13

HLA-Transparent Distributed Simulation of Agent-based Systems

Daniele Gianni[1],
Andrea D'Ambrogio[2], Giuseppe Iazeolla[2] and Alessandra Pieroni [2]
[1]*Computing Laboratory, University of Oxford, United Kingdom*
[2]*Dept. of Computer Science, University of Rome TorVergata Italy*

Abstract
The adoption of the agent-based approach to the modelling and simulation of physical systems has proved to considerably increase the simulation realism and accuracy. Simulation systems of such a kind, however, require computational resources that might become unfeasible when the number of simulated agents scales up. A *Distributed Simulation (DS)* approach might improve the execution of such systems, particularly in the case of scenarios populated by a large number of agents. Building an agent-based DS system, however, requires both specific expertise and knowledge of distributed simulation standards and a non-negligible amount of effort with respect to conventional local simulation systems. In this book chapter, we introduce a simulation framework named *Transparent_DS* (TDS), which enables the use of distributed environments while making affordable the development of agent-based simulators. TDS raises developers from the specific issues of the distributed environment. By use of TDS, the simulation agents can be locally developed and tested, and then transparently deployed in a DS environment, bringing significant savings in terms of effort and development time. In addition, TDS provides a uniform interface to the JADE framework, which further facilitates the work of developers of JADE-based *Multi-Agent Systems* (MAS) in the production of agent-based DS systems. By the TDS approach, any HLA- and agent-based distributed simulation system can practically be developed as conventional local MAS, with no extra effort and no extra knowledge. An example development of a simulation system is presented which is a common abstraction in several domains that involve the motion of individuals in a multi-storey building to simulate operations in normal or emergency situations.

1. Introduction

Distributed Simulation (DS) gives three main advantages with respect to *Local Simulation* (LS), namely [Fujimoto]: 1) scalability of needed computational resources, 2) reusability of existing simulation systems, 3) composability of available simulation subsystems. These

features can be particularly exploited by agent-based simulation systems for at least three reasons:

1) the modelling of the agents (i.e.: the incorporation of intelligence, adaptation, and learning abilities into agents [Jennings]) requires computational capabilities that might become inadequate in a LS approach. Such capabilities can be scaled up in the DS approach by connecting various execution platforms;
2) a wide variety of agent-based software components is available and could be incorporated into an agent-based simulation system. Such components are however in many cases heterogeneous for implementation languages or platforms, and therefore their incorporation into an LS system might be problematic. This problem can be overcome by use of existing DS standards (e.g. *IEEE High Level Architecture (HLA)* [IEEE-HLA]) that give ways to interconnect heterogeneous systems;
3) by use of the same standards, various and heterogeneous LS subsystems can be aggregated into a unique DS system.

The use of DS environments, however, requires both specific expertise and knowledge of the DS standard and a non-negligible amount of effort to develop ad-hoc components, or to reuse existing ones, or else aggregating them. In this chapter, we address the problem of making the development of distributed agent-based simulators HLA-transparent. To this purpose, in this chapter we introduce the *Transparent_DS* (TDS) framework that easies the development of DS agent-based systems by raising the system developer from all the concerns of the HLA standard. Moreover, TDS is built on top of the popular *Multi-Agent System (MAS)* platform JADE [JADE] and provides a uniform interface with it, both in LS and DS environments.

The chapter is organized as follows. Section 2 presents the related work and outlines the TDS contribution with respect to state-of-the-art works. Section 3 is the background section that recalls concepts and terminology of HLA and of JADE. Section 4 introduces the TDS framework, and Section 5 gives an example case study which is a common abstraction in several domains that involve the motion of individuals in a multi-storey building to simulate operations in normal or emergency situations.

2. Related work

The TDS framework aims to make the development of distributed agent-based simulation systems HLA-transparent. The framework also introduces two main innovations:

(i). the incorporation of DS facilities into existing agent-based frameworks;
(ii). the effortless development of DS systems by transparent extension of conventional LS systems.

Current state-of-the-art systems that relate to TDS can be identified in: the *Agent-based DEVS/HLA* system [Sarjoughian], the *JAMES* system [Uhrmacher], the *HLA_AGENT* system [Lees], and the *JADE-HLA* system [Wang].
The *Agent-based DEVS/HLA* shares TDS's objectives and peculiarities, but differs considerably in the adopted solutions. *Agent-based DEVS/HLA* aims to introduce a layered

architecture that can indifferently execute simulated agents either in local or distributed environment. This system is based on seven layers and includes the agent reasoning layers besides the simulation layers. TDS, on the other hand, aims to 1) make the development of DS agent-based systems effortless by transparently extending conventional LS systems; and to 2) make transparent the development of the simulation system once the agent system is available. Although objectives 1 and 2 are also achieved by *Agent-based DEVS/HLA*, the way these objectives are achieved is considerably different from the TDS one. TDS integrates with widely adopted and standard MAS frameworks, rather than introducing new ones. By this approach, TDS gains several advantages, such as the availability for immediate reuse of agent adds-on, such as agent reasoning and planning [Pokahr], agent inference [Jess], or agent ontologies [Ontology-Bean] – all of which can be transparently and effortlessly incorporated into the DS agent-based.

