Control Aspects in Multiagent Systems

Franco Cicirelli and Libero Nigro

Abstract The work described in this chapter develops a control framework for modelling, analysis and execution of parallel/distributed time-dependent multi-agent systems. The goal is to clearly separate agent behaviours from crosscutting control concerns which in general are orthogonal to a specific application and transparently affect and regulate its evolution. The approach centres on a minimal computational model based on actors with asynchronous message-passing and actions. Actors are the basic building blocks for modelling the business logic of an application. Actions model activities needed by actors, which have a time duration and require specific computing resources to be executed. Action execution can be either preemptable or not preemptable. Actions are the only abstraction units which have to be reified when passing from model analysis to model implementation. Therefore, the use of actions favours model continuity, i.e., a seamless transformation from model analysis by simulation to model implementation and real time execution. Different pluggable control strategies ranging from pure concurrent to time sensitive (real-time or simulated-time) were implemented. Control strategies are compliant with agent mobility and resource availability. For demonstration purposes, the realized control framework was tailored to the JADE distributed agent infrastructure. This chapter first describes the control framework and its prototyping in JADE. Then presents two case studies. The first one is devoted to a thorough assessment of the timing behaviour and performance of a company help desk system. The second one is concerned with the schedulability analysis of a real-time tasking set. Finally, directions of on-going and future work are drawn in the conclusions.

F. Cicirelli · L. Nigro (✉)
DIMES, Universitá della Calabria, 87036 Rende (cs), Italy
e-mail: l.nigro@unical.it

F. Cicirelli
e-mail: f.cicirelli@dimes.unical.it

© Springer International Publishing Switzerland 2016

J. Kołodziej et al. (eds.), *Intelligent Agents in Data-intensive Computing*,
Studies in Big Data 14, DOI 10.1007/978-3-319-23742-8_2

1 Introduction

The runtime support of a multi-agent system normally rests on a control structure based on multi-threading and on asynchronous message-passing [1]. The internal organization of an agent hosts a hidden data status and a hidden behaviour. The behaviour is responsible of defining how the arrival of messages alters the agent data status. Typically, sent messages to an agent get stored into a mailbox owned by the destination agent. The agent control thread then extracts, one at a time, a message from the mailbox and eventually processes it. In the case the mailbox is empty, the agent thread goes to sleep awaiting the arrival of new messages.

The multi-threaded control structure of a multi-agent system is normally felt sufficient to ensure the basic abilities of agents [2], namely autonomy, proactivity, adaptivity to the surrounding perceived/acted-upon environment, sociality, mobility, and so forth.

In order to widen/tailor multi-agent systems to specific application domains, it is important to modularly adapt the basic agent control structure so as to guarantee, e.g., a time-sensitive behaviour, or the fulfilment of dependency/precedence constraints, etc. are achieved. Flexibility of control design is also advocated, for instance, in mechanism design problems [2] where a proper control strategy is needed in a group of agents competing for the allocation of scarce resources in order to coordinate the decision process.

A more general and challenging goal of this work is supporting *model continuity* [3–5], i.e., favouring a seamless transformation of a time-dependent multi-agent system model from property analysis (e.g., based on parallel/distributed simulation [6]) to real-time implementation.

An original and flexible control framework [3, 7, 8] is proposed in this chapter which makes it possible to transparently aggregate a control module chosen from a library of reusable control forms to a distributed multi-agent system, so as to ensure its dynamic evolution is influenced by a given time notion (real-time or simulated-time) and to the availability of computational resources. The proposed control framework purposely depends on a minimal computational actor model [9–11]. The actor model, though, actually used in this work is novel in that it hosts a notion of *actions* which are a key for transparently switching from simulation to real execution, and naturally map onto the available processing units managed by a control strategy. Action execution can be preemptive as required, e.g., in the support of priority-driven embedded real-time systems, or it can be non preemptive.

Actions are well-suited for modelling activities whose execution consumes time and requires computational resources not owned by actors in an exclusive way (e.g., shared CPUs in a computing system). Actions can abstract operations which need to be reified when switching from model analysis to real execution. They do not affect/trigger actor behavior, i.e., the business logic of a model remains captured in terms of message processing only. The framework hides and makes orthogonal all the aspects related to action scheduling and their dispatching on the available computational resources thus simplifying modelling activities.

The proposed control approach was prototyped in JADE [12, 13]. JADE was chosen because it is open source, it adheres to FIPA communication standards [14] which in turn favour application interoperability, it is based on Java. JADE rests on a multi-threaded agent model and on asynchronous message-passing. As a side benefit, the embedding of the actor model in JADE has the effect of simplifying the use of the JADE behavioural constructs.

It is worth noting that widespread agent-based tools and infrastructures (e.g., JADE, Repast [15, 16]) do not have built-in solutions for customizable control extensions working with a specific (real/simulated) time notion. In addition, approaches aimed to model continuity normally rest on a special case implementation of the modelling language (e.g., DEVS [5]) but in no case the modeller is able to modify the runtime support scheduler.

The chapter first introduces basic concepts of JADE. The control framework is then proposed and a description of its implementation in JADE is provided. After that a library of achieved control modules is detailed which exposes both untimed and timed (respectively based on real-time and simulated-time), concentrated (sequential) and parallel/distributed control structures. Subsequently some guidelines are given about how to use the control framework. The practical application of the approach is demonstrated by presenting two examples: (i) a complex multi-agent system modeling a help desk offering services to a company's customers. The help desk is organized into service centres which can exploit one or multiple operators. The goal is a thorough assessment of the help desk properties, i.e., timing behaviors and performance, through the use of distributed simulation; (ii) the schedulability analysis of a real-time tasking model under fixed priority scheduling. The latter model admits periodic and sporadic tasks, non deterministic execution times and precedence constraints due to the access to some shared data guarded by locks.

Finally, conclusions are presented with an indication of on-going and future work.

2 JADE Concepts

JADE [12, 13] is a known open source middleware for building Java-based distributed multi-agent systems. It conveniently hides heterogeneity aspects (e.g., hardware architectures, operating systems, etc.) of a distributed context and provides a suitable API and runtime support for developing, controlling and executing agents. Agents live in and can move among the so-called *containers* which in turn are composed into *platforms*. A platform is a particular distributed system which is established by starting a *main-container* which hosts some built-in agents which implement such fundamental services as naming/addressing, mobility, information sharing (through yellow-pages) etc., for user-defined agents. Normal containers typically join an existing main-container at their launch time. At its boot a container can be assigned some initial agents executing in the container. Agents can also be created interactively and intuitively through the services of the RMA (Remote Management Agent) GUI,

which has platform scope. Finally, agents can dynamically be created through programming, according to the application logic, by exploiting the available API.

Agents in JADE are thread-based. Agent lifecycle builds on receiving messages via a local mailbox [1] and processing them, one at a time, through a properly designed behavioural structure. Some basic and simple behaviours are available which can easily be specialized so as to meet the modeller needs. Complex behaviours (i.e., sequential, finite state machine, parallel) can also be adapted to the application requirements. Behaviour objects can be flexibly added/replaced dynamically to/in a given agent.

The communication model of JADE is founded on asynchronous messages [1] expressed using the FIPA Agent Communication Language (ACL) [14]. The content of a message can be a simple textual information or, in the general case, it can be bound to a complex serialized Java object. A JADE multi-agent system can define and use a family of ACL messages sharing a certain ontology. JADE (serialized) agents can migrate dynamically from a container to another. The available API permits to an agent to request migration (doMove() method), and to specify what to do just before migration (by overriding the beforeMove() method) and after migrated to the destination container (by adapting the afterMove() method).

