DisSimJADE: A framework for the development of Agent-based Distributed Simulation Systems

Daniele Gianni
Computing Laboratory
Oxford University
Oxford, UK
gianni@comlab.ox.ac.uk

Andrea D'Ambrogio and Giuseppe Iazeolla
Dept. of Computer Science
University of Rome TorVergata
Rome, Italy
{dambro,iazeolla}@info.uniroma2.it

ABSTRACT

The adoption of an agent-based approach that incorporates intelligence, adaptation and learning abilities has proved to significantly increase the realism and the accuracy of the simulation. Simulation systems of such a kind, however, require computational resources that might be considerable for a single agent, so to become unfeasible when the number of simulated agents scales up. A distributed environment is thus needed to allow the execution of such simulation systems, particularly in the case of scenarios populated by a large number of agents. Building an agent-based distributed simulation system, however, requires both specific expertise and knowledge of distributed simulation standards and a non-negligible amount of effort to develop ad-hoc components. This paper introduces a simulation framework named DisSimJADE, which enables the incorporation of distributed simulation facilities into existing agent-based systems. DisSimJADE is built on top of the popular agent-based framework JADE and allows to define agent-based simulation systems that can be transparently executed either in a local or distributed, therefore bringing significant savings in terms of effort and development time. In addition, DisSimJADE provides a uniform interface to the JADE framework, which further facilitates the production of distributed simulation systems to developers of JADE-based multi-agent systems.

Categories and Subject Descriptors

D.2.13 Software Reusability, D.2.10 Design, D.3.2 Language Classification, D.3.3 Language Constructs and Features, I.2.11 Distributed Artificial Intelligence, I.6.5 Model Development, I.6.7 Simulation Support Systems, I.6.8 Discrete Event, I.6.8 Distributed, I.6.2 Simulation Language.

General Terms

Design, Experimentation, Languages.

SIMUTools '09, Rome, Italy

Keywords

Discrete event simulation, Distributed Simulation, Agent-based Simulation, Framework, JADE, HLA

1. INTRODUCTION

Physical systems are often composed of autonomous, interacting, possibly intelligent entities that cooperate, compete and carry out tasks to achieve individual or collective goals [1]. When simulating such systems, an agent-based modeling approach offers an effective conceptualization paradigm that easily allows to capturing the interactions and the individual/collective intelligence that such systems exhibit. The incorporation of sophisticated intelligence often requires computational resources, in terms of memory for the data representation and CPU cycles for the reasoning rules or criteria, that are often not available on a single host. The use of distributed execution environments can be seen as a solution to the problem of guaranteeing the needed accuracy and efficiency when largely populated scenarios are to be simulated [2]. On the other hand, developing a simulator in a distributed environment requires specialized know-how that goes far beyond the agent-based modelling techniques. In addition, acquiring such knowledge is a considerable initial investment that can prevent the adoption of such techniques.

In this paper, we face the problem of making easier the development of distributed agent-based simulation systems. To this purpose, the paper introduces DisSimJADE, a framework that makes transparent the development of distributed agent-based simulation systems by raising the agent-based developer from all the concerns of the local or distributed simulation environment. At the same time, DisSimJADE provides a simulated agent container that can also be used to host conventional agent components.

Therefore, the benefits of DisSimJADE are amplified by combining the effortless development of distributed simulation systems with the incorporation of distributed simulation facilities into existing agent-based frameworks. In such a setting, DisSimJADE allows developers of multi-agent systems to easily produce distributed versions of agent-based simulation systems with a very limited effort and without being required to gain specific knowledge about distributed simulation standards and implementations.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

To this purpose:

(a) DisSimJADE is built on top of the popular agent-based framework JADE [3] and provides a uniform interface with it, both in local and distributed environments

(b) DisSimJADE has been integrated into SimArch, a layered simulation architecture that allows to define simulation systems that can be transparently executed either in a local or distributed environment [4].

A side benefit of point (a) is that DisSimJADE is compliant with the FIPA specifications [4], as in the case of JADE, while point (b) provides a uniform approach to develop agent-based simulation systems without explicit knowledge about the execution environment (local or distributed) and the specific distributed simulation infrastructure (e.g., HLA).

The paper is organized as follows. Section 2 points out this work contribution compared to other state-of-art works, while Section 3 introduces the technologies upon which DisSimJADE has been built (i.e., JADE and SimArch). Section 4 gives a detailed description of the proposed framework and, finally, Section 5 illustrates an example scenario of use.

2. RELATED WORK

DisSimJade provides the following two main innovative contributions:

(i) the incorporation of distributed simulation facilities into existing agent-based frameworks;

(ii) the effortless development of distributed simulation systems as a transparent extension of the corresponding conventional (i.e., centralized) simulation system.

As regards contribution **(i)**, DisSimJADE can be compared to similar works, such as SIM_AGENT [6], the Time-Extension for MAS [7] and JADE-HLA [8].

SIM_AGENT provides a framework to develop agent-based modeling and simulation systems. It differs from DisSimJADE because it does not formulate the simulation in terms of agent-based system and binds the reasoning, planning, etc. mechanisms to the framework. Differently, DisSimJADE deals only with the issues related to the simulation, and therefore allows the use of JADE-compliant frameworks currently available (e.g.: Jess rule engine [9] or JADEx [10]).

The Time-Extension shares with DisSimJADE the partial objective of bringing the simulation time concept into agent-based systems. However, there are considerable differences. First of all, DisSimJADE has a wider scope. It presents a formulation of *discrete-event simulation* (*DES*) systems as agent-based systems. Secondly, the Time-Extension uses innovative aspect-oriented methodologies [11] to bring the duration concept within agent-based systems. Differently, DisSimJADE shows how conventional object-oriented techniques can effectively support this through the mere application of the *Decorator Pattern* concept [12]. Thirdly, the Time-Extension mechanism introduces some discontinuities between an agent-based system and the corresponding simulated agent-based system. They are due to the use of the aspect oriented technology, which is specifically used to produce the simulated agent-based system, and to the non-encapsulation of the implicit wall-clock time concept that agents have.

