interactions like those between Consumer and Buffer in Fig. 2, it is assumed that the GetIF interface (see also Fig. 4) specifies the Get message *and* within it a reply JobArr message which the buffer fills in with the returned job and then sends back to the requestor. All of this has an obvious analogy with postal letters which anticipate the message to be used for giving an answer to the sender.

Java Programming Style

Actors can directly be programmed in Java. In the following, the supported type-safe programming style is clarified using the example of Fig. 2. Figg. from 3 to 6 show respectively the Java code of InputIF, GetIF, Generator and Consumer.

```
public interface InputIF extends MessageIF{
   public final class Input extends Message{
      private Job job;
      public Input( Job job ){ this.job = job; }
      public Job getJob(){ return job; }
   }
}//InputIF
```

Fig. 3. The InputIF message interface

The Generator implements a timed reactivation through the local (hidden) message Next. Interarrival time between consecutive generated jobs is uniformly distributed within G0..G1. Each generated job has also a temporal size which expresses its service time. The size is uniformly distributed within S0..S1.

```
public interface GetIF extends MessageIF{
   public static class Get extends Message{
      public static class JobArr extends Message{
         private Job job;
         public Job getJob(){ return job; }
         public void setJob( Job job ){ this.job = job; }
      }
      private MessageIF sender;
      private JobArr reply;
      public Get( MessageIF sender ){
         this.sender = sender;
         reply = new JobArr();
      }
      public MessageIF getSender(){ return sender; }
      public JobArr getReply(){ return reply; }
   }//Get
}//GetIF
```

Fig. 4. The GetIF message interface

```
public class Generator extends Actor{
   private static class Next extends Message{}
   public static final byte ACTIVE=0;
   private InputIF input; //output port
   private int G0, G1, S0, S1;
   private int jobCount=0;
   private Random random=new Random();
   public Generator( InputIF input , int G0, int G1, int S0, intS1 ){
      this.input=input; this.G0=G0; this.G1=G1;
      this.S0=S0; this.S1=S1;
      int d=G0+random.nextInt( G1-G0 );
      send( new Next(), now()+d );
      become( ACTIVE );
   }
   protected void handler( Message m ){
      switch( currentStatus() ){
         case ACTIVE:
            if( m instanceof Next ){
               int d = G0+random.nextInt( G1-G0 );
               int s = S0+random.nextInt( S1-S0 );
               Job job = new Job( jobCount++, now(), s );
               input.send( new InputIF.Input(job) );
               send( m, now()+d );
            }
      }
   }//handler
}//Generator
```

Fig. 5. The Generator

Actor Consumer (Fig. 6) is a simple server. It cyclically requests a job to the buffer; when a job arrives it consumes the job by a timed End message sent to itself. For demonstration purposes, the behaviour is organized in two states: IDLE (awaiting a job from buffer) and BUSY (consuming the arrived job). At the end of the consuming activity, a new job is requested and so forth.

```
public class Consumer extends Actor implements MessageIF{
   private static class End extends Message{}
   public static final byte IDLE=0, BUSY=1;
   private GetIF get; //output port
   private Job job;
   private int consumed=0;
   public Consumer( GetIF get ){
      this.get=get;
      get.send( new GetIF.Get( this ) );
   }
   protected void handler(Message m) {
      switch( currentStatus() ){
         case IDLE:
            if( m instanceof GetIF.Get.JobArr ){
               GetIF.Get.JobArr dispMsg =(GetIF.Get.JobArr)m;
               job = dispMsg.getJob();
               send( new End(), now()+job.getSize() );
               become( BUSY );
            }
            break;
         case BUSY:
            if( m instanceof End ){
               get.send( new GetIF.Get( this ) );
               job = null; consumed++;
               become( IDLE );
            }
      }
   }//handler
   public String toString(){
      return "No of consumed job="+consumed;
   }//toString
}//Consumer
```

Fig. 6. The Consumer

The Buffer component is implemented as an unbounded buffer of jobs. The actor can find itself into one of three states: EMPTY (no buffered job), REQ_PEND (a request for job is pending) and NOT_EMPTY (one or more jobs buffered). Since only one consumer is admitted, at most one pending request can exist at each time. This is mirrored in the REQ_PEND state where only an Input message is expected. The arrival of a job causes it to be replied to the requester and the buffer to come back to EMPTY state.

