A Discrete-Event Simulation Framework for the Validation of Agent-based and Multi-Agent Systems

Giancarlo Fortino, Alfredo Garro, Wilma Russo
DEIS – Università della Calabria, I-87036 Rende (CS), Italy
{g.fortino, garro, w.russo }@unical.it

Abstract

Simulation of agent-based systems is an inherent requirement of the development process which provides developers with a powerful means to validate both agents' dynamic behavior and the agent system as a whole and investigate the implications of alternative architectures and coordination strategies. In this paper, we present a discrete-event simulation framework which supports the validation activity of agent-based and multi-agent systems which are modeled and programmed as a set of event-driven agents by means of the Distilled StateCharts formalism and related programming tools. The simulation framework is equipped with a discrete-event simulation engine which provides support for the execution of agents by interleaving their events processing, the exchange of events among agents, the migration of agents, and the clustering of agents into agent servers interconnected by a logical network. Using this framework, an agent-based complex system can be easily validated and evaluated by defining a simulator program along with suitable test cases and performance measurements.

1. Introduction

Agent-based and multi-agent systems (MAS), like other complex software systems, must be tested and evaluated before being deployed [10]. Simulation of agent-based systems is an inherent requirement in all phases of the development process. Modeling and simulation help developers learn more about agents' interactive behavior and investigate the implications of alternative architectures and coordination strategies. In particular, discrete-event simulators are highly required for evaluating how complex agent-based systems work on scales much larger than those achievable in real testbeds.

Currently few development processes for agent-based systems which explicitly incorporate a simulation phase have been proposed. In [13] an integrated development environment for the engineering of MAS as Electronic Institutions is presented. An Electronic Institution is a performative structure of multi agent protocols (or scenes) along with a collection of normative rules that can be triggered off by agents' actions. The development environment is composed of a set of tools supporting the design, validation through simulation, development, deployment and the execution of MAS as Electronic Institutions. Such a development environment is aimed at facilitating the iterated and progressive refinement of the development cycle of MAS. In particular, SIMDEI, a simulation tool, allows for the animation and analysis of the specification of the rules and protocols in an Electronic Institution. In [12, 15] a modeling and simulation framework (DynDEVS) for supporting the development process of MAS from specification to implementation is proposed. The authors advocate the use of controlled experimentation in order to allow for the incremental refinement of agents while providing rigorous observation facilities. The benefits of using modeling and simulation for the evaluation of cooperative agents is illustrated through a simple example based on the Contract Net Protocol. The exploited simulation framework is JAMES, a Java Based Agent Modeling Environment for Simulation, which aims at exploring the integration of the agents paradigm within a general modeling and simulation formalism for discrete-event systems. JAMES follows a formal approach for discrete-event simulation based on DEVS (Discrete Event Systems Specification) which allows to specify (atomic and coupled) models and execute them by sending typed messages between simulator objects. In [11] a logic based prototyping environment for multi-agent systems, CaseLP (Complex Application Specification Environment Based on Logic Programming) is presented. CaseLP integrates simulation tools for visualizing the prototype execution and for collecting the related statistics. The CaseLP visualizer tool provides documentation about events that happen at the agent level during the MAS execution. Developers according to their needs can instrument the code of some agents after it has been loaded by adding probes to the code of agents. In this way, events related to state changes and /or exchanged messages can be recorded and collected for on-line and/or off-line visualization. It is worth pointing out that from a

simulation point of view CaseLP is a time-driven centralized simulator with a global time known from all the agents in the system.

In this paper, we present a Java-based discrete-event simulation framework which supports the validation activity of agent-based and multi-agent systems which are modeled and programmed as a set of event-driven agents by means of the Distilled StateCharts formalism and the related programming tools [8]. The simulation framework is organized in four layers: (i) *low-level simulation framework*, which provides the basic mechanisms and classes to simulate general purpose systems; (ii) *agent platform*, which is built atop the low-level simulation framework and provides a distributed infrastructure formed by a network of interconnected agent servers; (iii) *ELA adapter*, which allows to map event-driven DSC-based lightweight agents onto the agent platform layer; (iv) *user*, which provides abstractions representing interacting users and users' behaviors. Using this framework, an agent-based complex system can be easily validated and evaluated by defining a simulator program along with suitable test cases and performance measurements.

The remainder of the paper is structured as follows. Section 2 overviews the Distilled StateCharts-based approach for the modeling and validation of agent-based system which adopts the proposed simulation framework as validation tool. In section 3, the simulation framework is described in detail whereas section 4 reports some results concerning with the performance evaluation of an agent-based e-Marketplace by means of the simulation framework. Finally conclusions are drawn and directions of future work delineated.

2. A Distilled StateCharts-based approach for the modeling and validation of agent-based systems: an overview

The Distilled StateCharts-based approach [5, 6], which aims at supporting the modeling and validation of agent-based and multi-agent systems, consists of the following phases (Fig. 1): High-Level Modeling, Detailed Design, Coding and Simulation.

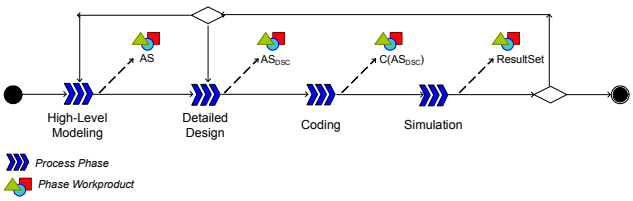


Figure 1. Process schema of the DSC-based approach.