Both SIM_AGENT and the Time-Extension do not deal with distributed simulation of agent-based systems, as instead DisSimJADE does.

A contribution that, similarly to DisSimJADE, provides distributed simulation facilities is JADE-HLA, which is built on top of JADE and makes us of the *High Level Architecture* (*HLA*) distributed simulation standard [13]. However, the following differences between JADE-HLA and DisSimJADE can be found:

- DiSSimJADE adopts a general DES modelling approach, and therefore is not related to any specific distributed simulation standard;
- DisSimJADE implements an agent-based conceptualisation of DES systems;
- DisSimJADE is compliant with the JADE design outline, and therefore enables JADA developers to easily carry out agent-based modeling and simulation activities.

With respect to contribution **(ii)**, i.e., the effortless development of distributed simulation systems as transparent extension of the corresponding conventional – local – simulation system, DisSimJADE can be compared to works carried out in the distributed simulation community, such as PDNS [14], DisSimJava [15], DEVS/HLA [16], OSA [17] and JAMES [18].

All such works provide valuable contributions in the field of distributed simulation, but fail to address point (i), which refers to the issue of incorporating distributed simulation facilities into agent-based frameworks.

Therefore, in this paper case, the benefits of contribution (ii) are amplified by combination to contribution (i), which is the application of (ii) to the development of distributed agent-based simulation system. In such a setting, DisSimJade allows developers of multi-agent systems to easily produce distributed versions of agent-based simulation systems with a very limited effort and without being required to gain specific knowledge about distributed simulation standards and implementations.

The distributed simulation of multi-agent systems using HLA as underlying platform has already been targeted in [19]. This paper contribution however differs from the above one since it is not a methodology to produce distributed agent-based simulation systems but only a method to effortlessly incorporate distributed simulation facilities into existing agent-based frameworks, as results from the combination of contributions (i) and (ii).

Moreover, the proposed approach does not pretend to give an answer to overcome the pitfalls of agent-based systems outlined by Jennings and Wooldridge [1]. Indeed, the approach is only intended to reuse existing agent-based systems (e.g., JADE-based systems) into distributed simulation contexts.

3. BACKGROUND

The following sub-sections introduce the JADE framework and the SimArch software architecture, respectively.

3.1 JADE

JADE [3] is a Java-based framework for the implementation of agent-based systems. It provides a base element, the *agent*, which maintains an internal state and whose dynamics can be configured through a set of pluggable behaviours. Each behaviour consists of

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

a sequence of internal operations and interactions with other agents, or other behaviours, which can be composed according to several constructs (e.g. parallel, serial, etc.).

The fundamental JADE aspect is the communication [20], which is carried out according to FIPA specifications [4] through an asynchronous mailbox-based mechanism. As FIPA defines, JADE messages are composed of the following attributes: sender, list of recipients, performative action, content, content language reference, content ontology reference, and a set of minor fields to control concurrent conversations. Besides attributes of immediate understanding, the message contains a performative action attribute, and two references to the content coding language and to the shared ontology, which needs further details.

The *performative action* attribute specifies the type of communication, which has been classified by FIPA into twenty-five different communicative acts. For example, it can be of value REQUEST when the sender agent asks for a service request to the recipient agents, or can be of value INFORM in the case of a "notification" of state change.

Concerning the reference attributes to the content *language* and content *ontology*, they provide the information needed to decode and interpret the semantics of the content field, respectively. JADE ontologies are in turn to be built on top of the basic ontology, which provides basic concepts for primitive data types, and can define three types of elements [20]: predicates, concepts, and actions.

Predicates represent facts in the modelled world, and can be true or false. *Concepts* represent complex data structures, which are composed of standard simple types like String, Integer, etc., while *actions* are a specialization of concepts that are internally associated to the actions performed by agents.

3.2 SimArch

SimArch is a software architecture that offers a layered view of simulation systems. Figure 1 illustrates the four layers, whose detailed description is given in [4].

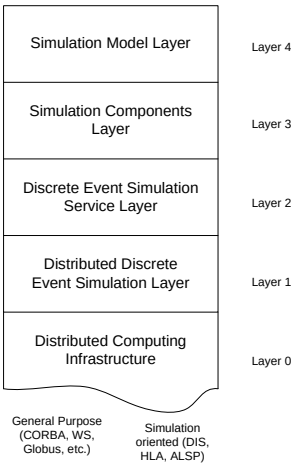

Figure 1 SimArch's layered architecture [4]

Layer 4 is the top layer where the simulation model is defined through the invocation of the simulation language primitives.

The primitives' implementation, i.e., the components' simulation logic and the model configuration services, are provided by *Layer 3*, while *Layer 2* deals with the transparent synchronization and communication among simulation components, for both local and distributed execution. The distributed version of this layer uses in turn *Layer 1* to achieve global time synchronization and to provide communication with the remote simulation components.

Finally, *Layer 1* provides a DES (discrete-event simulation) abstraction [21], such as *sendEvent*, *waitNextDistributedEvent* and *waitNextDistributedEventBeforeTime*, on top of the distributed computing infrastructure conventionally identified by *Layer 0*. Such bottom layer does not belong to SimArch and therefore the interfaces between Layers 1 and 0 are not defined. In the case of a *HLA*-based implementation of Layer 1, such interfaces are subsets of the *RTI-Ambassador* and *FederateAmbassador* interfaces for the communication between Layers 1 and 0 and between Layers 0 and 1, respectively.