The *JAMES* system provides a similar framework using the DEVS formalism in a Java-based environment. This system differs from the TDS for similar reasons to the ones already mentioned for *Agent-based DEVS/HLA*. In particular, *JAMES* uses platforms and formalisms that are not commonly adopted by agent-based systems. Differently, TDS uses JADE, a widely adopted MAS framework, and inherits all the add-ons already available for popular agent-based systems.

The *HLA_AGENT* system also provides a framework to develop agent-based simulation systems, and mainly gives a distributed extension of the *SIM_AGENT* framework [Sloman]. This system differs from TDS in two ways: 1) the use of the HLA standard is not made transparent; and 2) the reasoning, the planning, and the other agent peculiarities are directly incorporated into the simulation framework.

The system that, similarly to TDS, provides distributed simulation facilities and integrates with JADE is *JADE-HLA*. Despite of these similarities, TDS presents the following differences:

- TDS uses SimArch [Gianni08a] and makes the use of HLA completely transparent;
- TDS adopts a general *Discrete Event Simulation* (DES) modelling approach, and thus is not tied to any specific distributed simulation standard;
- TDS implements an agent-based conceptualisation of DES systems [Gianni08b];
- TDS is compliant with the JADE design outline, and therefore enables JADE developers to easily carry out agent-based modeling and simulation activities.

3. Background

3.1 High Level Architecture

The *High Level Architecture* (HLA) is an IEEE standard [IEEE-1516] that provides a general framework within which software developers can structure and describe simulation applications. The standard promotes interoperability and reusability of simulation components in different contexts, and is based on the following concepts [Kuhl]:

- *Federate*, which is a simulation program and represents the unit of reuse in HLA;
- *Federation*, which is a distributed simulation execution composed of a set of federates;
- *Run Time Infrastructure (RTI)*, which is the simulation oriented middleware that provides the services to build federates. The middleware consists in *RTI*

Locals, which reside on each federate site, and a *RTI Executive*, which is deployed on a central server.

The standard is defined by four documents:

- *1516 HLA Rules*, which govern the behaviour of both the federation and the federates [IEEE-HLAA];
- *1516.1 Interface Specification*, which defines both the RTI - federate (*RTIAmbassador*) and the federate - RTI (*FederateAmbassador*) interfaces [IEEE-HLA1];
- *1516.2 Object Model Template*, which defines the formats for documenting HLA simulations [IEEE-HLA2];
- *1516.3 Federate Execution and DEvelopment Process*, which provides a reference process for the development of HLA simulations [IEEE-HLA3].

The major improvement that HLA brings in with respect to its predecessors is an API-oriented development of distributed simulation systems. Compared to Protocol-oriented techniques, this approach raises developers from all the concerns related to the communication and synchronization of the distributed simulation systems. Despite of this improvement, the HLA approach still suffers from a considerable drawback. This derives from the set of service which is rather complex. These services consist of a wide set of generic simulation services, beyond the scope of *Discrete Event Simulation (DES)*, that ranges from the management of the simulation life cycle to advanced updating techniques. Furthermore, the services concern only with the distributed environment and a considerable effort is always needed to develop the synchronization and communication logic between the local and the distributed environment. As a consequence, a considerable extra effort is necessary when using HLA to develop a DS system (compared to an equivalent LS one).

3.2 JADE

JADE [Bellifemmine] is a popular Java-based framework for the implementation of *Multi-Agent Systems* (MAS). JADE's base element is the agent, which maintains an internal state and whose dynamics is described through a set of pluggable behaviours. The behaviour is a sequence of internal operations and communication acts with other agents, and it is possibly composed of sub-behaviours according to several composition structures (e.g. parallel or serial).

The most fundamental JADE aspect is the communication, which is carried out according to the FIPA specifications [FIPA] through an asynchronous mail-box 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 to be further detailed.

The *performative action* attribute specifies the type of communication, which has been classified by FIPA into twenty-five different communication acts. For example, this attribute 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 "notification" of state change.

Concerning the reference attributes to the *content language* and to the *content ontology*, these provide the recipient agents with the information needed to decode and interpret the semantics of the content field, respectively. JADE ontologies are in turn built on top of the basic ontology, which provides basic concepts for primitive data types, and can define three types of elements: 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 and Integer. Finally, *actions* define special concepts that are internally associated to the actions an agent can perform.

4. The TDS Framework

The *Transparent_DS (TDS)* framework consists of a set of software libraries that can be configured and used to transparently implement HLA- and agent-based DS systems. The framework achieves this by integrating the *SimArch* layered approach [Gianni08, Gianni07, D'Ambrogio08] with the JADE framework. *SimArch* is a software architecture that structures simulation systems in four layers, which separate the program (coded in a high level simulation language) from the HLA-based distributed simulation infrastructure (denoted as layer 0, and not belonging to the architecture). The layers numbering proceeds bottom-up, from layer 1 to layer 4 as follows [Gianni08a]:

Layer 4: the simulation model program
Layer 3: the simulation language implementation
Layer 2: the execution container (i.e., the simulation engines)
Layer 1: the Distributed Discrete Event Simulation (DDES) abstraction
Layer 0: the distributed simulation infrastructure (HLA in this work)

In conventional development scenarios, the primitives of the distributed simulation standard (e.g. HLA) are used to code the simulation model. By use of SimArch, differently, a DS system can be developed as a conventional LS system, because developers are not to be concerned with the knowledge, the details and the practice of the specific distributed simulation infrastructure. Moreover, the developers are saved from the non-negligible efforts needed to implement the synchronization logic between the local and the distributed environment, which every distributed simulation infrastructure currently needs.
As a result, effort savings can be obtained of about 60% for beginners and of about 30% for experienced developers with respect to the development effort usually required by a DS system. Besides that, an additional saving can be recognized in an estimated 1.25 man_months per federate according to the SED effort model [Grable] when applied to the saving of about 3.5 Klines of code per federate [D'Ambrogio06a].
The TDS approach has the potential of obtaining similar results [Gianni09] and its architecture is illustrated in Fig.1, as resulting from the integration of the 4 layers structure of *SimArch* and of the JADE framework, as better described in next section.

