CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3802

SPECIAL ISSUE PAPER

# Control centric framework for model continuity in time-dependent multi-agent systems

Franco Cicirelli and Libero Nigro*,†

*Laboratorio di Ingegneria del Software, Dipartimento di Ingegneria Informatica Modellistica Elettronica e Sistemistica (DIMES), Università della Calabria, Rende (CS) 87036, Italy*

SUMMARY

This work centres on a control framework for general multi-agent systems, which separates the design of agents behaviours (the application layer) from crosscutting control concerns (the control layer) regulating message exchange and processing. The goal is to support modelling and execution of a multi-agent system whose evolution is transparently governed by a pluggable control structure. A library of different control structures, including pure-concurrent and time-sensitive (real-time and simulation) strategies, was developed. The paper describes the developed control framework and focusses on the achievement of control strategies compliant with agent mobility and resource availability. The control design rests on a minimal actor computational model extended with actions, which are independent computational tasks able to execute in parallel. The approach enables *model continuity*, that is, the same model is used from analysis by simulation to implementation and real-time execution. The framework is prototyped on top of the JADE agent infrastructure. Usability and effectiveness of the resultant approach are demonstrated by a case study based on a complex closed queue network of services. On-going and future work is finally pointed out in the conclusions. Copyright © 2016 John Wiley & Sons, Ltd.

Received 27 January 2016; Accepted 28 January 2016

KEY WORDS: control framework; model continuity; multi-agent systems; parallel/distributed simulation; real-time execution; JADE

## 1. INTRODUCTION

Usually, the runtime infrastructure of a multi-agent system provides a control structure, which rests on agent multi-threading and on the exchange of asynchronous messages [1]. Messages are buffered into a mailbox owned by the recipient agent and subsequently processed by the control thread of the agent itself. The agent blocks waiting for new incoming messages when the mailbox becomes empty.

The way the messages are handled ultimately depends on the behaviour of the agent, that is, message handling is a concern of the application layer. The built-in control structure proves able to support basic agent features [2] like autonomy, proactivity, adaptivity to the surrounding perceived/acted-upon environment, sociality and mobility.

This work argues that in order to widen/tailor the applicability of multi-agent systems to specific application domains, it is important to adapt the basic control structures of agents. Control structures regulate system evolution by determining the way messages are gathered, ordered, dispatched and processed by recipient agents. For instance, different time notions (real-time or simulated time) can be adopted, and the message dispatching (sequential or parallel) can ultimately be realized by con-

*Correspondence to: Libero Nigro, Laboratorio di Ingegneria del Software, Dipartimento di Ingegneria Informatica Modellistica Elettronica e Sistemistica (DIMES), Università della Calabria, Rende (CS) 87036, Italy.
†E-mail: l.nigro@unical.it

sidering the availability of dedicated processing units. Flexibility of control design is also advocated to deal with problems of *mechanism design* [2] where a suitable control strategy is required to regulate/coordinate the decision process in a group of agents during the allocation of scarce resources.

This paper proposes an original and flexible control framework for distributed multi-agent systems. The approach makes it possible to transparently aggregate a given control module extracted from a library to a multi-agent system. One of the challenging goals in this work is to support *model continuity* [3] whose aim is to favour the use of a same model from property analysis (possibly based on parallel/distributed simulation) to real-time execution. A particular implementation of the approach, devoted to the schedulability analysis of real-time systems, is described in [4].

The proposed control framework purposely depends on a minimal computational actor model [5–7]. The actor model actually used in this paper is novel in that it owns a notion of *actions,* which are a key for transparently switching from simulation to real execution. Actions naturally map on to *processing units* (PUs) managed by a given *control strategy*. PUs model computational resources of the external environment of the application. Both the availability and the behaviour of processing units can affect, in an orthogonal way, the execution of the application. In particular, the evolution of a realized multi-agent system constitutes an emerging property of the interaction between the application and its environment, mediated by the provided control layer.

For demonstration purpose, the framework is prototyped in JADE [8, 9]. JADE was chosen because it is a representative of nowadays agent frameworks, which permit the development of general, untimed, distributed multi-agent systems using Java. JADE is open source, it adheres to FIPA communication standards [10], which in turn favour application interoperability. JADE rests on a multi-threaded agent model and on asynchronous message passing.

A preliminary version of this work appeared in [11]. The current version of the framework described in this paper greatly improves the previous one and that described in [4] by (i) supporting pre-emptive control forms (both during simulation and real-time execution); (ii) adding flexibility to the mapping of actions on to the available processing units; (iii) augmenting the kinds of supported actions (normal, incremental-accuracy and multi-body actions); and (iv) enabling behaviour modelling of processing units. The use of the achieved control framework is demonstrated by a case study concerned with a closed queue service network where all the newly added features are exploited.

The remainder of this paper is organized as follows. Section 2 provides some related work. Section 3 reviews basic concepts of JADE. Section 4 describes the adopted actor model with actions. Section 5 discusses some methodological guidelines about the use of actors and actions with control forms for modelling multi-agent systems. Section 6 proposes the control framework design in JADE. Section 7 summarizes a library of achieved control strategies. Section 8 details modelling, thorough analysis and model continuity aspects of the chosen case study. Finally, conclusions are drawn in Section 9 with an indication of on-going and future work.

## 2. RELATED WORK

The work described in this paper can be first related to a number of multi-agent runtime infrastructures and frameworks developed over the past years. These infrastructures allow the simulation and/or the execution of models, which are based on the agent metaphor.

The DIVAs framework [12] supports the development of large-scale agent-based simulation systems where agents are situated in open environments. DIVAs include high-level abstractions for the definition of agents and open environments, a microkernel for the management of the simulation workflow, domain-specific libraries for the rapid development of simulations and reusable, extendable components for the control and visualization of simulations.

GALATEA multi-agent systems [13] combine two lines of research: simulation languages based on the Zeigler's theory of simulation [14] and logic-based agents. There is in GALATEA a proposal to integrate, in the same simulation platform, conceptual and concrete tools for multi-agent, distributed, interactive, continuous and discrete event simulation.



DOI: 10.1002/cpe

Differently from more traditional approaches (e.g. in [15]), the problem of distributing agent-based simulation systems is considered in [16] where, instead of distributing work packages over distinct nodes of a high performance cluster, the simulation workload is distributed over the Internet and the computation is carried out in the JavaScript rendering engine on off-the-shelf computers, smart phones and Internet-connected devices. The same problem is addressed in [17] by using the Java-based middleware Terracotta, which permits to cluster the JVM by the concept of a network-attached heap memory holding shared object graphs, so as to improve specifically the availability and scalability of web-based enterprise applications.

In [18], a real-time agent-based middleware solution with a reliable mobile message protocol in wireless networks for data transfer is proposed . The infrastructure was purposely developed to deal with healthcare service environments.

An agent-based approach exploitable in the domain of policy modelling and simulation is proposed in [19]. The term policy refers to strategic areas of complex decision-making with various stakeholders having conceivably diverging interests. The approach relies on a policy development process and a software toolbox supporting this process.

An agent-based framework directed to the field of digital humanities is described in [20]. It integrates an agent design template, a transparent and layered mechanism to translate model-level agents actions to timestamped events and a distributed simulation kernel. All the previously mentioned frameworks and approaches, although, do not allow an adaptation of the provided control policy. In addition, no one addresses the problem of model continuity.

Some specific efforts directed to experimenting with the concept of model continuity in the development lifecycle of networked real-time embedded systems are carried out in the context of the discrete event system specification (DEVS) [14] research community.

In [21], a systematic method for designing, testing and executing intelligent systems with timing constraints is proposed. The DEVS-based approach provides a modelling–simulation–execution methodology with several stages, to develop real-time software. In the modeling stage, atomic and coupled models capture behaviours and structures of both the system and the environment. In the simulation stage, models are tested in an incremental way. Finally, in the execution stage, the verified model is executed by a real-time execution engine. DEVS state Activities (originally introduced in the RT-DEVS [22] formalism) in a case can model sensors/actuators hardware interfaces (APIs) with which the control model interacts during its decision process. In a different case, an Activity can be modelled with its own control logic. Activities are expressed as SimActivities with a common interface, during simulation. For instance, a sensor SimActivity obtains input from the environment model, in the same way as the real sensor (RT)Activity obtains input from the physical environment. The goal is to ensure the control model to transit (almost) unchanged from simulation to real execution.

The approach proposed in [21] was applied to decentralized control of a distributed robotic system in [23] using the leader–follower pattern. Here, the robot sensor/actuator Activities act as hardware interfaces. During simulation, different simulator engines, including a fast-mode simulator, real-time simulator and distributed simulator, can be chosen. All of this allows to validate alternative design models and to incrementally check the logical and the temporal behaviour of the models. Distributed simulators and execution engines take care of distributed communication, which is kept transparent to the models. A modeller might not be able to completely replicate a complex real environment in the environment model used during simulation. Therefore, design problems can surface during the real execution, which in turn can require re-iterations of the overall design process.

The Activity concept was refined in [24] where real-time simulation of a robotic system is considered. An Activity is seen as a computation task that an atomic model owns and can trigger. The Activity communicates its results to its owner through messages. In [3, 24], both real and virtual versions of a robot coexist and must be kept synchronized. Abstract (or simulated) Activities, real Activities (RTActivities) and hardware-in-the-loop Activities (HILActivities) are used in combination. An HILActivity is both a real and a simulated activity. It drives the real sensor/actuator of a robot in the real environment while, in the meantime, it is coupled (by message passing) with the environment model. All of this would maintain unchanged the decision-making model. For the



DOI: 10.1002/cpe

critical time synchronization problem, the approach requires a suitable execution engine able to ensure the environment model simulation runs in real-time [25–28].

In [29, 30], a progressive simulation-based design (PSBD) framework is proposed where Activities are removed and directly reproduced by DEVS component models. A virtual sensor/actuator is modeled as an atomic or coupled (if it is complex) model. Such virtual components are replaced by physical counterparts when the system model is transitioned to the real execution environment. The PSBD framework distinguishes four development stages. In the first stage, a centralized simulation on a single computer is used to analyse/test the system model along with its associated environment model (e.g. by reproducing the communication latency of a networked architecture). In the second stage, a distributed real-time simulation where models are deployed onto real network nodes is executed with the goal of validating communication delays. In the third stage, a hardware-in-loop simulation is used where the environment model is simulated by a DEVS real-time simulator on one computer, whereas the control model is executed under a real execution engine on real hardware. In the last stage, the system control model is run with a real-time execution engine, and the control model interacts with the real environment through sensors/actuators interfaces that drive the real sensors/actuators. According to PSBD, model continuity is preserved only for the control models.