The communications between the layers are bidirectional and the provided interfaces have to be implemented to successfully use the available layers implementations. For example, when using Layer 1, the Layer1ToLayer2 interface is to be implemented and its implementation is to be provided as reference to Layer 1. In the specific case, Layer1ToLayer2 interface intercepts the distributed events and takes care of scheduling a proper handler in the local event list.

4. DisSimJADE

DisSimJADE is a Java framework for agent-based modelling and simulation. It is built on top of JADE and structured according to SimArch. Distinctive features of DisSimJADE are the compliance with the FIPA specifications [4], inherited from JADE, and the integration with *HLA*, given by *SimArch*.
The framework is implemented with the objective of simplifying the development of distributed agent-based simulation systems. Specifically, DisSimJADE aims to:

1. making the development of agent-based simulation systems similar to the development of conventional agent-based systems;
2. enabling the transparent execution of agent-based simulation systems either in a local or a distributed environment.

To achieve objective 1, DisSimJADE introduces a set of software components that conform to the JADE and FIPA standards and that can encapsulate conventional JADE components, while objective 2 is achieved by integrating DisSimJADE into the SimArch software architecture. Specifically, DisSimJADE uses the Layer 1 provided by SimArch and implements the SimArch Layer 2 interface.
In particular, the DisSimJADE framework consists of the following components:

- a simulation ontology;
- a simulation agent society and a set of agents;
- an interaction protocol;
- a set of simulation behaviours;
- a set of simulation event handlers.

The simulation ontology, named *DES-Ontology* and illustrated in Section 4.1, defines the semantic base for the communications among the simulation agents. It consists of DES *concepts*

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

(simulation time) and *actions* (DES and simulation life cycle management services), and allows the incorporation of any other JADE ontology thus enabling the reuse of standard agent-based components.

The *simulation agent society*, illustrated in Section 4.2, is structured hierarchically and is based on two types of simulation agents, the *simulation entity agent* and the *simulation engine agent*, with the former encapsulating the simulation logic, i.e. the sequence of states and DES service requests, and the latter managing the agents. The society defines which agents (types and names) can be part of the simulation execution. DisSimJADE defines local societies, which are composed of a specified number of simulation entity agents and are managed by a locally running simulation engine agent, and a global society, which interconnects the local societies. A local society can be run in isolation, in case of local simulation execution, or can be interconnected with other societies, in case of distributed simulation execution.

The *interaction protocol*, illustrated in Section 4.3, defines the communication rules between agents belonging to the same society. Due to the hierarchical structure of the society, the communication takes place only between the entity agents and the engine. The distributed execution extends the interaction protocol for the local version by transparently masking the synchronization and communication issues behind SimArch and HLA services, which are out of entity agents visibility.

The *simulation behaviours* define the actions taken by both types of agents in response to the reception of any of the DES-Ontology action, by implementing the interaction protocols. They conform to the JADE interfaces and can encapsulate standard JADE behaviours.

The *simulation event handlers* define the routines that must be locally processed by the engine agent to deal with the scheduled requests, such as wake up or event notification. They can be considered as support components that are visible to the engine only.

4.1 DES-Ontology

The *DES-Ontology* extends the JADE standard ontology [19] introducing concepts and actions that characterize the simulation domain. The *concepts* are related to the simulation time, while the *actions* are related to the interaction between simulation entities and simulation engines.

As regards *concepts*, the DES-ontology defines two different representations of the simulation time: *AbsoluteSimulationTime*, for absolute values of the simulation time; and *RelativeSimulationTime*, for relative values of the simulation time, with “relative” having default semantics “with respect to the current time”. The two concepts are related by the fact that the AbsoluteSimulationTime is given by the sum of the current AbsoluteSimulationTime and the RelativeSimulationTime. Nevertheless, the definition of a relative time concept is included in the ontology because it is a parameter required by several DES services.

As regards *actions*, the ontology defines *simulation management services* and *DES services*.

A *simulation management service* defines an action that manages the simulation life cycle, i.e.,:

- *register agent*: to request to join a simulation society;
- *registration successful*: to acknowledge the acceptance of a registration request;
- *remove agent*: to resign from the society;
- *move agent*: to move the agent to another society;
- *simulation end*: to notify that the society objective has been reached.

The actions *register agent* and *remove agent*, which are both of performative type REQUEST, have no attributes because the action object, i.e. the name of the agent requesting the action, can be inferred from the message envelope.
The *move agent* action is of performative type REQUEST and is characterized by the name of the recipient engine where the agent is to be started with the initial state (also provided).
The actions *registration successful* and *simulation end*, which are both of performative type INFORM, inlcude an instance of *AbsoluteSimulationTime* that specifies either the simulation start time (in case of *registration successful* action) or the simulation end time (in case of *simulation end* action).

The *DES services* define actions of the following types:

- *conditional hold time*: to request an hold for a given simulated time, under the condition that no event notifications are received;
- *hold time*: to request an unconditional hold for a specified simulated time;
- *notify time*: to inform that the specified time has been reached;
- *notify message*: to inform that the specified event was requested to be scheduled for the receiving agent, at the current time;
- *send message*: to request the delivery of the specified event at the specified time to another simulation entity agent;
- *wait message*: to request a wake up when a simulation message is to be notified.

The *conditional hold time* and *hold time* actions, which are both of performative type REQUEST, are characterized by a relative simulation time that specifies the simulation sleep time.
The *notify time* action, which is of performative type INFORM, informs the receiving agent of the absolute simulation time reached. The *notify message* action, which instead notifies a message, is described by the following four attributes: sender agent, recipient agent, message and time. The first three attributes are of type *String*, while the fourth is of type *AbsoluteSimulationTime*.
The s*end message* action is complementary to the *notify message* action. It is described by the same attributes, but it is of performative type REQUEST. In the specific case, to maintain a logical uniformity with the common practice in DES, the time is of type *RelativeSimulationTime*.
Finally, the *wait message* action, which is of performative type REQUEST, informs the engine that the sender agent is blocked and waiting for new messages.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

With the exception of the *move agent* action, all the actions are indifferently used by the entity agents either with the a local or a distributed engine agent.