The High-Level Modeling phase can be supported by well-established agent-oriented methodologies (such as the Gaia methodology [17]) which cover the phases of requirements capture, analysis and high-level design. The product of this phase is the agent-based system model (AS) defined as follows:

$$AS = <AT, LCL, act, serv, prot>,$$

where:

AT (Agent Types) is the set of types of agents embodying activity, offering services and interacting with each other;

LCL (Logical Communication Links) is the set of logical communication channels among agent types which embody interaction protocols;

act: AT $\rightarrow$ *activity description* is the activity relation which associates one or more activities to an agent type;

serv: AT $\rightarrow$ *service description* is the service relation which associates one or more services to an agent type;

prot: LCL $\rightarrow$ *interaction description* is the protocol relation which associates an interaction protocol to a logical communication channel.

The Detailed Design phase is enabled by a Statecharts-based formalism, namely the Distilled StateCharts (DSC) [8], which supports the specification of the behavior of the agent types and the interaction protocols among the agent types of AS. In particular DSC allow for the specification of the behavior of lightweight agents (see §2.1) which are event-driven, single-threaded entities capable of transparent migration and executing chains of atomic actions. The DSC-based specification of an AS, denoted as AS_{DSC}, can be expressed as follows:

$$AS_{DSC} = \{Beh(AT_1), \ldots, Beh(AT_n)\},$$

where Beh $(AT_i) = <S_{Beh}(AT_i), E_{Beh}(AT_i)>$ is the DSC-based specification of the dynamic behavior of the i-th agent type. In particular, $S_{Beh}(AT_i)$ is a hierarchical state machine incorporating the activity and the interaction handling of the i-th agent type and $E_{Beh}(AT_i)$ is the related set of events to be handled triggering state transitions in $S_{Beh}(AT_i)$.

The Coding phase is carried out by using the Java-based Mobile Active Object Framework (MAO Framework) [8] and produces the work product $C(AS_{DSC})$ representing the code of AS_{DSC}. In particular, $Beh(AT_i)$ can be seamlessly translated into a composite object, which is the object-based representation of $S_{Beh}(AT_i)$, and into a set of related event objects representing $E_{Beh}(AT_i)$.

The Simulation phase is supported by MASSIMO, a Java-based discrete-event simulation framework for multi-agent systems (see §3). On the basis of the framework, a simulator program can be implemented and executed to obtain a ResultSet containing validation traces and performance parameter values. The validation of agent behaviors and interactions is carried out on execution

traces automatically generated, whereas the performance evaluation relies on the specific agent-based system to be analyzed; the performance evaluation parameters are therefore set ad-hoc. The ResultSet can also be used to feed back the High-level Modeling and Detailed Design phases.

2.1. The reference agent model

The agent model is based on the abstraction of event-driven state-based lightweight agent [7] which can be represented by the tuple:

<Id, Beh, DS, TC, EQ>,

where:
- Id is the unique identifier of the agent;
- Beh is the DSC-based dynamic agent behavior;
- DS is the data space hierarchically organized of the agent;
- TC is the single thread of control supporting agent execution;
- EQ is the event queue of the agent containing received and to-be-processed events.

The event-driven state-based lightweight agent is programmed by specifying its Beh through the FIPA-compliant agent behavioral template [2], reported in Figure 2, which is a Distilled StateChart [8] consisting of a set of basic states (Initiated, Transit, Waiting, Suspended, and Active) and transitions labeled by events. In particular, the agent performs computations and interactions in the Active Distilled StateChart (ADSC) composite state, inside the Active state, which is to be refined by the agent programmer. The presence of the deep history connector (H*) inside the Active state allows for a coarse-grained strong mobility-based agent migration [9]. An event reaction can produce computations, which can affect the DS, and/or the generation of one or more events, or a migration. While the reception of incoming events (or IN-events) is implicit and decoupled by the EQ, the transmission of events is explicitly carried out by means of the `generate(<event>(<parameters>))` primitive which allows to asynchronously raise outcoming events (or OUT-events). The execution semantics of the event-driven state-based lightweight agent are defined in terms of the Event Processing Cycle (EPC): the next available event is cyclically fetched from EQ and is passed to the Beh which can handle it so triggering one reaction. OUT- and IN-events are classified in:
- *internal* events, which can be defined at programming level for self-triggering active and/or proactive behavior. In the case of *internal* events, IN and OUT events coincide. In fact, an emitted *internal* event or OUT-event is received as IN-event by the emitting agent itself.
- *management* events, which include requests and notifications of services at agent server level such as agent lifecycle management, creation, cloning, and migration.
- *coordination* events, which enable coordination acts between agents according to a specific coordination model. In this paper the considered coordination model is the asynchronous *Direct* model, even though the *Tuple-based* and the *Publish/Subscribe event-based* models could also be exploited as shown in [7].

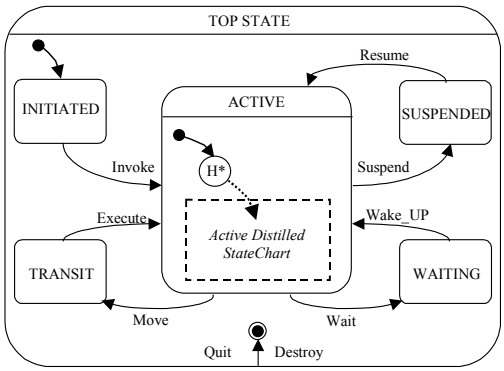
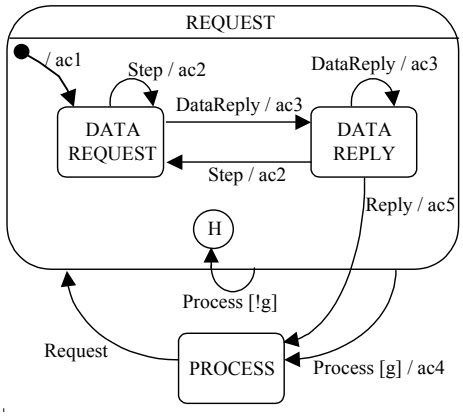


Figure 2. The FIPA-based template of the event-driven DSC-based lightweight agent.