4.1 Description of the TDS Layers

Each TDS layer has a well defined scope and its interfaces are for communication with the adjacent layers only. The interfaces and the communication protocols are defined independently from the layers internal implementation so that they are completely decoupled and can be replaced by custom layer implementations without any extra rework.

This is essential in many scenarios, for example when porting a simulation system onto a different distributed simulation infrastructure, when running the same simulation model in a different distributed environment, or when needed to meet specified performance requirements.

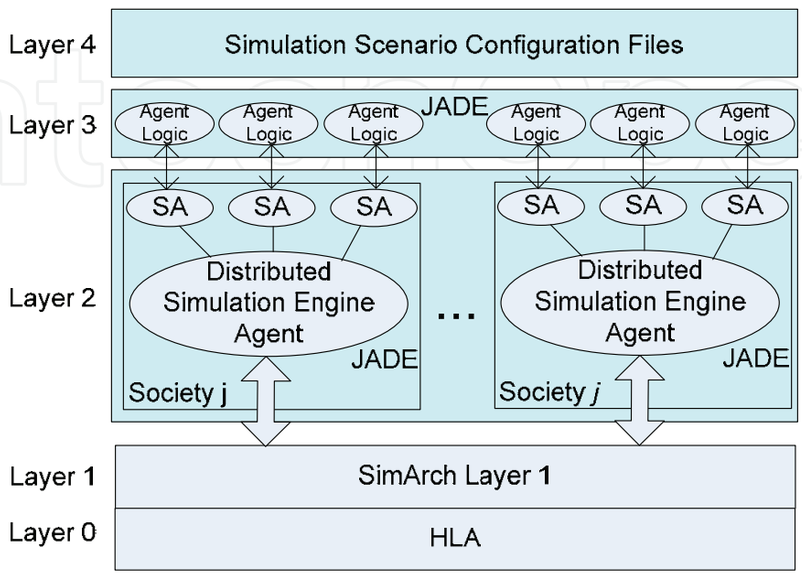


Fig. 1. The TDS architecture

The bottom layer, *Layer 1*, provides general synchronization and communication services in the distributed environment. These include *sendEvent*, *waitNextDistributedEvent* and *waitNextDistributedEventBeforeTime*, on top of the distributed simulation infrastructure conventionally identified by *Layer 0*, which is HLA in the current implementation, with the corresponding *RTIAmbassador* and *FederateAmbassador* communication interfaces, for communication from Layer 1 to 0 and from Layer 0 to 1, respectively. The current implementation of Layer 1 is coded on top of HLA, and thus benefits of the above mentioned reduced effort when developing HLA-based simulation systems because of the higher abstraction level, the higher components reusability, and the reduced necessary know-how that is obtained by this approach. Layer 1 also incorporates the *Federation Manager* [Kuhl], which manages the federation life cycle. The manager first creates the federation, then waits for all the federates to join and finally starts the simulation. When the run ends, the manager also coordinates the resigning of the federates. The role of the Federation Manager is to guarantee the DES-based execution throughout all the federation life cycle. The reader may refer to [Gianni07, D'Ambrogio06a] for further details on this layer implementation.

Layer 2 is the core of TDS as it makes transparent the actual execution environment, either local or distributed. The framework provides the execution container for the agents at Layer 3 and is based on the JADE framework to achieve component integration for the reuse of conventional JADE components. By these features, TDS provides an agent-based formulation of discrete event simulation systems and inherits all the compliance with the

FIPA standard and JADE framework. In addition, TDS operates transparently either in local or in distributed environments, with the latter being implemented on top of Layer 1, by thus obtaining the full transparent use of the distributed simulation standard (HLA in the current implementation). Section 4.2 gives an in-depth description of Layer 2.

Layer 3 deals with the implementation of the agents, the building blocks for the description of simulation scenarios. The agents are domain-specific and can be customarily developed to satisfy the modelling requirements of the specific application. To provide an application example for the TDS framework, Section 5 gives an example of a set of agents that model the movement of human on a space area represented as a graph. This scenario is a common abstraction in several domains that involve motion of individuals in a multi-storey building to simulate operations in normal or emergency situations.

Layer 4 is the top layer, where the simulation model is defined by configuring the simulation scenario, in other words the number and types of the agents involved in the simulation, and the definition of each individual agent. In the current implementation, this is obtained with XML files describing the number and types of agents, in addition to parameters of the Layer 3 individual agents. Currently, the set of agents available at Layer 3 allows the specification of main features such as the world model, the decision model, the motion model, and the health model.