The JADE agent programming model is assisted by some Java classes/interfaces like Agent, Location, AID (Agent unique IDentifier), Behaviour, ACLMessage along with associated attributes and methods. It is important to note, though, that no primitive support exists in the API about a time notion or about mechanisms useful e.g., for building a simulation model. All of this motivated the work described in this chapter aimed at making it possible to experiment in JADE with general control strategies, and in particular with model continuity [4, 5] i.e., the possibility of turning, seamlessly, a simulation model used for property analysis into a real execution model.

3 Modelling with Actors and Actions

A minimal computational model easily hosted by JADE is adopted in this work, which makes it possible to design control aspects modularly separated from the application logic but reflectively governing the evolution of a multi-agent system. Control aspects can be transparently interchanged so as to ensure, for example, a given model can smoothly be transformed from its version used for analysis purposes (based on simulation) to its implementation version used for real-time execution. Main concepts are communicating actors and schedulable actions.

Actors [9–11] are a variant of the Gul Agha model [1]. They hide some internal data variables and a behaviour (finite state automaton) for responding to messages. The communication model relies on asynchronous message-passing. An actor is a reactive entity which answers to an incoming message on the basis of its current state and the arrived message. An actor is at rest until a message arrives. Message processing is carried out in the *handler(msg)* method of the actor, which implements actor

behaviour and accomplishes data/state transitions. During a message processing, one or multiple actions can be activated (scheduled). An action encapsulates a basic timing consuming activity to be performed on behalf of the requesting actor. Action execution requires a computational resource (processing unit) for it to be carried out. Actors are mapped onto JADE agents (see also Fig. 1). Basic mechanisms like naming, setup, message-passing, migration etc. are directly borrowed from JADE. The following are the basic operations on actors:

- *newActor*, for creating a new actor
- *become*, for changing the behaviour (state) of the actor
- (non-blocking) *send* for transmitting messages to acquaintance actors (including itself for proactive behaviour). The send operation carries a message with a timestamp which specifies when the message has to be delivered to its recipient
- *do* action, for scheduling the execution of a given action. The *do* operation can specify also if the requesting actor wants to receive a completion message when the action terminates
- *abort* action, for aborting a previously scheduled action.

Every action is modelled as a black box having a set of input parameters, a set of output parameters and an execution body. Actions have no visibility to the actor internal data variables. Action termination is communicated by a message to the requesting actor. This actor then can retrieve the output parameters from the action object and update its internal data variables.

It is worth noting that the action design is lightweight and safe. It purposely avoids interference problems arising when multiple actions activated by a same actor get concurrently executed. In no case, indeed, there is the need to use a synchronization mechanism (e.g., semaphores) to protect actions from one another, simply because actions have no visibility to the actor internal data variables.

Different roles are assigned to messages and actions. Messages mainly serve to maintain sociality relationships among actors (communication). In addition, messages are a key to trigger actor behaviour (e.g., making a state transition in the finite state automaton of the receiving actor). Actions, on the other hand, capture execution concerns, i.e., tasks to be accomplished and which affect the temporal evolution of actors. Message processing is atomic: it cannot be preempted nor suspended. Action execution, instead, can possibly be suspended and subsequently resumed, as demanded by a preemptive priority-driven real-time system model (see later in this chapter). In other models (also shown in this chapter), though, actions too can be required to execute in a not preemptive way.

A collection of actors (i.e., a Logical Process or LP) is assigned to an execution locus (i.e., a JADE container) which is regulated by a control machine (CM). The control machine hides a specific control strategy which is responsible for handling sent messages and submitted actions. Action execution ultimately depends on a collection of parallel computational resources, i.e., the processing units (PUs), hosted by the CM and administrated by an action scheduler (AS) (see Fig. 1). A control machine can be in charge of managing a time notion (real-time or simulated-time) regulating actor behaviour.

4 A Framework for Control Experiments in JADE

The previously described actor model was prototyped into JADE as shown in Fig. 1. Control machines, as the actors, are mapped onto JADE agents (see Fig. 1). As a consequence, actors and control machines can interact to one another through a suitable protocol of *ACLMessages*. The control framework is founded on the following abstract classes:

- *Message.* It has fields for the involved sender/receiver actors and a timestamp information. Message is the common ancestor from which all the applicative messages derive. A message object is thinked to be embodied, in serialized form, as the object content of an *ACLMessage*.
- *Action.* Contains the submission time, two free slots for hosting respectively the input and output parameters (array of serializable Objects), the deadline, the action priority and an indication about the set of PUs to use for its execution. In the case no information is provided, the action can be executed on any PU. For an action it is possible also to express if an indicated PU is *preferred* or if it is *mandatory*. On the base of the above rules, a PU is said to be *exploitable* if it could be potentially used to execute an action. A specific flag can be set to indicate also if an action is pre-emptable or not during its execution. The abstract method *execute()* must be overridden in a concrete action class. An action object is created by an actor and (transparently) submitted to a control machine as a serialized content object of an *ACLMessage*.
- *ControlMachine.* Is the base class for application-specific control structures. Typically, a control machine repeats a basic control loop. At each iteration of the loop one message is extracted, according to a control policy, from the set of pending messages, and consigned to its target actor for its processing. At message processing termination, the activated actor replies the control machine with an *ACLMessage* containing the set of the just sent messages and the set of submitted actions of the actor. Following such a reply, the new messages are copied to the pending set whereas the submitted actions are passed to the action scheduler. The behaviour of a time-sensitive control machine can require, before the actual

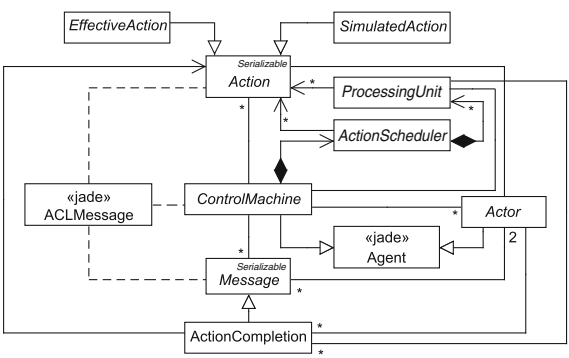


Fig. 1 Basic classes

delivery of a pending message, to first synchronize with a time server (see Fig. 2) toward the achievement of the necessary grant to proceed with the message and its timestamp.

- *ActionScheduler.* Imposes an application-specific execution policy to the actions. An action scheduler controls a set of processing units. On the basis of the adopted execution policy, a scheduler can (i) assign the action to a free processing unit, (ii) assign the action to a busy processing unit by firstly preempting the ongoing action and saving its execution status or (iii) add the action to a pending set for its subsequent execution. Preempted actions are added to the pending set too and marked as *suspended*. Of course, the number of available processing units along with the action execution policy affect model evolution both in simulation and in real-time execution.
- *ProcessingUnit.* Denotes an action executor capable of processing one action at a time. In a basic case it can coincide with an instance of a thread in a pool, which maps onto a physical core of the underlying machine hardware. The use of PUs naturally permits to take into account the computational capabilities of a multi-core architecture both during analysis (i.e., simulation) and real execution. Methods of a PU include *start*, *preempt* and *stop*, whose meaning should be self-explanatory. An *ActionCompletion* message is used to communicate that an action has terminated its execution.
- *Actor.* It is the common ancestor for applicative actors and exports all the basic operations, i.e., the send, become, do/abort action and the abstract handler methods. The hidden JADE behaviour in the Actor class is in charge of receiving an *ACLMessage* from a control machine, extracting from it the Message content (deserialized) object, and causing message processing by invoking the *handler()* method on the destination actor. At handler termination, all the newly generated messages and collected actions are sent back to the control machine as a part of an *ACLMessage*. A binding exists between an actor and the control machine belonging to the hosting JADE container. Actor migration is controlled by a redefinition of the *afterMove()* method whose responsibility is updating the binding to the local control machine of the destination container. Following a migration, an actor can receive messages originated from a previously bound control machine. In such a case the actor has to (transparently) forward these messages to the currently bound control machine so as for them to be properly managed. This provision is necessary to guarantee that message scheduling and dispatching activities, which are control sensitive, are ultimately handled by the control machine associated with the JADE container currently hosting the recipient actors.