Model continuity is also at the basis of the DEVSRT approach proposed in [31]. DEVSRT adopts hardware–software co-design where hardware and software components are uniformly integrated in the DEVS modeling and simulation. DEVSRT extends DEVS by adding a relative deadline to the generation of the output at each state, measured from the end of the state. In addition, driver objects are added to the border of a control system, which are a key for modularly interfacing the control model with the environment sensors/actuators. DEVSRT was implemented using the E-CD++ tool and runs on top of the Xenomai real-time kernel.

A specific approach directed to control timing (both virtual-time, real-time and physical-time) in an integration of real-time simulation and computational-physical systems is the action-level real-time DEVS (ALRT-DEVS) proposed in [32]. ALRT-DEVS borrows concepts from RT-DEVS [22] and real-time statecharts [33]. Actions (i.e., activities) with a time interval are associated to state locations, which have a deadline. Guarded state transitions provide dynamic decision-making capabilities. Timing is associated only with atomic models, which are priority defined. Coupled models in ALRT-DEVS do not specify time. The input/output couplings are timeless, meaning that messages traversing couplings consume zero times (instantaneous message transfer), which can be a problem in real-time simulation. The underlying platform used to execute a ALRT-DEVS model can introduce limitations to the model timing in the form of a maximum accuracy provided to real-time simulation. As a consequence, real-time guarantees are not specified in the model but come into play when the model is realized on a specific platform.

With respect to the previously described approaches, this paper proposes an original project whose aim is to support model continuity in general time-dependent multi-agent systems. The approach shares some basic concepts with the previously mentioned DEVS work. A distinguishing feature of the approach is the definition of a control centric framework designed according to aspect-oriented programming concepts [34]. Pluggable control strategies are usable without modifying the application models. Processing units purposely model, in a transparent way with respect to the business model, entities belonging to the execution environment of an application. The behaviour and availability of processing units implicitly affect the evolution of a realized application. Actions are introduced to represent boundary elements joining an application model with its external environment. An action can either be based on incremental-accuracy computing or be structured as a multi-body action. The reification of the actions along with the flexibility of replacing the control forms naturally supports model continuity.

## 3. AN OVERVIEW OF JADE CONCEPTS

Java Agent DEvelopment framework (JADE) [8] is an open-source multi-agent infrastructure developed in Java. Its runtime support hides system heterogeneity thus allowing the realization of distributed MASs. JADE agents are thread based. A behaviour can be dynamically added/removed



DOI: 10.1002/cpe

to/from an agent. Some already available basic behaviours can be specialized in order to fulfil modelling needs. Complex behaviours (e.g. sequential or parallel) are available too.

Execution loci for agents are the so-called *containers* further organized into *platforms*. A platform constitutes a realized distributed system, which is booted by starting a *main container*. Other containers can be launched to join with an existing main container thus establishing a given platform.

Besides the *agent* abstraction, JADE provides a simple yet powerful task execution and composition model. It supports a peer-to-peer agent communication model based on asynchronous message passing. Messages are expressed through the FIPA ACL (Agent Communication Language). They can carry either simple textual information or complex serialized Java objects. An application can rely on a family of ACL messages sharing a common ontology. Messages are received via a local mailbox [1] from where they are extracted and processed, one at time, through the agent-behaviour structure.

JADE provides the fundamental services of agent naming, mobility and searching through yellow-pages. Agents can be created by using the RMA (Remote Management Agent) GUI or, during the runtime, by using the available APIs as a part of the application logic.

JADE (serialized) agents can be migrated dynamically from a container to another. The `doMove()` method serves to request a migration, whereas the `beforeMove()` and `afterMove()` methods are respectively used to specify what to do just before and just after a migration.

Programming agents are supported by some basic classes/interfaces such as `Agent`, `Location`, `AID` (Agent unique IDentifier), `Behaviour` and `ACLMessage`. It is worthy of note, although, that no APIs exist in JADE to natively support neither a time notion nor mechanisms for building, for example, a simulation model.

## 4. AN ACTOR MODEL WITH ACTIONS

In this work, a minimal actor computational model [1] is adopted, which modularly separates the application logic from the control aspects, which reflectively govern the evolution of the application. A transparent interchanging of control aspects can be exploited, in an important case, to favour a smooth transition of a given model from the analysis phase (based on simulation) down to the implementation and real-time execution.

Actors [5–7] have a hidden behaviour (finite state automaton), which is in charge of responding to incoming messages and modifying a hidden set of data variables. An actor reacts to messages on the basis of its current state and the type and content of a received message. An actor is at rest until a message arrives. The communication model relies on the exchange of asynchronous messages.

Actor behaviour is implemented in the *handler(msg)* method, which processes a received message by actuating corresponding data/state transitions and, possibly, by creating and submitting for execution one or multiple *actions*. An action defines an activity, which has a duration and requires, for its execution, a computational resource, that is, a *processing unit*.

An action can be seen as a black box, which has a set of input parameters, a set of output parameters and an execution body with a deadline within which its execution should complete. At action termination, the submitter actor can be notified so as to retrieve the output parameters from the action object and possibly update its own data variables.

Actions have no visibility to the internal variables of actors. All of this avoids interference problems when multiple actions submitted by a same actor are concurrently executed. As a consequence, no synchronization mechanism (e.g., locks) needs to be used. The following are the basic actors operations:

- *newActor*, for creating a new actor;
- *become*, for changing the state of the actor;
- (non-blocking) *send*, for transmitting (scheduling) messages to acquaintance actors (including itself for proactive behaviour);
- *do* action, for submitting the execution of a given action;
- *suspend* action, for temporarily suspending the execution of a given action. A suspended action is de-scheduled and its processing frozen until a subsequent resume operation;



DOI: 10.1002/cpe

- *resume* action, for making a previously suspended action ready to be executed again. A resumed action is re-scheduled and its computation will continue from the point it was last suspended; and
- *stop* action, for aborting the execution of an action.

Each actor is mapped onto a JADE agent, and a subsystem of actors (logical process) is assigned for execution to a JADE container. Although JADE does not have any built-in solutions for developing time-sensitive applications [35–38], it was mainly exploited for its basic services like naming, setup, message passing and migration.

For each logical process, a *control machine* (CM) is the component, which hides a specific control strategy, and it is responsible for administering scheduled (send) messages. A control machine can be in charge of managing a time notion (real-time or simulated time) regulating actor behaviours. Submitted actions are manage by the *action scheduler* (AS), whereas action execution, both during simulation and real-time, ultimately depends on a collection of parallel *processing units* (computational resources) administrated by an AS. For generality, a complex processing unit can have its behaviour explicitly modelled.

## 5. MODELLING WITH THE CONTROL FRAMEWORK

This section provides some guidelines to develop an application based on actors and actions, together with the rationale underlying the furnished abstractions. A discussion about the supported application lifecycle is also given.

### *5.1. Modeling with actors and actions*

As described in the previous sections, the basic abstractions upon which a model is built are actors, messages, actions and processing units. It is useful to point out the different roles played by the previous entities. Actors and messages capture the business logic of a model. Messages mainly serve to maintain sociality relationships among actors (communication) and to trigger actor behaviour, that is, making a state transition in the finite state automaton of the receiving actor. From a model point of view, the processing of a message is instantaneous. This means that during real-time execution, the time needed to process a message should be negligible. Message processing cannot be pre-empted nor suspended. A timestamp can be used to specify when the message has to be delivered to its recipient. If the timestamp is not specified, the message has to be processed at current time.

Differently to messages, actions can be pre-empted, resumed and/or aborted. For an action, it is not allowed to specify a timestamp. Actions model activities whose execution consumes time and requires computational resources not owned by the submitting actors. Such computational resources, abstracted by means of processing units, are used to represent entities, which do not belong to the actor model but which are part of the environment (or context) where the model operates. The number, behaviour and status (e.g. available, busy and out of work) of the processing units regulate if and when an action is executed and, as a consequence, determine the action parallelism degree. The action parallelism degree, on the other hand, affects the evolution of the actor model both in simulation and real execution. After an action is submitted for execution, an actor remains able to receive and process other incoming messages. Multiple actions can be submitted without necessarily waiting for the completion of a previously submitted one.

For each action, it is possible to specify a set of exploitable processing units among which to choose the action executor. An action is immediately ready to be executed after its submission, but its execution can be deferred until an exploitable processing unit become usable. Choosing the right executor is a matter of an adopted control strategy. For example, a processing unit can be exploitable when idle, or it can become exploitable by first pre-empting an on-going computation.

Actions naturally can be used to abstract tasks, which need to be reified when switching from model analysis to model real execution. The use of actions along with the flexibility in changing the regulating control strategy are the key to foster model continuity. In the case the modeller is just



DOI: 10.1002/cpe

interested in property assessment of his/her system through simulation, that is, model continuity is not an issue, the use of actions can still be recommended to improve the speedup by spawning in parallel those activities occurring at a same time.

As a final remark, actions could entirely be unused in an actor-based model. In this case, action computation can be explicitly achieved in the actor behaviour, that is, their effect equivalently obtained by message exchanges and message processing.

### *5.2. Application lifecycle*

The prototyped control framework supports an application lifecycle, which is made up of the following phases: *modelling* through the exploitation of the actor metaphor, *property analysis* through simulation, *preliminary execution* and, finally, *real execution*.

The modelling phase can be carried out by following the guidelines provided in the previous section. To proceed with the simulation and subsequently with the execution, it is required to respectively setup the *simulation environment* and the *execution environment* of the framework. Each kind of environment is set up by choosing in a coherent way:

- a control machine;
- an action scheduler;
- the type of actions to use; and
- the number of processing units and, if needed, the behaviour of each of them.

A detailed description of the made available control forms and action types is provided respectively in the Sections 6 and 7.

The previous points refer to issues, which are orthogonal with respect to the modelling phase. This means that the business logic of the application remains unchanged while switching from analysis to execution.

During property analysis, the simulation environment is set where PUs and actions are simulated components. For real execution, instead, the execution environment is settled where PUs can be specific hardware devices or Java threads of a multi core machine and actions have their final implementation carrying out concrete activities.