Fig. 7 depicts a skeleton main which configures the subsystem in Fig. 2 and launches simulation on a standalone machine. The simulation time limit is furnished as an argument to the constructor of the Simulation control machine.

```
public class Driver{
   public static void main( String... args ){
      ControlMachine cm=new Simulation( 1000 /*tEnd*/ );
      Buffer b=new Buffer();
      … input values for G0, G1, S0, S1
      Generator g=new Generator( b, G0, G1, S0, S1 );
      Consumer c=new Consumer( b );
      cm.controller(); //simulation start
      System.out.println( c ); //statistics output
   }
}//Driver
```

Fig. 7. Configuration and launch of a simulation

Composition and Coupled Models

Components off-the-shelf can be built by composing existing actors in order to form a new component (coupled model or composite) which can immediately be reused as a unit. The new component behaves as a folder for the internal components which in turn can be elementary actors or composed actors (hierarchical composition). A composite is not distinguishable from a normal actor. It exhibits to its external environment a collection of input/output ports together with required and provided interfaces. Such interfaces are then delegated to internal components.

An example of a composite is portrayed in Fig. 8 which relates to a computing Node which hides a Buffer, a Dispatcher and a collection of Server*s*. Node has an input port exporting the InputIF message interface, and two output ports respectively associated with required StatIF and OutputIF interfaces. Node receives external generated jobs through the input port, and stores them in the Buffer.

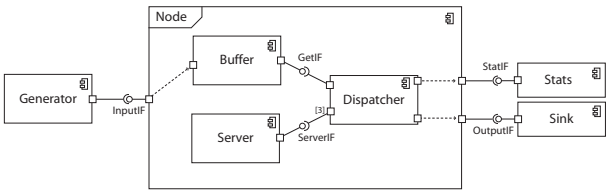

Fig. 8. A Node composite

The Dispatcher requests one job at a time to the Buffer and assigns it to an idle Server (here, a fixed number of servers is assumed). Server behaviour is similar to that of Consumer in Fig. 6, except it now follows a *push* instead of a *pull* model. In other words, instead of asking the buffer for a job, it now waits for a job submission from the Dispatcher. When a server terminates with its job, the Dispatcher gets informed of this fact and sends the processed job to an external Sink. In addition, information about each served job (its id, generation time, service finish time etc.) are captured in a message according to the StatIF interface and sent out for proper statistical processing.

Node composition is specified by its internal structure diagram. Fig. 8 indicates that input messages coming from the external Generator are actually routed to the internal Buffer. All of this is witnessed by the dashed (<<delegate>>) dependency relationship which states that the InputIF message interface is really implemented by Buffer. In a similar way, requests toward Stats and Sink are effectively originated (delegated) by the internal Dispatcher.

From the programming point of view, a coupled model is easy to build. At its construction time, the composite receives, among the other, the acquaintances corresponding to its required interfaces. The composite then creates and links together the instances of its internal components. Sub-components which generate output external messages, are supposed to be directly connected to composite acquaintances.

Internal routing of external incoming messages is achieved by overriding the send() method of MessageIF, so as to forward these messages to delegated sub-components. Forwarding is accomplished by invoking the send() method of the delegate.

Fig. 9 summarizes the configuration of Node by showing operations in its constructor. Fig. 10 depicts the overridden send() method.

```
public Node( StatIF stat, OutputIF out ){
   create Buffer instance b
   create Server instances s1, s2 and s3
   create Dispatcher instance d as
          new Dispatcher( b, stat, out, s1, s2, s3)
}//Node
```

Fig. 9. Node configuration

```
public void send( Message m, long… at ){
   if( m instanceof InputIF.Input ) b.send(m,at);
   …
}//send
```

Fig. 10. Node's send() method

It should be noted that although a coupled model can implement multiple interfaces, it only needs one redefinition of the send() method, which queries the incoming message type through the instanceof operator for detecting the target delegate.

A RELOCATION SERVER MODEL

The computing model of Theatre makes it possible to design and execute variable structure systems. As in Kiltera (Posse & Vangheluwe, 2007) adaptivity depends on link mobility, i.e. the possibility of reconfiguring during runtime the interconnection infrastructure of components by adjusting the acquaintance network of the system. The approach preserves the contract of component interfaces and is very flexible when paired with the mobile agent capabilities of actors which can migrate among the theatres allocated to different physical nodes of a distributed system.

The following describes the design, implementation and distributed execution of a relocation server system which models a collection (pipeline) of interconnected computing nodes. Each node receives from its environment a stream of jobs, stores them in a buffer and ultimately processes them using a variable number of server components. An example of an open system with three nodes is portrayed in Fig. 11. A system can also be configured as a ring.