As one can see in Fig. 1, two specializations of the *Action* base class actually exist, whose goal is to favoring model continuity. *SimulatedAction*s are used in a simulation context. *EffectiveAction*s replace the simulated ones when switching to the real execution of an actor model. It should be noted that effective actions related to a controlled physical environment, are logically part of so called *interface actors* whose responsibility is translating environment stimuli into internal messages and vice versa.

A smooth transition from analysis to implementation of an agent-based model is possible by replacing the control machine and turning any simulated action into a real one. Any other aspect of the model remains exactly the same in simulation and in real-time execution. In particular, no change is needed in the number and type of exchanged messages as well as in the application communication patterns.

5 Library of Control Forms

Figure 2 shows a prototyped library of reusable control structures whose design handles in a particular way messages and actions of an actor model. Other control strategies can be added as well. A common design principle of all the control machines in Fig. 2 concerns the actors *handler* methods which are always executed one at time in an interleaved way (co-operative concurrency) as long as the considered actors are hosted in a same container. Actions are instead executed in parallel according to the number of processing units which in turn mirrors the assumed parallelism degree of the model.

The way actions are ultimately executed is determined by the adopted implementation of an *action scheduler* (AS). Prototyped schedulers, along with the implemented kind of processing units, are summarized in Fig. 3 and detailed later in this section.

In the course of prototyping the various control machines, the following JADE problem emerged which is tied to the dynamic creation of actors/agents. Indeed, the Actor newActor() method retrieves the container controller and then invokes on it the createNewAgent() method for completing actor creation. Due to the action parallel execution model it was found necessary to synchronize the above mentioned JADE operations which in turn can degrade the parallel execution schema. A description of available control forms is provided in the next sections.

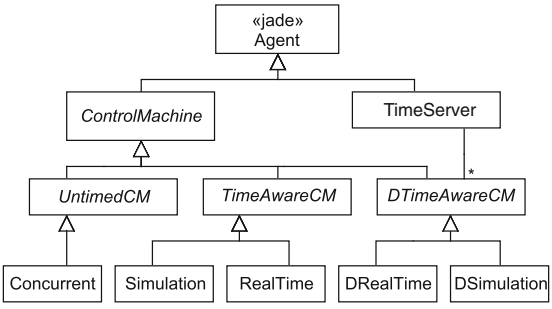


Fig. 2 Hierarchy of developed control machines

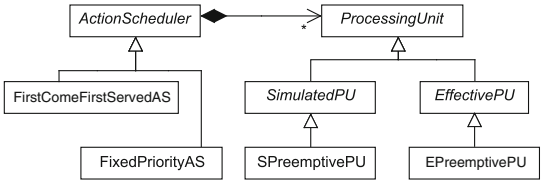


Fig. 3 Hierarchy of action schedulers

5.1 Prototyped Control Machines

Three families of control machines can be identified in Fig. 2. The `UntimedCM` family contains control structures which do not manage an explicit notion of time. These control machines can be used both in a centralized or in a parallel/distributed scenario where an actor model is partitioned among multiple JADE containers. The `TimeAwareCM` family groups time-sensitive control machines which operate in a not distributed context, that is the actor model cannot be partitioned into multiple JADE containers. The `DTimeAwareCM` control machines, however, provide their services to time-dependent models allocated for execution on a parallel/distributed scenario. In this case, a `TimeServer` has to be used to ensure a coherent time notion among all the participating control machines is ultimately met.

`Concurrent` realizes an untimed parallel control structure which rests on a FIFO message queue (MQ) as the message pending set and on a certain number of processing units corresponding to the parallelism degree of the model. The control machine can work with either `SimulatedActions` or `EffectiveActions`. Therefore, it is sufficient to replace the nature of actions for switching from simulation to real execution. The `Concurrent` prototype enables an actor system to terminate when an explicit application-level END control message is received by all the involved control machines.

`Simulation` implements a classical discrete-event simulation schema driven by a simulated time notion. Messages are tagged with an absolute timestamp and are buffered into a time ranked queue (TQ) where the head message holds the (or is one message with) minimum timestamp. At each iteration of the `Simulation` control loop, the most imminent (head) message is extracted from TQ, its timestamp assigned to the virtual time clock, then the message is dispatched to its receiver actor. Of course, the control machine expects to work only with `SimulatedActions`. A simulated action carries the time duration of the associated activity. At its submission, a simulated action is assigned to an exploitable PU (if there are any) which in this case simply means that its action completion message is scheduled onto TQ with a timestamp which is the sum of the action duration and the value of the current time. Since the execution of an action is only simulated, the processing units are actually fake objects without threads. It is worth noting that, although in a simulation context, the execute() method of a submitted action can still be useful. Its invocation notifies the action about its execution. All of this can be exploited, e.g., for gathering statistical data or to purposely generate output parameters which will be furnished

to the actor that submitted the action. The use of `Simulation` is conveniently assisted by a package (actor.distributions) of common density distribution functions (including uniform, exponential, hyper exponential, Erlang, normal etc.) which are based on the java.util.Random pseudo-random number generators. The behavior of `Simulation` terminates when the virtual time goes beyond the adopted simulation time limit.

`Realtime` is another time-sensitive control machine with a real time notion built on top of the `System.currentTimeMillis()` Java service. `Realtime` is useful for non hard real-time applications. Messages can be or not time-constrained, in the sense that they can specify their occurrence (or firing) time. Not time-constrained messages (created without an explicit timestamp) are assumed to be processed in FIFO order and when there are no fired time-constrained messages. Time-constrained messages must be dispatched as soon as the current time exceeds their firing time. `Realtime` uses a configurable time tolerance *EPS*, so that a time-constrained message which should occur at absolute time t, is considered to be still in time if the current time is less than or equal to $t+EPS$. The quality with which timing constraints are eventually satisfied is assessed in the so called *preliminary execution* of the model (see later in this chapter). Two message buffers are used: MQ as in `Concurrent`, and (time ranked) TQ similar to that used by `Simulation`. `Realtime` control loop is never-ending. When no messages are found in MQ and current time is lesser than the most imminent message in TQ, the control structure simply awaits current time to advance to the firetime of the first message in TQ. The control machine is developed to work with effective actions only. Effective actions are ultimately executed on Java threads abstracted by effective processing units (see Fig. 3) so as to naturally exploit the underlying hardware multi-core architecture. The message notifying the action completion will be scheduled as soon as the action execution completes.