A hybrid scenario between simulation and real execution occurs in the so called preliminary execution phase. Here, the control machine and the action scheduler are real time, but actions are not the simulated ones, nor the final effective ones. In this phase, actions are pure resource-consuming tasks, that is, they have a time duration and keep busy a processing unit (see also Section VI for some implementations of such actions). The processing units could be the same used for the pure execution environment or more specific because some thread-based PUs might be substituted by some real hardware devices.

The goal of this phase is to assess if real-time constraints, previously checked in simulation, are satisfied during real-time execution, which in turn means that the message processing overhead (including distribution issues) is effectively negligible. When the message processing is not negligible, it can be necessary to relax some timing constraints (e.g., by increasing the application, time tolerance factor and/or the model can be revised/optimized). For the sake of completeness, it is important to highlight that, from the model perspective, the right kind of actions can be established by using a factory object that is responsible for the creation of the proper kind of actions on the basis of the chosen environment.

Both the simulation and the execution environment are able to operate in a parallel/distributed context. Distributed simulation can be used in the case it is necessary to cope with a very large or complex model and/or in the case it is known in advance that the application under development is distributed in character. In the latter case, it would be of value to develop, analyse and carry out a preliminary execution by considering the distribution concerns.

## 6. CONTROL FRAMEWORK IN JAVA AGENT DEVELOPMENT FRAMEWORK

The actor framework described in the previous section was prototyped by using the JADE platform. An UML class diagram of the basic entities is shown in Figure 1. It is important to highlight that the



DOI: 10.1002/cpe

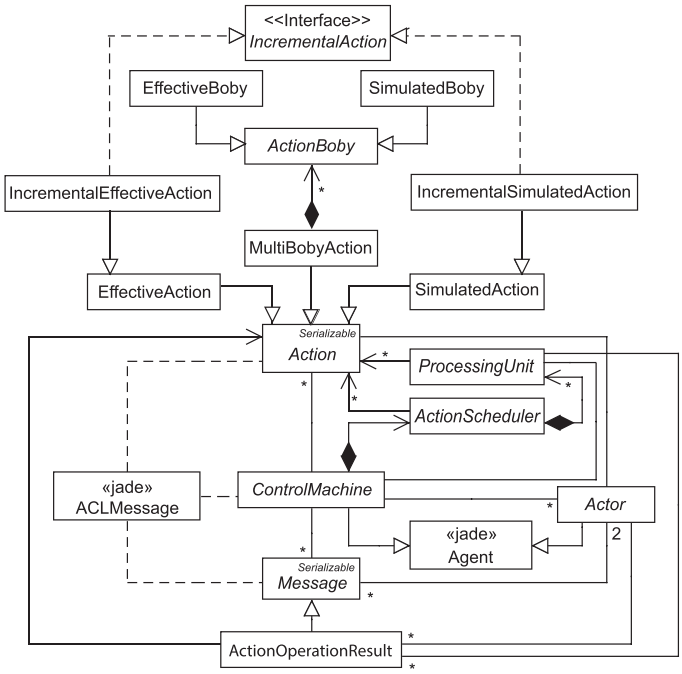



Figure 1. Basic classes of the control framework in JADE.

framework mainly relies on abstract classes and interfaces. This implies that it is open for extensions. In other words, if a new kind of control strategy is needed, a further kind of control machine and/or action scheduler can be implemented in order to meet specific requirements.

In the current implementation, both actors and control machines are mapped onto JADE agents, which communicate to one another by exchanging *ACLMessages*. Basic roles of the control framework are assigned to the following abstract classes.

*Message*. It is the common ancestor from which all the applicative messages derive. A message object is designed to be embodied, in serialized form, as the object content of an *ACLMessage*. A message has fields for the involved sender/receiver actors and a timestamp information.

*Action*. It contains the submission time, two free slots for hosting respectively the input and output parameters (array of serializable Objects), the relative deadline (measured from the submission time), the action priority and an indication about the set of PUs exploitable for its execution. In the case no indication 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 previous 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-emptible 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 (through the *do* operation) to a control machine as a serialized content object of an *ACLMessage*.

*ControlMachine*. It 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 first extracted from the set of pending messages according to a control policy; then the message is 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 added to the pending set, whereas the submitted actions are passed to the action scheduler. A time-sensitive control machine can require to synchronize with a time server (Figure 2) in order to obtain the necessary grant to proceed with the actual delivery of a timestamped pending message.

*ActionScheduler*. It imposes an application-specific execution policy to the actions and controls a set of processing units. On the basis of the adopted execution policy, a scheduler can (i) assign



DOI: 10.1002/cpe

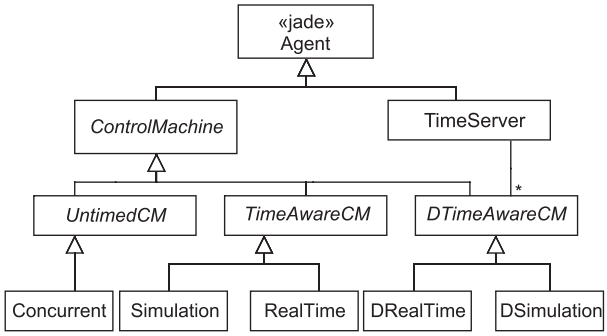



Figure 2. The hierarchy of developed control machines.

an action to a free processing unit, (ii) assign an action to a busy processing unit by firstly pre-empting the on-going action and saving its execution status, (iii) add an action to a pending set for its subsequent execution and (iv) suspend, resume or stop the execution of an action according to an actor request. Both pre-empted and resumed actions are added to the pending set. Suspended actions are instead added to a waiting-set looking for a subsequent resume operation. Stopped actions are definitively removed from the scheduler.

*ProcessingUnit*. It denotes an action executor capable of processing one action at a time. Methods of a PU include *start*, *pre-empt*, *resume* and *stop*, whose meaning should be self-explanatory. The previous methods are invoked by the action scheduler administrating the PU either by following a request made by an actor or either to agree to the control strategy established for the actions. An *ActionOperationResult* message is used to communicate to the actor, which submitted the action that (i) an action has terminated its execution or (ii) a requested operation, for example, suspend or resume, took place. A further method *whenAvailable* is used to ask to a processing unit about its availability. If a PU is already available, the method returns to zero; otherwise, it returns an estimation of the time needed to become available again. This method provides a common way to know if and when a PU is usable independently from its implementation, behaviour and status (for example, under maintenance or busy). The status of a PU can be accessed by using the method *getStatusInfo*.

*Actor*. It is the base class for applicative actors; it offers all the basic methods such as send, become, do/abort action and the abstract handler. The built-in JADE behaviour within the Actor class takes care of receiving an *ACLMessage* delivered by a control machine, extracting from it the (de-serialized) *Message* content object and triggering the corresponding message processing by invoking the *handler()* method on the recipient actor. At the handler termination, all the newly generated messages and actions are collected and sent back to the control machine, packaged in an *ACLMessage*. A binding relationship is established between the actors and the regulating control machine co-located in a same JADE container. Migration of actors is assisted by a redefinition of the *afterMove()* method, which updates the binding of a moved actor to the control machine of the reached container. A migrated actor can continue receiving messages deriving from a previously bound control machine. However, in this case, the messages are first (and transparently) forwarded to the actually bound control machine. All of this is necessary in order to guarantee that the control sensitive message scheduling and dispatching activities get ultimately managed by the control machine of the JADE container currently hosting the destination actors.

Besides the abstract classes so far described, Figure 1 are also shown some interfaces and concrete classes, which are related to actions. Such classes and interfaces can be roughly divided into two categories. The entities having ‘Simulation’ into their names are related to the simulation environment. On the contrary, a name containing ‘Effective’ refers to an entity usable within the real execution environment including the preliminary execution of a model. The built-in effective actions have the goal to furnish some ready-to-use implementations. Would it be necessary, new kinds of effective actions could be introduced as well.



DOI: 10.1002/cpe

Three different families of `Action` implementations are provided (Figure 1): (i) `SimulatedAction` and `EffectiveAction`, (ii) `MultiBodyAction`, (iii) `IncrementalSimulatedAction` and `IncrementalEffectiveAction`. `Simulated Action` and `EffectiveAction` implement the basic roles of the action as previously described in this section. A `MultiBodyAction` allows specifying a task, which can be accomplished by choosing one among different `ActionBodys`. All the `ActionBodys` are related to a same `MultiBodyAction` that are functionally equivalent. `ActionBodys` differ in the adopted algorithm and its corresponding execution time. Choosing the 'right' body to execute is a decision of an action scheduler. Specialization of `ActionBoby` is the `EffectiveBody` and `SimulatedBody`. Incremental actions are used instead to model activities whose duration is not statically fixed. A temporal threshold $IA_{th}$ is defined for each incremental action. The temporal threshold indicates that an action is able to furnish a usable although approximate result after $IA_{th}$ time units of its execution. The accuracy of the computation improves as the execution time goes beyond $IA_{th}$. A nominal execution time $IA_{ne}$ indicates the time after which the action surely provides the more accurate result.

All the provided action implementations related to the simulation environment have the *execute()* methods carrying out no computation. The method can be overridden, for instance, for gathering statistical data during simulation.

The *execute()* method related to implementation of actions exploitable during the preliminary execution is obtained by iterating a basic *computational step*. In the current prototyped version of the framework, such basic step may rely either on *busy-waiting* (achieved through a while-true based on a time lapse) or on a waiting strategy based on *sleeping*. Other realizations of the *execute()* method are possible depending on the application needs. The duration of each computational step is specified as a parameter at the action construction. The computational step gets repeated a number of times necessary to guarantee the fulfilment of the action duration. The described approach allows possibly (i) to modify the action duration during the runtime by varying the number of the remaining steps to be performed and (ii) to make action execution complaint with the operations of suspend and resume. In fact, an action terminates when all its computational steps are executed; despite the real time, the action remains suspended.

## 7. A CATALOGUE OF CONTROL STRUCTURES

A library of control structures, namely, a library of control machines and action schedulers, was prototyped as described in the following two sections and as depicted respectively in Figures 2 and in 3. Other control mechanisms can be added as well.