4.2. The core of TDS

The core of the TDS framework is represented by Layer 2. This layer has the objective of making the development of HLA-distributed agent-based simulation systems transparent by raising the abstraction level offered by the underlying Layer 1 to a uniform level for both local and distributed agent-based systems. This is obtained by providing communication and synchronization services that conform to the JADE interfaces (thus making transparent the development of the simulation system) and that are specialized for the local and distributed environments. To this purpose, the TDS framework introduces components such as [Gianni08b]:

- a simulation ontology;
- a simulation agent society and a set of simulation agents (SAs in Fig. 1);
- an interaction protocol;
- a set of simulation behaviours;
- a set of simulation event handlers.

The simulation ontology, named *DES-Ontology* and illustrated in Section 4.2.1, defines the semantic base for the communications among the simulation agents. The ontology consists of DES *concepts* (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 4.2.2, 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. The TDS 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 the case of local simulation executions, or can be interconnected with other societies, in the case of distributed simulation executions.

The *interaction protocol*, illustrated in Section 4.2.3, 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 not visible to the entity agents.

The *simulation behaviours* define the actions taken by both types of agents in response to the reception of any of the DES-Ontology actions, by implementing the interaction protocols. These behaviours conform to the JADE interfaces and can encapsulate standard JADE behaviours. The reader can find details of this in [Gianni08b].

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. The handlers can be considered as support components that are visible to the engine only. The reader can refer to [D'Ambrogio06] for further details.

4.2.1 DES-Ontology

The *DES-Ontology* extends the JADE standard ontology [Bellifemmine] 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: *Absolute Simulation Time*, for absolute values of the simulation time; and *Relative Simulation Time*, for relative values of the simulation time, with "relative" to be intended as "with respect to the current time". The two concepts are related by the fact that the Absolute Simulation Time is given by adding up the current Absolute Simulation Time and the Relative Simulation Time. Nevertheless, the definition of a relative time concept is included in the ontology, being a parameter required by several DES services.

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

A *simulation management service* defines actions to manage the simulation life cycle [Gianni08b], namely:

- *register agent*: to request joining a simulation society;
- *registration successful*: to acknowledge the acceptance of a registration request;
- *remove agent*: to resign from a 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 in which 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, include an instance of *Absolute Simulation Time* 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;
- *move agent*: to request the transfer of the agent to a different simulation engine
- *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 *move agent* action represents a performative type REQUEST that a simulation entity agent can submit to a simulation engine. 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 *Absolute Simulation Time*. The s*end message* action is the complement of the *notify message* action. This action 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 *Relative Simulation Time*. 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.

All the actions are indifferently used by the entity agents either with a local or a distributed engine agent, with the exception of the *move agent* action, which is accepted and processed only by an engine operating in a distributed environment.

4.2.2 Simulation Agents

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

Simulation entity agent

Fig. 2 illustrates the state diagram that defines the lifecycle of a simulation entity agent. The states are simulation states built on top of the standard states of a JADE agent [Bellifemmine] 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 entity, and therefore we only focuses on the differences between the two diagram – additional details can be found in [D'Ambrogio06]. Such differences concern the *Waiting for Registration Acknowledgement* state and the *Mobility* state. In the former, the simulation engine collects the registration requests and checks whether the society is ready to execute the simulation. In the latter, the agent forwards the request to the engine and terminates the life cycle. The introduction of these states is 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 logical processes.

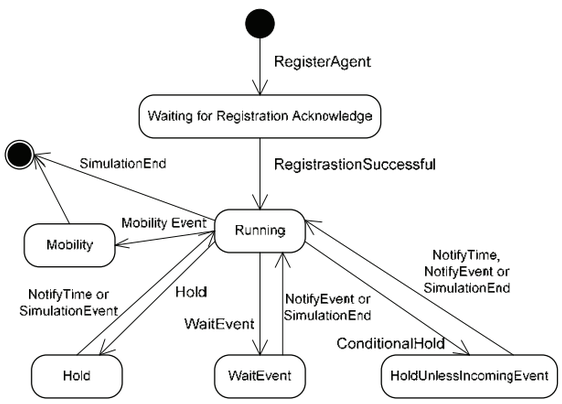


Fig. 2. State diagram of the simulation entity agent [Gianni09]

To implement such dynamics, the entity's behaviour is configured as a serial composition of the *Register Agent Behaviour* and the *Entity Main Cycle Behaviour*, with the latter to be configured according to the model specifications.
In order to allow the easy plug-in of any conventional JADE behaviour into the *Entity Main Cycle Behaviour*, the *simulation entity agent* interface must be consistent with the JADE agent standard interface. To achieve this, the *simulation entity agent* must 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.

Simulation engine agent
The *simulation engine agent* can be similarly described both for local and distributed engines. The local engine has already been described in [Gianni08], which the interested reader may refer to for further details. The distributed engine is built extending the local one, and by adding the following functionalities:

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

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 between two different societies.

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 Fig. 1. By this, the TDS framework transitively makes the implementation of HLA- and agent-based DS systems transparent because no details of the distributed computing infrastructure, HLA in particular, are to be known in order to develop such systems. Besides from that, however, the following additional advantages can be obtained:

- SimArch and its HLA-based implementation allow the integration with other simulation systems developed by use of such technologies, such as SimJ [D'Ambrogio06];
- 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 [Bellifemmine];
- the implementation remains extremely simplified and conforms to a general reuse and integration trend currently observed in the software and simulation industry.