4.2 Simulation Agents

A simulation agents' society is populated by two types of agents: the *simulation entity agents* and the *simulation engine agents*.
The simulation entity agents incorporate the simulation logic by use of custom simulation behaviours, while a simulation engine agent is in charge of coordinating the society, and therefore includes a list of the simulation events and a record of the society composition, as detailed in the following sub-sections.

4.2.1 Simulation entity agent

Figure 2 describes the state diagram that defines the lifecycle of a simulation entity agent. The states in the diagram of Figure 2 are simulation states built on top of the standard states of a JADE agent [19] and are transparently integrated with them.
The state diagram of a simulation entity agent looks similar to the state diagram of a conventional DES simulation and therefore this section only focuses on the differences, while additional details on the rest of the diagram can be found in [22].
The changes introduced by the state diagram of a simulation entity agent concern the *Waiting for Registration Acknowledge* state and the *Mobility* state. In the former, the simulation engine collects the registration requests and checks when the society is ready to execute the simulation. In the latter, the agent forwards the request to the engine and terminates the life cycle. These differences are due to the decentralised and dynamic nature of the agent-based simulation framework, which differently from a conventional DES framework allows the creation and termination of logic processes.

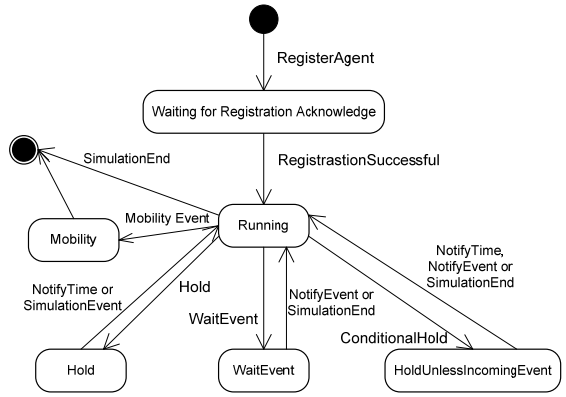

Figure 2 State diagram of the simulation entity agent

To implement the above described dynamics, the entity's behaviour is configured as a serial composition of the *RegisterAgentBehaviour* and *EntityMainCycleBehaviour* behaviours, with the latter to be configured according to the model specifications.
In order to allow the easy plugging of any conventional JADE behaviour into the *EntityMainCycleBehaviour* behaviour, the *simulation entity agent* interface must be consistent with the JADE agent standard interface. To achieve this, the *simulation entity agent* must therefore invoke the simulation actions *conditional hold time*, *hold time*, *send event*, and *wait event* by use of the JADE standard methods *blockingReceive(millisecs)*, *doWait()*, *send()*, and *blockingReceive()*, respectively.

4.2.2 Simulation engine agent

The *simulation engine agent* can be similarly described both for local and distributed engines. The distributed engine is indeed built by extending the local, which is therefore presented first in the following sub-section.

4.2.2.1 Local engine

Figure 3 describes the state diagram of the simulation local engine agent.
The local engine state diagram consists of a sequence of states that can be grouped in three phases, denoted as *Phase 0* through *Phase 2* in Figure 3.
Phase 0 is the registration phase that takes care of synchronizing the start-up phase through the *Waiting for Registration Requests* and *Confirm Registration Successful* states. In such a phase, the engine accepts incoming *register agent* requests while checking whether the simulation society becomes complete. Once the society is completed, the engine notifies the *registration successful* to all the registered agents. Such states are not present in a conventional DES framework because the entities registration is carried out through the static invocation of local methods at coding time. Similarly to the entity agent state diagram, the Phase 0 states originate from the inherent decentralised nature of the system.

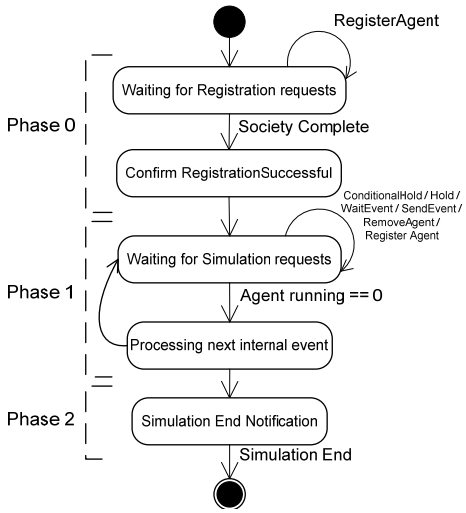

Figure 3 State diagram of the simulation local engine agent [24]

After completing this phase, the engine proceeds to the *Phase 1* that consists of the states *Waiting for Simulation Requests* and *Processing Internal Event*, which contribute to define the *EngineMain-CycleBehaviour*. Such behaviour executes the following algorithm:

```
While (numberOfRunningAgents > 0) {
   wait for a simulation message;
   Case of:
      SendEvent:  schedule sendEventHandler;
                  break;
```

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

```
        HoldTime:   numberOfRunningAgents--;
                    schedule wakeUpHandler;
                    break;
        WaitEvent:  numberOfRunningAgents--;
                    break;
        RemoveAgent:
                   numberOfRunningAgents--;
                    start RemoveAgentBehaviour;
                    break;
        ConditionalWaitEvent:
                    numberOfRunningAgents--;
                    schedule wakeUpHandler;
                    store agentName in conditionalList;
                    break;
        RegisterAgent:
                    numberOfRunningAgents++;
                    registerAgent;
                    start RegistrationSuccessfulBehaviour;
      } // end case
} // end while

If (eventsList.size() > 0) {
   nextEvent = eventsList.remove(0);
   nextEvent.process();
} else {
   setSimulationEnd();
}
```

The algorithm is composed of two main blocks: a *while* block for the requests collection at a given simulation time, and an *if-then-else* block to process the next scheduled event and advance the simulation time.