A common design principle of control machines (Figure 2) concerns co-operative concurrency of the actors handler methods, which are always executed one at time and in an interleaved way. A parallel execution schema, instead, applies to actions, which depends on the model parallel degree and the corresponding configured number of idle processing units. The way actions are ultimately executed that is determined by the adopted implementation of an AS (Figure 3). On receiving a message relevant to an operation to be performed on an action, a control machine forwards the message to an action scheduler, which takes care of it. Each time an action is completed, an *action operation result* message containing an *action completion* informative is raised by the executing processing unit. This message is in charge of (i) informing the control machine that a previously

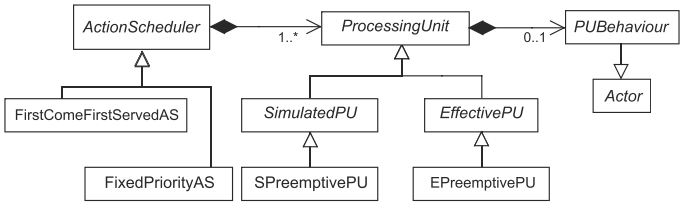



Figure 3. The hierarchy of developed action schedulers.



DOI: 10.1002/cpe

busy processing unit is now ready and can re-used (ii) notifying action completion to the submitter actor in the case it specified its willingness in receiving such a notification. The information carried by an action completion message includes the action starting processing time (which can be greater than the submission time), the completion time and the action identifier.

When a control machine is tailored to a simulation context, action execution is simulated, and processing units are realized as fake objects. However, despite the simulation context, the `execute()` method of a submitted action is anyway invoked to notify about action execution. This can be useful, for example, to gather statistical data and/or to predispose specific output parameters for the submitter actor. When an action is allocated to an exploitable processing unit, the action completion message is scheduled to occur at the time the action terminates. This mechanism allows the virtual time, during simulation, to increase accordingly to the time required to simulate action execution.

### *7.1. Prototyped control machines*

Three families of control machines were identified (Figure 2). The `UntimedCM` family contains control structures, which only manage untimed messages. Such control machines can be exploited either on a standalone or on a parallel/distributed machine where an actor model is split into multiple logical processes. `TimeAwareCM` denotes a class of control machines, which are capable of time management although in a not distributed setting. The `DTimeAwareCM` control machines can, instead, be used in the case a time-sensitive model has to be handled in a parallel/distributed context. In this case, a `TimeServer` is needed to enforce a common time notion among the participating control machines. For modelling simplicity, timestamps for messages are specified by relative times with respect to current (implicit) time. A relative time is automatically turned into an absolute one by the control machines. In the case no timestamp is provided, a message is considered to be scheduled at the current time. All the control machines own an action scheduler to which the submitted actions are delivered.

The Concurrent control machine implements an untimed parallel control structure, which depends on a FIFO message queue for storing pending messages and an assigned number of processing units. A Concurrent-based actor system terminates when an application-level `END` control message is received by all the involved control machines.

`Simulation` implements a classical discrete event simulation schema with a virtual time notion. Messages are ranked according to absolute timestamps. They are buffered into a time-ordered queue (TQ). At each iteration of the control loop, the most imminent (or one of the most imminent) message is extracted from TQ and its timestamp assigned to the virtual time. Then, the message is dispatched to its recipient actor. The `Simulation` control loop exits when the virtual time exceeds the simulation time limit. A package (actor.distributions) of common density distribution functions (including uniform, exponential, hyper exponential and Erlang, normal) based on the java.util. Random pseudo-random number generators can be exploited by the modeller when using `Simulation`.

The `Realtime` control machine adopts a notion of real-time based on the Java `System.currentTimeMillis()` service. The control machine is suited for non-hard real-time models. Similarly to Simulation, a message-based TQ is used. All messages in TQ are dispatched as soon as the current time exceeds their firing time. If the current time is lesser than the timestamp of the most imminent message in TQ, the control structure simply awaits. A configurable time tolerance `EPS` is used by `Realtime`. A time-constrained message scheduled to occur at absolute time `t` can then be considered *still* in time if the current time is within the time window `[t,t+EPS]`.

When switching from a control machine used for simulation to a control machine used for real-time execution, it is necessary to convert the simulation time units into real-time units. The default mapping associates a virtual time unit to a second of real-time. Other mappings, for example, a virtual time unit associated with an hour of real time, can be defined.

Prototyped time-aware control machines, which are able to work in a distributed context, are `DSimulation` and `DRealTime` (Figure 2). Differently from `Simulation`, `DSimulation` relies on a specialization of the `TimeServer`, which coordinates time advancement among all the involved control machines.



DOI: 10.1002/cpe

The following conservative time synchronization strategy is adopted [25]. In the case the next timed message has a timestamp $\tau$, which is greater than the current simulation time, the control machine first asks the time server for a grant to advance to $\tau$. All the proposals of time advancement are collected by the time server. The minimum of these proposals is finally provided as grant to requesting control machines. However, for the grant to be generated, the condition of 'no in-transit messages'has to hold in the system. Toward this, separate counters of sent and received messages [25] are introduced for each actor/agent. The counters are kept updated by the control machines, and the counter values are furnished as accompanying information to the time-advancement proposals to the time server. These fine-grain counters are necessary to cope with actors migration, which implies a same actor can dynamically be handled by distinct control machines.

The `DRealTime` control machine is very similar to `RealTime`. However, an extended interpretation of time tolerance `EPS` is needed. In fact, despite the use of a time server, in a distributed context, it is impossible to completely eliminate the time misalignments occurring among computational nodes. As a consequence, it can happen that a control machine $CA$ at time $tA$ receives a timed message from a control machine $CB$ with time $tB$ such that $tB < tA$. In such a case, $CA$ receives a message in the past. The message is still considered in time if $tA - tB <$`EPS`. In other words, `EPS` in the distributed context is also used to ascertain if a clock misalignment can be considered negligible. The prototyped `DRealTime` rests on a `TimeServer`, which transparently synchronizes the clocks of the computing nodes. In particular, the tool Dimension 4 [39], based on the SNTP protocol, was used for experimental purposes. Other techniques, for example, based on UTC [40] or GPS [41], could be used to keep time-aligned the control machines.

### *7.2. Action schedulers*

Prototyped schedulers (Figure 3) immediately put into execution a newly scheduled action on an idle *exploitable* processing unit (if there are any). In the event no such idle PU exists, the scheduler `FirstComeFirstServerAS` organizes actions in a pending list. This list is ranked according to the submission (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 when it is not exploitable by any of the actions in the list. The `FixedPriorityAS` scheduler, instead, uses an action priority to rank the pending list. Action execution is priority driven and pre-emptive. The duration of a pre-empted action is shortened by the time the action was running.

Both the previous schedulers are able to work with incremental or multi-body actions. Supposing one of such an action is scheduled, the absolute deadline of the action is exploited by the scheduler to figure out the proper duration of the action. In the case of a multi-body action, the scheduler chooses for execution the longest body of the action whose duration permits to meet the deadline. If all the bodies do not permit to satisfy the deadline, the shortest one is chosen. After being chosen, the body cannot be changed anymore. For an incremental action, the scheduler chooses for the action, the longest execution time capable of fulfilling the deadline. In any case, the computation cannot have a duration, which is lesser than the temporal threshold $IA_{th}$ defined for the action or greater than the value $IA_{ne}$ (Section 6). The use of incremental actions can be more flexible than that of multi-body actions because the execution time of an incremental action can be changed if needed (i.e., shortened again) after the action gets resumed following a pre-emption or a suspension.

The behaviour of an action scheduler remains the same during simulation or real-time execution. A proper processing unit is instead required when switching from system analysis to preliminary or real execution. For simulation purposes, one can use `SPreemptivePUs`, that is, some passive objects without internal threads. The `EPreemptivePUs` are instead active objects, that is, thread based, allowing the actual execution of effective actions. Each pre-emptive PU manages a pool of Java threads where only one thread at time can be running. This ensures a unitary degree of parallelism within the PU. The use of multiple threads allows a running action in a PU to be pre-empted and replaced by another running action and then subsequently resumed.

A `PUBehaviour` can be attached to a processing unit in order to implement a specific, possibly complex, behaviour. All of this can be exploited to model processing units having a limited or



DOI: 10.1002/cpe

discontinuous availability (due for instance to an overheat, ageing, shift or failure behaviour). The `PUBehaviour` is simply removed during real system execution because its functionalities will be implicitly implemented by the real processing unit. A PU behaviour is selected and set during the configuration phase of the framework, and its definition remains independent with respect to the application model. The design of PU behaviours purposely contributes to maintain unchanged an application model when transitioning it from the analysis down to the implementation stage.

## 8. CASE STUDY

The agent-based modelling and analysis of a complex closed queuing network named CSM (Central Station Model) was considered as a case study. The model (Figure 4) is representative of a wide class of systems. As an example, a specialization of the CSM has been used in [42] for studying, in the context of a seaport logistic problem, the optimal assignment of berth slots and cranes to shipping services at a modern terminal of marine containers. The goal of this section is to provide evidences about the effectiveness and usability of both the proposed approach and supporting control framework. More in particular, issues related to (i) modelling activities, (ii) property analysis and (iii) preliminary execution of the CSM are detailed. The latter point is specifically devoted to estimating how message transmission, message processing and action management affect real-time execution.

### *8.1. Problem statement*

The CSM model is based on $K$ re-circulating clients, and it is composed of a reflective station S0 and four service stations S1, S2, S3 and S4, which provide services to clients. The number of clients is fixed, and they re-enter S1 after each reflection in S0. The service stations along with the router (Figure 4) constitute the processing system. Each station needs some computational resources (processing units) for its operation. The number of admitted processing units determines the maximum number of clients a station can handle at the same time, that is, the *parallel degree* (Pd) of the station. A client arriving at the station has to *wait* when it cannot be served immediately.

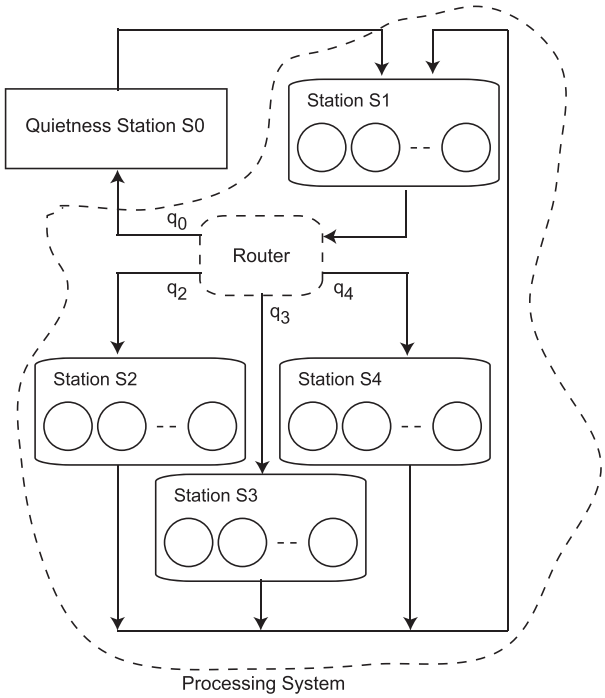



Figure 4. A CSM model.