Fig. 3 shows the state diagram of the *simulation distributed engine agent*. This diagram defines a lifecycle that 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 [Gianni07, D'Ambrogio06a]. *Phase 1* is the registration phase and cares of synchronizing the start-up phase through the *Waiting for Registration Requests* and *Confirm Registration Successful* states. In this 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. After completing this phase, the engine proceeds to the *Phase 2,* which represents the *Engine Main – Cycle Behaviour* and consists of the states *Waiting for Simulation Requests, Advancing Distributed Simulation Time,* and *Processing Internal Event*. In the first state, the engine waits for simulation requests from the agents, which are executing the associated simulation logic. These requests can be: *hold for a time*, *wait message*, *send message*, and *move to another engine*. The hold and the wait involve the scheduling of a local event, to which an event processing routine is associated. The sending of a message, differently, requires some processing to determine whether the recipient agent is local or remote. In the case of local agent, the request is dealt with the scheduling of a local event as for the hold and wait services. Conversely, in the case of remote recipient, the message is passed to the underlying layer 1 using the data interfaces provided by the SimArch architecture. Layer 1 in turns delivers the message to the specified remote engine. Finally, the mobility request is dealt with removing the agent from the society composition, and by conventionally invoking the send event service of layer 1 with a special tag that discriminates the type of service invocation on the recipient engine side. When all the agents have sent their requests and entered a blocking state, the engine proceeds to the *Advancing Distributed Simulation Time* state. In this state, the engine invokes layer 1 services to advance

the simulation time to the time of the next internal event. This is needed to guarantee that no incoming events from other engines are being delivered at a lesser simulation time than the time of the next internal event. The transition from this state to the next state only occurs when either a distributed event has been received or the time has been granted. In the latter case, the distributed event is transparently scheduled as a local event by SimArch Layer 1, and becomes the next internal event. In both cases, the next event is retrieved from the list and processed. If this event is a send message event, it is delivered and the execution of the relevant recipient agent is resumed. Differently, if the event is of type mobility event of an incoming agent in the engine, the simulation temporarily blocks until the agent is loaded up and joins the local simulation society. This is obtained by updating the society composition so that 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 Fig. 3.
Phase 3 is activated by Phase 2 when receiving the corresponding event from the distributed environment, and consists in notifying the simulation end event to all the local agents.
Finally, *phase 4* concludes the engine life cycle and restores the distributed environment set up. This phase consists in the operations specified in [Gianni07, D'Ambrogio06a].

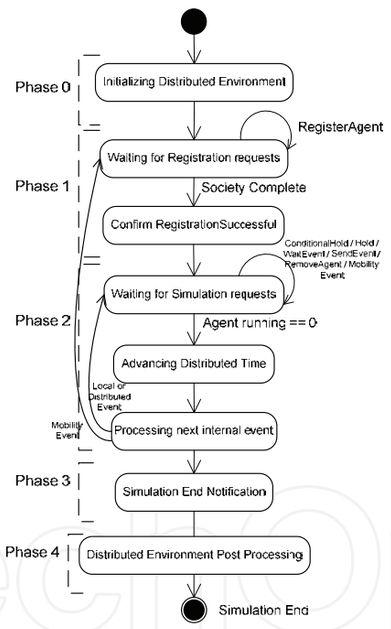


Fig. 3 State diagram of the simulation distributed engine agent

4.3 Interaction Protocol

The interaction protocol defines the rules upon which the conversation between the agents occurs, e.g. which agent talks, which listens, which expects what. The protocol 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 of a given society to request and acknowledge the simulation actions defined in the *DES-Ontology*. This protocol 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* require the agent to interrupt 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' requests and which activates the individual agents by responding to their requests. For the correct execution of the simulation it is fundamental that the entity agents are aware of and comply with such protocol.

The *inter-society protocol* complements the intra-society rules when operating in a distributed environment. The distributed engine implements this 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 whether the recipient is running locally or on a remote society. 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 by HLA, and therefore is not compliant with the FIPA standard. However, such an approach brings several advantages – as shown above, and does not affect the peculiarities of the local interactions, which are still FIPA compliant.

The *agent mobility* is based on a similar approach but is more complex. Fig. 4 shows an example of agent mobility with the actors of this interaction and the sequence of steps. A *resource manager* agent is needed on the remote site to start the moved agent. The presence of this agent is essential to guarantee the proper application specific initialization typical of an agent start-up.

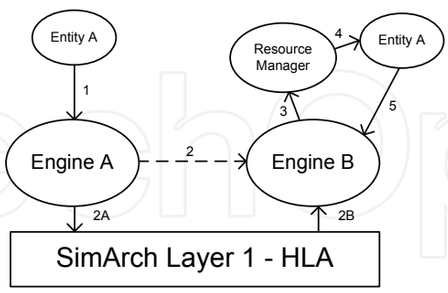


Fig. 4. Example of agent mobility [Gianni09]

Shall we assume that Agent Entity A in Fig. 4 wants to move from Society (Engine) A to Society (Engine) B at simulation time *t*. Entity A first sends a JADE-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 in turn processes

the event by updating the society composition and by 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 from the locally running Entity A. After having operated the initialization of the agent parameters, the Resource Manager activates Entity A agent with the attached state (step 4). Once running, the agent first requests to join the local society and, after having received the respective acknowledgement, this starts its simulation cycle as at it was a first activation (step 5). Such mechanism guarantees that the mobility is operated transparently and in synchronization with the simulation clock, local and distributed.