The algorithm is based on the following assumptions:

- the cardinality of the simulation society is known from the previous phase and stored in the local variable *numberOfRunningAgents*;
- the agents requesting *hold time*, *conditional hold* or *wait event* simulation services block their execution and do not process further requests until they receive proper simulation notifications.

The second assumption defines the *interaction protocol* between entity and engine agents, and guarantees that within the *if-then-else* block the actual number of running agents is zero, as verified through the value of the local variable *numberOfRunningAgents*.

The *while* block executes until there are running agents in the society. In this block, the activities follow a sequential wait-and-serve cycle that processes the requests by properly updating the *numberOfRunningAgents* variable and by performing the relevant action: either the scheduling of a new *event handler* in the list or the activation of a *simulation service behaviour*. As an example, upon receiving a *send event* request, the engine schedules a new local *SendEventHandler* with the proper data (recipient, time, message, etc.). Similarly, in case of *wait event* requests, the engine verifies that the requesting agent is blocked and will not proceed until a local event unblocks the agent.

The algorithm also manages the dynamic composition of the agent society by processing *register agent* and *remove agent* requests.

Once the *simulation end event* is reached, the engine stops and the *EngineMainCycleBehaviour* terminates.

After that the engine proceeds to the last phase, denoted as *Phase 2*, which includes the *Simulation End Notification* state. In such a phase, the engine notifies a *simulation end* message to the entire society before terminating its life cycle and removing itself from the agent container.

4.2.2.2 Distributed engine

The distributed simulation engine agent makes use of the JADE framework for the local interactions and uses *SimArch Layer 1* and *HLA* for the synchronization and communications among distributed entities, as illustrated in Figure 4.

The choice of not using JADE as distributed platform is motivated by the following considerations:

- SimArch and its HLA-based implementation allow the integration with other simulation systems developed by use of such technologies;
- the integration with SimArch allows to obtain a multi-paradigm (e.g. agent-based, process interaction, event scheduling, etc.) distributed simulation environment;
- HLA proves to perform better in terms of simulation workload compared to RMI-based communications between the JADE nodes [19];
- the implementation remains extremely simplified and conforms to a general reuse and integration trend currently observed in the software and simulation industry.

The distributed engine is compliant with the local engine for what concerns the interactions to be carried out with the simulation entity agents, which can be therefore deployed regardless the type of engine. Vice versa, the distributed engine deals with the following extra issues:

1. synchronization and communication between local and distributed environment;
2. agent mobility between simulators;
3. handling of distributed events in the framework.

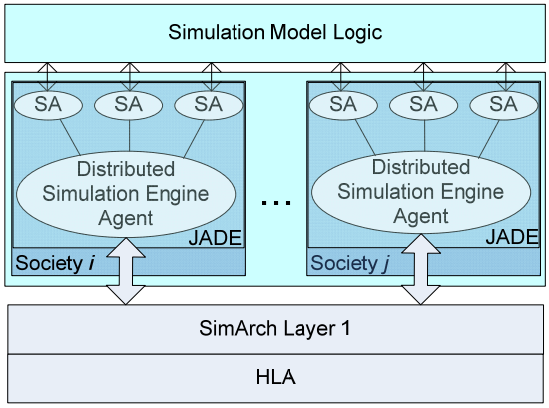

Figure 4 DisSimJADE architecture

The synchronization and communication concern the consistency between the local and distributed environments with the addition of event delivery to agents running on remote simulators. The agent mobility allows simulation time-stamped transfer of an agent from a given simulator to a remote one. All such

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

functionalities require that new event handlers are introduced to properly processing such requests.

The engine deals with the above issues using SimArch Layer 1 in conjunction with:

- An improved life cycle and algorithm;
- Mobility Event;
- Distributed event handlers, for which interested users are sent to [4].

The *agent life cycle* is described by the diagram in Figure 5. It consists of five phases, denoted as Phase 0 through Phase 4. Phase 0 is the initialization of the distributed environment and proceeds as illustrated in [4][22]. *Phase 1* is shared with the local version; and similarly *Phase 2* (Simulation main cycle), which however presents two significant differences.

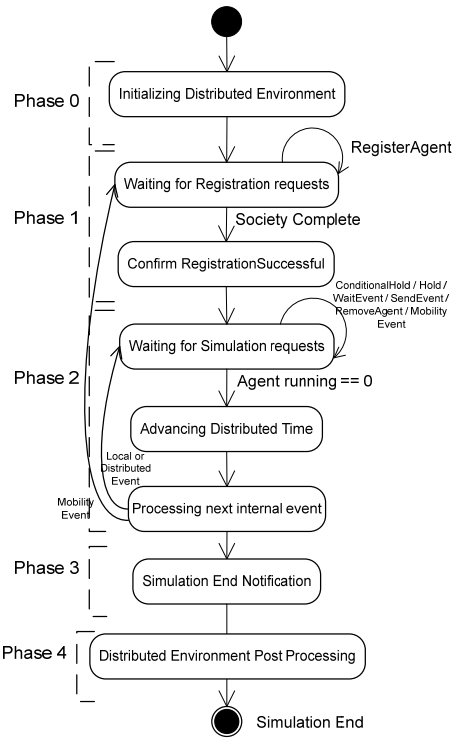

Figure 5 State diagram of the simulation distributed engine agent

The *first* is that the local events cannot be processed before advancing the distributed time, which is carried out within the *Advancing Distributed Time* state. The transition from this state to the next state only occurs when either the time has been granted or a distributed event has been received. In this last case, the distributed event is transparently scheduled as a local event by SimArch layer 1.