DOI: 10.1002/cpe

Table I. CSM parameter values.

| Parameter | Values |
|---|---|
| Exponential distribution for S0 | $mu0 = 1.38 \times 10^{-4}$ t.u.$^{-1}$ |
| Exponential distribution for S1 | $mu1 = 0.01$ t.u.$^{-1}$ |
| Exponential distribution for S2 | $mu2 = 2.85 \times 10^{-3}$ t.u.$^{-1}$ |
| Hyper-exp distribution for S3 | $mu31 = 0.002$ t.u.$^{-1}$, $mu32 = 0.005$ t.u.$^{-1}$, $a31 = 0.7$, $a32 = 0.3$ |
| Erlang distribution for S4 | $n4 = 5$, $mu4 = 8.33 \times 10^{-3}$ t.u.$^{-1}$ |
| Uniform distribution for router | [5, 10] t.u. |
| Probabilities for router | $q0 = 0.6$, $q2 = 0.1$, $q3 = 0.2$, $q4 = 0.1$ |
| $K$ | 10, 20, 30, 40, ... 100 |
| $K(\pi_l)$ | $0.7 \times K$ |
| $K(\pi_m)$ | $0.2 \times K$ |
| $K(\pi_h)$ | $0.1 \times K$ |
| $U_{trs}$ | 80% |
| $T_{ut}$ | 200 t.u. |
| $T_m$ | 200 t.u. |
| $CU$ | 500 |
| Pd(S0) | Infinite |

Each client owns a priority. Three different priorities are considered, namely, $\pi_h = 3 > \pi_m = 2 > \pi_l = 1$ from the highest, to average and to the lowest one. The number of clients with a specific priority is denoted respectively by $K(\pi_h)$, $K(\pi_m)$ and $K(\pi_l)$. Obviously, $K(\pi_h)+K(\pi_m)+K(\pi_l) = K$. To avoid starvation of low priority clients, it should be $K(\pi_h) < K(\pi_m) << K(\pi_l)$ (Table I). A client $c$ under service at a given station $s$ is pre-empted and becomes waiting if a highly priority client arrives at $s$. The execution of $c$ can be resumed when no more waiting higher-priority clients exist in $s$.

Initially, the $K$ clients are injected into the reflective station S0 where they reflect a certain amount of time before entering the system. Within the reflective station, all the clients reflect in parallel, that is, Pd(S0) is $K$. Stations S1, S2, S3 and S4 have instead a bounded parallel degree denoted respectively by Pd(S1), Pd(S2), Pd(S3) and Pd(S4).

A client enters the processing system by arriving at the central (front-end) station S1 whose service time is assumed exponentially distributed. After service in S1, the client, with certain probabilities (q0, q2, q3 and q4), can be routed (see the `Router` station in Figure 4) in input to S0, or to one of the service stations S2, S3 or S4. Each router output is supposed to be affected by a uniform distributed communication delay.

Station S2 has an exponentially distributed service time. Station S3 has a second-order hyper-exponential distribution, which is characterized by the rate of each exponential component (mu31 and mu32) and the probability of choosing one distribution or the other (a31 and a32). The hyper-exp is configured to reproduce a burst phenomenon, where *silence times* are due to mu32 and *burst repetitions* are due to mu31. Station S4, finally, uses an Erlang distribution composed of $n$ identically distributed exponentials with the same rate.

A client exiting S2, S3 or S4 comes back in input to S1. It is worth noting that when a client enters S0, it actually exits the processing system, and it is annotated with the exit time. Similarly, when a client exits S0, it enters the processing system, and it is time-stamped with the enter time. This way, passage through the S0 permits to check the timing behaviour of the whole system.

Computational resources allocated to a station are not continuously available. In particular, the resources in S1, S3 and S4 can become temporarily unavailable for overheating when their mean utilization goes over a given threshold $U_{trs}$ within a time interval $T_{ut}$. Computational resources of S2, instead, may become temporarily unavailable for a time interval $T_m$ because of maintenance reasons after $CU$ consecutive uses.

The parameter values of CSM, except for the values of the parallel degree Pd(S1), Pd(S2), Pd(S3) and Pd(S4), are collected into Table I. The parallel degree of each service station will be assessed during properties analysis.



DOI: 10.1002/cpe

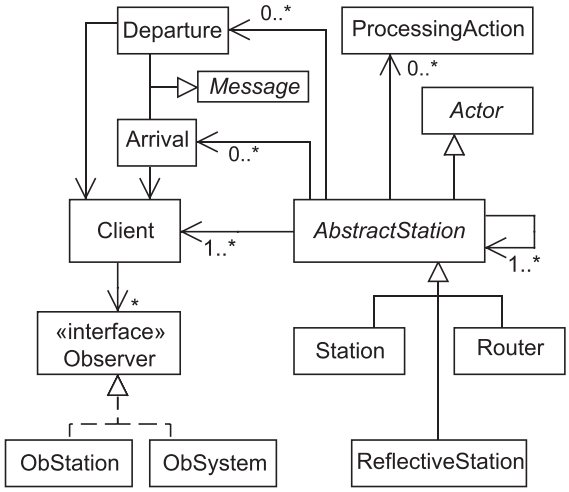



Figure 5. A CSM model class diagram.

### 8.2. *Modelling the Central Station Model system*

A UML class diagram of the actor-based CSM model is shown in Figure 5. The generic station is defined through the `AbstractStation` class. Concrete classes specializing such abstract class are `Station`, `ReflectiveStation` and `Router`. Basic messages are identified by `Arrival` and `Departure`.

`Router` and `ReflectiveStation` do not use actions because they do not need to model activities that require to be reified when switching from analysis to real execution. As a consequence, the behaviour of the `ReflectiveStation` and that of the `Router` is regulated by message passing only.

`Station` is used to model processing stations, which provide a service to clients; hence, the use of actions is useful. For demonstration purposes, `Station` does not introduce any buffering for the clients awaiting to be served. As soon as a new client arrives, a new action is created and submitted. In this way, the buffering, the dispatching, the priority management and the execution of actions become a responsibility of the control framework, thus simplifying the modelling activities.

All the stations have a next attribute indicating where a client should be routed after its processing. An array of next stations is instead provided to the `Router`. Each element of the array is paired with a probability value.

Implementations of the `Observer` interface are used to monitor client arrival and departure events. Two different kinds of observers are considered: the `ObStation` and the `ObSystem`. The first type is used to monitor a single service station (S1, S2, S3 or S4); the second one serves to watch, according the viewpoint of the reflective station (S0), the overall system behaviour.

Other two classes, heir of the `PUBehaviour` abstract class (Figure 3), were introduced. Such classes are the `ProcessingUnitOverheatBehaviour` and the `ProcessingUnitMaintenaceBehaviour`, which are used respectively to model the processing units with overheat and the processing units needing maintenance. These classes are not reported in Figure 5 because no entity of the CSM model directly interacts with them. All of this ensures (Section 7.2) that when switching from property analysis to real execution, the emulated processing units will be transparently substituted by real processing units without affecting the CSM model.

### 8.3. *Property analysis of the Central Station Model system*

Parallel/distributed simulation was used to estimate quantitative properties of the processing system by varying the number of re-circulating clients and that of processing units. Some properties such as the *response time* (waiting time plus service time spent by a client into a station), the *number of*



DOI: 10.1002/cpe

*provided services* and *utilization* of the whole CSM and of each single station S1, S2, S3 and S4 were studied.

The model was partitioned into two JADE containers: the first one (main container) hosts all the stations except S1, which is allocated to the second container. The `DSimulation` control machine, the `FixedPriorityAS` action scheduler and the `SPreemptivePUs` as processing units were used to configure the control framework. Simulation experiments were executed with a time limit of $3 \times 10^6$. Experimental results confirmed that this simulation time guarantees, among others, that the average service time parameter of each station is eventually met. Experiments were carried out on two Win 7, 12 GB, Intel Core i7, 3.50 GHz, 4 cores with hyper-threading.

Four scenarios were considered according to the parameter values in Table I. In the first one (maximum parallelism hypothesis), an arbitrary number of processing units were admitted for each service station. The goal was to estimate the effective parallel degree of the models, that is, the maximum number of used processing units, as the number of clients increases. In the second scenario, it was fixed; the number of processing units and the system behaviour was evaluated as the number of clients increases. This scenario was also dedicated to identify potential bottlenecks of system performance. In the third scenario, a performance optimization of the system was obtained by adopting incremental-accuracy actions. In the fourth scenario, instead, a performance optimization is studied by adding some extra processing units to those service stations identified as bottlenecks in the second scenario. Each scenario was configured without modifying the CSM model but by varying only the framework set-up.

*8.3.1. Scenario 1: System level behaviour under maximum parallelism.* In this scenario, the use of priorities does not introduce any advantage nor penalty for any client. Each time a client arrives at a station, it is immediately served because the number of processing units is unbounded. In addition, there is no need to use `ProcessingUnitOverheatBehaviour` or `ProcessingUnitMaintenaceBehaviour` because new PUs are always available. `SimulatedActions` are used within stations from S1 to S4.

Obviously, for each station, the waiting time is zero, and the measured response time coincides with the station service time determined by the adopted exponential distribution. For the whole processing system, instead, the response time was found to be about 470 t.u. despite the number of re-circulating clients. For the hypothesis of maximum parallelism, it is easy to predict a linearly increasing number of provided services as the number of re-circulating clients increases. The same trend is expected for the utilization factor of each station. All of this was confirmed by the simulations, as reported in the Figure 6(a), (b) and (c).

In particular, Figure 6(a) depicts the measured number of services of the whole system versus the number of clients, for the different categories of client priority and for all the clients. Figure 6(b) portrays the number of accomplished services according the viewpoint of the single service stations.

Figure 6(c) portrays the utilization of the stations S1, S2, S3 and S4 versus the number of re-circulating clients. An utilization factor below or equal to 100% mirrors the fact that a station is able to work with a single processing unit. An utilization factor belonging to $(100\%, 200\%]$ implies instead the need of two processing units. More in general, let $U_{\%}(s)$ be the utilization factor of a station $s$, and the number of required processing unit is $\lceil U_{\%}(s)/100 \rceil$.