5. TDS application example

The TDS framework serves as a general container for domain-specific agents, which are to be developed for the particular application. In this Section, we consider a scenario that involves the motion of individuals in a multi-storey factory building to simulate operations in a normal or emergency situation.
For the sake of conciseness, we consider a simplified manufacturing system where workers move around the factory premises in order to reach the machines they need to use or to reach building evacuation points. The modelling of the system includes the modelling of the space and of the worker agents. 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. The worker agents are autonomous entities, each characterized by a set of objectives, a decision model and a motion model.

The reasons for the investigation of such a system can be multiple. For example, the optimization of the factory resources, such as mechanical machines, or the validation of architectural design choices in the factory plan or the assessment of the evacuation capabilities of the factory building in presence of emergency situations. These studies, however, are not in the scope of this chapter and the example is here used to illustrate a TDS framework application only.

The agent-based modeling approach consists of a world model and a set of agents. The *world model* represents the simulated world within which the agents move and interact with each other. The world model consists of a graph representing the physical space, plus a description of the world status for each element of the physical world. The nodes of the graph represent the physical positions reachable by the agents, while the edges define the walking access between two points, as fully described in [Gianni08c]. The *agents*, including their dynamics and the parameterisation, are defined at Layer 3 of the TDS architecture. Their design is based on the key principle of *Separation of Concerns* [Mens] that suggests designing components with a maximized cohesion. Each agent is defined by a behavioural logic, which specifies the interaction with the external world, and a set of parameters that do

not affect the pattern of the logic, according to the design outlines [Mernik]. Adopting this approach, the cohesion of each agent is maximised to make it reusable across the several values the parameters might assume. A straightforward, but effective, methodology to identify the candidate parameters comes from the analogy with the physical agents. A *Resource Manager* manages the world model, and the active actors, *human agents*, use and affect the conditions the world model resources. The in-depth description of the behavioural logic and the parameters of both the Resource Manager and the human agents is given in [Gianni08c].

The next subsection presents the design details of the scenario configuration.

5.1 Configuration of the scenario

The development of the distributed agent-based simulation system proceeds with the definition of the simulation scenario at Layer 4 of the TDS framework. In the specific case of the factory scenario, this includes: 1) the definition of the world model upon which the workers operate; 2) the partitioning of the simulated world and, 3) the definition of the number and the characteristics of the workers.

Point 1 can be achieved by first deciding the simulated world, and then by deriving the graph for each of its segments. In our application we considered a two storey factory, whose plan is shown in Fig. 5 for floor 1 and 2, respectively, and whose graph schematization is illustrated in Fig. 6.

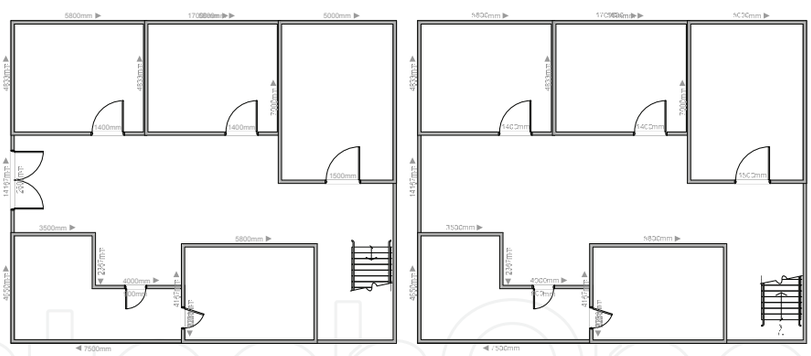


Fig. 5. Factory plan, floor 1 (left) and floor 2 (right)

Point 2 can be straightforwardly obtained by choosing floor 1 and floor 2 as segments of the simulated world to be allocated on independent hosts. The graphs of such floors were adapted for the distributed simulation and finally coded as XML files using GraphML [Brandes]. Each node and each edge are characterized by the relevant properties concerning the point in the physical world (e.g. type of room, type of machine, exit node, stairwell node, and the name of agent – resource manager – in charge of the nodes management). In particular, the nodes corresponding to the exit and the stairwell have a different semantics for the worker agents. The former specifies that the life cycle of the agent must terminate, as it reaches the main exit. Differently, a worker agent that approaches the stairwell knows that it must pass the management of its simulation requests to another resource manager, which is likely to be running on another host. It is also important to note that these graphs are not

augmented with a condensed view of the remaining remote world. To simplify the prototypal implementation of this demo, the worker agents are specifically provided with the intermediate objective of reaching the stairwell node before directing towards any of the node being managed in the remote resource manager container.