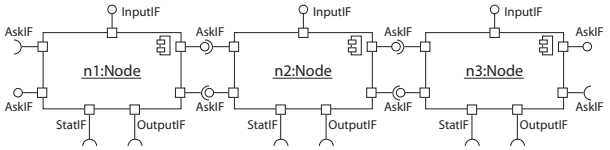

Fig. 11. A pipeline of nodes

The Node component in Fig. 11 only differs from the one in Fig. 8 because it has an AskIF interface for interacting with peer nodes. From the Node internal structure diagram depicted in Fig. 12, one can see that AskIF is delegated to the Dispatcher sub component.

A system is assumed to work with a fixed number of servers. Servers cannot be dynamically generated because they model physical computing resources. However, a high loaded node, that is a node with a pending job but without idle servers, can ask for a server to its neighbours. A node which receives a request, can reply to it with a server if one is available, otherwise ignores the request. Buffer and Server in Fig. 12 are identical to those of Fig. 8. The Dispatcher, though, is now in charge of handling the server relocation issues.

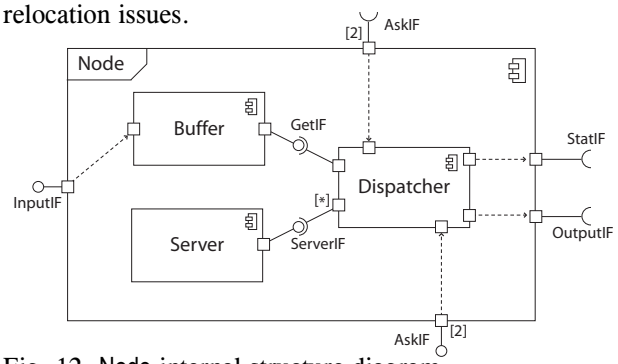

Fig. 12. Node internal structure diagram

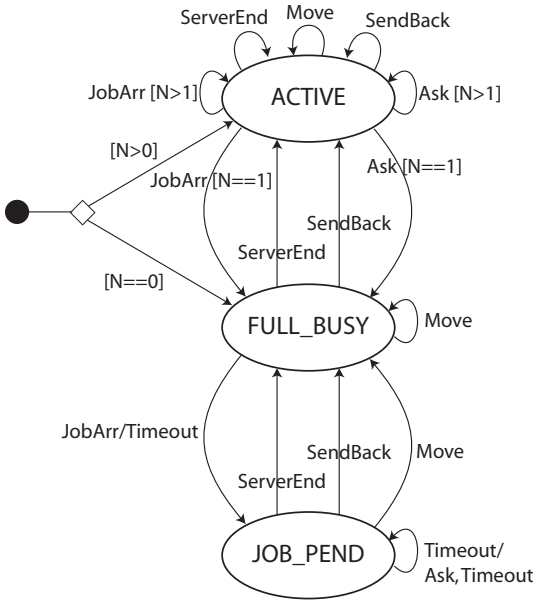

Fig. 13. Behaviour of Dispatcher

Fig. 13 shows an abstract description of Dispatcher behaviour. The actor maintains a set of available servers. Let N be the number of free servers at any moment. At start time, the dispatcher receives the intial number of assigned servers and sends a Get message to the buffer in order to achieve the first job.

Fig 14. shows in pseudo-code the dispatcher events/actions in its three possible states.

```
when ACTIVE{
   on a JobArr{
      assign job to a server and send a next Get to buffer
      if( no idle server ) become( FULL_BUSY)
   }
   on a Move{
      send a SendBack to sender with the received server
   }
   on a SendBack or ServerEnd{
      add received server to the list of idle servers
      if( ServerEnd ) send job info for stats calculation
   }
   on a Ask{
      send a Move to the requestor with a server
      if( no idle server ) becomes( FULL_BUSY )
   }
}

when FULL_BUSY{
   on a JobArr{
      send a Timeout to itself waiting for a server
      become( JOB_PEND )
   }
   on a Move{
      send a SendBack to sender with the received server
   }
   on a SendBack or ServerEnd{
      add received server to the list of idle servers
      if( ServerEnd ) send job info for stats calculation
      become( ACTIVE )
   }
}

when JOB_PEND{
   on a Timeout{
      if( valid Timeout ){
         send an Ask to neighbour nodes for a server
         send Timeout to itself waiting for a server
      }
   }
   on a SendBack or ServerEnd{
      invalidate Timeout
      assign pending job to available server
      send a Get to buffer for next job
      if( ServerEnd ) send job info for stats calculation
      become( FULL_BUSY )
   }
   on a Move{
      invalidate Timeout
      assign job pending to available server
      send a Get to buffer for next job
      become( FULL_BUSY )
   }
}
```