The *second* difference is that, when an agent is moving to the simulator, the simulation has to temporarily block until the agent is loaded up and joins the local simulation society. In case of an agent mobility event, the society composition is updated and the society complete condition is no longer satisfied. This corresponds to the transition from the *processing next internal event* state to the *waiting for registration requests* state of Phase 1 in Figure 5.

Phase 3 follows the main processing cycle and concerns the notification of the simulation end event to all the local agents. It is activated by Phase 2 when receiving the corresponding event from the distributed environment.

Phase 4 concludes the engine life cycle restoring the distributed environment set up. It follows the operations specified in [4] and [22].

The distributed engine life cycle is implemented similarly to the locale engine one, with the addition of the advancement of the distributed time and the transition from the *processing next internal event* state to the *waiting for registration requests* state. The adapted algorithm is as follows:

```
While (numberOfRunningAgents > 0) {
   wait for a simulation message;
   Case of:
      SendEvent:  schedule sendEventHandler;
                  break;
      HoldTime:   numberOfRunningAgents--;
                  schedule wakeUpHandler;
                  break;
      WaitEvent:  numberOfRunningAgents--;
                  break;
      RemoveAgent:
                  numberOfRunningAgents--;
                  start RemoveAgentBehaviour;
                  break;
      ConditionalWaitEvent:
                  numberOfRunningAgents--;
                  schedule wakeUpHandler;
                  store agentName in conditionalList;
                  break;
      RegisterAgent:
                  numberOfRunningAgents++;
                  registerAgent;
                  start RegistrationSuccessfulBehaviour;
      MoveAgent:
                  forward mobility request to recipient simulator;
                  numberOfRunningAgents--;
                  start RemoveAgentBehaviour;
                  break;

      } // end case
} // end while

If (eventsList.size() > 0) {
   nextEvent = eventsList.read(0);
   waitNextDistributedEventBeforeTime(nextEvent.time);
   nextEvent = eventsList.remove(0);
   nextEvent.process();
} else {
   waitNextDistributedEvent();
   nextEvent = eventsList.remove(0);
   nextEvent.process();
}
```

where the **bold** text denotes the changes with respect to the algorithm of the local engine. The handling of a mobility event

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

request consists of the forwarding of the request to the remote simulator, the update of the society cardinality and finally the removal of the agent from the society.

When all the agents have been executed and are blocked waiting for a response from the engine, differently from the local engine, the distributed version proceeds by checking the size of the events list. If the list contains at least one event, the distributed engine will verify if events are available in the distributed environment before the next local event, by use of a call to the SimArch service *waitNextDistributedEvent*. Once the service call returns, the next local event is processed and one or more agents will be reactivated, as in the local version. The case of the events list empty is slightly different. In such a case, the service *waitNextDistributedEvent* is to be invoked. Similarly, when the service call returns, the next local event is to be retrieved from the list and processed.

The *mobility event* logic, which is triggered in response to a *Move Agent* request, is to be implemented by accepting the incoming agent and locally removing the migrated agent. This is achieved using *SimArch Layer 1*. To use this layer, the engine has to implement the Layer 1 to Layer 2 interface to allow the reception of the distributed simulation events. The interface implementation includes the code to carry out the transition from the *processing next internal event* state to the *waiting for registration requests* state.

In case of a standard event, e.g., a communication between two agents, the event will be scheduled as an internal event that specifies the remote sender. Differently, in case of an incoming agent, the engine state variable denoting the society cardinality is properly updated to include the incoming agent. The engine will then block the processing and will wait for the registration request to proceed on. Concurrent requests will still be collected by the engine, but none of them will be actually processed because of the extra running agent not yet in a blocked state. Such request collection does not compromise the validity of the algorithm because it does not allow any agents to run over the current local simulation time.
The discrimination between a standard event and a mobility event is to be specified when sending the *RemoveAgent* event. This can be easily achieved with SimArch Layer 1 service *sendEven*t, which allows the specification of an event tag that discriminates the type of event. By setting a different value for both types, the discrimination becomes trivial for the receiving engine.

The distributed execution requires that event handlers are introduced to specifically deal with distributed events. They are similar to the handlers introduced within the SimArch software architecture and are presented in the following subsection.

4.3 Interaction Protocol

The interaction protocol defines the rules upon which the conversation between the agents takes place, e.g. which agent talks, which listens, which expects what. It can be distinguished in intra-society protocol and inter-society protocol. The former takes place for the communications in a local environment, both in the case of local and distributed simulation. Differently, the latter is used in the distributed environment only and involves agents, either engines or entities, which are running within different societies.

The *intra-society protocol* is used between the entity agents and the engine agent to request and acknowledge the simulation actions defined in the *DES-Ontology*. It is based on the blocking and non-blocking properties of the simulation services. On the entity agent side, the action requests such as register agent, wait time or hold time need that the agent interrupts its execution until given proper conditions are met. Such conditions are monitored by the engine agent, which has the entire view of the society and the agents' request and which activates the individual agents by responding to their request. For the correct execution of the simulation it is fundamental that the entity agents are aware and respect such protocol.

The *inter-society protocol* complements the intra-society rules when operating in a distributed environment. The distributed engine implements such a protocol in addition to the intra-society one and therefore can immediately replace the local engine without modifying the simulation entity agents. The inter-society protocol defines two types of interactions: the sending of an event to a remote entity agent, and the mobility of an agent on a remote society.