In order to exemplify the DSC-based modeling of agent behavior, the specification of the ADSC of a mobile event-driven state-based lightweight agent is shown in Figure 3; Table 1 reports state variables, methods and events of the example agent specification. The agent overall goal is that of moving across a set of agent servers according to a predefined itinerary for monitoring a set of remote processes. In order to fulfill its goal, the agent alternates the following three phases:
- *Data acquisition*, which is performed by generating DataRequest coordination events targeting N different local agents which are controlling the local process. As soon as the monitoring data are collected (after the reception of all the DataReply coordination events), the internal Reply event is generated.
- *Data processing*, which is performed upon reception of the Reply event and carried out by means of the *process* method. It can also occur upon reception of the Process coordination event sent by another agent (e.g. the owner agent) if data are *enough* (the guard g holds), otherwise the agent returns in the substate of request which abandoned most recently.
- *Migration*, which depends on the data processing which, if successful, enables the agent to autonomously migrate to another site according to its itinerary; otherwise, the monitoring process is re-executed.



Action expressions:

```
ac1:count=0;
 generate(new Step(0));
 generate(new DataRequest(recipients[0], 0));
ac2:i=((Step)e).getI();
 if (i<N-1) generate(new Step(i+1))
 generate(new DataRequest(recipients[i+1], i+1));
ac3:i=((DataReply)e).getI();
 data[i]=((DataReply)e).getData();
 count++;
 if (count==N) generate(new Reply());
ac4:if (process()) {
 next=(next+1)%itinerary.length;
 generate(new Move(self(), itinerary[next]));}
 generate(new Request());
ac5:ac4;
```

Guards:

```
g:enough()
```

Figure 3. The Active Distilled StateChart of the example agent.

VAR	DESCRIPTION
N	Number of requests the agent issues to the local monitoring agents
itinerary	List of agent servers to be visited
recipients	List of identifiers of the interacting agents
data	Collector of the data coming from the replying agents
next	Index of the last visited agent server
count	Number of replies received in a monitoring cycle
i	Temporary integer variable
e	Reference to the last received event instance
METHOD	
process	Specific method for processing data which returns true if the processing was successful
enough	Specific method for evaluating if there are enough data for processing
self	Method which returns the identifier of the agent
EVENT	
Step	Internal event pacing the generation of *DataRequest*
DataRequest	Coordination event of the asynchronous Direct model sent by the agent to the local monitoring agents for requesting data
DataReply	Coordination event of the asynchronous Direct model sent by a local monitoring agent for replying to *DataRequest*
Reply	Internal event indicating data gathering completion
Process	Coordination event enabling a forced processing
Request	Internal event activating a monitoring cycle

Table 1. State variables, methods and events of the example agent.

3. MASSIMO: a discrete-event simulation framework for MAS

The Multi-Agent Systems SIMulation framewOrk (MASSIMO) is a Java-based discrete-event simulation framework which allows for the validation and evaluation of:

- the dynamic behavior (computations, interactions, and migrations) of individual and cooperating agents;
- the basic mechanisms of the distributed architectures supporting agents, namely agent platforms;
- the functionalities of applications and systems based on agents.

The architecture of MASSIMO (Fig. 4) is composed of four basic layers:

(i) *Low-level simulation framework*, which provides the basic mechanisms and classes to simulate general purpose systems;

(ii) A*gent platform*, which is built atop the low-level simulation framework and provides a distributed infrastructure formed by a network of interconnected agent servers;

(iii) *ELA adapter*, which extends the MAAF (Mobile Agent Adaptation Framework) [8] and allows to map event-driven DSC-based lightweight agents, provided by the MAO Framework, onto the agent platform.

(iv) *User*, which provides abstractions representing interacting users and users' behaviors.

In the subsections 3.1-3.4 the four layers are described in detail. In section 3.5, the basic structure of a MAS simulator program is exemplified.

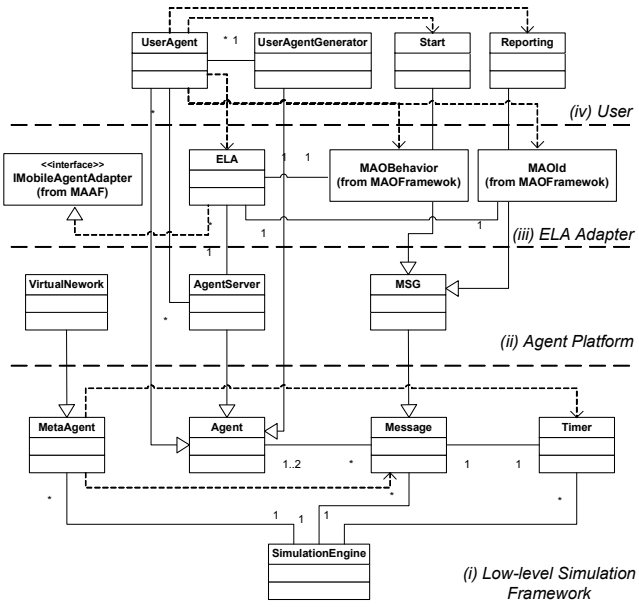


Figure 4. The architecture of MASSIMO.

3.1. The low-level simulation framework

The low-level simulation framework is composed of the following base Java classes which support agent-based programming and simulation of general-purpose systems:

- Agent, which represents a computational state-based agent communicating through asynchronous messages.
- MetaAgent, which represents a meta-level agent able to capture and constrain messages sent by computational agents or by other meta-agents.
- Message, which represents a message sent by an agent (source) to another agent (target).
- Timer, which is an object encapsulating a *Message* instance and a timeout. The message is delivered to its target at the timeout expiration.

The basic components of the simulation engine (Fig. 5) are:

- Global System Message Queue (GSMQ), which stores all the messages to be delivered.
- Global System Timer Queue (GSTQ), which stores all the timers ranked by timeout value.
- Simulation Clock (SC), which represents the simulation time. It is incremented every time that a timer expires.
- Filter (FT), which receives the messages generated by the computational agents and insert them into GSMQ if they are not subjected to the meta-level agent capture; otherwise FT forwards the messages to their associated meta-agents.
- Scheduler (SD), which cyclically extracts a message from GSMQ and dispatches it to the target agent. If there are not messages in GSMQ, SD forces a timer (the one with the lowest timeout) to fire and dispatches the associated message to its target.

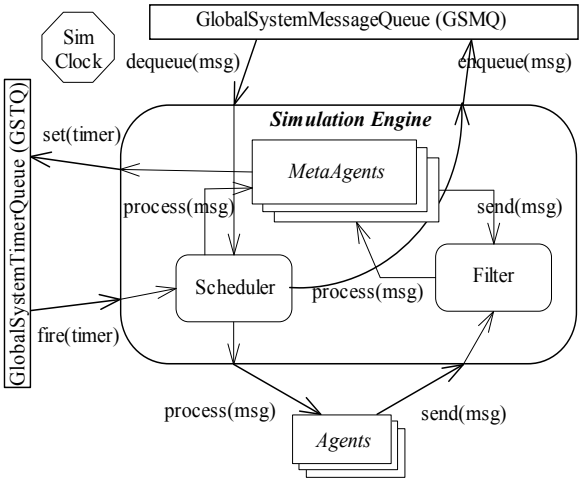


Figure 5. The architecture of the simulation engine.

3.2. The agent platform

The agent platform layer, which is built atop the low-level simulation framework, provides two basic abstractions: the AgentServer, which represents the infrastructure where event-driven lightweight DSC-based agents (ELAs) run, and the VirtualNetwork, which represents a network of hosts on which AgentServers can be mapped.

The AgentServer, which is an extension of the Agent class, provides the following functionalities:

- agent management lifecycle, which supports (de)registration and execution of ELAs;
- agent migration, which supports the migration of an ELA from one AgentServer to another;
- agent interaction, which supports the event-based interaction among ELAs;
- inter-agent-server service signaling.

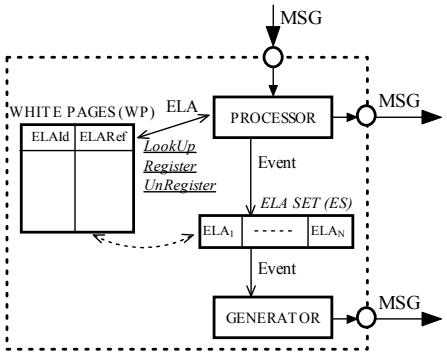


Figure 6. The architecture of the AgentServer.

The architecture of the AgentServer (Fig. 6) consists of the following components:

- *White Pages (WP)*, which keeps archived the ELAs running in the AgentServer. It consists of pairs <ELAId, ELARef>, where ELAId is the ELA identifier and ELARef is either (i) the reference to the ELA identified by ELAId and belonging to the set of ELAs (ES) running in the AgentServer or (ii) the proxy of the ELA identified by ELAId and migrated to another AgentServer. A proxy is a triple <AS, MBX, active>, where AS is the address of the AgentServer to which the ELA migrated, MBX is the ELA mailbox containing the events targeting the ELA during the ELA migration transitory, and active is a boolean variable indicating whether or not the forwarding activity of the proxy is on.
- *Processor*, which receives and processes incoming MSGs, extensions of the Message class, which can contain one of the following objects:
  (i) *an Event targeting an ELA*. The ELA target of the Event is looked up and the Event passed to it if the ELA is present in the AgentServer; otherwise, the ELA

Proxy is returned and the Event is encapsulated in a MSG and the resulting MSG redirected to the AgentServer address contained in the proxy.
(ii) *a created ELA*. The created ELA is registered in the WP.
(iii) *an incoming migrating ELA*. The incoming ELA is registered in the WP. If it is not the first time that the ELA is hosted by the AgentServer, the previously left proxy is substituted by the incoming ELA.
(iv) *an outcoming migrating ELA*. The outcoming ELA is encapsulated in a MSG and the resulting MSG is transmitted to the target AgentServer. Finally, the outcoming ELA is unregistered from the WP and its associated Proxy is set.
(v) *an inter-AgentServer service message*. The basic service messages are those for the management of the MBX of the ELA proxy:

- GetMBX, which is a request issued by a remote AgentServer to activate the proxy and obtain the MBX of an ELA which migrated from the AgentServer to the remote AgentServer. Upon reception of GetMBX, the AgentServer first looks up the proxy of the ELA whose identifier is contained in the GetMBX; then, it retrieves the MBX associated to the ELA and, if the MBX is not empty, sends an MBX message containing the MBX to the remote AgentServer. Finally, the proxy forwarding is activated (active=true).
- MBX, which contains the mailbox of an ELA previously requested from a remote AgentServer by a GetMBX service request. Upon reception of an MBX message, the AgentServer looks up the ELA whose identifier is contained in the MBX message and, if the ELA is present in the AgentServer, encapsulates the events contained in the MBX message in Messages targeting the AgentServer itself and inserts them in the GSMQ. If the ELA is not present in the AgentServer, MBX is sent to the AgentServer where the ELA migrated if the proxy is on; otherwise, the events contained in the MBX message are inserted in the MBX of the ELA proxy.

- *Generator*, which processes the following events generated by the hosted ELAs:
(i) *Internal self-triggering Event*. The event is encapsulated in a MSG whose target is the AgentServer itself to which the MSG is then transmitted.
(ii) *External Event*. The event is encapsulated in a MSG whose target is the AgentServer hosting the ELA target of the event and the MSG is then transmitted to the target AgentServer.
(iii) *Creation Event*. The event contains the identifier and the dynamic behavior of an ELA created in the AgentServer. These parameters are used to create a new ELA agent which is then encapsulated in a MSG whose target is the *AgentServer* itself to which the MSG is then transmitted.
(iv) *Timer Event*. The event is encapsulated in a MSG whose target is the AgentServer itself and the MSG then is encapsulated in a Timer which is set to the timeout contained in the timer event.

The VirtualNetwork, which is an extension of the MetaAgent class, is able to set Timers on transmitted MSGs. It relies on a graph-based network structure in which a network link is completely reliable and based on an end-to-end delay model by which the delay of event/message transmissions [3] and agent migrations [14] can be calculated. The calculated delay is used as timeout value of a Timer containing a MSG.

3.3. ELA adapter

The ELA (Event-drive Lightweight Agent) adapter (Fig. 7) allows to plug a MAOBehavior object encapsulating the DSC-based behavior of an event-driven lightweight agent into the simulation framework.

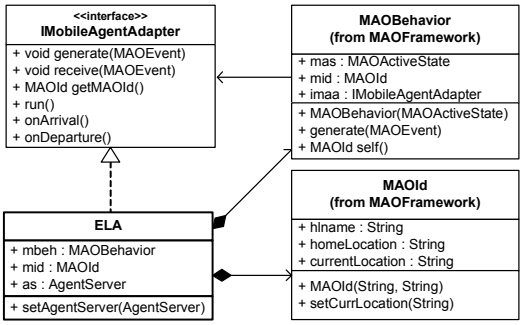


Figure 7. The ELA adapter layer.

The ELA class is an extension of the MAAF (Mobile Agent Adaptation Framework) [8] designed to provide the basic support for the adaptation of a MAOBehavior to a mobile agent class which is made available by a specific Java-based mobile agent platform. The ELA class implements the IMobileAgentAdapter interface and is associated with a MAOBehavior and a MAOId encapsulating the high-level agent identifier. IMobileAgentAdapter declares the following methods for adapting agent interaction, execution and migration:

- *receive*, which is invoked to pass MAOEvents to agents;
- *generate*, which interprets the MAOEvents generated within MAOBehavior and translates them into calls of platform-dependent methods;
- *run*, which is the method supporting agent execution;
- *onDeparture,* which is invoked just before the migration initiates;
- *onArrival*, which is invoked after the migration is completed.

To completely adapt an ELA to the agent platform layer the ELA class needs only to implement the methods *receive* and *generate*. The method *receive* is invoked by the AgentServer to deliver MAOEvents to ELAs. The method *generate*, which is invoked by the MAOBehavior, passes a MAOEvent to the AgentServer.

3.4. User

The User level makes it available two abstract classes UserAgent and UserAgentGenerator which are extensions of Agent. UserAgent represents a user directly connected to an AgentServer who can create, launch and interact with ELAs. UserAgentGenerator models the generation process of UserAgents. In particular, the UserAgentGenerator is able to create and start UserAgents according to a given logic (e.g. statistical distribution). Moreover, the Start message allows for the activation of a UserAgent or a UserAgentGenerator, whereas the Reporting message which targets a UserAgent contains a report sent from an ELA owned by the UserAgent.

3.5. Simulator programming

A MAS simulator can be programmed on the basis of the simulation entities described in the previous subsections: VirtualNetwork, AgentServer, ELA, UserAgent and UserAgentGenerator. A general simulator program can be constructed in the following steps:

1. creation of the VirtualNetwork;
2. creation of one or more AgentServers;
3. mapping of the created AgentServers onto distinct nodes of the VirtualNetwork;
4. creation of the ELAs that will not be created, directly or undirectly, by a UserAgent;
5. mapping of the created ELAs onto AgentServers;
6. creation of one or more UserAgentGenerators and/or one or more UserAgents. In the latter case, the created UserAgents are to be bounded to one or more AgentServers;
7. generation of the Start messages targeting the UserAgentGenerators and/or the UserAgents;
8. start of the simulation engine.

Figure 8 sketches the code of the simulator program of an example MAS. The MAS is composed of N stationary service agents (SA) distributed on N different AgentServer, a UserAgent (UA) which creates and launches a mobile agent (MA). MA travels along the N different AgentServers, interacts with the N SAs and, finally, comes back home by reporting to the UA.

```
//initialize the simulation engine
 SimulationEngine.init();

//create an Homogeneous Small Network of N_AS+1 nodes
 VirtualNetwork vn = new VirtualNetwork(N_AS+1, VirtualNetwork.HSN);

//add the VirtualNetwork to the set of MetaAgents
 SimulatiomEngine.addMetaAgent(vn, MetaAgent.ALL_MSG);

//create N_AS agent servers
 AgentServer [] ass = new AgentServer[N_AS];
 String [] ass_url  = new String[N_AS];
 for (int i=0; i<N_AS; i++){
  ass_url[i] = "agentserver"+i;
  ass[i] = new AgentServer(ass_url[i], "typeX");
 }

//map agent servers to network nodes
 for (int i=0; i<N_AS; i++)
  vn.map(ass[i], i);

//create the service agents and map them to the agent servers
 for (int i=0; i<NUM_AS; i++)
  ELA sa = new ELA(new MAOId(ass_url[i]+"#sa", null,
                   ass_url[i]), new MAOServiceActiveState(100));
  Msg msg = new Msg(ass[i], ass_url[i], ass_url[i],
                    Msg.AGENT_CREATION, sa );
  ass[i].process(msg);
 }

//create the user agent and map it to the N_AS node
 UserAgent ua = new UserItineraryAgent("useragentX", ass_url);
 vn.map(ua, N_AS);

//send the Start message to the UserAgent
 Agent.send(new Start(ua));

//start the simulation of a duration of 1000000
 SimulationEngine.start(1000000);
```

Figure 8. An example MAS simulator program.

4. Performance evaluation of a consumer-driven agent-based e-Marketplace

An Agent-based e-Marketplace (AEM) is a distributed multi-agent system formed by both stationary and mobile agents which provide e-Commerce services to end-users within a business context. AEMs are distributed large-scale complex systems which require tools which are able to analyze not only the AEM at the micro level, i.e. behaviors and interactions of their constituting agents, but also the AEM at the macro level, i.e. the overall AEM dynamics. Although useful insights about AEM micro and macro levels can be acquired by playing e-Commerce simulation games and, then, analyzing the obtained results, or by evaluating real e-Commerce applications, discrete-event simulators are essential for evaluating how AEMs work on scales much larger than that achievable in games or in applications which involve humans. This section shows the application of the proposed discrete-event simulation framework to the analysis of micro level issues of a consumer-driven AEM, i.e. an e-Marketplace in which the exchange of goods is driven by the consumers that wish to buy a product.

4.1. An Agent-based Consumer Driven e-Marketplace model

The modeled AEM, inspired by the systems presented in [1] and [16], consists of a set of stationary and mobile agents which provides basic services for the searching, buying, selling, and payment of goods.

The identified types of agents are:

- *User Assistant Agent* (UAA), which is associated with users and assists them in: (i) looking for a specific product that meets their needs; (ii) buying the product according to a specific buying policy.
- *Access Point Agent* (APA), which represents the entry point of the e-Marketplace. It accepts requests for buying a product from a registered UUA.
- *Mobile Consumer Agent* (MCA), which is an autonomous mobile agent dealing with the searching, contracting, evaluation, and payment of goods.
- *Vendor Agent* (VA), which represents the vendor of specific goods.
- *Yellow Pages Agent* (YPA), which represents the contact point of the distributed Yellow Pages Service (YPS) providing the location of agents selling a given product. The organization of the YPS can be: (i) *Centralized* (C), each YPA stores a complete list of Vendor Agents; (ii) *One Neighbor Federated* (1NF), each YPA stores a list of VAs and keeps a reference to only another YPA; (iii) *M-Neighbors Federated* (MNF), each YPA stores a list of VAs and keeps a list of at most M YPAs.
- *Bank Agent* (BA), which represents a reference bank supervising money transactions between MCAs and VAs

The identified types of interactions between the agent types are described below by relating them to the system workflow triggered by a user's request:

1. *Request Input* (UAA→APA): the UAA sends a request to the APA containing a set of parameters selected by the user for searching and buying the desired product, i.e. the product description (*Prod_Desc*), the maximum product price (P_{MAX}) the user is willing to pay, and the type of buying policy (*BP*).
2. *Service Instantiation* (APA→MCA): the APA creates a specific MCA and provides it with the set of user's parameters, the type of searching policy (*SP*), and the location of the initial YPA to be contacted. Upon creation, the MCA moves to the initial YPA location.
3. *Searching* (MCA↔YPA): the MCA requests a list of locations of VAs selling the desired product to the YPA. The YPA replies with a list of VA locations and, possibly, with a list of linked YPA locations.
4. *Contracting & Evaluation* (MCA↔VA): the MCA interacts with the found VAs to request an offer (P_{offer}) for the desired product, evaluates the received offers, and selects an offer, if any, for which the price is acceptable (i.e., $P_{offer}<=P_{MAX}$) according to the type of *BP*.
5. *Buying* (MCA↔VA↔BA): the MCA moves to the location of the selected VA and pays for the desired product using a given amount of e-cash (or bills) triggering the following money transaction: (i) the MCA gives the bills to the VA; (ii) the VA sends the bills to a BA; (iii) the BA validates the authenticity of the bills, disables them for re-use, and, finally, issues an amount of bills equal to that previously received to the VA; (iv) the VA notifies the MCA.
6. *Result Report* (MCA→UAA): the MCA reports the buying result to the UUA.

4.2. Agent behaviors

A model of MCA is defined on the basis of the tuple:

<SP, BP, TEM>,

where:

- SP is a searching policy in {ALL, PA, OS}:
  a. *ALL*: all YPAs are contacted;
  b. *Partial* (PA): a subset of YPAs are contacted;
  c. *One-Shot* (OS): only one YPA is contacted.
- BP is a buying policy in {MP, FS, FT, RT}:
  a. *Minimum Price* (MP): the MCA first interacts with all the VAs to look for the best price of the desired product; then, it buys the product from the VA offering the best acceptable price;
  b. *First Shot* (FS): the MCA interacts with the VAs until it obtains an offer for the product at an acceptable price; then, it buys the product;
  c. *Fixed Trials* (FT): the MCA interacts with a given number of VAs and buys the product from the VA which offers the best acceptable price;
  d. *Random Trials* (RT): the MCA interacts with a random number of VAs and buys the product from the VA which offers the best acceptable price.
- TEM is a task execution model in {ITIN, PAR}:
  a. *Itinerary* (ITIN): the *Searching* and *Contracting & Evaluation* phases are performed by a single MCA which fulfils its task by sequentially moving from one location to another;
  b. *Parallel* (PAR): the *Searching* and *Contracting & Evaluation* phases are performed by a set of mobile agents in a parallel mode. In particular, the MCA is able to generate a set of children (generically called workers) and to dispatch them to different locations; the workers can, in turn, spawn other workers.

Thus, each one of the defined models implements the product buying service differently.

An MCA task execution model is chosen by the Access Point Agent (APA) when it accepts a user input request; the choice can depend on the <*SP, BP*> pair selected by the user and on the e-Marketplace characteristics. If the

chosen task execution model is of the *Parallel* type then the MCA is named PCA (*Parallel Consumer Agent*) otherwise if the chosen task execution model is of the *Itinerary* type then the MCA is named ICA (*Itinerary Consumer Agent*). Therefore, a PCA model is defined by a triple <*SP*, *BP*, PAR> whereas an ICA model is defined by a triple <*SP*, *BP*, ITIN>.

The DSC-based behavior of the PCA models is reported in [4] whereas the DSC-based behavior of the ICA models, that can be seen as a particular case of the PCA behaviour, is described in detail in [6].

4.3. Simulation parameters and results

The goal for which the simulation phase was performed is twofold:

- to validate the behavior of each type of agent, the different models of MCA agents on the basis of the different YPS organizations, and the agent interactions.
- to gain a better understanding of the effectiveness of the simulation for evaluating MAS performances.

In order to analyze and compare the MCA models, the Task Completion Time (T_{TC}) parameter was defined as follows: $T_{TC}=T_{CREATION}-T_{REPORT}$ where, $T_{CREATION}$ is the creation time of the MCA and T_{REPORT} is the reception time of the MCA report. The simulation scenario was set up as follows:

- each stationary agent (UAA, APA, YPA, VA, BA) executes in a different agent server;
- agent servers are mapped onto different network nodes which are completely connected through links having the same characteristics. The communication delay (δ) on a network link is modeled as a lognormally distributed random variable with a mean, μ, and, a standard deviation, σ [3];
- each UAA is connected to only one APA;
- the price of a product, which is uniformly distributed between a minimum (PP_{MIN}) and a maximum (PP_{MAX}) price, is set in each VA at initialization time and is never changed; thus the VAs adopt a fixed-pricing policy to sell products;
- each YPA manages a list of locations of VAs selling available products.
- an UAA searches for a desired product, which always exists in the e-Marketplace, and is willing to pay a price P_{MAX} for the desired product which can be any value uniformly distributed between PP_{MAX} and $(PP_{MAX}+PP_{MIN})/2$.

Simulations were run by varying the organization of the Yellow Pages (C, 1NF and 2NF organized as a binary tree or 2NFBT), the number of YPA agents in the range [10..1000] and the number of VA agents in the range [10..10000]. These ranges were chosen for accommodating small as well as large e-Marketplaces. The duration of the performed simulations were set so as

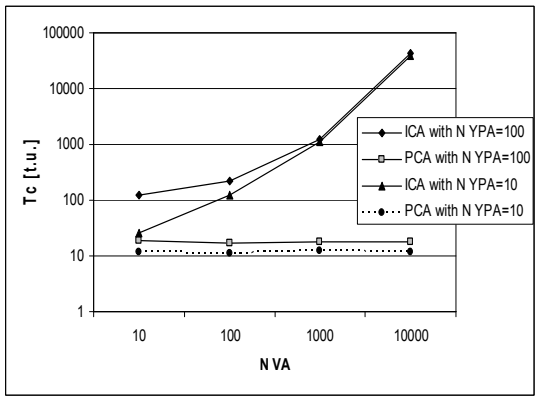


Figure 9. Performance evaluation of the <ALL, MP, *TEM*> models for an e-Marketplace with YPS=2NFBT, N_{YPA} ={10, 100} and variable N_{VA}.

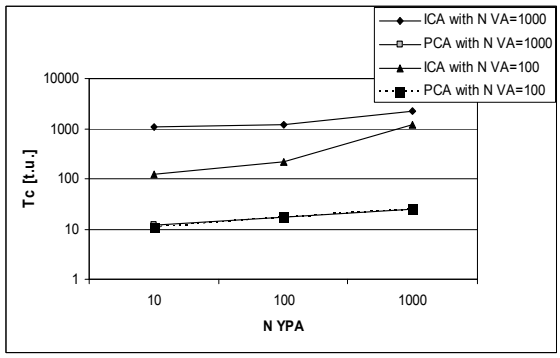


Figure 10. Performance evaluation of the <ALL, MP, *TEM* > models for an e-Marketplace with YPS=2NFBT, N_{VA} ={100, 1000} and variable N_{YPA}.

to allow for the completion of the buying task carried out by the MCA. The results obtained from the simulations made it possible to:

(a) evaluate which task execution model is more appropriate with respect to SP and BP policies (see §4.2) and for the characteristics of the e-Marketplace;

(b) validate the analytical model proposed in [16] regarding the sequential and parallel dispatching of mobile agents.

With respect to point (a), the ICA performs better than the PCA in the following cases:

- SP ={ALL, PA, OS}, BP =FS, YPS ={C, 1NF};
- SP ={PA, OS}, BP =FS, YPS =2NF.

Thus, the APA can choose the itinerary task execution model if such cases occur.

With respect to point (b), the performance evaluation focused on <ALL, MP, *TEM*> models (see §4.2) as these are the only models of MCA which guarantee both a successful purchase and the best purchase since they are successful at identifying the VA selling the desired product at the minimum price. In order to compare the performances of PCA and ICA models, the results

obtained for the <ALL, MP, *TEM*> MCA models adopting a YPA organization of the 2NFBT type are reported in Figures 9 and 10. The results shown in Figure 9 were obtained with N_{YPA}={10, 100} and varying N_{VA}, whereas the results shown in Figure 10 were obtained with N_{VA}={100, 1000} and varying N_{YPA}. In agreement with the analytical model reported in [16], the PCA, due to its parallel dispatching mechanism, outperforms the ICA when N_{VA} and N_{YPA} are increased.

5. Conclusions

Validation tools for agent-based and multi-agent systems are highly required before such systems get completely deployed on distributed execution platforms. In order to support the validation phase of agent-based systems at different levels of granularity, from agent behaviors, protocols and services (micro-level) to global system behavior (macro-level), flexible and robust agent-oriented, discrete-event simulation frameworks should be carefully designed and developed. This paper has proposed a Java-based discrete-event simulation framework (MASSIMO – Multi-Agent Systems SIMulation framewOrk) which aims at supporting the validation activity of agent-based and multi-agent systems modeled and programmed by using an integrated approach based on the Distilled StateCharts formalism and the related programming tools. In particular, MASSIMO allows for the validation and the performance evaluation of the dynamic behavior (computations, interactions, and migrations) of individual and cooperating agents, the basic mechanisms of the distributed architectures supporting agents, namely agent platforms, and the functionalities of applications and systems based on agents. Finally, some results about the exploitation of MASSIMO for the validation of a consumer-driven agent-based e-Marketplace have been reported. Current efforts are being devoted to applying MASSIMO for the validation and performance evaluation of workflow instances enacted by agent-based enactment engines in the context of agent-based workflow management systems.

Acknowledgements

The work reported in this paper was partially supported by the M.I.U.R. (Italian Ministry of Instruction, University and Research) in the framework of the M.ENTE (Management of integrated ENTErprise) research project PON (N°12970-Mis.1.3).

References

[1] D.J. Bredin, D. Kotz, and D. Rus. Market-based Resource Control for Mobile Agents. Proc. of ACM Autonomous Agents, May 1998.

[2] FIPA Agent Management Support for Mobility Specification, DC00087C, 2002/05/10. http://www.fipa.org.

[3] S. Floyd, V. Paxson, Difficulties in simulating the Internet. IEEE/ACM Transactions on Networking, 9(4), pp. 392-403, 2001.

[4] G. Fortino, A. Garro, and W. Russo. An Integrated Approach for the Development and Validation of Multi Agent Systems. Computer Systems Science & Engineering, 20(4), pp.259-271, Jul. 2005.

[5] G. Fortino, A. Garro, W. Russo. Modelling and Analysis of Agent-Based Electronic Marketplaces. IPSI Transactions on Internet Research, 1(1), pp. 24-33, Jan. 2005.

[6] G. Fortino, A. Garro, W. Russo. E-commerce Services based on Mobile Agents. in Mehdi Khosrow-Pour, editor, Encyclopedia of E-Commerce, E-Government and Mobile Commerce, Idea Publishing Group, Hershey (PA), USA, 2006, to appear.

[7] G. Fortino, W. Russo. Multi-coordination of Mobile Agents: a Model and a Component-based Architecture. Proc. of the ACM Symposium on Applied Computing, Special Track on Coordination Models, Languages and Applications, Santa Fe, New Mexico, USA, 13-17 Mar, 2005.

[8] G. Fortino, W. Russo, and E. Zimeo. A Statecharts-based Software Development Process for Mobile Agents. Information and Software Technology, 46(13), pp. 907-921, Oct. 2004.

[9] N.M. Karnik and A.R. Tripathi. Design Issues in Mobile-Agent Programming Systems. IEEE Concurrency, 6(3), 52-61, 1998.

[10] M. Luck, P. McBurney, and C. Preist. A Manifesto for Agent technology: Towards Next Generation Computing. Autonomous Agents and Multi-Agent Systems, 9(3), pp. 203-252, 2004.

[11] M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. Proc. of Appia-Gulp-Prode Joint Conf. on Declarative Programming, L'Aquila, Italy. M.C. Meo and M. Vilares-Ferro (eds), pp. 13-28, 1999.

[12] M. Röhl, A. M. Uhrmacher. Controlled Experimentation with Agents - Models and Implementations. Proc. of the 5th International Workshop "Engineering Societies in the Agents World", 20-22 October 2004, Toulouse, France.

[13] C. Sierra, J. A. Rodríguez-Aguilar, P. Noriega, M. Esteva, and J. L. Arcos. Engineering Multi-agent Systems as Electronic Institutions. Novática, 170, July-August 2004.

[14] M. Strasser and M. Schwehm, A Performance Model for Mobile Agent Systems, Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), June 1997, 1132-1140.

[15] A.M. Uhrmacher, M. Röhl, and B. Kullick. The Role of Reflection in Simulating and Testing Agents: An Exploration Based on the Simulation System James. Applied Artificial Intelligence, (9-10):795-811, October-December, 2002.

[16] Y. Wang, K-L. Tan, and J. Ren. A Study of Building Internet Marketplaces on the Basis of Mobile Agents for Parallel Processing. *World Wide Web: Internet and Web Information Systems*, 5(1): 41-66, 2002.

[17] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems, 3(3):285–312, 2000.