An entire actor model can be partitioned into multiple JADE containers each supplied with a control machine. Containers can run on different cores of a same CPU or they can be allocated to distinct processors of a distributed system. When the multi-agent system is time sensitive, a time server has to be adopted for ensuring a global time notion (simulated-time or real-time).

`DSimulation` (see Fig. 2) is an example of control machine for making distributed simulation experiments. It differs from `Simulation` because the time advancement mechanism is now negotiated (i.e., a conservative synchronization algorithm [6] is adopted) by the various control machines, with a specialization of the `TimeServer`. Before processing the next timed message whose timestamp is greater than the local simulation time, a control machine asks the time server for a grant to advance to the next time. The time server collects all the time advancement proposals and the minimum of those proposals is furnished as time-grant to control machines which demanded to advance to this time. A subtle point for the time server is that it can actually generate the grant provided no in-transit messages exist in the system. Towards this a distinct counter about the number of sent and received messages [6] related to each actor/agent, are kept by the control machines and transmitted as accompanying information to the proposal messages sent to the

time server. These fine-grain counters, instead of coarse-grain counters at the level of the control machines (LPs), are necessary for correctly managing actor migrations. As a consequence of migration, indeed, a given actor can dynamically be handled by different control machines.

The prototyped `DRealTime` control structure is similar to `RealTime`. However, a common time reference among the various control machines is now assumed. Global time, e.g., based on UTC [17] or GPS [18], is kept by the time server. Other versions of the control form can be designed, e.g., based on local clock synchronization at runtime.

5.2 Action Schedulers

Prototyped schedulers (see Fig. 3) immediately put into execution a newly scheduled action on a idle *exploitable* PU (if there are any), otherwise, different scheduling strategies can be adopted. In the case no such idle PUs exist, the scheduler `FirstComeFirstServerAS` organizes actions in a pending list. This list is ranked according to the scheduling (arrival) time of actions. Each time a PU becomes idle, the pending list is iterated and the first action for which the PU is exploitable is removed from the list and assigned to the PU. The PU remains idle in the case it is not exploitable by any of the actions in the list. The `FixedPriorityAS` scheduler uses an action priority to keep ordered the pending list. Action execution is priority driven and preemptive. The duration of a preempted action is shortened by the time the action was running. It is worth noticing that switching from simulation to real execution, implies only a redefinition of PUs.

For simulation purposes, the `SPreemptivePUs` can be used which are passive objects without internal threads. Assigning an action to a `SPreemptivePUs` implies an action completion message with an absolute timestamp achieved by adding the current time to the action duration is scheduled. This provision (i) permits the virtual time to grow accordingly to the time needed to simulate the execution of the action, (ii) allows the scheduler to be informed that a previously busy processing unit is now ready to be used again, (iii) notifies action completion to the submitter actor in the case it expressed the willingness to receive such notification. In the case an action is preempted, the related action completion message is simply descheduled (see the association between `ProcessingUnit` and `ControlMachine` in Fig. 2).

The `EPreemptivePUs` are instead active objects, i.e., thread-based objects, allowing to emulate the real action execution through busy-waiting. Each preemptive PU manages a pool of Java threads where only one thread at time can be running thus ensuring a unitary degree of parallelism within the PU. The use of multiple threads allows a running action in a PU to be preempted, replaced by another running action and then resumed later. With an `EPreemptivePU` the action completion message is scheduled at the end of the action execution.

6 Development Lifecycle

A key factor of the prototyped control framework is its support to a development lifecycle which is made up of the following phases: *modelling*, based on an exploitation of the actor metaphor; *property analysis*, centred on discrete-event simulation; *preliminary execution*, aimed at estimating if the timing constraints can be satisfied in real-time execution and, finally, *real execution*. In the previous sections it has been described that the basic abstractions for building a model are actors, messages, actions and processing units. The main goal of actions is the modelling of activities whose execution both consumes time and depends on the availability of suitable computational resources. In addition, actions can abstract tasks which need to be reified for properly switching from model analysis to model real execution. From the actor viewpoint, message processing is assumed to be instantaneous, i.e., it requires a negligible amount of computational resources. Messages serve the purpose of triggering behavior evolution of actors. An important consequence is that the business logic captured by message processing and actor behavior remains unmodified when moving from simulation to real execution.

Property analysis by simulation of a chosen model is directly influenced by the number of computational resources existing in the real execution environment. These resources define the action parallelism degree. The so called preliminary execution phase uses an hybrid of simulation and real time execution. This is because the control machine is a real time one but actions are not the simulated ones, nor the final effective ones, but they are pure resource consuming ones, that is, actions require both time and a processing unit. The goal of preliminary execution is to assess if real-time constraints, previously checked in simulation, can be fulfilled by real time execution, which in turn means the message processing overhead is effectively negligible. In the case message processing is not negligible, some timing constraints are to be relaxed, e.g., the application time tolerance factor can be increased and/or the model can be revised/optimized.

Computational resources can in general be shared among actors and used to execute actions in different ways. The proposed framework allows a fine-grained control on computational resources assignment. First of all, processing units are assigned to a control machine. Then an action can be configured to execute either on a specific PU, on a set of PUs or on any available PU. Since a specific control machine is required for each used JADE container, model partitioning can also be guided by the assignment of resources to actors. The number of PUs allocated to a control machine can be also unbounded. This is the hypothesis of *maximum parallelism*. In such a case, an idle PU is always considered available for any newly submitted action. This assumption can be used during simulation but it is not suitable for preliminary execution.

It should be noted that after an action is submitted for execution, an actor remains able to receive and process other incoming messages. Multiple actions can be submitted without waiting for the completion message of a previously submitted action. Obviously, if the way of processing actions is not appropriate for a specific applica-

tion domain, other kinds of control machines could be developed and transparently used. When the modeller is more interested to property assessment of his/her system through simulation than to model continuity, the use of actions can still be recommended for improving speedup by spawning in parallel those activities which can be carried out at a same time.

As a final remark, it is useful to observe that the modeller can, in principle, avoid an explicit recourse to object actions. In this scenario, the corresponding activities can be in-line coded in the handler() method or the effect of actions (e.g., time advancement) can be equivalently reproduced by messages.

7 Modelling and Analysis Examples

The use of the control framework is demonstrated by means of two case studies. The first example is devoted to modelling, analysis and preliminary execution of a help desk system, the second one is related to the schedulability analysis of a real-time tasking set designed to run under priority preemptive scheduling over a parallel/distributed architecture. In both cases, the analysis is carried out by using distributed simulation. The goal is to highlight the flexibility and the effectiveness of the control infrastructure in modelling and analyzing complex time-dependent agent based systems.

7.1 Modelling and Analysis of a Help Desk System

A help desk (HD) is aimed at providing the customer with information and support related to a company's or institution's products and services. The goal is to troubleshoot problems or furnishing guidance about various kind of products like computers, electronic equipment, food, apparel, or software. Corporations can provide HD support to their customers in various ways such as by means of toll-free numbers, websites, instant messaging or email. In the considered scenario, a certain number of customers are supposing to contact a help desk by phone.