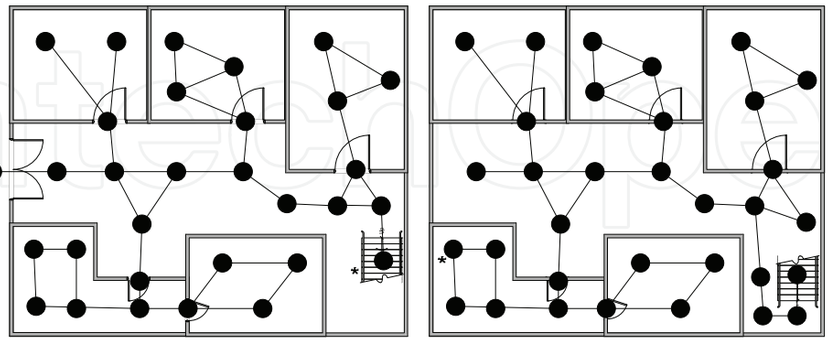


Fig. 6. World model for floor 1 (left) and floor 2 (right)

The number of worker agents is arbitrarily defined to be 40, more specifically 20 per each floor. Each agent is provided with an initial instance of the local graph and with an assigned set of objectives to accomplish before exiting the factory premises. An example of agent state model is reported below:

```
<agentWorker>
    <name>Worker 1</name>
    <initialNode>20</initialNode>
    <initialFloor>1</initialFloor>
    <motionModel>
        <UniformGenerator> <!-- motion over the edges -->
                <lowerBound>45</lowerBound>
                <upperBound>55</upperBound>
                <offset>15</offset>
        </UniformGenerator>
        <Constant>
                <value>0.3</value> <!-- motion over the nodes -->
        </Constant>
    </motionModel>
    <objectives class="SequenceOfObjectives">
        <objective class="SimpleObjective">
                <destination class="LocalDestination">
                <nodeId>34</nodeId>
                <groupId>0</groupId>
                <resourceManager>ResourceManager1</resourceManager>
                <areaName>Floor1</areaName>
        </objective>
        <objective class="SimpleObjective">
                <destination class="LocalDestination">
                <nodeId>62</nodeId>
                <resourceManager>ResourceManager2</resourceManager>
                <areaName>Floor2</areaName>
        </objective>
    </objectives>
</agentWorker>
```

The agent model specifies the agent name, the initial position, the motion model and the objectives that the agent must achieve before leaving the factory premises form the main

exit. The motion model consists in two components: the motion over the edges, which is the first parameter, and the motion over the nodes, which is the second parameter. In this specific case, the objective is of type composite and consists of a sequence of objectives. Particularly, the first objective is reaching the stairwell node 34 of floor 1 and the second one is reaching node 62 on second floor, both denoted with a *star* in Fig. 6. When the worker agent reaches the node 34, it recognizes that this node is of type stairwell and then automatically performs the mobility action to the floor 2 resource manager, which is running on a different host. Once in the new segment of the simulated world, the worker agent is loaded with the local world map, which also includes node 62. The agent finally uses the new map to compute the best path towards the assigned objective node 62.
In the simulated scenario, the 40 worker agents are similarly defined and activated in the HLA federation execution. The whole process proceeds activating first the Runtime Infrastructure, then the Federation Manager federate, and finally the two TDS engines. Each engine is provided with a local *Resource Manager* (Resource Manager 1 for Floor 1 and Resource Manager 2 for Floor 2) which eventually uses the XStream library [XStream] to first load the agent state of the agents, and then activates each of them. These agents register into the simulation society, and then the simulation cycle starts as described above.

6. Conclusion

The development of an agent-based distributed simulation system requires considerable effort in terms of HLA code and knowledge, compared to conventional local simulation systems. In this chapter, we have presented a simulation framework, named TDS, which reduces such an effort by making the development of agent-based distributed simulation systems HLA-transparent. This is achieved by introducing several abstraction layers on top of the distributed environment so that the simulation system developers deal with the same uniform interface when developing either local or distributed simulation systems. Moreover, this interface is uniform with conventional Multi-Agent System (MAS) framework, such as JADE, and therefore this reduces the development of such simulation systems to the one of conventional MAS. By such an approach, any HLA-based agent-based distributed simulation system can practically be developed as a conventional local MAS one, with no extra effort and no extra knowledge. An example development of a simulation system has also been presented to illustrate the application of the proposed framework in a domain that involves the motion of individuals in a multi-storey factory building to simulate operations in normal or emergency situations.

7. Acknowledgments

This work has been partially supported by the FIRB Project “Performance Evaluation of Complex Systems”, funded by the Italian Ministry of Research and by the University of Roma TorVergata; by the FIRB Project “Software frameworks and technologies for the development of open-source distributed simulation code”, funded by the Italian Ministry of Research; by CERTIA Research Center of the University of Roma TorVergata; and by the FP7 euHeart Project, funded by the European Commission (FP7-ICT-2007-224495).

8. References

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

[Brandes] Brandes, U.; Eiglsperger, M.; Herman, I.; Himsolt, M.; and Marshall, M.S.; "GraphML Progress Report: Structural Layer Proposal," *Proceedings of the 9th Intl. Symp. Graph Drawing (GD '01)*, LNCS 2265, Springer-Verlag, pp. 501-512.

[D'Ambrogio06] D'Ambrogio, A.; Gianni, D.; and Iazeolla, G.; "SimJ: a Framework to Distributed Simulators", *Proceedings of the 2006 Summer Computer Simulation Conference (SCSC06)*, Calgary, Canada, July, 2006, pp. 149 – 156.

[D'Ambrogio06a] D'Ambrogio, A.; Gianni, D.; and Iazeolla, G.; "jEQN: a Java-based Language for the Distributed Simulation of Queueing Networks", LNCS vol. 4263/2006, *Proceedings of the 21st International Symposium on Computer and Information Sciences (ISCIS'06)*, Istanbul, Turkey, Nov, 2006.