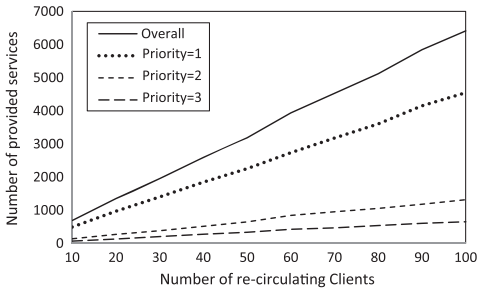
Figure 6(d) shows the number of required processing units versus the number of re-circulating clients. In the worst case, that is, when $K = 100$ clients are considered, eight processing units are required: three for S1, two for S3, two for S4 and one processing unit for S2.

*8.3.2. Scenario 2: System level behaviour with constrained parallelism.* In this scenario, the processing system was studied under the more realistic hypothesis of a finite number of processing units, which can possibly become unavailable. In particular, the CSM was configured with a number of re-circulating clients $K = 100$, three PUs for S1, two PUs for S3, two PUs for S4 and one PU for S2, as emerged in the previous scenario and stated in Figure 6(d). In addition, the processing unit of S1 was equipped with the `ProcessingUnitMaintenanceBehavior`. For all the other PUs, the `ProcessingUnitOverheatBehavior` was set. `SimulatedActions` are still used.

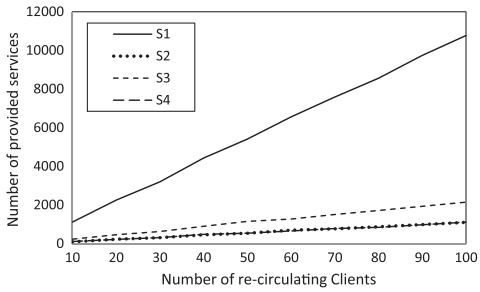
By cross-referencing Figures 6(a) and 7(a), it emerges that the number of services provided by the processing system remains almost the same both in the case of maximum parallelism and for



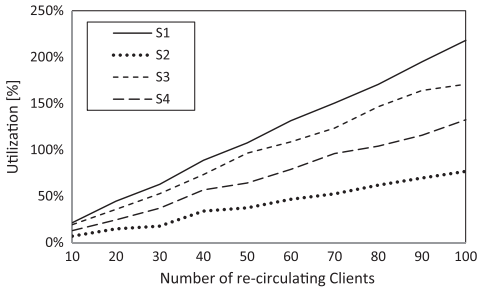
DOI: 10.1002/cpe



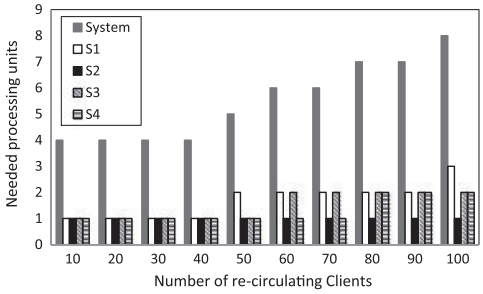

(a) Number of services provided by the processing system vs. the number of re-circulating clients




(b) Number of services provided by each station vs. the number of re-circulating clients




(c) Utilization of the service stations vs. the number of re-circulating clients




(d) Needed processing units vs. the number of re-circulating clients

Figure 6. CSM outcomes in the case of maximum parallelism.

constrained parallelism with possibly PU unavailability. All of this witnesses the estimation made in the first scenario about the number of processing resources needed for $K = 100$ clients was correct.

The response time of the whole system and that of the various service stations is now increasing with respect to the number of re-circulating clients (Figure 7(b)). This means that clients wait within stations before being processed.

This is also confirmed by Figure 7(c), which shows an increasing average size of the waiting queue at each service station as the number of re-circulating clients increases.

By analysing Figure 7(b) and (c), it results that the whole system is able to effectively handle a number of clients up to 80. After that value, the response time and the average waiting-queue size of the stations S2 and S3 suddenly increase. As a consequence, the quality of service of the entire system diminishes. Moreover, the Figure 7(b) and (c) suggest that stations S2 and S3 are a potential bottleneck for the system.

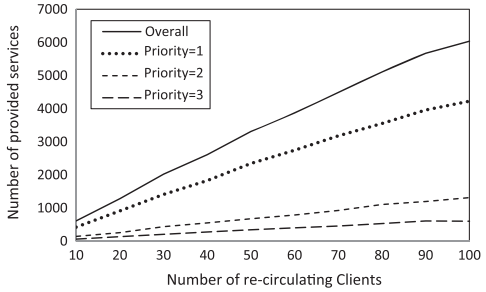
It is useful to observe that because of the fixed number of PUs, in this scenario, clients' priority implies action pre-emptions thus affecting system evolution. Figure 7(d) depicts the number of pre-emptions experienced by the different categories of client priority within the service station S1. Figure 7(d) confirms high priority clients are never pre-empted.

A greater value of the observed response time, with respect to the first scenario, suggested the introduction of an index of quality of service based on the sojourn time of clients in the system. In particular, because in the case of maximum parallelism, a sojourn (service) time of about 470 t.u. was estimated for the whole system, in the current scenario, a *soft* deadline for the clients was added equal to 500 t.u.

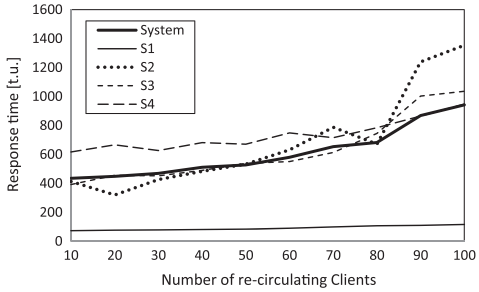
Figure 7(e) and (f) shows respectively the number of clients missing their deadline and the average time with which such deadlines were missed. The number of clients missing the deadline remains almost the same despite client priority, whereas the delay, a deadline is missed, becomes unacceptable for clients having the lowest priority as the number of re-circulating client goes over 50. The delay suddenly increases with $K > 80$. This value too confirms (Figure 7(b) and (c)) that the overall system seems inappropriate for managing more than 80 clients.



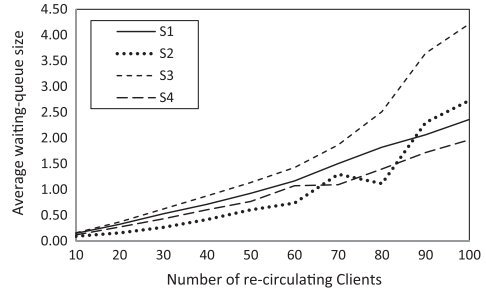
DOI: 10.1002/cpe



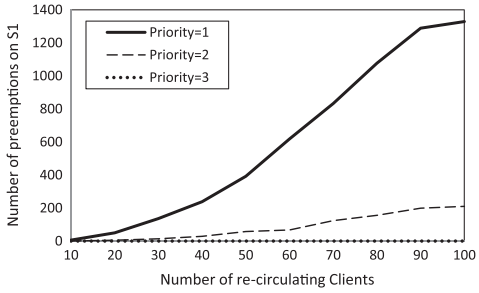

(a) Number of services provided by the processing system vs. the number of re-circulating clients



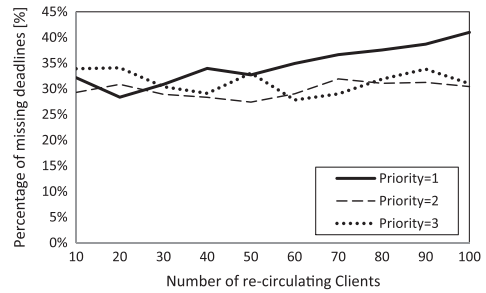

(b) Response time vs. the number of re-circulating clients



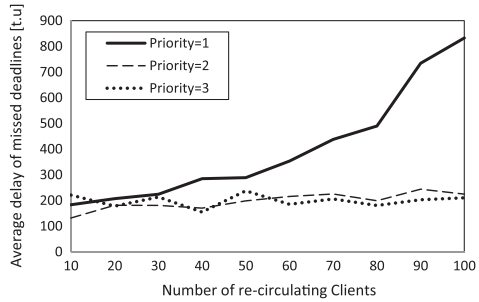

(c) Average waiting-queue size vs. the number of re-circulating clients




(d) Number of pre-emptions experienced by clients on station S1 vs. the number of re-circulating clients




(e) Percentage of missed deadlines with respect to provided services grouped by priority vs. the number of re-circulating clients




(f) Average time of missed deadline vs. the number of re-circulating clients

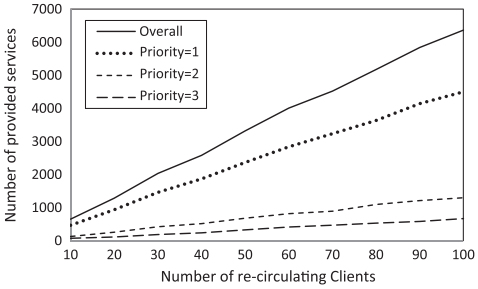
Figure 7. CSM outcomes in the case of constrained parallelism.

*8.3.3. Scenario 3: System level behaviour with incremental-accuracy actions..* The goal here was to try to improve the system performance in the case the allocations of the processing units are left unchanged with respect to the second scenario. Performance optimization was studied by exploiting incremental actions. More precisely, `SimulatedActions` are replaced by `IncrementalSimulatedActions` with a computational threshold (Section 6) $IA_{th}$ set to the 75% of the nominal execution time $IA_{ne}$. This percentage was used for experimentation purposes.

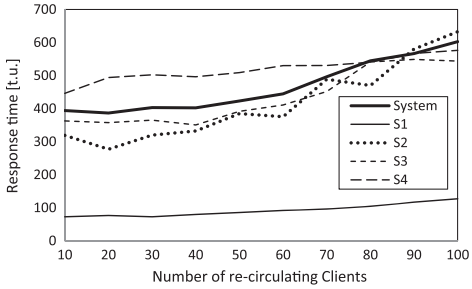
By cross referencing Figures 7(a) and (a), it emerges that the number of provided service remains almost the same in both Scenarios 2 and 3. What is significantly reduced is instead the waiting time (Figure 8(b)) and the average size of the waiting queues (Figure 8(c)); thus, mirroring the incremental actions is able to improve the quality of service of the overall system. By comparing Figure 8(b) with Figure 7(b), it results the pick behaviour in the response time when S2 disappears, and the response time of the system decreases of about 25%. From Figure 8(c) compared with Figure 7(c), it emerges that the average waiting-queue size of S3 is halved and that of S2 reduces of about 60%. It appears that the bottleneck character of S2 and S3 is decreased.