The HD is composed of a single point of contact, i.e., a call center which constitutes the first line of support, which gathers all the requests. For simple questions, e.g., about ongoing promotions and/or general services offered by the company and/or well known problems, the call center is directly able to meet the needs of the customer. If the issue is not resolved, the call center routes the request to a more appropriate and specific center of support, i.e., a second line of support. The second line of support comprises three centers specialized respectively to solve billing problems, technical problems and to offer specific and detailed description of corporation products and services. After a customer has been served by the second line of support, he/she is redirected to the call center in order to evaluate the degree of customer satisfaction or in order to re-route the customer again to the second line of support in the case

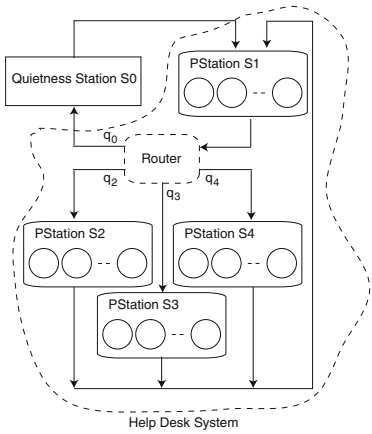


Fig. 4 A help desk model

the original problem remained unsolved. Customers are assumed to have the same priority.

The considered HD system (see Fig. 4) is modelled as a closed queue network. It is based on K recirculating customers and it is composed of a quietness station $S0$ and four service stations $S1$, $S2$, $S3$ and $S4$. The service stations along with the router $Router$ constitute the HD system whereas $S0$ hosts the clients which are not using the HD. Initially the K clients are injected into the quietness station where they stay a certain amount of time before re-asking issues to the HD system. The time a customer stay in $S0$ is determined by using an exponential distribution. A client enters the HD system by arriving at the station $S1$ (i.e., the call center) whose service time is exponentially distributed too. After the service in $S1$, the customer, with certain probabilities $q0$, $q2$, $q3$, $q4$, can be routed in input to $S0$ (i.e., he/she exits from the HD), or to one of the service stations $S2$, $S3$ or $S4$ which respectively provide billing service support, description support about specific company's services/products, and technical assistance support. Each router output is supposed to be affected by a uniform distributed communication delay. $S2$ has an exponentially distributed service time. Station $S3$ has a second order hyper-exponential distribution. Station $S4$, finally, has an Erlang distribution composed of n identically distributed exponential with the same rate. A customer exiting from $S2$, $S3$ or $S4$ comes back into input to $S1$. It is worth noting that when a customer enters $S0$ it actually exits the HD system and is annotated with the exit time. Similarly, when a customer exits $S0$, i.e., it enters the HD system, it is time-stamped with its enter time. This way, passage through the $S0$ permits to infer the HD system timing behaviour.

It is assumed that the company has $K = 300$ customers and that the HD is operating 24 h/day. Moreover, a customer uses the HD, on the average, one time per 24 h. A telephone call with the call center lasts 5 min on the average, 8 min is the average time needed to serve a customer in $S2$ and $S3$, 15 min is the average service time for $S4$. The call center is able to resolve a problem of a customer in the 70 % of cases. A customer asks for station $S2$, $S3$ and $S4$ respectively in the 5 %, 15 %

Table 1 HD parameter values

Entity	Type	Values
Station S0	Exp	$mu0 = 1.154^{-5}$ s^{-1}
Station S1	Exp	$mu1 = 3.333^{-3}$ s^{-1}
Station S2	Exp	$mu2 = 2.083^{-3}$ s^{-1}
Station S3	Hyper-Exp	$mu31 = 1.667^{-3}$ s^{-1}, $mu32 = 1.333^{-3}$ s^{-1}, a31 = 0.6, a32 = 0.4
Station S4	Erlang	$n = 3$, $mu4 = 3.333^{-3}$ s^{-1}
Router		q0 = 0.70, q2 = 0.05, q3 = 0.15, q4 = 0.1, delay$\in$ [5, 10]s
#Customers		300

and 10 % of the cases. The parameter values of the HD system are collected into Table 1. The second order hyper-exp is characterized by the rate of each exponential component ($mu31$, $mu32$), and the probability for choosing one distribution or the other ($a31$, $a32$). This mirror the fact that $S2$ can provide the description of two different kinds of services each of them requiring a specific time to be described. In a more complex scenario, more than two services can exist. The Erlang distribution is used to reproduce the fact that work of $S4$ is divided into sequential steps. In this case 3 steps are considered, i.e., $n = 3$, each of them having a service rate $mu4$. The overall goal of the analysis is to evaluate the timing behavior of the HD system when the number of employes in the service stations is varied.

7.1.1 Agent Model for the HD System

Figure 5 shows a class diagram of the realized HD model. `AbstractStation` defines a generic component and introduces the basic `Arrival` and `Departure` messages mirroring the arrival of a customer in a station and its subsequent departure, i.e., the end of a service. `PStation`, `QuietnessStation` and `Router` are concrete heirs of `AbstractStation`. `QuietnessStation` and `Router` do not use actions because they do not need to model activities that require to be reified when switching from analysis to real execution. `PStation` is used to model processing stations which provide a service to customers. As a consequence, in these cases, the use of actions is required. For demonstration purposes of the features of the achieved control framework, `PStation` does not introduce any buffering for customers awaiting to be served. As soon as a new customer arrives, a new action is created and immediately scheduled. In this a way, buffering, dispatching and execution of actions become a responsibility implicitly transferred to the control framework, thus simplifying the modelling activities. Moreover, the analysis of the timing behavior of the system can be carried out by simply changing model

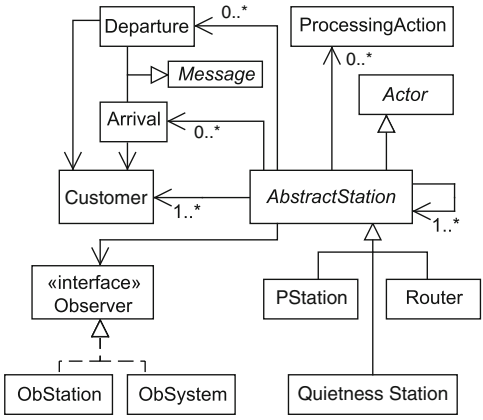


Fig. 5 HD model class diagram

configuration, i.e., by specifying for each station the number of exploitable processing units. Stations have a next station attribute which specifies where a processed customer should be re-directed. `Router` has an array of next stations each one paired with a probability value. Internally to each station, a non-agent `Observer` object is used for monitoring the occurrence of arrival/departure events. Two different kinds of observers are used for monitoring respectively a service station (i.e., $S1$, $S2$, $S3$ or $S4$) or the entire system through the perspective of the quietness station ($S0$).

7.1.2 Property Analysis of the HD System

The timing behavior of the HD system was assessed by parallel/distributed simulation. The model was split in two JADE containers. The call center was allocated on one container, whereas the remaining components of the model were allocated on the second container. The `DSimulation` control machine and the `FirstComeFirstServedAS` action scheduler together with `SPreemptivePUs` as processing units were used to set up the control framework. Each simulation experiment was executed with a time limit of 2×10^6 time units which guarantees, as confirmed experimentally, the average service time of each station is eventually met. Experiments were carried out on a Win 7, 12 GB, Intel Core i7, 3.50 GHz, 4 cores with hyper-threading.