The *sending of an event to a remote entity agent* occurs when a local entity agent requests the delivery of a message to a specified entity agent. The engine collects the request and verifies if the recipient is running locally or on a remote machine. In both cases, the intra-society protocol is applied for the interaction between the engine and the entity agent. In the case of a distributed recipient, the protocol assumes that the engine forwards the request to the remote agent before continuing the local processing. The communication between the two engines is obtained by SimArch and HLA and therefore is not compliant to the FIPA standard. However, such an approach brings several advantages – as shown above, and does not affect the peculiarities of the local interaction, which is still FIPA compliant.

The *agent mobility* is based on a similar approach but is more complex. Figure 6 shows an example of agent mobility with the actors of this phase and the sequences of steps. Besides the entity agent and the engines of the source and destination sites, another agent supports this action. It is the *resource manager*, which is in charge of starting the agent on the remote site. The presence of this agent is essential because to guarantee the proper application specific initialization typical of an agent start-up.

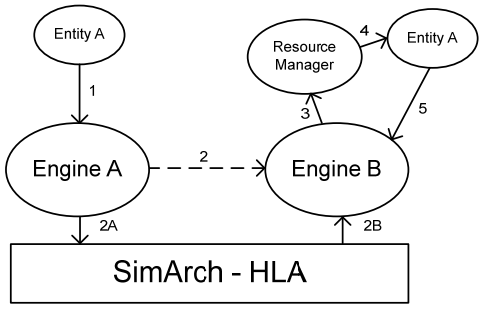

Figure 6 Example of agent mobility

Let's assume that Agent A in Figure 6 wants to move from Society A to Society B at simulation time t. It first sends a JADE-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

compliant *mobility request* to engine A (step 1). The request consists of a simulation event to be delivered to the remote Resource Manager and an attached serialization of agent state. Engine A sends the event to engine B by specifying that the event is of type mobility (step 2). At the specified time t, SimArch and HLA deliver the event to engine B (steps 2A and 2B), which processes the event by updating the society composition and delivering the event to the Resource Manager (step 3), as initially specified by Entity A. Differently from a conventional event, the delivery and the processing of the mobility event does not allow Engine B to continue. The local society on site B is now incomplete and engine B cannot proceed until it receives the request of joining the society. After having operated the initialization of the agent parameters, the Resource Manager activates entity A agent with the provided state (step 4). Once running, the agent first requests to join the local society and after proper acknowledge starts its simulation cycle as at the first activation (step 5). Such mechanism guarantees that the mobility is operated transparently and in synchronization with the simulation clock, local and distributed.

5. EXAMPLE SCENARIO OF USE

A significant scenario for the application of DisSimJADE can be found within the domain of manufacturing system simulation. For the sake of simplicity, let us consider a simplified system where workers move around a manufacturing factory premises in order to reach the machines they need to use. In such a scenario, a significant aspect for the paper scope is represented by the movement of the workers. A possible space modelling for this system is represented by a graph whose nodes identify the possible positions, and whose edges represent the possible movements of workers between two positions. The nodes also represent physical resources that can exclusively be used by only one worker at a time, whereas the edges can simultaneously be traversed by more workers at the same time.

A possible agent-based modelling of such system could include two types of agents: a *ResourceManager* agent, which coordinates the access to the physical points; and a *Worker* agent, which is provided with a self-updated view of the world, a decision model, a motion model and a set of machines to use. A synthetic sequence diagram of a local simulation system for two worker agents and a resource manager is shown in Figure 7. The workers inform the ResourceManager when they reach the node and then wait for an authorization event. The ResourceManager authorizes the movement when the required node is free and delays authorizations when the node is busy. The diagram does not include the engine agent because it is not visible at system modelling level.

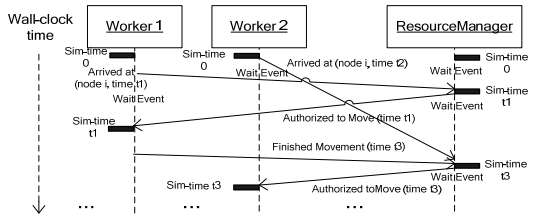

Figure 7 Example sequence diagram [24]

Assuming that the simulated space is vast and largely populated, it could be appropriate, if not necessary, to execute it in a distributed environment. This can be obtained by simply substituting the local engine with the distributed engine, for what concerns the framework, and by instructing the resource manager and the worker agents to support the application specific logic for the mobility event. This regards in particular three aspects, which are however related to the specific application:

1. the definition of the mobility trigger condition;
2. the serialization of the state to transfer on the remote simulator;
3. the deserialization of such state at the remote simulator.

The first aspect concerns the boundaries of the world simulated by each of the simulators. It can be easily addressed by specifying the ResourceManager name on each node. In such a way, the workers can easily determine whether the mobility triggering condition is true by checking that the resource manager name of the destination node differs from the name of the current node resource manager.

The second and the third aspects are standard operations in Java and many libraries support the automatic serialization of object on String, such as XStream String [25]. The control of the data to be serialized remains in charge of the application, however. Only at application level it is possible to determine which data has to be carried on and which is to be reconstructed on the destination simulator. In the shown application, the workers are provided with a detailed map of the local world, as in the local simulator, and a condensed representation of the remote world. This includes, for example, main gateways, stairs, as well connections between the simulated areas. To the purpose of demonstrating the framework, we locally stored constant data and reduced the mobility data. In particular, each resource manager maintains a copy of the world view from the local area and replicates for each of the incoming workers. At the same time, each worker brings with it only the specific parameters of the decision and motion model, in addition to the list of the machines to use. The mobility of the workers that reach a border node follows the procedure described in Section 4.3. The procedure and the functionalities of the framework are independent from the modelling characteristics of the workers and more accurate decisions and motion models can also be used. Their implementation is out of the scope of this paper and therefore not discussed here.