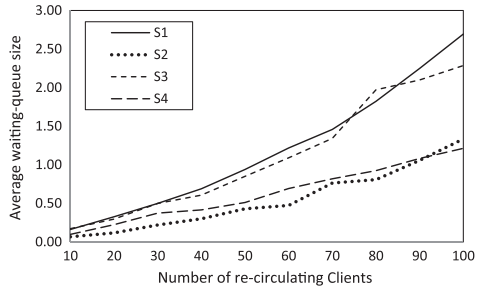
DOI: 10.1002/cpe



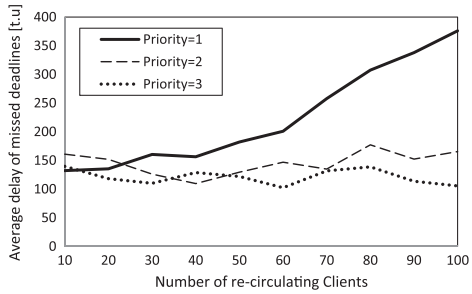

(a) Number of services provided by the processing system vs. the number of re-circulating clients



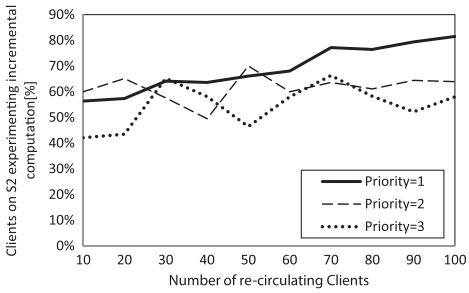

(b) Response times vs. the number of processing units




(c) Average waiting-queue size vs. the number of re-circulating clients




(d) Average delay of missed deadline vs. the number of re-circulating clients




(e) Percentage of the number of clients in station S2 which experiments the effect of incremental action computations

Figure 8. CSM outcomes in the case of incremental computation.

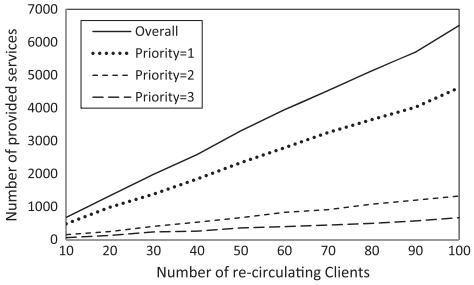
As expected, in this scenario, the average delay of missed deadlines notably reduces (compare Figures 7(f) and 8(d)). More in particular, the highest reduction occurs for clients having the lowest priority. In the case of $K = 100$ clients, the reduction is of about 54%. The average number of missed deadlines, although, remains almost the same with respect to the Scenario 2. Indeed, an action duration is actually shortened only when it would complete beyond the deadline. All of this, paired with clients with traverse multiple stations as occurs in the CSM, can make it difficult to control deadline fulfilment.

To give an idea of how many times the incremental computation was actually exploited, Figure 8(e) depicts the percentage of clients in the station S2 for which the incremental action behaviour is concretely used. Figure 8(e) confirms that low priority clients require incremental actions most of the time. However, also the other categories of clients in about 50% of the cases make recourse to incremental computation.

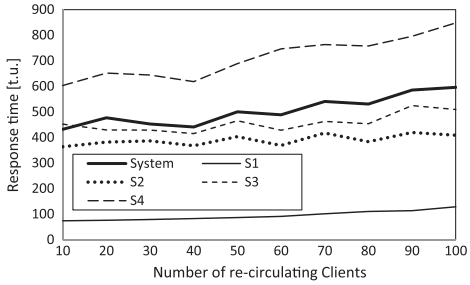
*8.3.4. Scenario 4: System level behaviour with augmented constrained parallelism.* In the Scenario 3, system performance optimization was investigated by using imprecise-accuracy actions. In this scenario, the same goal is pursued without imprecise actions but by adding instead further



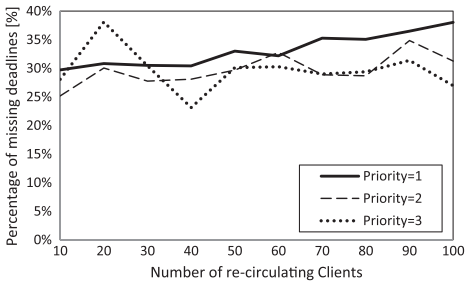
DOI: 10.1002/cpe



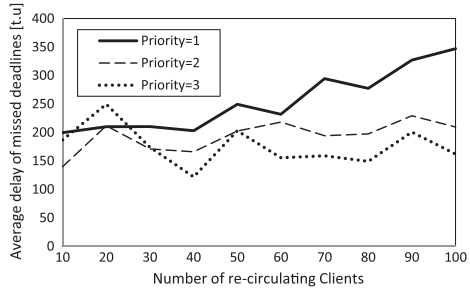

(a) Number of services provided by the processing system vs. the number of re-circulating clients




(b) Response times vs. the number of processing units




(c) Percentage of missed deadline with respect to provided services grouped by priority vs. the number of re-circulating clients




(d) Average delay of missed deadline vs. the number of re-circulating clients

Figure 9. CSM outcomes in the case of augmented constrained parallelism.

processing units to the CSM. Because Stations S2 and S3 emerged, in the second scenario, as a source of bottleneck for the system, in the current scenario, one extra PU was added to S2 and one for S3. Figure 9(a) shows the number of provided services of the system in the new configuration. By comparing Figures 7(a) and 9(a), it emerges that the number of provided services does not increase. This fact complies with previous results achieved with maximum parallelism hypothesis. However, the newly added resources are capable of improving the systems quality of service. This can be confirmed by comparing Figure 9(b) with Figure 7(b). As one can see, the response time of the system and of its component stations notably diminishes. Moreover, the response times augment almost linearly as the value of $K$ increases. This means that system bottlenecks were correctly identified in the second scenario and now are removed. For completeness, in the current scenario, the number of missed deadlines and the time delay of missed deadlines were checked too (Figure 9(c) and (d)). By comparing Figure 7(e) with Figure 9(c), it emerges that the number of missed deadlines for clients of priority 2 and 3 is almost the same. For the lowest priority clients, instead, the additional PUs cause a little improvement for $K = 100$ where a reduction of the number of missed deadlines of about 10% occurs. The same behaviour arises when the time delay of missed deadlines is considered. By comparing Figure 7(f) with Figure 9(d), a more significant improvement is observed for the lowest priority clients, which have a reduction of about 56%.

### *8.4. Preliminary execution of the Central Station Model system*

After property analysis based on simulation, the CSM model was tested according to preliminary execution in real time. Preliminary execution was devoted to assess the quality of fulfilment of timing constraints by measuring the amount of deviation by which actions and time-constrained messages are executed out of their due time. More in particular, in the considered case study, the goal was to evaluate the computational load related to action and message management (considering both message processing and transmission overhead due to communication delay). Effective actions



DOI: 10.1002/cpe

with the sleep-waiting strategy (Section 6) were adopted, which reproduce action durations without introducing execution costs.

Preliminary execution refers to the CSM configuration used during analysis where constrained parallelism and incremental computation of actions were used. As for the simulation phase, the model was partitioned into two JADE containers: the first one (main container) hosts all the stations except S1, which is allocated to the second container.

The time tolerance `EPS` was set to 750 ms, and the simulated incremental actions were substituted by incremental effective actions. Incremental effective actions have, as for the simulation context, a computational threshold $IA_{th}$ set to 75% of its nominal execution time $IA_{ne}$. The `DRealTime` control machine was used, and the `SPreemptivePUs` were turned to `EPreemptivePUs`. All the other entities of the CSM model remained unchanged. Simulation time units were converted into seconds, and a real execution time of $8.64 \times 10^4$ s (24 h) was considered.

Two cases were taken into account. In the first one, the actor model was executed on a single multicore machine. In the second one, the two containers were allocated on two distinct workstations connected through our Department LAN with Internet traffic. In the latter case, the Dimension 4 software tool [39] was used to keep aligned the clocks of the two exploited computational nodes.

The time interval between two consecutive time synchronizations among the physical nodes was set to 30 s, which experimentally proved sufficient to ensure an acceptable accuracy of clock alignment while introducing a negligible overhead. Preliminary execution was carried out on Win 7 platforms having 12 GB, Intel Core i7, 3.50 GHz, 4 cores with hyper-threading.

Figure 10 shows the distribution of the time deviation during the preliminary execution upon a single multi core machine. The maximum time deviations observed in the two containers are respectively 312 ms for the container holding all the stations except S1, and 141 ms for the other container.

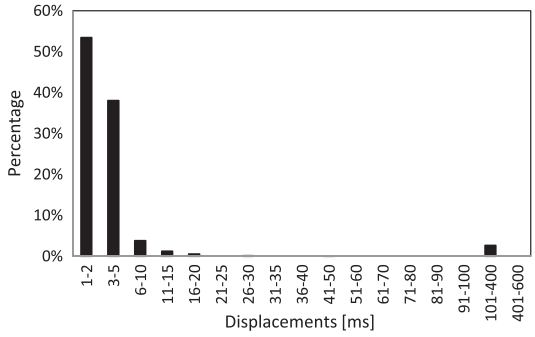
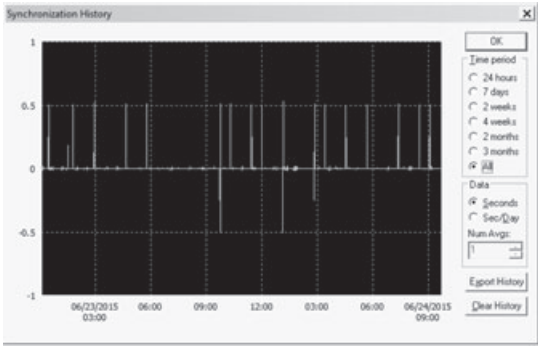
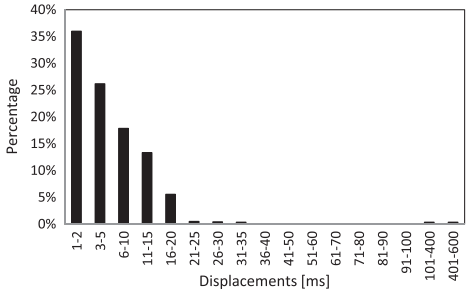



Figure 10. Percentage of measured time deviations during preliminary execution on a multicore machine (incremental computations).