A preliminary simulation was conducted under the assumption of maximum parallelism (unbounded number of operators) in all the operating stations. It emerged an utilization factor for the stations as follows: $\mathrm{u}(S1) = 1.492$, $\mathrm{u}(S2) = 0.123$, $\mathrm{u}(S3) = 0.348$, $\mathrm{u}(S4) = 0.457$, with a maximum number of PUs simultaneously used in $S1$ of 9. As a consequence, for a deeply exploration of system performance, subsequent simulation experiments were carried out by using in $S1$ a variable number of operators (PUs) in the range from 1 to 9, while keeping just 1 operator into each other operating stations $S2$, $S3$ and $S4$. Obviously, it could happen that, despite the call

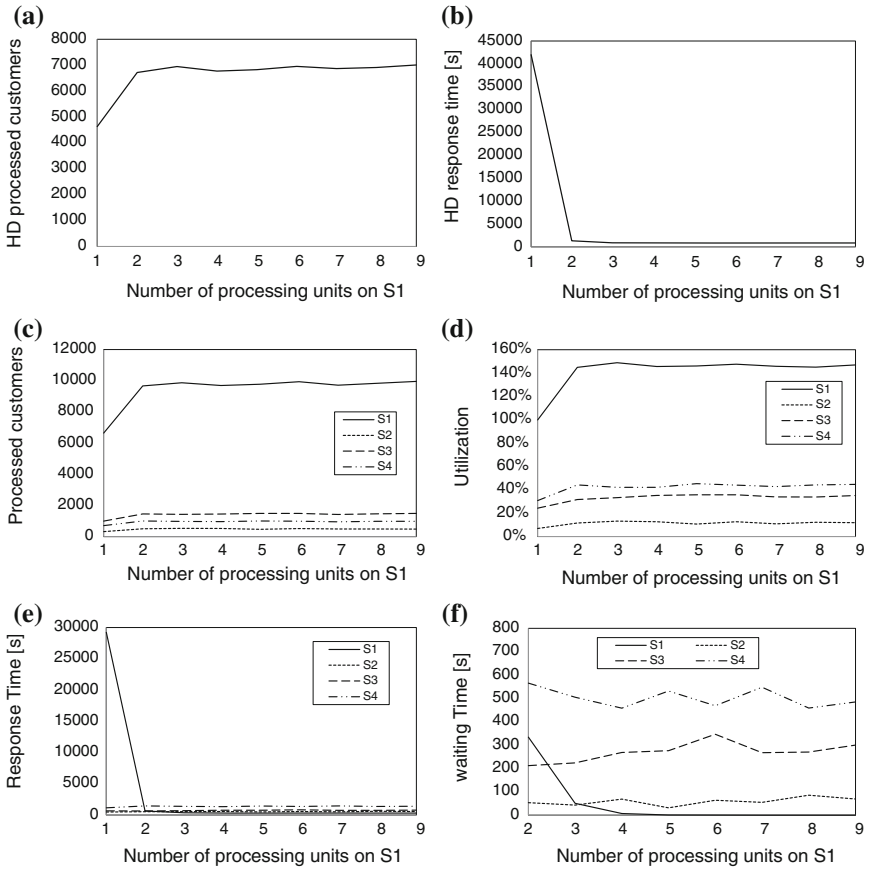


Fig. 6 Simulation results of the HD system versus the number of operators in the call center

center ability to quickly process the incoming customers, other stations may become a bottleneck thus impairing the achievable performance of the whole system. Gathered simulation results are shown in Fig. 6.

From Fig. 6a, b it emerged that with 2 operators assigned to the call center the number of served customers saturates, and the response time (i.e., waiting time plus service time) reduces and also saturates. By cross referencing data in Fig. 6c, d it is possible to infer that the call center is effectively capable of exploiting an increasing number of the available operators because the number of processed customers increases and the utilization goes beyond 100 %. However, from these figures too, it seems in reality that it is not useful to have a number of operators greater than 2. The behavior of the other stations, indeed, stays almost the same. Moreover, Fig. 6d makes it clear that the utilization of $S2$, $S3$ and $S4$ keeps well under the 40 % in all the cases. All of this indicates that the second line of support is not fully utilized and that no bottleneck exists in the system.

Table 2 Configurations of the second line of support

Name	Description	Rationale behind the configuration
a	3 PUs shared among $S2$, $S3$, $S4$	This way, resources may be exploited in a more flexible way. A temporarily idle PU can be used by another station
b	1 PU for $S2$, 1 PU for $S3$, 2 PUs for $S4$	Station $S4$ exhibits the highest waiting time. A further dedicated PU to $S4$ might help increasing its performance
c	1 PU shared between $S2$ and $S3$, 2 PUs for $S4$	Station $S4$ exhibits the highest utilization whereas utilization of $S2$ and $S3$ is low. A single PU may be shared by $S2$ and $S3$ thus freeing computational resources for $S4$
d	1 PU for $S2$, 2 PUs for $S3$, 2 PUs for $S4$	Station $S3$ and $S4$ have a high waiting time. A further dedicated PU to $S3$ and $S4$ may help increasing their performance

Figure 6e, f show the response times of the stations and the waiting times of the stations. From these figures it results that the call center, as one expects, has a diminishing waiting time as the number of operators increases. The waiting time is almost zero when the number of the operators becomes 6, i.e., with 6 operators the behavior of the system is practically equivalent to the behavior of the system under the hypothesis of maximum parallelism. However, with a value of 4 operators the waiting time is about 6 s which can be considered negligible from a practical point of view. In the light of the above discussion, it follows that the number of operators which virtually optimizes system behavior is 4.

A further optimization was looked for with the aim of reducing the waiting time of the second line of support (see Fig. 6f). Four different configurations were considered by varying the number and the way PUs are assigned to $S2$, $S3$ and $S4$. Such configurations are described in Table 2. In all the cases, station $S1$ is always assigned 4 PUs.

Figure 7 shows the simulation results corresponding to the 4 configurations of Table 2. As one can see, the configuration a is the best one because it minimizes the waiting time. In addition, it does not require further operators in the second line of support (see Fig. 7a). It should be noted that in all the considered configurations, the number of processed customers remains almost unchanged (see Fig. 7b).

7.1.3 Preliminary Execution of the HD System

After simulation, the HD model was tested for a preliminary execution in real time, by using 4 processing units for $S0$ and three processing units shared among $S2$, $S3$ and $S4$. Model partitioning is the same as used for the simulation experiments. The `DRealTime` control machine and the action scheduler `FirstComeFirstServedAS` were involved to set up the control framework. The execution time limit

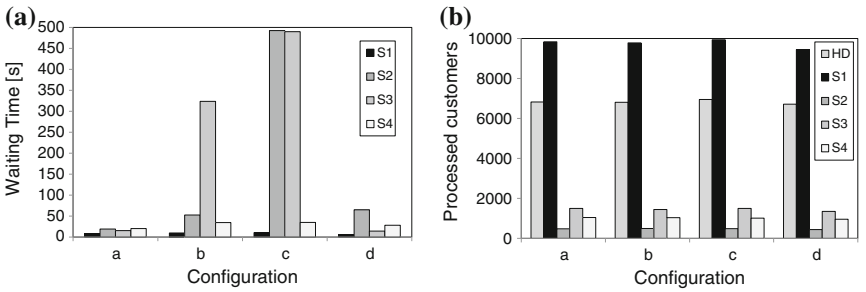


Fig. 7 Performance of the second line of support versus configurations

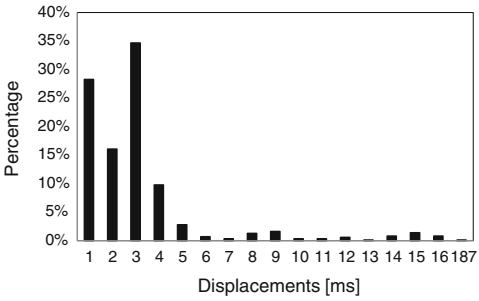


Fig. 8 Preliminary execution for the HD system: percentage of the measured time deviations

was set to 86400 s in order to execute 24 h of service and the time tolerance `EPS` was set to 500 ms. The simulated actions were changed into "effective actions" that simulate services through busy-waiting by adopting the `EPreemptivePU` concrete class for the PU.