6. CONCLUSIONS

The adoption of an agent-based approach has proven effective when simulating complex scenarios consisting of a large number of autonomous and interacting entities. In such settings, it is often required to exploit distributed simulation to deal with the required scalability and accuracy characteristics.

This paper has introduced *DisSimJADE*, a simulation framework which provides a uniform approach to develop agent-based simulation systems that can be transparently executed either in a local or distributed environment.

The paper has described the several benefits that DisSimJADE provides with respect to comparable contributions. Most of such benefits come from the use of JADE as the underlying agent-based platform and from the integration with the SimArch simulation architecture, and can be summarized as follows:

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725

- the incorporation of distributed simulation facilities into conventional agent-based frameworks;
- the effortless development of distributed simulation systems as a transparent extension of the corresponding conventional (i.e., centralized) simulation system;
- the provision of a mobility facility to easily migrate simulation agents from a given simulation society to a remote simulation society.

An example scenario of use has also been illustrated to give the flavor of the effectiveness provided by the DisSimJADE framework.

7. ACKNOWLEDGMENTS

This research has been partially funded by the ALADDIN project, funded by BAE and EPSRC, by the euHeart project, funded by European Union FP7, by the FIRB project on "Software frameworks and technologies for the development and maintenance of open-source distributed simulation code", funded by the Italian Ministry of Research, and by the University of Roma TorVergata CERTIA Research Center.

8. REFERENCES

[1] N.R. Jennings, and M. Wooldridge, "Application of Intelligent Agents", *Agent technology: foundations, applications, and markets*, Springer-Verlag, 1998, pp. 3 – 28.

[2] R. Fujimoto, Parallel and Distributed Simulation Systems, Wiley (2000).

[3] JADE project home, http://jade.tilab.it,Telecom Italia.

[4] D. Gianni, A. D'Ambrogio and G. Iazeolla, "A Layered Architecture for the Model-driven Development of Distributed Simulators", *The First International Conference on Simulation Tools and Technologies (SIMUTOOLS08)*, Marseille, March, 2008.

[5] FIPA Specification, http://www.fipa.org.

[6] A. Sloman and B. Logan, "Building cognitively rich agents using the SIM_Agent toolkit", *Communication of the ACM*, vol. 42, n. 3, 1999, pp. 71 – 77.

[7] A. Helleboogh, T. Holvoet, D. Weyns, and Y. Berbers, "Extending Time Management Support for Multi-agent Systems". *Proceedings of the 2004 Workshop on Multi Agent Simulation and Multi Agent-based Systems*, LNCS 3415/2005, Springer Verlag, 2004, pp. 37 – 48.

[8] F. Wang, S.J. Turner, and L. Wang, "Agent Communication in Distributed Simulations", *Proceedings of the Multi-Agent and Multi-Agent-Based Simulation* (MABS 2004), Springer-Verlag, LNAI 3415, 2005, pp. 11–24.

[9] Jess Project http://www.jessrules.com.

[10] A. Pokahr, L. Braubach, and W. Lamersdorf, "JADEx: Implementing a BDI-Infrastructure for JADE Agents", *EXP - In Search of Innovation (Special Issue on JADE)*, vol 3, n. 3, Telecom Italia Lab, Turin, Italy, 2003, pp. 76-85.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming", *Proceedings of the European Conference on Object-Oriented Programming*, Vol. 1241, Springer-Verlag, 1997, pp. 220 – 242.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley (2000).

[13] IEEE 1516, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules.

[14] G.F. Riley, M.H. Ammar, R.M. Fujimoto, A. Park, K. Perumalla, and D. Xu, "A federated approach to distributed network simulation", *ACM Transaction on Modeling and Computer Simulation (TOMACS)*, Vol. 14 N. 2, April 2004.

[15] E.H. Page, R.L. Moose and S.P. Griffin, "Web-Based Simulation in SimJava using Remote Method Invocation", *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, pp 468-474, December 1997.

[16] B.P. Ziegler, G. Ball, H. Cho, J.S. Lee, and H. Sarjoughian, "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions", *Proceedings of the 1999 Simulation Interoperability Workshop (SIW99)*.

[17] O. Dalle, "The OSA Project: an Example of Component Based Software Engineering Techniques Applied to Simulation", *The 2007 Summer Computer Simulation Conference (SCSC'07)*, San Diego, USA, July 15–18, 2007.

[18] A.M. Uhrmacher and B. Schattenberg, "Agents in Discrete Event Simulation," *European Simulation Symposium (ESS98)*, 1998, pp. 129 – 136.

[19] M. Lees, B. Logan, G.K. Theodoropoulos, Distributed simulation of agent-based systems with HLA, *ACM Transaction on Modeling and Computer Simulation (TOMACS)*, vol. 17, n. 3, 2007.

[20] F. Bellifemine, G. Caire, and D. Greenwood, "Developing Multi-Agent Systems with JADE", Wiley (2007).

[21] Richard E. Nance, "The time and state relationships in simulation modeling", *Communications of the ACM*, vol. 24, n. 4, April 1981, pp. 173-179.

[22] D. Gianni and A. D'Ambrogio, "A Language to Enable Distributed Simulation of Extended Queueing Networks", *Journal of Computer*, Vol. 2, N. 4, July, 2007, Academy Publisher, pp. 76 – 86.

[23] I. Sommerville, Software Engineering, 7th ed., Addison Wesley (2007).

[24] D. Gianni, "Bringing Discrete Event Simulation Into Multi Agent Systems", 10th International Conference on Computer Modelling and Simulation, EuroSIM/UKSIM, Cambridge, April, 2008.

[25] XStream project home page, http://xstream.codehaus.org/.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5725
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5725