Fig. 14. States/Events/Actions of Dispatcher

The dispatcher can find itself in one of three states: ACTIVE (at least one server is idle), FULL_BUSY (no server is available) and JOB_PEND (waiting for a server). The dispatcher can receive a JobArr message carrying a job from the buffer, a ServerEnd message from a server which has terminated its service, an Ask message from a neighbour node which requests a server, a Move message from a neighbour node thus responding to a causal Ask request by sending a server, a SendBack message from a neighbour node which kindly returns a moved but not really useful server. The dispatcher has an own Timeout message which is sent to itself as a timeout mechanism.

When a job is pending and the dispatcher has no idle server (see state JOB_PEND in Fig. 14) the dispatcher asks neighbours and waits (using a Timeout message) a given amount of time for a server to become available. In the case a server notifies its existence before the timeout expires, the timeout is invalidated. The timeout message is re-sent at its expiration would the server missing condition persist. An invalidated timeout is simply ignored when subsequently received. This mechanism which avoids direct cancellation of a message in the message queue of the control machine, was adopted because it is more compliant with general requirements of distributed simulation.

For the purposes of simulation experiments, a second protocol for server relocation was also designed and implemented. The variation consists in the introduction of a debit concept for server movement. A node which receives a server from a neighbour, annotates the identification of the furnishing node. As soon as the dispatcher of a debitor node has no pending job but has at least one idle server, it exhausts one debit by anticipating restitution of the server to its creditor node. In the following the former protocol which freely distributes servers on-demand will be referred to as OnDemProt, whereas the second protocol based on debits will be denoted as DebtProt.

Simulation Experiments

A closed system with a variable number of nodes was configured and equally partitioned between two theatres allocated for execution on two Win platforms Pentium IV 3.4Ghz, 1GB RAM, interconnected by a 1GB Ethernet switch, using HLA/RTI (Cicirelli *et al.*, 2007b). Server relocation exploits the agent migration capability of Theatre actors.

The number of nodes was varied from one (single isolated node) to ten and the average size of buffers and the mean waiting time of jobs were measured. For the experiments, each node was fed by a similar uniform traffic of jobs (see Generator in Fig. 5). Table 1 depicts the adopted simulation parameters which refer to a single node (tu=time unit).

Figg. 15 and 16 portray respectively the average buffer size and the job mean waiting time vs. the number of nodes separately in the two cases of OnDemProt and DebtProt protocols. Each point in the figures is the mean of five runs, each lasting 10^7 time units.

Table 1. Simulation parameters per node

Job interarrival time	2-4 tu
Job size	8-15 tu
Timeout time	1 tu
Number of servers	4

As one can see, the positive effect of server relocation immediately appears as soon as the number of nodes is increased above 1. The reduction in buffer size almost stabilizes when the number of nodes goes beyond five. In reality, with OnDemProt the buffer size slightly increases when the number of nodes grows toward ten, mirroring the fact that the freely diffusion of servers in the system caused by the protocol tends to favour "selfish" nodes and to slightly penalyze "suffering" nodes. Reduction in the average buffer size obviously improves job processing, by diminishing the job mean waiting time (Fig. 16) which has definitely the same evolution of the buffer size.