The goal of preliminary execution was to assess the quality of fulfillment of the timing constraints, by checking the amount of deviation by which actions and time-constrained messages are executed after their due time. It was found that the maximum time deviation in the two containers is 187 ms, which occurs in the container holding all the stations except *S*1. Such value was registered just once and at system startup. After that, the measured deviations fall within the interval 1–16 ms and are distributed as shown in Fig. 8.

7.2 Schedulability Analysis of a Real-Time Tasking Set

This example focuses on modelling and temporal analysis of embedded real-time systems with timing constraints, executed on top of a parallel/distributed context [7, 19]. The modelling phase, similarly to Preemptive Time Petri Nets (PTPN) [20, 21], allows one to specify the control flow of each real-time task (or process). In particular the modeller can specify if the task is to be activated periodically or

sporadically, if it competes for the access to shared data guarded by locks, if it interacts with other tasks by message-passing. In addition, the individual computational steps in which a task body is articulated have timing constraints in the form of non deterministic execution times, and scheduling parameters such as a fixed priority, deadline, and specific processing unit to be used for the execution. Exhaustive verification of such systems is known to be undecidable in the general case. In the Oris tool [22], which supports PTPN and Fixed Priority (FP) scheduling only, the analysis of a model is assisted by a posteriori phase of cancellation of false behaviours in the enumerated state classes of a model. In [23] PTPN were mapped onto UPPAAL [24, 25] for model checking. As in [25], the approach permits either FP or Earliest Deadline First (EDF) scheduling. However, the use of stopwatches, necessary to properly implement task preemptions, forces model checking to depend on over-approximation [25] in the generation of the state graph zones. As a consequence, some properties can be checked but only with some uncertainty. In the following, a task system analysis is based on simulation. As a consequence, temporal analysis can show a deadline miss in a multi-core based model, but obviously it cannot guarantee deadlines are always met. Nevertheless, the approach is of practical value in that it allows to flexibly adapt the scheduling algorithm, and to check system behaviour under general conditions thus achieving a good confidence level about the system properties.

The chosen tasking set (see Fig. 9) model is made up of two periodic processes ($P1$ and $P2$ with period T_{P1} and T_{P2} respectively) and a sporadic one ($P3^s$ with minimal interdistance between two consecutive occurrences of the triggering event being T_{P3^s}) all having non-deterministic execution times. The three tasks are supposed to be ready at time 0, i.e., the first task instances (jobs) arrive at time 0. In addition, the relative deadlines coincide with task periods. Process $P1$ has the highest priority (i.e., 3) whereas process $P3^s$ has the lowest one (i.e., 1). An intermediate priority is assigned to process $P2$ (i.e., 2). Mutual exclusion, based on a *mutex* semaphore,

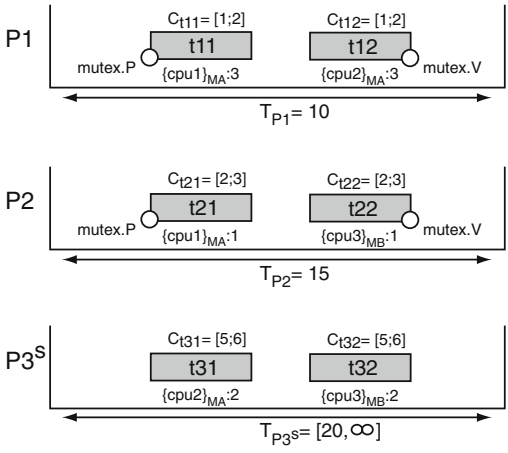


Fig. 9 A task set model

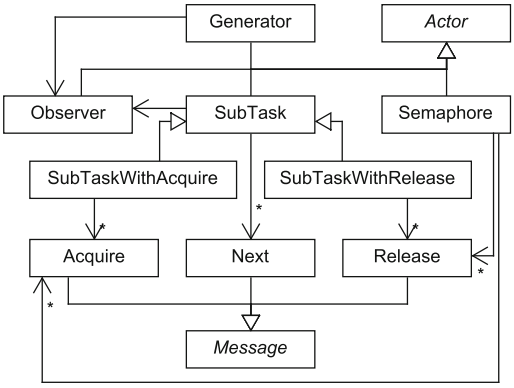


Fig. 10 Task set class diagram

is required between processes $P1$ and $P2$ to regulate access to some shared data. Each process task is split in two sub-tasks t_{xy} each allocated to a different CPU. The computation time of a sub-task is denoted in Fig. 9 by C_{txy}. Three CPUs, namely $cpu1$, $cpu2$ and $cpu3$, are used for supporting the task set execution. More specifically, $cpu1$ and $cpu2$ are supposed to be hosted by a same multi-core machine *MA* whereas $cpu3$ resides on a dedicated computer *MB*. It is worth noting that the analysis of one such tasking model is not covered by the classical scheduling theory. In addition, the use of a multi-processor context opens, in general, to possible scheduling anomalies [20, 23, 26]. The case study, though, is intended to address basic problems and to highlight the achieved programming style.

7.2.1 Agent Model for the Task Set

The following kinds of agents were developed (see Fig. 10). A *Generator* agent is in charge of creating the task instances on the basis of task periods. A *SubTask* agent models sub-tasks. The acquaintance relationship among sub-task agents mirrors the precedence schema of the task model. Coordination among sub-task executions is achieved by exchanging *Next* messages. One such a message informs a *SubTask* agent that the previous sub-task completed thence its sub-task can be scheduled. Each *SubTask* agent models its assigned computational step as an action. A *Semaphore* agent is introduced to manage mutual exclusion. The *Acquire* and *Release* messages are used to negotiate semaphore acquisition and its subsequent release. Two specializations of sub-task agents are implemented, namely the *SubTaskWithAcquire* and *SubTaskWithRelease*, which model respectively the case a sub-task requires to acquire/release the semaphore. Finally, an *Observer* agent in used to gather data about the start-time and completion-time of any task instance in order to evaluate the maximum and minimum response times of tasks. When an instance of a given task begins before the completion of the previous one, the observer notifies the task model is not schedulable.

The following reports the *handler* method (behaviour) of the *SubTaskWithRelease* agent, which confirms simplicity of the resultant programming style. As one can see, the code completely hides all the issues related to the scheduling policy, execution, preemption etc. of actions.

```
public void handler(Message m) {
  if (m instanceof Next) {
    double subTaskDuration = random.nextSample(minDuration,maxDuration);
    MyAction subTask = new MyAction(subTaskDuration, cpu, priority);
    do(subTask, true);
  } else if (m instanceof ActionCompletion) {
    Observer.End end = new Observer.End(observerAID, subTaskName);
    end.setTimestamp(now());
    send(end);
    Mutex.Release release = new Mutex.Release(semaphoreAID);
    release.setTimestamp(now());
    send(release);
    if (nextAgentExists){
      Next next = new Next(nextAgentAID);
      next.setTimestamp(now());
      send(next);
    }
  }
}//handler
```

7.2.2 Property Analysis of the HD System

The model was partitioned in two JADE containers, one simulating the machine *MA* and the other simulating *MB*. The model was configured by using `DSimulation` for the control machines, `FixedPriorityAS` for the action schedulers and `SPreemptivePU` for the processing units. Two PUs were assigned to the container simulating *MA* and one PU to the container simulating *MB*.

