ELDATool: A Statecharts-based Tool for Prototyping Multi-Agent Systems

Giancarlo Fortino, Alfredo Garro, Samuele Mascillaro and Wilma Russo

Abstract— **This paper briefly describes the ELDATool, a Statecharts-based visual tool for the rapid prototyping of Multi-Agent Systems based on the Event-driven Lightweight Distilled Statecharts-based Agents (ELDA) model. In particular, the ELDATool, which is implemented in Java as an Eclipse plug-in, supports an iterative process involving the following phases: detailed design, automatic code generation and simulation. The high-level design, which is the input to this iterative process, can be obtained through currently available agent-oriented methodologies such as PASSI and GAIA. In order to show the main characteristics of the ELDATool, a simple case study is presented.**

Index Terms—**Visual Tools, Multi-Agent Systems, Distilled StateCharts, State-based Programming.**

I. Introduction

Agent oriented software engineering [1] aims at providing methodologies and tools for the development of complex and distributed software systems through the agent paradigm in terms of Multi-Agent Systems (MASs).

Several agent-oriented methodologies (PASSI [2], GAIA [3], SODA [4], INGENIAS [5], DSC-based [6, 7], etc.) have been to date proposed for supporting the development life-cycle of MASs. Few of them are also equipped with visual tools capable of supporting all the phases of the development life-cycle. The availability of such tools is widely considered to be strategic for supporting a rapid prototyping of the MAS under-development.

This paper introduces the ELDATool which aims at supporting the DSC-based agent-oriented methodology proposed in [6, 7, 8]. This methodology covers the modelling, implementation and simulation phases of MAS based on the ELDA (Event-driven Lightweight Distilled Statecharts-based Agents) model.

According to the ELDA model [6, 9] a multi-agent system is modelled at high-level as a set of different types of agents and a set of interaction events among the agents. An ELDA agent consists of a unique identifier, a data space, a dynamic behaviour, a single thread of control, and a queue of the received events. In particular, the dynamic behaviour of an agent is specified through the Distilled Statecharts formalism [6], derived from the well-known Statecharts [10]. Modelling an agent is basically carried out by specifying its behaviour as a hierarchical state machine compliant with the state-based template of the FIPA agent [11] and by defining the events which can be received and generated. While events are implicitly received through the event queue, they are explicitly emitted through the *generate* primitive. The received events are called IN-events, whereas the generated events are called OUT-events. In particular, events formalize three kinds of interactions [9]: (i) internal, which are sent by the agent to itself to proactively drive its activity; (ii) management, which are used to interact with the agent management system for requesting services and resources; (iii) coordination, which are exploited to interact with local or remote agents/entities through a given coordination space.

The ELDA model is implemented in the ELDA framework, an object-oriented framework which provides the programming abstractions to implement ELDA-based MAS. Currently, the ELDA framework is implemented in Java.

The ELDATool incorporates the ELDA framework and provides a graphical integrated development environment based on Eclipse [12]. The ELDATool is exemplified through a simple case study concerning with the modelling of a contractor mobile agent within an agent-based e-Marketplace.

The rest of this paper is organized as follows. Section 2 enumerates the system requirements of the ELDATool. Section 3 and section 4 respectively present the ELDATool design and implementation. Section 5 describes the use of the ELDATool through the simple case study developed. Finally conclusions are drawn and on-going work anticipated.

II. System Requirements

ELDATool aims at supporting the MAS designer for the rapid prototyping of MASs based on the ELDA model according to the iterative process shown in Figure 1.

G. Fortino is with the Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), 87036 Italy. (corresponding author; phone: +39.0984.494063; fax: +39.0984.494713; e-mail: g.fortino@unical.it).

A. Garro is with the Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), 87036 Italy. (e-mail: garro @unical.it).

S. Mascillaro is with the Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), 87036 Italy. (e-mail: samuele.mascillaro@deis.unical.it).