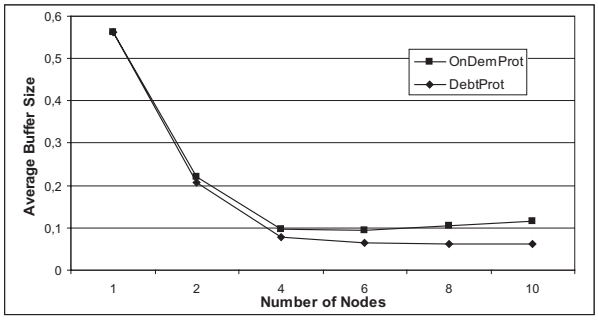

Fig. 15. Average buffer size vs. number of nodes

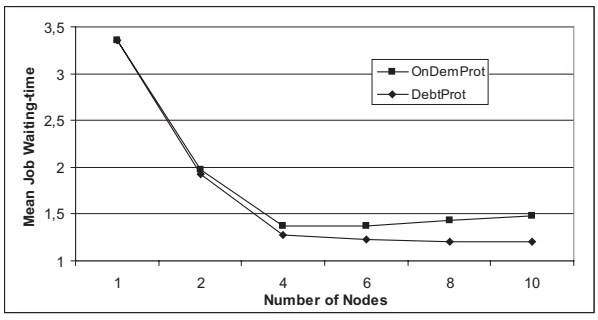

Fig. 16. Job mean waiting time vs. number of nodes

From Figg. 15 and 16 it emerges that DebtProt outperforms OnDemProt. Both buffer size and job waiting time regularly decrease and stabilize as the number of nodes is augmented. Results confirm the intuition that DebtProt tries to keep equilibrated the number of servers in each node.

CONCLUSIONS

Variable structure systems challenge for the availability of suitable modelling and simulation frameworks. This paper suggests Theatre as a concrete software architecture enabling M&S of adaptive systems. Theatre is founded on the concept of actors (agents) as the basic components. Theatre also supports composition of existing components (basic of composed) in order to facilitate construction of reusable coupled models.

A lean and efficient implementation of Theatre in Java was realized which supports both centralized and distributed simulation of complex dynamic structure systems. Theatre can work with common transport layers like Java Socket, Java RMI and recently HLA/RTI which also provides, among others, time management services. As an example, the paper reports modelling and distributed simulation of a relocation server model, under two different protocols of server movements. Prosecution of the research aims at

- experimenting with complex variable structure systems using e.g. biological or social paradigms
- using Theatre as a starting point for supporting other formalisms, e.g. PDEVS (Zeigler *et al.*, 2000). A preliminary prototype which maps PDEVS models on to actors is described in a recent paper (Cicirelli *et al.*, 2006b)
- developing graphical tools which allow visual modelling and automatic code generation of basic and coupled components.

REFERENCES

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

Brown A.W. and K.C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37-46, 1998.

Cicirelli F., A. Furfaro, L. Nigro. A distributed agent-based simulation model for large wireless sensor networks. In *Proc. of Agent Directed Simulation Symposium (ADS'06), SCS SpringSim*, pp. 115-122, 2006a.

Cicirelli F., A. Furfaro, L. Nigro. A DEVS M&S framework based on Java and actors. In *Proc. of 2nd European Modelling and Simulation Symposium (EMSS'06)*, pp. 337-342, 2006b.

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

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

DEVSJAVA Reference Guide, www.acims.arizona.edu.

Hu X., B.P. Zeigler and S. Mittal. Variable structure in DEVS component-based modelling and simulation. *Simulation*, 81(2):91-102, Feburary 2005.

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

Jang M.-W. and G. Agha. Agent framework services to reduce agent communication overhead in large-scale agent-based simulations. *Simulation Modelling Practice and Theory*, 14(6):679–694, 2006.

Posse E. and H. Vangheluwe. Kiltera: a simulation language for timed, dynamic structure systems. In *Proc. of 40th Annual Simulation Symposium*, Norfolk (VA), USA, 26-28 March, 2007.

Shaw M. and D. Garlan. *Software architecture: perspective on an emerging discipline*. Prentice-Hall, 1996.

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

AUTHORS BIOGRAPHIES

FRANCO CICIRELLI holds a PHD in computer science from the University of Calabria (Unical), DEIS. As a postdoc, he is making research on agent and service paradigms for the development of distributed systems, parallel simulation, Petri nets, distributed measurement systems. He holds a membership with ACM.

ANGELO FURFARO, PHD, is a computer science assistant professor at Unical, DEIS, teaching object-oriented programming. His research interests are centred on: multi-agent systems, modeling and analysis of time-dependent systems, Petri nets, parallel simulation, verification of real-time systems, distributed measurement systems. He is a member of ACM.

LIBERO NIGRO is a full professor of computer science at Unical, DEIS, where he teaches object-oriented programming, software engineering and real-time systems courses. He directs the Software Engineering Laboratory (www.lis.deis.unical.it). His current research interests include: software engineering of time-dependent and distributed systems, real-time systems, Petri nets, modeling and parallel simulation of complex systems, distributed measurement systems. Prof. Nigro is a member of ACM and IEEE.

FRANCESCO PUPO, PHD, is a computer science assistant professor at Unical, DEIS, teaching Java introductory programming and computer architecture courses. His research interests include: Petri nets, discrete-event simulation and real-time systems. Dr. Pupo is a member of ACM.