Simulation experiments were carried out using a time limit of 10^6, on a Win 7, 12 GB, Intel Core i7, 3.50 GHz, 4 cores workstation. It emerged that the original model in Fig. 9 is not schedulable due to a priority inversion problem occurring for the task $P1$ which misses its deadline in the case the sub-task $t22$ is executing but gets preempted by the sporadic task $P3^s$. By raising the priority of $t22$ to 3 (which is the priority of $P1$), i.e., by (partly) emulating a priority ceiling protocol, the task model appears to be schedulable and the estimated response times (after five runs) of $P1$, $P2$ and $P3^s$ were found to be respectively [2.002; 9.907], [4.006; 9.875] and [10.014; 16.815].

8 Conclusions

This chapter proposes a control framework in JADE which makes it possible to develop application-specific control strategies, e.g., time-dependent, regulating the evolution of multi-agent systems (MASs) in a parallel/distributed context. Key factors

of the approach are (a) a realization of the control structures modularly separated from the business logic of the multi-agent system, (b) a support of a development lifecycle where modelling, based on actors and actions, can be uniformly exploited both for property analysis through distributed simulation, and for implementation and real-time execution. This important feature, namely model continuity, relies on the possibility of adapting action concretizations together with a replacement of the control structure, but keeping unaltered the agent behaviours and the exchange of messages. Flexibility and practical usefulness of the proposed control framework were demonstrated by two non trivial modelling and analysis examples.

Prosecution of the research is aimed to:

- modelling actions by Java 8 lambda expressions;
- optimizing/extending the library of control forms, e.g., by providing other real-time schedulers such as Earliest Deadline First (EDF);
- experimenting with the use of the control approach in the analysis and implementation of complex agent-based models, e.g., time-constrained workflow modelling, analysis and enactment, virtual environments etc.;
- supporting adversary simulators [27] for large sporadic task models, to evaluate global fixed-priority over multiprocessors;
- implementing the approach directly in Java (without JADE) to improve its performance and porting it to Real Time Java;
- exploting the proposed multi-agent approach in the context of Big Data applications (e.g., [28–30]). A key potential is concerned with massive parallel execution of actions devoted to the management of ever-increasing amount of information.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Wooldridge, M.: An Introduction to Multi-agent Systems, 2nd edn. Wiley (2009)
3. Cicirelli, F., Nigro, L.: A control framework for model continuity in JADE. In: Proceedings of IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications, pp. 97–104. Toulouse, France (2014)
4. Hu, X., Zeigler, B.P.: Model continuity in the design of dynamic distributed real-time systems. IEEE Trans. Syst. Man Cybern. Part A **6**(35), 867–878 (2005)
5. Mittal, S., Martin, J.L.R.: Netcentric System of Systems Engineering with DEVS Unified Process. CRC Press (2013)
6. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley (2000)
7. Cicirelli, F., Nigro, L.: Modelling and Analysis of Parallel/Distributed Time-dependent Systems: An Approach based on JADE. In: Proceedings of 7th International Conference on Internet and Distributed Computing Systems (IDCS 2014), LNCS 8729, pp. 204–214. Springer (2014)
8. Cicirelli, F., Nigro, L., Pupo, F.: Agent-based control framework in JADE. In: Proceedings of 28th European Conference on Modelling and simulation, pp. 25–31. Brescia, Italy (2014)
9. Cicirelli, F., Furfaro, A., Nigro, L.: An agent infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination. SIMULATION Trans. SCS **85**(1), 17–32 (2009)

10. Cicirelli, F., Furfaro, A., Nigro, L.: Modelling and simulation of complex manufacturing systems using statechart-based actors. Simul. Model. Pract. Theory **19**(2), 685–703 (2011)
11. Cicirelli, F., Giordano, A., Nigro, L.: Efficient environment management for distributed simulation of large-scale situated multi-agent systems. Pract. Experience Concurrency Comput. (2014). doi:10.1002/cpe.3254
12. Jade. http://jade.tilab.com. Accessed on June 2014
13. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-agent Systems with JADE. Wiley (2007)
14. Foundation for Intelligent Physical Agents FIPA. http://www.fipa.org. Accessed on June 2014
15. North, M.J., Macal, C.M.: ManaginG Business Complexity: Discovering solutions with Agent-Based Modeling and Simulation. Oxford University Press (2007)
16. RePast. http://repast.sourceforge.net. Accessed on June 2014
17. Schmid, U.: Synchronized UTC for distributed real-time systems. Ann. Rev. Autom. Program. **18**, 101–107 (1994)
18. Leick, A.: GPS Satellite Surveying. Wiley (2004)
19. Brekling, A.W., Hansen, M.R., Madsen, J.: Models and formal verifications of multiprocessor system-on-chips. J. Log. Algebr. Program. **77**(1–2), 1–19 (2008)
20. Bucci, G., Fedeli, A., Sassoli, L., Vicario, E.: Timed state space analysis of real-time preemptive systems. IEEE Trans. Softw. Eng. **30**(2), 97–111 (2004)
21. Carnevali, L., Ridi, L., Vicario, E.: Putting preemptive time petri nets to work in a V-model SW lifecycle. IEEE Trans. Softw. Eng. **37**(6), 826–844 (2011)
22. Bucci, G., Carnevali, L., Ridi, L., Vicario, E.: Oris: a tool for modeling, verification and evaluation of real-time systems. Int. J. Softw. Tools Technol Trans. **12**(5), 391–403 (2010)
23. Cicirelli, F., Angelo, F., Nigro, L., Pupo, F.: Development of a schedulability analysis framework based on PTPN and UPPAAL with stopwatches. In Proceedings of IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications, pp. 57–64, Dublin, Ireland (2012)
24. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems, LNCS 3185, pp. 200–236. Springer (2004)
25. David, A., Illum, J., Larsen, K.G., Skou, A.: Model-based framework for schedulability analysis using UPPAAL 4.1. In: Model-Based Design for Embedded Systems, Chap. 3, pp. 93–120. CRC Press (2009)
26. Andersson, B., Jonsson, J.: Preemptive multiprocessor scheduling anomalies. In: Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium, pp. 12–19, Ft. Lauderdale, FL, USA (2002)
27. Silva de Oliveria, R., Carminati, A., Starke, R.A.: On using adversary simulators to evaluate global fixed-priority and FPZL scheduling of multiprocessors. J. Syst. Soft. **86**, 403–411 (2013)
28. Markic, I., Stula, M., Maras, J.: Intelligent multi-agent systems for decision support in insurance industry. In: Proceedings of International Convention on Information and Communication Technology, Electronics and Microelectronics, pp. 1118–1123. Opatija, Croatia (2014)
29. Ravindra, B.T, Narasimha, M.M., Subrahmanya, S.V.: Big Data Abstraction Through Multi-agent Systems in Compression Schemes for Mining Large Datasets Advances in Computer Vision and Pattern Recognition, Chap. 8, pp. 173–183, Springer (2013)
30. Twardowski, B., Ryzko, D. Multi-agent architecture for real-time big data processing. In: Proceedings of IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), pp. 333–337. Varsavia, Poland (2014)

Springer

http://www.springer.com/978-3-319-23741-1

Intelligent Agents in Data-intensive Computing
Kołodziej, J.; Correia, L.; Manuel Molina, J. (Eds.)
2016, XVIII, 216 p., Hardcover
ISBN: 978-3-319-23741-1