W. Russo is with the Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), 87036 Italy. (e-mail: w.russo @unical.it).

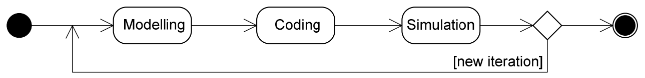


Figure1: Iterative process for prototyping ELDA-based MASs.

To support the *Modelling* phase, the tool offers the basic functionality of visual modelling of the active state of the agent behaviour through a DSC-based Hierarchical State Machine. The active state is a composite state in which the agent performs its main activity. In particular, the following modelling features are supported:

- definition of the internal states of the active state;
- definition of the events, generated (or OUT-events) and received (IN-events) by/from the ELDA agent, by extending appositely the base events provided by the Java implementation of the ELDA framework (or ELDAFramework) or events previously defined by the user;
- definition of the transitions between states which involves:
  - the use of the IN-events previously defined for labelling the transitions;
  - (possibly) the definition and the use of the guards associated to the transitions;
  - (possibly) the definition and the use of the actions associated to the transitions.

The obtained graphical modelling is serialized into XML-like files.

To support the *Coding* phase, the tool offers the functionality of automatic code generation by translating the XML-like files produced after the *Modelling* phase into Java code based on the ELDAFramework.

Finally, to support the *Simulation* phase, the tool is based on the MASSIMO framework [13] and offers the following functionalities:

- implementation of the simulator through the definition of the network topology of the agent platform, the initial location of the agents, the definition of the performance parameters, etc;
- execution of the simulator for performing the simulation;
- gathering of the values of the parameters defined for the performance measurements.

The ELDATool is implemented in Java as an Eclipse plug-in to exploit several frameworks which fully support the development of visual editors. Moreover, the high diffusion of Eclipse in the research community makes the tool immediately available to the Eclipse users and the learning process of the tool is so quicker.

III. Design

The architecture of the ELDATool is component-based; each component is responsible of the specific aspects of the *Modelling*, *Coding* and *Simulation* phases.

In particular, for each different modelling aspect the following editors have been identified and designed:

- DSCEditor, for modelling the active state of an ELDA agent;
- EventEditor, for defining the events;
- GuardEditor, for defining the guards;
- ActionEditor, for defining the actions;
- FunctionEditor, for defining the supporting functions.

Each editor is capable of handling (visually or not) the elements of its reference meta-model and producing an instance of this meta-model (or specific model) as output.

The CodeGenerator component uses the models produced by the editors as input to offer the functionalities needed for the code generation according to the classes constituting the ELDAFramework.

The Simulator component uses the code produced by the CodeGenerator component to support the *Simulation* phase.

Figure 2 shows the components, the dependence relationships among them, and their contextualization with respect to the process phases.

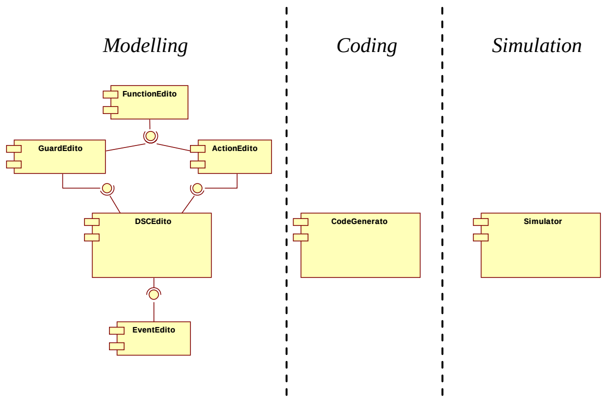


Figura 2: The ELDATool components

IV. ELDATool Implementation

Currently the ELDATool supports the first two phases of the process: *Modelling* and *Coding*. The architectural components described in section II are implemented in Java by exploiting:

- the Eclipse platform [12], which is a widely-used Integrated Development Environment (IDE) with extensible architecture based on plug-ins, i.e. independent components which can be easily installed and integrated in the IDE;
- the Graphical Editing Framework (GEF) [14] which allows for the development of visual editors in Eclipse by offering high support for the management of the user interactions;
- the Eclipse Modelling Framework (EMF) [15] which supports the modelling phase of a structural model and the automatic generation and manipulation of its Java implementation.

The editor components (see section II) are implemented according to the architectural pattern Model-View-Controller (MVC) to support the user-interaction handling (View-Controller) and the manipulation of the model in response to the generated events (Model).

In particular, user-interaction handling is implemented by extending the classes provided by GEF whereas the model manipulation is carried out by the plug-ins automatically generated by EMF. It is worth noting that EMF generates a plug-in exposing the interfaces needed for the instantiation of the implemented meta-model. Accordingly, each editor component is constituted by an EMF-generated plug-in which manages the model and a plug-in which handles the user interaction.

In order to ease the deployment of the ELDATool the number of its constituting plug-ins was minimized. In particular, the plug-ins which manage the models are separately implemented whereas the plug-ins handling the user-interaction and supporting the code generation are integrated in a unique plug-in, the ELDAEditor.

As a consequence, the following plug-ins are implemented:

- *DSCModel*, which contains the implementation of the DSC meta-model;
- *EventModel*, which contains the implementation of the Event meta-model;
- *ActionGuardModel*, which contains the implementation of the Action and Guard meta-model;
- *FunctionModel*, which contains the implementation of the Supporting Function meta-model;
- *ELDAEditor*, which contains all the editor and the code generator.

It is worth noting that the models, obtained through instantiating the related meta-models and by using the editor made available by the ELDATool, are serialized into independent XML-like files with different extensions (see Table 1).

Table 1: Extensions of the XML-like files associated to the models

Model	**File Extension**
Event	event
Action	action
Guard	guard
Function	function
Active State	dsc

Figure 3 highlights and clarifies the dependence relationships among the implemented plug-ins and the GEF/EMF plug-in.

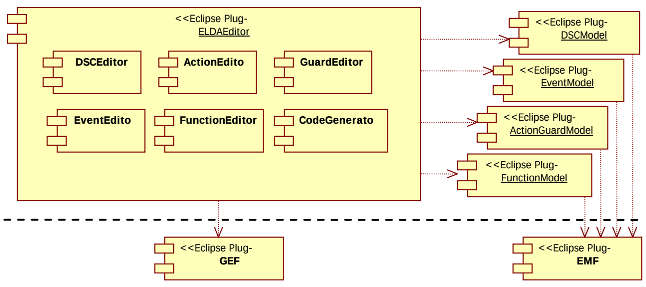


Figura 3: The ELDATool Plug-ins.

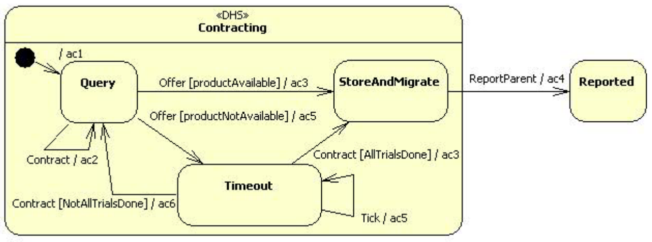
The ELDATool will be released as a set of plug-ins and a jar named ELDAFramework.jar which contains the Java implementation of the ELDA framework. It is worth noting that to install the ELDATool it is only necessary to copy the set of plug-ins and the ELDAFramework.jar into the plugins folder of Eclipse and restart Eclipse. The software requirements of the ELDATool are: Eclipse ver. 3.3, GEF ver. 3.3, EMF ver. 2.3.0 and JRE ver. 1.5.

V. A CASE STUDY

In this section the use of the ELDATool is shown by illustrating the modelling of an ELDA agent named Contractor Mobile Agent (CMA) which operates within an agent-based e-Marketplace. After a discovery phase of the vendors offering a specific product which was carried out by another type of agent, the CMA has the goal of supporting the phase of contracting with the vendors found.

Figure 4 shows the active state of the CMA behaviour and Figure 5 reports its guards, actions, and supporting functions.

In particular, the CMA, received the identifier and the location of a given vendor and the product to buy, migrates to the vendor location (see action ac1) and starts the contracting phase (see action ac2). After obtaining the offer, if the product is immediately available (see guard productAvailable) the CMA archives the offer and comes back to the starting location (see action ac3); otherwise, the CMA waits for the product availability until a timeout expiration (see action ac5). After the timeout expiration, the CMA restarts the contracting phase if the number of trials is greater 0 (see guard NotAllTrialsDone); otherwise, the CMA archives the offer and migrates to the starting location (see action ac3). Finally, the CMA notifies the details of the contracting phase to its owner (see action ac4).



«DHS»
Contracting
/ ac1
Query
Offer [productAvailable] / ac3
StoreAndMigrate
ReportParent / ac4
Reported
Offer [productNotAvailable] / ac5
Contract / ac2
Contract [AllTrialsDone] / ac3
Timeout
Contract [NotAllTrialsDone] / ac6
Tick / ac5

Figure 4: The active state of the CMA

```
Guards Definitions
private boolean productAvailable(ELDAEvent e){
 OfferMsg offer=(OfferMsg) e;
 if (offer.getProductAvalaible())
  return true;
 return false;
}
private boolean productNotAvailable(ELDAEvent e){
 return !productAvailable(e);
}
private boolean NotAllTrialsDone(ELDAEvent e){
 return !AllTrialsDone(e);
}
private boolean AllTrialsDone(ELDAEvent e){
 if(trials==0)
  return true;
 return false;
}
Actions Definitions
private void ac1(ELDAEvent e){
 generate(new ELDAEventMoveRequest(self(), self(),
                                       VATarget.getCurrLocation()));
 generate(new Contract(self(), self()));
}
private void ac2(ELDAEvent e){
 PriceQueryMsg priceQuery= new PriceQueryMsg(self(), VATarget, null);
 generate(new ELDAEventMSGRequest(self(), VATarget, priceQuery));
}
private void ac3(ELDAEvent e){
 OfferMsg offer=(OfferMsg) e;
 storeVAOffer(offer);
 generate(new ELDAEventMoveRequest(self(), self(),
                                              owner.getCurrLocation()));
 generate(new ReportParent(self(), self()));
}
private void ac4(ELDAEvent e){
 PPriceMsg pPrice=new PPriceMsg(self(), owner, VAOffer);
 generate(new ELDAEventMSGRequest(self(), owner, pPrice));
 generate(new ELDAEventQuitRequest(self()));
}
private void ac5(ELDAEvent e){
 if(timeout>0){
  timeout--;
  generate(new Tick(self(), self()));
 }
 else{
  timeout=100;
  generate(new Contract(self(), self()));
 }
}
private void ac6(ELDAEvent e){
 trials--;
 ac2(e) ;
}
Functions Definitions
private void storeVAOffer(OfferMsg offer){
 //omissis
}
```
Figura 5: Guards, actions and functions of the CMA behavior

To exemplify the use of the ELDATool the following activities are briefly illustrated: (A) visual definition of the active state, (B) definition of events, (C) definition of guards, (D) definition of actions, and (E) definition of supporting functions.

A. *Definition of the active state*

The result of this activity is the model of the active state of the agent behaviour; the transitions defined among the states are based on events, guards, and actions which are previously defined. Moreover, during this activity, it is possible to define local variables for each state so constituting a hierarchical data space. Figure 6 shows a snapshot of the active state of the CMA behaviour obtained through the DSCEditor.