(a) Clock disalignments experienced by the two workstations as measured by Dimension 4




(b) Percentage of measured time deviations (incremental computations)

Figure 11. Preliminary execution of the CSM system in a LAN context.



DOI: 10.1002/cpe

Figure 11(a) and (b), instead, refers to the distributed scenario. In particular, Figure 11(a) shows the history of the observed time misalignments between the two computational nodes. The time drifts are small except for a few peaks of about $\pm 500$ ms. Figure 11(b) portrays the distributions of the time deviations. The maximum time deviations in the two containers are respectively 506 ms for the container holding all the stations except S1, and 156 ms for the other container.

By cross-referencing Figures 10 and 11(b), it emerges that, as expected, the presence of the local area network and clock misalignments increases time deviations. In any case, although, such deviations are small enough and remain below the `EPS`; thus, confirming the overhead of message processing and action management is acceptable both in the multicore and the distributed scenario.

## 9. CONCLUSIONS

This paper proposes a control framework for developing time-dependent multi-agent systems where control issues are managed as pluggable cross-cutting aspects. A major goal of the infrastructure is the support of an application lifecycle, which fosters *model continuity*, that is, the use of a same model for both analysis purposes (through parallel/distributed simulation) and real-time execution. Some flexible modelling entities were designed to capture both the business logic of an application and the external elements of the environment where the application runs. A key factor of the proposed approach is that the evolution of a realized system is an emerging property resulting from the interaction of the application model and its environment, mediated by the provided control forms. Current version of the framework was prototyped using JADE.

The approach is demonstrated by a case study concerned with a closed queuing network where model continuity is practically experimented. Prosecution of the research is aimed to the following:

- porting the implementation of the framework to the THEATRE infrastructure [5–7, 17, 43] which is a minimal and efficient middleware for actors with a customizable transport layer;
- specializing the control framework toward real-time scheduling (e.g. [44]) and real time Java;
- extending the developed control forms in order to include pluggable recovery strategies to cope with the missing of action deadlines or the reaching of system unsafe states;
- applying the approach to time-constrained workflow modelling, analysis and enactment; and
- experimenting the framework in such domains as symbiotic simulations, cyber physical systems, mixed reality and Internet of things.

## REFERENCES

1. Varela C, Agha G. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices* 2001; **36**(12):20–34.
2. Wooldridge M. *An Introduction to Multi-agent Systems* (2nd edn). John Wiley & Sons: New York, NY, USA, 2009.
3. Hu X, Zeigler B. Model continuity in the design of dynamic distributed real-time systems. *IEEE Transactions on Systems, Man and Cybernetics, Part A* 2005; **6**(35):867–878.
4. Cicirelli F, Nigro L. Control aspects in multiagent systems. In *Intelligent Agents in Data-intensive Computing*, vol. 14, Kolodziej J, Correia L, Molina J (eds)., chap. 2. Springer, 2016; 27–50.
5. 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. of SCS* 2009; **85**(1):17–32.
6. Cicirelli F, Giordano A, Nigro L. Efficient environment management for distributed simulation of large-scale situated multi-agent systems. *Concurrency and Computation: Practice and Experience* 2015; **27**(3):610–632.
7. Cicirelli F, Furfaro A, Nigro L. Modelling and simulation of complex manufacturing systems using statechart-based actors. *Simulation Modelling Practice and Theory* 2011; **19**(2):685–703.
8. Bellifemine F, Caire G, Greenwood D. *Developing Multi-agent Systems with JADE*. John Wiley & Sons: New York, NY, USA, 2007.
9. Jade. (Available from: http://jade.tilab.com) [Accessed on December 2015].
10. Fipa. *Foundation for Intelligent Physical Agents*. (Available from: http://www.fipa.org.) [Accessed on December 2015].
11. Cicirelli F, Nigro L. A control framework for model continuity in JADE. *Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '14, IEEE Computer Society, Washington, DC, USA, 2014; 97–104.



DOI: 10.1002/cpe

12. Al-Zinati M, Araujo F, Kuiper D, Valente J, Wenkstern RZ. DIVAs 4.0: A multi-agent based simulation framework. *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '13, IEEE Computer Society, Washington, DC, USA, 2013; 105–114.
13. Davila J, Gomez E, Laffaille K, Tucci K, Uzcategui M. Multiagent distributed simulation with GALATEA. *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT '05, IEEE Computer Society*, Washington, DC, USA, 2005; 165–170.
14. Zeigler BP, Praehofer H, Kim TG. *Theory of Modeling and Simulation* (2nd edn). Academic Press: New York, 2000.
15. Cabani A, Pecuchet JP, Itmi M. Distributed multiagent simulation on P2P architecture. *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '07, IEEE Computer Society, Washington, DC, USA, 2007; 76–79.
16. Zehe D, Aydt H, Lees M, Knoll A. Javascript distributed agent based discrete event simulation. *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '13, IEEE Computer Society, Washington, DC, USA, 2013; 21–29.
17. Cicirelli F, Furfaro A, Giordano A, Nigro L. Parallel simulation of multi-agent systems using terracotta. *Proceedings of the 2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '10, IEEE Computer Society, Washington, DC, USA, 2010; 219–222.
18. Arunachalan B, Light J. Agent-based mobile middleware architecture (AMMA) for patient-care clinical data messaging using wireless networks. *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, IEEE Computer Society, Washington, DC, USA, 2008; 326–329.
19. Lotzmann U, Wimmer MA. Evidence traces for multi-agent declarative rule-based policy simulation. *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '13, IEEE Computer Society, Washington, DC, USA, 2013; 115–122.
20. Craenen B, Murgatroyd P, Theodoropoulos G, Gaffney V, Suryanarayanan V. Mwgrid: A system for distributed agent-based simulation in the digital humanities. *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '12, IEEE Computer Society, Washington, DC, USA, 2012; 124–131.
21. Hu X, Zeigler B. An integrated modeling and simulation methodology for intelligent systems design and testing. *Proceedings of Performance Metrics for Intelligent Systems Workshop*, Gaithersburg, MD, USA, 2002; 379–386.
22. Hong JS, Song HS, Kim TG, Park KH. A real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems* 1997; **7**(4):355–375.
23. Hu X, Zeigler BP. Model continuity to support software development for distributed robotic systems: A team formation example. *J. Intell. Robotics Syst* 2004; **39**(1):71–87.
24. Hu X, Zeigler BP. A simulation-based virtual environment to study cooperative robotic systems. *Integrated Computer-Aided Engineering* 2005; **12**:353–367.
25. Fujimoto R. *Parallel and Distributed Simulation Systems*. John Wiley: New York, NY, USA, 2000.
26. Moen D, Pullen J. Enabling real-time distributed virtual simulation over the internet using host-based overlay multicast. *Proceedings of the 2003 IEEE/ACM 7th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '03, IEEE Computer Society, Washington, DC, USA, 2003; 30–36.
27. Fanchao Z, Turner SJ, Aydt H. Symbiotic simulation control in supply chain of lubricant additive industry. *13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, Singapore, October 25-28, 2009; 165–172.
28. Peusaari J, Ikonen J, Porras J. Distribution issues in real-time interactive simulation. *13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, Singapore, October 25-28, 2009; 227–230.
29. Hu X. From virtual to real - a progressive simulation-based design framework. In *Discrete-Event Modeling and Simulation: Theory and Applications*, Wainer GA, Mosterman PJ (eds). CRC Press, 2010; 271–289.
30. Hu X, Azarnasab E. Progressive simulation-based design for networked real-time embedded systems. In *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, Popovici K, Mosterman PJ (eds)., chap. 7. CRC Press, 2012; 181–198.
31. Moallemi M, Wainer G. Modeling and simulation-driven development of embedded real-time systems. *Simulation Modelling Practice and Theory* 2013; **38**:115 –131.
32. Sarjoughian HS, Gholami S, Jackson T. Interacting real-time simulation models and reactive computational-physical systems. *Proceedings of the Interacting real-time simulation Winter Simulation Conference: Simulation: Making Decisions in a Complex World*, WSC '13, IEEE Press, Piscataway, NJ, USA, 2013; 1120–1131.
33. Giese H, Burmester S. Real-time statechart semantics. *Technical Report*, TR-RI-03-239 University of Paderborn, 2003.
34. Kiczales G, Lamping J, Mendhekar A, Maeda C. Cristina. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*: LNCS 1241, Jyväskylä, Finland, 1997; 220–242.
35. Carzaniga A, Picco G, Vigna G. Agent.GUI: a multi-agent based simulation framework. *Proceedings of the Federated Conference on Computer Science And Information Systems (FedCSIS'11)*, Szczecin, Poland, 2011; 623–630.
36. Gianni D, Loukas G, Gelenbe E. A simulation framework for the investigation of adaptive behaviours in largely populated building evacuation scenarios. *Proceedings of International Workshop on Organised Adaptation in Multi-Agent Systems (OAMAS)*, Estoril, Portugal, 2008; 1–15.
37. Wang F, Turner S, Wang L. Agent communication in distributed simulation. *Proceedings of International Conference on Multi-agent and Multi-agent-based Simulation*, Utrecht, The Netherlands, 2005; 11–24.



DOI: 10.1002/cpe

38. Pawlaszyk D, Strassburger S. A synchronization protocol for distributed agent-based simulations with constrained optimism. *Proceedings of European Simulation and Modelling Conference (ESM 2009)*, Leicester, United Kingdom, 2009; 337–341.
39. Dimension 4. (Available from: http://www.thinkman.com/dimension4/) [Accessed on December 2015].
40. Schmid U. Synchronized UTC for distributed real-time systems. *Annual Review in Automatic Programming* 1994; **18**:101–107.
41. Leick A. *GPS Satellite Surveying*. John Wiley: New York, NY, USA, 2004.
42. Laganà D, Legato P, Pisacane O, Vocaturo F. Solving simulation optimization problems on grid computing systems. *Parallel Computing* 2006; **32**(9):688 –700.
43. Cicirelli F, Nigro L. An agent framework for high performance simulations over multi-core clusters. In *Communications in Computer and Information Science, CCIS*, Vol. 402, Tan G, Yeo G, Turner S, Teo Y (eds). Springer: Berlin Heidelberg, 2013; 49–60.
44. Halang WA. Load adaptive dynamic scheduling of tasks with hard deadlines useful for industrial applications. *Computing* 1992; **47**(3):199–213.



DOI: 10.1002/cpe