[D'Ambrogio08] D'Ambrogio, A.; Gianni, D.; Iazeolla, G.; and Pieroni, A.; "Distributed simulation of complex systems by use of an HLA-transparent simulation language", *Proceedings of the 7th International Conference on System Simulation and Scientific Computing*, ICSC 2008, Asia Simulation Conference, Oct, 2008, Beijing, China, pp. 460 – 467.

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

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

[Gianni07] Gianni, D.; and D'Ambrogio, A.; "A Language to Enable Distributed Simulation of Extended Queueing Networks", *Journal of Computer*, Vol. 2, N. 4, July, 2007, Academy Publisher, pp. 76 – 86.

[Gianni08] Gianni, D.; and D'Ambrogio, A.; "A Domain Specific Language for the Definition of Extended Queueing Networks Models", *Proceedings of the 2008 IASTED Software Engineering Conference (SE08)*, Innsbruck, Austria, February, 2008.

[Gianni08a] Gianni, D.; D'Ambrogio, A.; and Iazeolla, G.; "A Layered Architecture for the Model-driven Development of Distributed Simulators", *The First International Conference on Simulation Tools and Technologies (SIMUTOOLS08)*, Marseille, March, 2008.

[Gianni08b] Gianni, D.; "Bringing Discrete Event Simulation Into Multi Agent System", *10th International Conference on Computer Modelling and Simulation*, EuroSIM/UKSIM, Cambridge, April, 2008.

[Gianni08c] Gianni, D.; Loukas, G.; and Gelenbe, E.; "A Simulation Framework for the Investigation of Adaptive Behaviours in Largely Populated Building Evacuation Scenarios", *International Workshop on Organised Adaptation in Multi-Agent Systems (OAMAS) in the 7th International Conference on Autonomous Agents and Multi-Agent System (AAMAS)*, Estoril, Portugal, May, 2008.

[Gianni09] Gianni, D.; D'Ambrogio, A.; and Iazeolla, G., "DisSimJADE: A JADE-based framework for the Distributed Simulation of Multi-Agent Systems", *The Second International Conference on Simulation Tools and Techniques for Communications* (SIMUTOOLS09), March, 2009.

[Grable] Grable, R.; Jernigan, J.; Pogue, C.; and Divis, D.; "Metrics for Small Projects: Experiences at the SED", *IEEE Software*, March-April 1999, pp 21-29.

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

[IEEE-HLA1] IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA) - federate interface specification. Technical Report 1516.1, IEEE (2000).

[IEEE-HLA2] IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA) - object model template (OMT) specification. Technical Report 1516.2, IEEE (2000).

[IEEE-HLA3] IEEE: Recommended practice for High Level Architecture (HLA) federation development and execution process (FEDEP). Technical Report 1516.3, IEEE (2003).

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

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

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

[Kuhl] Kuhl, F.; Weatherly, R.; and Dahmann, J.; Creating Computer Simulation Systems: An Introduction to the High Level Architecture, Prentice Hall (1999).

[Lees] Lees, M.; Logan, B.; and Theodoropoulos, G.; "Distributed Simulation of Agent-based Systems with HLA," *ACM Transaction on Modeling and Computer Simulation*, Vol. 17, N. 8, Jul, 2007.

[Mens] Mens, T.; and Wermelinger, M.; "Separation of concerns for software evolution", *Journal of Software Maintenance*, vol. 14, n. 5, Sept, 2002, pp. 311 – 315.

[Mernik] Mernik, M.; Heering, J.; and Sloane, A.M.; "When and how to develop domain-specific languages", *ACM Computing Surveys*, 37(4):316–344, 2005.

[Ontology-Bean] Ontology Bean Generator, http://hcs.science.uva.nl/usr/aart/beangenerator/index25.html

[Pokahr] Pokahr, A.; Braubach, L.; and Lamersdorf, W.; "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.

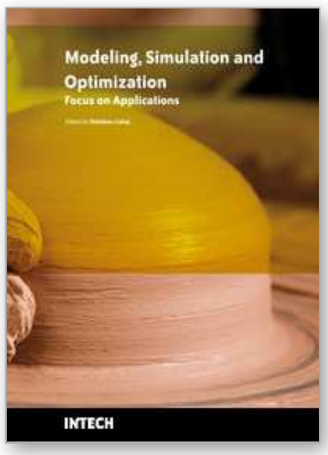
[Sarjoughian] Sarjoughian, H.S.; Zeigler, B.P.; and Hali, S.B.; "A Layered Modeling and Simulation Architecture for Agent-Based System Development," *Proceedings of the IEEE*, Vol. 89, N. 2, Feb, 2001, pp. 201 – 213.

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

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

[Wang] Wang, F.; Turner, S.J.; and Wang, L.; "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.

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



Modeling Simulation and Optimization - Focus on Applications
Edited by Shkelzen Cakaj

ISBN 978-953-307-055-1
Hard cover, 312 pages
Publisher InTech
Published online 01, March, 2010
Published in print edition March, 2010

The book presents a collection of chapters dealing with a wide selection of topics concerning different applications of modeling. It includes modeling, simulation and optimization applications in the areas of medical care systems, genetics, business, ethics and linguistics, applying very sophisticated methods. Algorithms, 3-D modeling, virtual reality, multi objective optimization, finite element methods, multi agent model simulation, system dynamics simulation, hierarchical Petri Net model and two level formalism modeling are tools and methods employed in these papers.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen