Modeling Multi-Agent Systems through Event-driven Lightweight DSC-based Agents

Giancarlo Fortino
DEIS
Università della Calabria,
Via P. Bucci cubo 41c
87036 Rende (CS) Italy,
+39.0984.494063
g.fortino@unical.it

Alfredo Garro
DEIS
Università della Calabria,
Via P. Bucci cubo 41c
87036 Rende (CS) Italy,
+39.0984.494795
garro@unical.it

Samuele Mascillaro
DEIS
Università della Calabria,
Via P. Bucci cubo 41c
87036 Rende (CS) Italy,
+39.0984.494754
s.mascillaro@unical.it

Wilma Russo
DEIS
Università della Calabria,
Via P. Bucci cubo 41c
87036 Rende (CS) Italy,
+39.0984.494691
w.russo@unical.it

ABSTRACT
To date several agent models and related programming frameworks have been introduced for developing distributed applications in terms of multi-agent systems in open and dynamic environments. Among them, those based on lightweight architectures, asynchronous messages/events and state-based programming such as Jade, Bond and Actors have demonstrated great effectiveness for modeling open and distributed software systems. In this paper, we propose the Event-driven Lightweight Distilled StateCharts-based Agent (ELDA) model which is based on the same basics of the aforementioned agent models and frameworks, and further enables a more effective design through (i) Statecharts-based specification of the agent behavior, (ii) multiple coordination spaces for local/remote inter-agent and agent/non-agent-component interactions, and (iii) a coarse-grained strong agent mobility. A MAS based on the ELDA model can be easily designed through the ELDA meta-model and programmed through the ELDAFramework, a Java-based implementation of the meta-model. MAS programming is supported by the ELDATool, an Eclipse-based visual tool which also automates code generation. A simple yet effective case study is provided to exemplify the proposed model and its related tools.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *State diagrams.* D.2.6 [**Software Engineering**]: Programming Environments – *Integrated environments.* I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *Multiagent systems*.

General Terms
Design, Languages.

Keywords
Agent Models, Statecharts, Events, Multi-coordination, Mobility.

1. INTRODUCTION
The agent paradigm is one of the mainstream paradigms for the modeling and implementation of complex software systems, especially in open and dynamic environments. As advertised by a manifesto for agent technology [18] agents can be seen as both a design metaphor and a source of technology. In particular, from the design metaphor perspective, several agent models and related frameworks have been to date introduced [26]. Such agent models basically can be classified in two large groups: (i) models based on intelligent agent architectures [18, 21] ranging from reactive agents (e.g. Brook's subsumption architecture) to deliberative agents (e.g. BDI agents); (ii) models based on the mobile active object concept encompassing mobile agent architectures [5, 24]. Models of the first group are mainly oriented to problem-solving, planning and reasoning systems whereas models of the second group are more oriented to distributed computation in open and dynamic environments like the Internet.

In the context of the Internet computing, agent models and frameworks based on lightweight architectures, asynchronous messages/events and state-based programming such as Jade [3], Bond [4], and Actors [2], have demonstrated great effectiveness for modeling and programming agent-based distributed applications.

This paper proposes the Event-driven Lightweight Distilled StateCharts-based Agent (ELDA) model which aims at modeling multi-agent systems (MASs) in the context of open and dynamic computing environments. The ELDA model is based on the same fundamentals of the aforementioned models/frameworks and also introduces new enabling features which allow for a more effective development of MASs. These new features result as an enhancement of basic characteristics provided by two previously defined models (the MAO model [8] and the MC model [11]) and their integration. In particular with reference to the three models (Behavioral, Interaction and Mobility) on which the ELDA model is centered:

- The Behavioral and Mobility models are mainly based on characteristics derived from the MAO (Mobile Active Object) model, which allows for a multi paradigm approach to the construction of distributed applications in highly dynamic distributed environments. In particular, modeling of the agent behavior is based on the Distilled StateCharts (DSC) formalism and agent migration relies on a coarse grain strong mobility model.
- The Interaction model is an extension of the MC (Multi-Coordination) model, which is based on high-level events unifying access to and exploitation of underlying coordination spaces and agent server resources.

Jung, Michel, Ricci & Petta (eds.): *AT2AI-6 Working Notes, From Agent Theory to Agent Implementation, 6th Int. Workshop,*
May 13, 2008, AAMAS 2008, Estoril, Portugal, EU.

Moreover, the paper presents the ELDA meta-model which effectively supports the modeling of multi-agent systems (MAS) based on the ELDA model. A MAS designed through the proposed meta-model can be seamlessly coded by using the ELDAFramework, a Java implementation of the meta-model. The coding phase is facilitated by the ELDATool, an Eclipse-based visual tool which supports graphical programming of the agent behavior and automatic generation of code according to the ELDAFramework. Finally, to exemplify agent programming a case study concerning distributed information retrieval based on coordinated set of agents is also described.

The remainder of this paper is organized as follows. Section 2 presents the ELDA model by describing the Behavioral, Interaction, and Mobility models, and discusses its distinctive features. In section 3 the ELDA meta-model is described by using a views-based approach. Section 4 describes the case study whereas section 5 discusses related work. Finally conclusions are drawn and on-going research briefly delineated.

2. THE ELDA MODEL

The Event-driven Lightweight Distilled Statecharts-based Agent (ELDA) model is based on the concept of event-driven lightweight agent which is a single-threaded autonomous entity interacting through asynchronous events, executing upon reaction, and capable of migration. In particular, an event-driven lightweight agent is represented by the following tuple: <Id, Beh, DS, TC, EQ>, where, Id is the unique identifier of the agent, Beh is the agent behavior, DS is the data space or world knowledge of the agent, TC is the single thread of control supporting agent execution, and EQ is the event queue containing the incoming events targeting the agent.

The ELDA model relies on the Behavioral, Interaction and Mobility models. The Behavioral model allows for the specification of the agent behavior through the definition of agent states, transitions among states, and agent reactions (i.e. atomic actions attached to transitions). In particular, the agent behavior is defined to react to a specific set of events and a reaction can produce computations, and/or generation of one or more events, or a migration. The Interaction model, which is based on asynchronous events, enables multi-coordination among agents and between agents and non-agent components through the exploitation of multiple coordination structures. The Mobility model is based on a coarse grain strong mobility model which allows for agent transparent migration (both autonomous and passive) and easy programming of the migration points.

These models are founded on the Distilled StateCharts (DSCs) formalism [12] which is derived from the Statecharts formalism [15], a visual formalism that gained valuable success in the Software Engineering community mainly due to its appealing graphical features and the means it offers for the modeling of complex software systems. Statecharts, formerly introduced by Harel, were included in UML [22] and currently are the most used formalism for modeling the behavior of object-oriented reactive systems [16]. In the following sections these models are presented in detail.

2.1 The Behavioral Model

The behavior of ELDA agents is specified through DSCs [12] which are obtained from Statecharts as follows: (i) deriving some basic and advanced characteristics from Statecharts (deriving process), (ii) imposing some constraints on Statecharts (constraining process), and (iii) augmenting Statecharts with some features (augmenting process). In particular:

- *Deriving process*. DSCs derive the following characteristics from Statecharts:
  - *Structure based on a higraph* consisting of rounded rectilinear blobs representing states, linked together with transitions.
  - *Transitions based on the ECA rule*: E[C]/A, when E(vent) occurs and C(ondition) holds, the transition fires and A(ction) is atomically executed.
  - *OR decomposition of states* in hierarchies of states. The enclosing state is called composite state,the nested states are called substates and a state without nested states is called simple state.
  - *Inter-level state transitions* that can originate from or lead to nested states on any level of the hierarchy.
  - *History entrance pseudostates* allow entering the substate which was most recently visited. With respect to the composite state on which the pseudostates appear, *shallow history* indicates that history is applied only at the level of the composite state whereas *deep history* applies the same rule recursively to all levels of the state hierarchy of the composite state.
  - *Default entrances* indicate the substate of a composite state to be entered when a transition targets its border.
  - *Default history entrances* indicate the substate of a composite state to be entered in absence of history.
- *Constraining process*. DSCs impose the following constrains:
  - Each DSC has an enclosing top state.
  - States do not include activity, entry and exit actions. So activity is only carried out under the form of atomic actions labeling transitions.
  - Transitions (apart from default entrances and default history entrances) are always labeled by an event.
  - Each composite state has an initial pseudostate from which the default entrance originates, which can only be labeled by an action.
  - Run-to-completion execution semantics: an event can be processed only if the processing of the previous event has been fully completed. The sequence of operations which starts from fetching an event from the event queue to its complete processing is called *run-to-completion step*.
- *Augmentation process*. DSCs augment Statecharts with the following features:
  - Events are implicitly and asynchronously received through an event queue.
  - To explicitly and asynchronously emit events the action language provides the primitive `generate(<event>(<parameters>))`, where `event` is an event instance and `parameters` is the list of formal parameters of `event` including the event sender, the event target, and (possibly) a list of specific event parameters (see Section 2.2).

- Variables can be declared in each state and inside the actions to form a hierarchical data space.

STATES:
Composite States = {TopState, ACTIVE, ADSC},
Simple States = {INITIATED, TRANSIT, SUSPENDED, WAITING},
Final State=FS, Deep History Pseudostate=DHS,
Initial State=TopState(INITIATED)

TRANSITIONS:

t_1 : TopState(INITIATED) $\xrightarrow{\text{Invoke}}$ TopState(ACTIVE(DHS))
t_2 : TopState(ACTIVE(x)) $\xrightarrow{\text{Move}}$ TopState(TRANSIT)
t_3 : TopState(TRANSIT) $\xrightarrow{\text{Execute}}$ TopState(ACTIVE(DHS))
t_4 : TopState(ACTIVE(x)) $\xrightarrow{\text{Suspend}}$ TopState(SUSPENDED)
t_5 : TopState(SUSPENDED) $\xrightarrow{\text{Resume}}$ TopState(ACTIVE(DHS))
t_6 : TopState(ACTIVE(x)) $\xrightarrow{\text{Wait}}$ TopState(WAITING)
t_7 : TopState(WAITING) $\xrightarrow{\text{WakeUP}}$ TopState(ACTIVE(DHS))
t_8 : TopState(x) $\xrightarrow{\text{Destroy}}$ TopState(FS)
t_9 : TopState(x) $\xrightarrow{\text{Quit}}$ TopState(FS)
Default Entrance of DHS = ADSC;
notation: A(B) = A encloses B, A(x) = any state inside A

Figure 1. The FIPA-based template of the agent behavior.

Each ELDA behavior is forged according to an extended version of the FIPA agent lifecycle template [13] in which the ACTIVE state is always entered through a deep history pseudostate (DHS) to restore the agent execution state after agent migration and, in general, after agent suspension (see Section 2.3). In particular, the ACTIVE state contains the active DSC (ADSC) composite state to which the default entrance of the DHS points. The active agent behavior can be obtained by refining the ADSC. The resulting FIPA template of an ELDA agent is shown in Figure 1 by using a term-rewriting formalism [19].

2.2 The Interaction Model

Interactions of ELDA agents are based on Events which formalize both self-triggering events (Internal events) and requests to or notifications from the local agent server (Management, Coordination and Exception events). Events are further classified into OUT-events which are generated by the agent and always target the local agent server and IN-events which are generated by local agent server and delivered to target agents.

2.2.1 Internal Events

Internal events are generated by agents for proactively driving their behavior. In particular, a generated internal event is placed into the event queue of the generating agent so an internal event can be considered as both OUT and IN.

2.2.2 Management Events

Management events (see Table 1) which include requests to and notifications from the local agent server are further classified with reference to the following functionalities/services: agent lifecycle management, timer setting, and resource access.

The *agent lifecycle management* events allow for the management of agent creation, cloning, migration, suspension and destruction. In particular:

- *agent creation* is supported by the OUT-event CREATE and the IN-event CREATENOTIFY, which respectively formalize the request for the creation of one or more agents and the creation notification (if requested);
- *agent cloning* is enabled by the OUT-event CLONE and the IN-event CLONENOTIFY, which respectively formalize the request for cloning of an agent and the cloning notification (if requested);
- *agent migration* is requested by the OUT-event MOVEREQUEST, which embodies the identifier of the agent to be migrated and the destination agent server location, and is actually carried out after delivering the IN-event MOVE to the agent; after migration the agent execution is resumed through the IN-event EXECUTE (see Section 2.3);
- *agent waiting*, *suspension*, and *quit* are respectively requested through the OUT-events WAITREQUEST, SUSPENDREQUEST, and QUITREQUEST, and actualized through the IN-events WAIT, SUSPEND and QUIT; a waiting agent is waken up through the IN-event WAKEUP whereas a suspended agent is resumed through the IN-event RESUME; finally, an agent is started and destroyed by the agent server through the IN-events INITIATE and DESTROY, respectively.

The *timer setting* events allow for timing agent activities. In particular, the OUT-events CREATETIMER, STARTTIMER, STOPTIMER, RESETTIMER, RELEASETIMER allow for the creation, start, stop, reset and release of timers. A created timer is notified through the IN-event TIMERNOTIFY whereas a timeout event (i.e. an event raised when the timeout expires) is derived from the IN-event TIMEOUTNOTIFY.

The *resource access* events allow for access to the resources of the agent server such as files, console, databases, and sensor/actuators. A resource is requested through the OUT-event RESOURCEREQUEST and granted through the IN-event RESOURCENOTIFY. An input operation on a resource is requested through the OUT-event RESOURCEINPUTREQUEST and the provided input is sent to the agent through the IN-event RESOURCEINPUT; an output operation on a resource is requested through the OUT-event RESOURCEOUTPUT; finally, a resource is released through the RESOURCERELEASE event.

Table 1. Classification of Management events

MANAGEMENT		
Class	*Event Type OUT*	*Event Type IN*
LIFECYCLE	CREATE	CREATENOTIFY
	CLONE	CLONENOTIFY
	MOVEREQUEST	MOVE, EXECUTE
	WAITREQUEST	WAIT, WAKEUP
	SUSPENDREQUEST	SUSPEND, RESUME
	QUITREQUEST	QUIT
		INITIATE
		DESTROY
TIMER	CREATETIMER	TIMERNOTIFY
	STARTTIMER	TIMEOUTNOTIFY
	STOPTIMER	
	RESETTIMER	
	RELEASETIMER	
RESOURCE	RESOURCEREQUEST	RESOURCENOTIFY
	RESOURCEOUTPUT	
	RESOURCEINPUTREQUEST	RESOURCEINPUT
	RESOURCERELEASE	

2.2.3 Coordination Events

Coordination events (see Table 2) enable coordination acts between agents and between agents and non-agent components (e.g. remote objects, web services) according to a specific coordination model. The considered inter-agent coordination models are the Direct (synchronous and asynchronous), the Tuple-based, and the Publish/Subscribe event-based models, whereas the considered agent/non-agent components interaction are a general RMI Object model and a Web Services model. In particular:

- The Direct model is supported by the OUT-event MSGREQUEST and the IN-event MSG for asynchronous message passing, and by the OUT-event RPCREQUEST and the IN-event RPCRESULT for synchronous message passing. MSGREQUEST formalizes a request for sending an asynchronous message and contains the actual message of the MSG type to be sent, whereas MSG contains the message content to be delivered to the target agent. RPCREQUEST formalizes a request for sending a synchronous message and contains the message of the MSG type to be delivered to the target agent along with the back event of the RPCRESULT type. When the receiving agent accomplishes the request, the return value is encapsulated in the RPCRESULT previously specified which is passed to the requesting agent.
- The Linda-like Tuple-based model is enabled by the OUT-events IN, OUT, and RD, and by the IN-event RETURNTUPLE. OUT, IN, and RD formalize the corresponding Linda primitives for insertion, extraction and reading of a tuple, respectively. IN and RD can be either synchronous or asynchronous whereas OUT is only asynchronous. RETURNTUPLE embodies the tuple/s associated to a previously submitted IN or RD event.
- The Publish/Subscribe event-based model is supported by the OUT-events SUBSCRIBE, UNSUBSCRIBE, and PUBLISH, and by the IN-event EVTNOTIFICATION. SUBSCRIBE and UNSUBSCRIBE respectively formalize subscription and unsubscription to given events/topics, PUBLISH embodies a generated event, and EVTNOTIFICATION, which is specified in a previously submitted SUBSCRIBE event, contains an event notification.
- The RMI Object model is supported by the OUT-event RMIINVOKE and the IN-event RMIRETURN for the invocation of methods on non-agent components. RMIINVOKE contains the information needed to invoke a remote method on a remote object along with the back event of the RMIRETURN type which will embody the return value, if any, of the invoked method.
- The Web Services model is supported by the OUT-event SERVICEDISCOVERY, WSDLREQUEST, SERVICEINVOKE and by the IN-event DISCOVERYRESULT, WSDLRESULT e SERVICERESULT. SERVICEDISCOVERY formalizes the service discovery request and the DISCOVERYRESULT, which is sent back to the agent, contains the list of discovered services. WSDLREQUEST formalizes the WSDL request of the chosen service and the corresponding reply is provided through the WSDLRESULT event. SERVICEINVOKE formalizes the service invocation request and a possible return value is sent back through the SERVICERESULT event.

Table 2. Classification of Coordination events

COORDINATION		
Model	*Event Type OUT*	*Event Type IN*
DIRECT	MSGREQUEST	MSG
	RPCREQUEST	RPCRESULT
TUPLE-BASED	RD, IN	RETURNTUPLE
	OUT	
P/S_EVENT-BASED	SUBSCRIBE	
	UNSUBSCRIBE	
	PUBLISH	EVTNOTIFICATION
RMI OBJECT	RMIINVOKE	RMIRETURN
WEBSERVICES	SERVICEDISCOVERY	DISCOVERYRESULT
	WSDLREQUEST	WSDLRESULT
	SERVICEINVOKE	SERVICERESULT

2.2.4 Exception Events

Exception events are modeled as IN-events which are sent from the local agent server to agents to notify the impossibility to execute a service which was requested through the generation of an OUT-event. An exception is defined per each OUT-event and includes the description of the raised exception and its typology. An exception also contains the causingEvent, i.e. the instance of the event which has not been served by the local agent server and caused the exception. The exceptions are organized into a hierarchy which mirrors that of the Management and Coordination OUT-events.

2.3 The Mobility Model

The mobility model of ELDA agents is based on a strong mobility model which allows retaining the agent execution state. With respect to a fine-grain mobility type in which the agent migration can occur on a per-instruction basis the offered strong mobility model is of the coarse-grain type as ELDA agents can migrate on a per-action basis (i.e. after the execution on an action where an action is a set of instructions atomically executed). In particular the migration points of an ELDA agent match with the end of the run-to-completion step (see Section 2.1) and represent the only agent execution points in which it is possible to process Move events (see Section 2.2).

The migration of ELDA agents can be either autonomous (i.e. triggered by the agent itself) or passive (i.e. enforced by the system or induced by other agents) [25]. Specifically, migration points are known by the agent for autonomous migration as they are specified in the agent (DSC-based) behavior through an appropriate definition of states, events and transitions. In case of passive migration, migration points are not known in advance as they are induced by other agents or by the system; then, to obtain a behavior more reactive to migration could be necessary to program an ELDA agent with finer granularity of its actions.

The ELDA migration process is defined as follows. According to the FIPA template (see Figure 1 for the referred transitions), an ELDA agent after receiving the Move event passes into the Transit state (see t2) where it rests until the migration is completed; at the destination location the ELDA agent receives the Execute event, generated by the system, which brings the ELDA agent back into the state it was before the migration (see t3) by retaining the same execution state. State retaining is intrinsic due to the properties of the DSCs, particularly empty states and run-to-completion semantics, and to the structure of the

FIPA-based template, specifically the entrance with deep history in the ACTIVE state (see Section 2.1). In fact, after processing an event the execution state of an ELDA agent is automatically stored into its ACTIVE state so when the ELDA agent migrates it goes into the TRANSIT state without modifying its execution state as no exit action is allowed; after migration it is resumed and the ACTIVE state is re-entered through the deep history pseudostate which allows to set the current state to the state prior to migration without modifying the execution state as no entry action is allowed.

2.4 Distinctive features of the ELDA Model

The distinctive features of the ELDA model, which derive from the characteristics of the *Behavioral*, *Interaction* and *Mobility* models on which it is centered, can be summarized as follows:

- *Visual modeling*. The use of a visual language, based on the *UML* Statecharts [22], for modeling the behavior of ELDA agents, reduces the learning curve for their modeling due to the pervasive exploitation of UML in Industry and Academia, and increases the productivity of designers and programmers as it facilitates application development.
- *Executable specifications*. As the DSC-based behaviors of the ELDA agents are executable according to operational semantics derived from Statecharts, ELDA agents can be effectively verified (e.g. by means of formal methods or simulation) prior to their actual implementation and deployment.
- *Multi-coordination*. Coordination based on multiple models can facilitate application design, improve efficiency, and enable adaptability in dynamic and heterogeneous environments as it allows agents to choose among a variety of different coordination spaces and patterns which best fit their dynamic communication and synchronization needs.
- *Coarse-grain strong mobility*. The ELDA mobility model allows to easily identify and define the migration points of an agent so letting agent designers choose and control the granularity of the offered coarse-grain strong mobility. Moreover this mobility model can be easily implemented through any language which only provides native support to weak mobility like the Java language.

These distinctive features make the ELDA model appropriate for the modeling and implementation of distributed applications characterized by:

- complex computations to be performed on huge data sets (distributed data mining);
- tasks that are inherently parallel and distributed in nature (distributed workflow execution);
- search in huge data repositories, especially in presence of high network latency (distributed information retrieval);
- management and delivery of replicated content (content delivery networks);
- computation in dynamic and resource-constrained environments (wireless sensor networks).

Moreover, the ELDA model can be used to design other agent-based behavioral and interaction models. In particular:

- the ELDA *Behavioral* model based on states and events can support the design of agents ranging from reactive to BDI agents;
- the ELDA *Interaction* model based on multiple coordination paradigms can support the design of agent protocols ranging from simple agent-to-agent interactions to complex multi-agent negotiation protocols possibly based on specific ACL messages.

3. THE ELDA-BASED MAS META-MODEL

Multi agent systems (MASs) based on the ELDA model can be designed through the ELDA meta-model which provides all the modeling elements needed to the design phase. This meta-model is organized in six views correlated as shown in Figure 2:

- *Agent View*, which represents the structure of an ELDA agent and its relationships with the coordination and system spaces.
- *Event View*, which represents the structure of events.
- *SystemSpace View*, which represents the structure of the system space.
- *CoordinationSpace View*, which represents the hierarchy of the coordination spaces.
- *DSC View*, which represents the structure of a DSC.
- *FIPATemplate View*, which represents the structure of the FIPA template of the ELDA behavior.

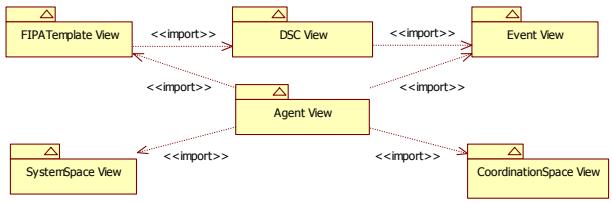


Figure 2. ELDA meta-model: Top-Level View

As shown in the Agent View (see Figure 3) an ELDA agent is composed of a single behavior which is specified through the FIPA template (see Figure 1) whose structure is contained in the FIPATemplate View (not reported for the sake of space). In particular, the FIPA template as well as the ADSC of an ELDA agent is modeled according to the DSC structure shown in the DSC View (see Figure 4). The Agent View also shows that an ELDA agent can interact with the System Space, which provides system services, through the ManagementOUT and ManagementIN events and with the Coordination Space, which provides coordination services, through the CoordinationOUT and CoordinationIN events. These events along with Internal and Exception events, defined in Section 2.2, are included in the Event View (not reported for the sake of space).

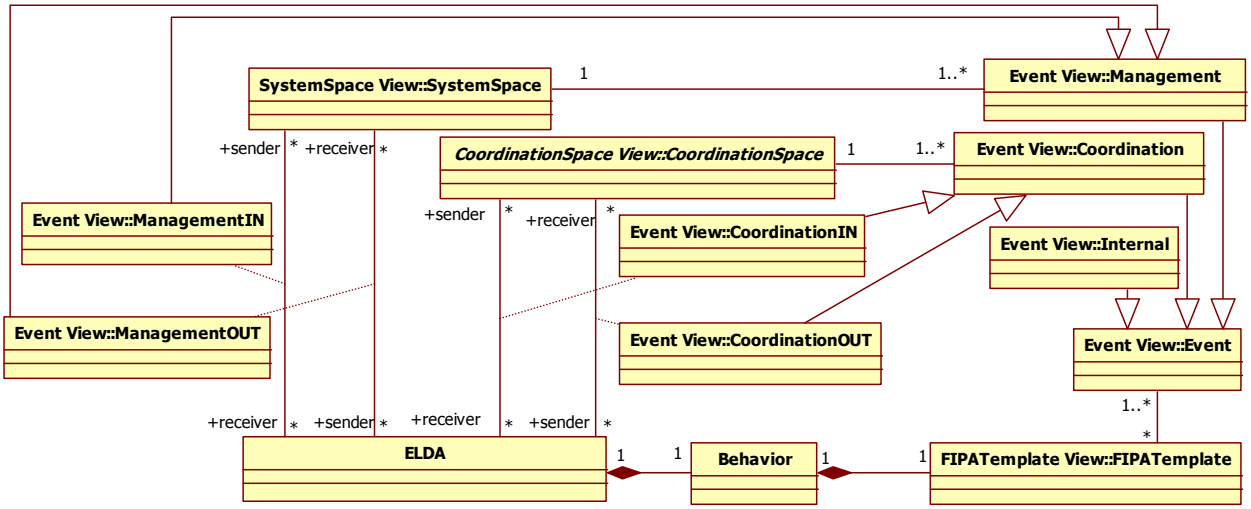


Figure 3. ELDA meta-model: Agent View

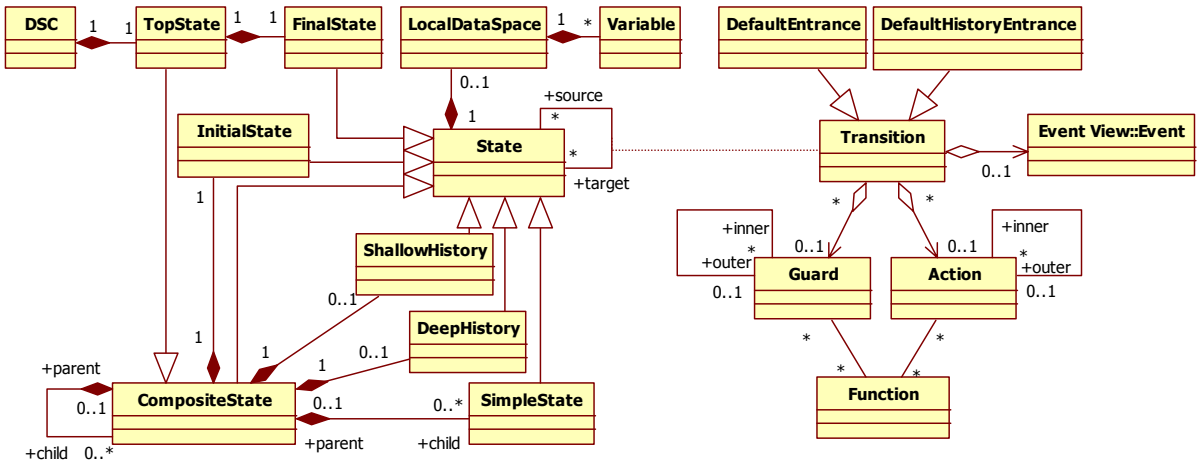


Figure 4. ELDA meta-model: DSC View

As shown in the SystemSpace View (see Figure 5), the System Space is composed of three basic managers, LifeCycleManager, TimerManager, and ResourceManager which handle the *Management* events of the *Lifecycle*, *Timer*, and *Resource* classes, respectively (see Table 1 and Section 2.2). It is worth noting that the ResourceManager provides access services to consoles, databases, files, sensors and other available local resources through associated sub-managers (ConsoleManager, DBManager, FileManager, SensorManager, etc) which handle such specific resources. Moreover, to extend the provided system services new special-purpose managers can be defined by the designer along with the related OUT- and IN-events.

The Coordination Space represents a local or global coordination structure based on a given coordination model through which agents interact. As shown in the CoordinationSpace View (see Figure 6), six coordination spaces are currently defined: DirectSpace (AsynchronousMsgSpace and SynchronousMsgSpace), TupleSpace, PublishSubscribeSpace, RMIObjectSpace, and WebServicesSpace. The interaction with these spaces is regulated by the *Coordination* events reported in Table 2 and described in Section 2.2. New coordination spaces can be introduced by defining new coordination space structures along with their related OUT/IN events.

To actually program ELDA-based MASs, the ELDA meta-model is currently implemented as a set of Java classes constituting the ELDAFramework. To enable rapid prototyping of MASs through the ELDAFramework, an Eclipse-based visual tool named ELDATool [9] is available which allows for visual programming of agent behaviors, graphical definition of events, and automatic generation of code (additional information about the ELDATool and the ELDAFramework can be found in [7]).

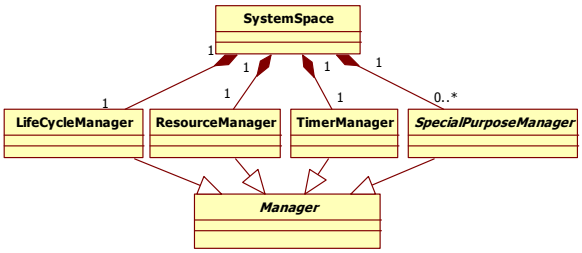


Figure 5. ELDA meta-model: SystemSpace View

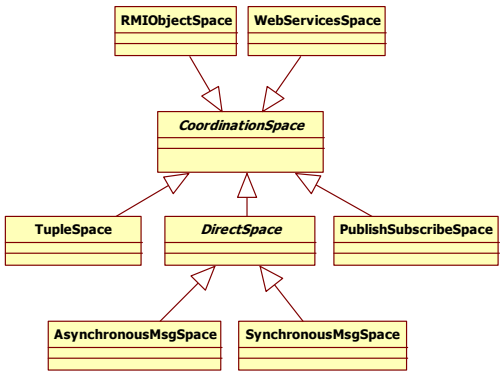


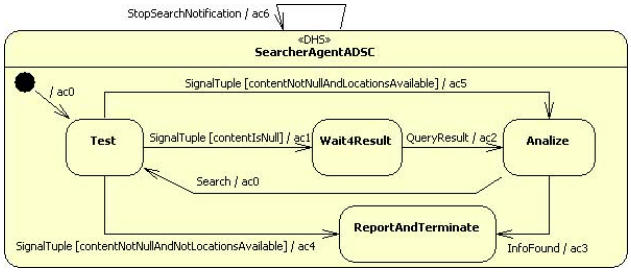
Figure 6. ELDA meta-model: CoordinationSpace View

4. A CASE STUDY

The proposed case study concerns with distributed information retrieval in a distributed computing system. In particular, a *User Agent* searches for specific information over networked federated locations by creating and launching a coordinated set of information *Searcher Agents* onto different locations. As soon as a *Searcher Agent* finds the desired information by locally interacting with distributed *Servant Agents*, stops all the other *Searcher Agents* and finally reports the found information to its owner *User Agent*. Owing to the multi-coordination features provided by the ELDA model, the application design choices are the following:

- *Searcher Agents* locally coordinate using a *local tuple space* to avoid duplicate search on the same location. Before searching for the information on a given location, the *Searcher Agent* checks for the presence of a signaling tuple in the *local tuple space* which is left by another *Searcher Agent* that has already visited the location. If the signaling tuple is not present, the *Searcher Agent* can search for the information; otherwise, it migrates to a new location, if a new location is available, or quits.
- *Searcher Agents* are stopped by exploiting an *event-based Publish/Subscribe* service. This allows a *Searcher Agent* to emit a searching stop event to easily stop all the other agents as soon as it finds the desired information.
- The *Searcher Agent* reports the found information to the *User Agent* through an *asynchronous message* instead of moving to the location of the *User Agent*. This is done to avoid the migration of the agent along with the found information, which would very likely take longer time.

In the following the behavior of the *Searcher Agent* (see Figure 7), programmed through the ELDATool [9], is fully described.




```
private void ac0 (ELDAEvent e) {
 generate(new ELDAEventRd(self(), signalTuple, false,
                  new SignalTuple(self())));
}
private void ac1 (ELDAEvent e) {
 if (firstHost)
  generate(new ELDAEventSubscribe(self(), stopSearch,
         new StopSearchNotification(self()) ));
 firstHost=false; servant=getServant();
 generate(new ELDAEventOut(self(), signalTuple,
         null, false));
 generate(new ELDAEventMSGRequest(self(),
         new QueryMsg(self(), servant, query)));
}
private void ac2 (ELDAEvent e) {
 Object result = ((QueryResult) e).getData();
 info=analize( result );
 if (info.found()) generate(new InfoFound(self()));
 else{
  locations.addLocations(info.getNeighbourLocations());
  if (locations.hasMoreLocations()) ac5(e);
  else ac6(e);
 }
}
private void ac3 (ELDAEvent e) {
 generate(new ELDAEventMSGRequest(self(),
         new Report(self(), owner, info)));
 generate(new ELDAEventPublish(self(), null,
          stopSearch));
 ac6(e);
}
private void ac4 (ELDAEvent e) {
 if (!firstHost)
  generate(new ELDAEventUnsubscribe(self(), stopSearch));
 generate(new ELDAEventQuitRequest(self()));
}
private void ac5 (ELDAEvent e) {
 generate(new ELDAEventMoveRequest(self(),
          locations.nextLocation()));
 generate(new Search(self()));
}
private void ac6 (ELDAEvent e) {
 generate(new ELDAEventUnsubscribe(self(), stopSearch));
 generate(new ELDAEventQuitRequest(self()));
}
```

Figure 7. The *Searcher Agent*: active behavior with the related action code

A *Searcher Agent*, once created and started on a given location, requests the reading of the `signalTuple` in the TupleSpace for checking if another *Searcher Agent* has already visited the location (see action `ac0`). In particular, the reading of the `signalTuple` is asynchronous and the back event to be delivered to the *Searcher Agent* is represented by the SIGNALTUPLE event (derived from RETURNTUPLE) which is handled as follows:

1) If the SIGNALTUPLE event has content null (i.e. no other *Searcher Agent* has searched in this location), the *Searcher Agent* subscribes to the `stopSearch` topic through the

SUBSCRIBE event, if the location visited is the first one (`firstHost`=true), inserts the signalling tuple through the OUT event, and sends to the `servantAgent` a query through the QUERYMSG event (derived from MSG) asking for the searched information (see action `ac1`). In particular, SUBSCRIBE embodies the STOPSEARCHNOTIFICATION event (derived from EVTNOTIFICATION), which is delivered to the *Searcher Agent* when another one finds the searched information.

2) If the SIGNALTUPLE event has content not null and other locations are available, the *Searcher Agent* generates a MOVEREQUEST to the next available location and the internal event SEARCH to keep searching (see action `ac5`). At the new location the migrated *Searcher Agent*, after receiving the internal SEARCH event, enters again into the Test state by executing `ac0`.
3) If the SIGNALTUPLE event has content not null and no other location is available, the *Searcher Agent* unsubscribes from the `stopSearch` topic through the UNSUBSCRIBE event, if `firstHost`=false, and generates a QUITREQUEST event (see action `ac4`).

Consequently to the case 1, as soon as the QUERYRESULT event (derived from MSG) sent by `servantAgent` is delivered to the *Searcher Agent*, its content is analyzed (see action `ac2`): if the searched information is found, the internal event INFOFOUND is generated; otherwise, possible new locations included in the obtained `info` are added to the list of available locations (`locations`) and, if this list has more locations, a MOVEREQUEST to the next available location and the internal event SEARCH are generated; else the *Searcher Agent* unsubscribes from the `stopSearch` topic and quits. When INFOFOUND is received by a *Searcher Agent*, the agent reports to its owner through the REPORT event (derived from MSG), publishes through the PUBLISH event a `stopSearch` topic, unsubscribes from the `stopSearch` topic and quits (see action `ac3`).

In whatever state the *Searcher Agent* is, when it receives STOPSEARCHNOTIFICATION, unsubscribes from the `stopSearch` topic and quits (see action `ac6`).

5. RELATED WORK

Several agent models and frameworks are related to the proposed ELDA model with respect to the three main dimensions of agent modeling: behavior, interaction and mobility. In the following, the comparison is restricted to those agent models which share the following basic features with the ELDA model: lightweight agent architectures, asynchronous agent interaction and state-based programming. In particular, we have considered the agent models on which Jade [3], Bond [4, 20] and Actors [1, 2] are based.

With reference to the agent behavior model, the behavioral model of ELDA agents is based on Distilled StateCharts, Jade offers, among different agent behavior types, an agent behavior (called FSMBehaviour) based on flat finite state machines (FSMs), an add-on of Jade (SmartAgent) [17] provides an extension of the Jade FSMBehaviour (named HSMBehaviour) based on hierarchical finite state machines (HSMs), Bond defines the agent behavior as a multi-plane state machine in which each plane is modeled as an FSM, and Actors are based on agents modeled as active objects with state variables and action methods. The execution semantics of the HSMBehaviour, the ELDA behavior, the Bond agent behavior and the actor behavior is very similar: a message/event triggers the execution of an action; when the action execution is terminated the next available message/event is fetched and processed. Conversely, the execution semantics of the Jade FSMBehaviour is not driven by messages/events but by action completions triggering transitions. The advantages of the ELDA behavior with respect to the other agent behaviors relies on the DSC formalism which (i) overtakes the limited features of the FSMBehaviour by also introducing hierarchy and history and of the HSMBehaviour which do not exploit history, (ii) derives from the well formalized Statecharts formalism whereas a well founded formalization and correlated tools for the Bond multi-plane state machine and the actor behavior are not yet available, (iii) lends itself to be easily supported by a visual tool.

With reference to the agent interaction model, the ELDA model provides the interesting notion of multi-coordination [11] which enables a holistic exploitation of multiple coordination spaces, each based on a different coordination model. This is strategic in the context of open and dynamic environments where agents to fulfill their goal should interact with other agents or with other components through different coordination models. Jade, Bond and Actors are mainly based on asynchronous message passing, even though Bond agents can also interact through synchronous message passing, a tuple space based on the IBM TSpace and a publish/subscribe event model. From the interaction perspective further and interesting related work is represented by the coordination infrastructures [23] such as reactive tuple spaces (e.g. TuCSon), environmental and organizational artifacts. These infrastructures/artifacts can be easily integrated and used as new coordination spaces (see Section 3) for ELDA agents.

With reference to the agent mobility model, Jade, Bond and Actors (in particular the implementation of Actors carried out in the ActorFoudry framework [1]) are based on a weak mobility model [14]. Conversely, the ELDA model is based on strong mobility of the programmable coarse-grain type which enables active and passive migration by simplifying the management code of the agent migration with respect to agents based on weak mobility whose programming is complicated by the explicit management (save and restore) of the agent execution state.

6. CONCLUSION

This paper has proposed the ELDA model and its exemplification through a case study developed by using ELDA-based design methods and programming tools. In particular, the ELDA model provides the following distinctive features: (i) DSC-centered behavioral specification which supports formal-driven and visual-based modeling of the agent behavior so enabling rapid prototyping due to both the graphical representation of the agent behavior and the availability of Statecharts-based formal tools for its validation; (ii) event-based interaction between agents and the hosting local agent server through an easily extensible system space which provides basic and advanced services for agent lifecycle management, timer handling and resource access; (iii) local/remote inter-agent and agents/non-agent-components interaction based on multiple coordination spaces which rely on both already available models (e.g. message passing, Linda-like tuple spaces, publish/subscribe, etc) and models to be purposely

defined; (iv) autonomous and passive agent migration based on strong mobility of the programmable coarse-grain type.

The aforementioned features make the ELDA model more effective than related agent models and frameworks currently available in the literature. Moreover, rapid prototyping of ELDA-based multi-agent systems is enabled by effective visual programming tools (ELDATool) and frameworks (ELDAFramework).

On the basis of the obtained results current research is focused on: (i) finalizing a simulated execution platform enabling functional and non-functional validation of ELDA-based MASs before their deployment stage; (ii) defining a full-fledged methodology supporting the development of ELDA-based MASs from analysis to implementation and validation; this research is based on the experiences gained by using PASSI [6] and GAIA [10] to drive the analysis and design phases in the development of MASs centered on Statecharts-based agents; (iii) formalizing the ELDA model through rewriting logic-based techniques.

7. REFERENCES

[1] Astley, M. 1999. Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures. Doctoral Thesis, University of Illinois at Urbana-Champaign.

[2] Astley, M. and Agha, G. 1998. Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management. In Proceedings of ACM SIGSOFT 6th International Symposium on Foundations of Software Engineering (FSE-6 SIGSOFT'98, Orlando, FL, USA, 1998).

[3] Bellifemine, F., Poggi, and A., Rimassa, G. 2001. Developing multi agent systems with a FIPA-compliant agent framework. Software Practice And Experience 31, 103-128.

[4] Boloni, L. and Marinescu, D.C. 1999. A Multi-Plane State Machine Agent Model. Technical Report CSD-TR-99-027, Computer Science Department, Purdue University.

[5] Braun, P. and Rossak, W. 2005. Mobile Agents: basic concepts, mobility models, & the tracy toolkit. Morgan Kaufmann Pub., Heildelberg, Germany.

[6] Cossentino, M., Fortino, G., Garro, A., Mascillaro, S. and Russo, W. 2008. PASSIM: a simulation-based process for the development of multi-agent systems. Int. J. Agent-Oriented Software Engineering 2(2), 132-170.

[7] ELDATool documentation and software, http://lisdip.deis.unical.it/software/eldatool.

[8] Fortino, G., Frattolillo, F., Russo, W. and Zimeo, E.. 2002. Mobile Active Objects for highly dynamic distributed computing. In Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), Workshop - Java for Parallel and Distributed Computing (JPDC'02, Fort Lauderdale, FL, Apr. 15-19, 2002).

[9] Fortino, G., Garro A., Mascillaro S., and Russo W. 2007. ELDATool: A Statecharts-based Tool for Prototyping Multi-Agent Systems. In Proceedings of Workshop on Objects and Agents (WOA'07, Genova, IT, Sept. 24-25, 2007), pp. 14-19.

[10] Fortino, G., Garro, A., and Russo, W. 2005. An Integrated Approach for the Development and Validation of Multi Agent Systems. Computer Systems Science & Engineering 20, 4, 94-107.

[11] Fortino, G. and Russo, W. 2005. Multi-coordination of Mobile Agents: a Model and a Component-based Architecture. In Proceedings of 20th Annual ACM Symposium on Applied Computing (SAC'05, Santa Fe, NM, USA, Mar. 13-17, 2005), Special Track on Coordination Models, Languages and Applications, vol. 1, pp. 443-450.

[12] Fortino, G., Russo, W. and Zimeo, E. 2004. Statecharts-based Software Development Process for Mobile Agents. Information and Software Technology 46, 13, 907-921.

[13] Foundation for Intelligent Physical Agents: Agent Management Support for Mobility Specification, DC00087C, 2002/05/10, http://www.fipa.org

[14] Fuggetta, A., Picco, G.P., and Vigna, G. 1998. Understanding Code Mobility. IEEE Trans. on Software Engineering 24, 5, 342-361.

[15] Harel, D. 1987. Statecharts: a visual formalism for complex systems. Science of Computer Programming 8, 231-274.

[16] Harel, D. and Gery, E. 1997. Executable Object Modelling with Statecharts. IEEE Computer 30, 7, 31-42.

[17] Kessler, R., Griss, M., Remick, B. and Delucchi, R. 2004. A hierarchical state machine using JADE behaviours with animation visualization. In Proceedings of Int'l Joint Conference on Autonomous Agents and Multi Agents Systems (AAMAS'04, New York City, NY, USA, July, 2004).

[18] Luck, M., McBurney, P., and Preist, C. 2004. A manifesto for agent technology: towards next generation computing. Autonomous Agents and Multi-Agent Systems 9, 3, 2004, 203-252.

[19] Lilius, J. and Paltor, I. P. 1999. The semantics of UML State Machines. Technical Report N. 273. Turku Centre of Computer Science (TUCS).

[20] Marinescu, D.C. 2002. Internet-based Workflow Management. John Wiley & Sons, Inc., New York.

[21] Nwana, H.S. 1996. Software Agents: an overview. Knowledge Engineering Review 11, 3, 205-244.

[22] Object Management Group.2005. Unified Modelling Language Specification (N. formal/2005-07-05) v. 2.0.

[23] Omicini, A., Ossowsky, S., and Ricci, A. 2004. Coordination infrastructures in the engineering of multi-agent systems. In Proceedings of Methodologies and Software Engineering for Agent Systems. (Kluwer, New York, 2004).

[24] Silva, A.R., Romao, A., Deugo, and D., Mira da Silva, M. 2001. Towards a reference model for surveying mobile agent systems. Autonomous Agent and Multi-Agent Systems 4, 3, 187-231.

[25] Xu, D., Yin, J., Deng, Y., and Ding, J. 2003. A Formal Architectural Model for Logical Agent Mobility. IEEE Trans. Software Eng. 29, 1, 31-45.

[26] Zambonelli, F. and Omicini, A. 2004. Challenges and research directions in agent oriented software engineering. Autonomous Agents and Multi-Agent Systems 9, 3, 253-284.