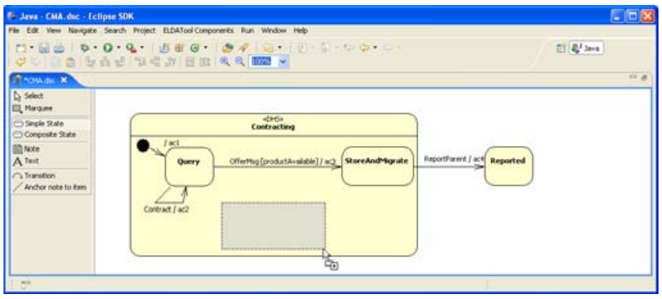
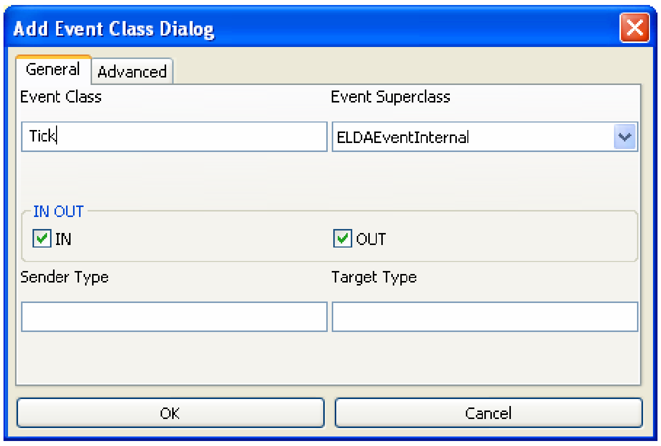


Figure 6: Definition of states

B. *Definition of events*

The EventEditor allows for the definition of new events by extending the events already offered by the ELDAFramework. In particular, for each event, the event name, the event class of the ELDAFramework to be extended, and (possibly) new parameters can be defined. Figure 7 shows the event definition dialog through which the event Tick is defined as extension of the base event ELDAEventInternal.

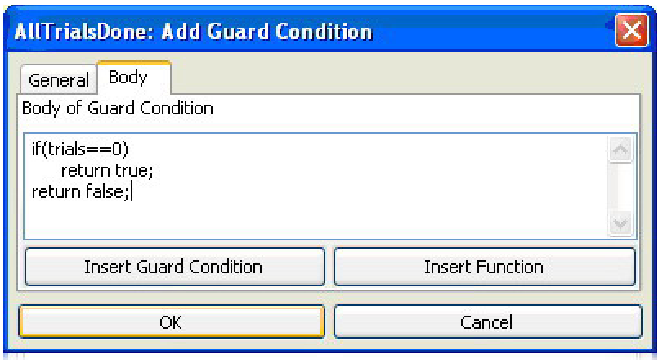


Event Class: Tick
Event Superclass: ELDAEventInternal
IN OUT: ✓ IN ✓ OUT
Sender Type:
Target Type:

Figure 7: Definition of an event

C. *Definition of the guards*

The definition of a guard involves the definition of its name and the boolean expression associated to it (or guard body). Within a guard body it is possible to use variables belonging to the data space (e.g. the integer variable *trials*), guards previously defined and supporting functions. Figure 8 shows the definition of the guard named AllTrialsDone to be associated to the transition between the TimeOut state and the StoreAndMigrate state.



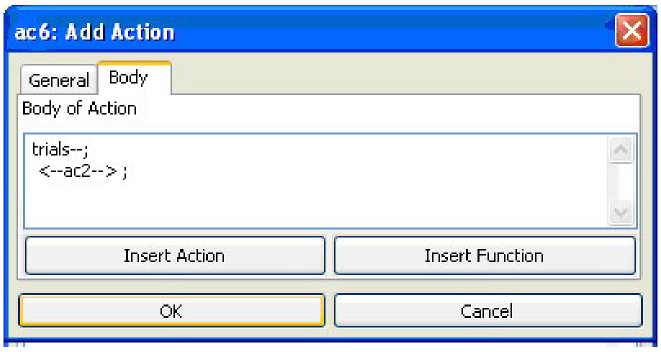
Body of Guard Condition

```
if(trials==0)
    return true;
return false;
```

Figure 8: Definition of a guard

D. *Definition of the actions*

The definition of an action involves the definition of its name and the instructions which costitute it. In an action, it is possibile to use variables belonging to the data space, actions previously defined, and supporting functions. Figure 9 shows the definition of the action ac6 which uses the action ac2 previously defined.

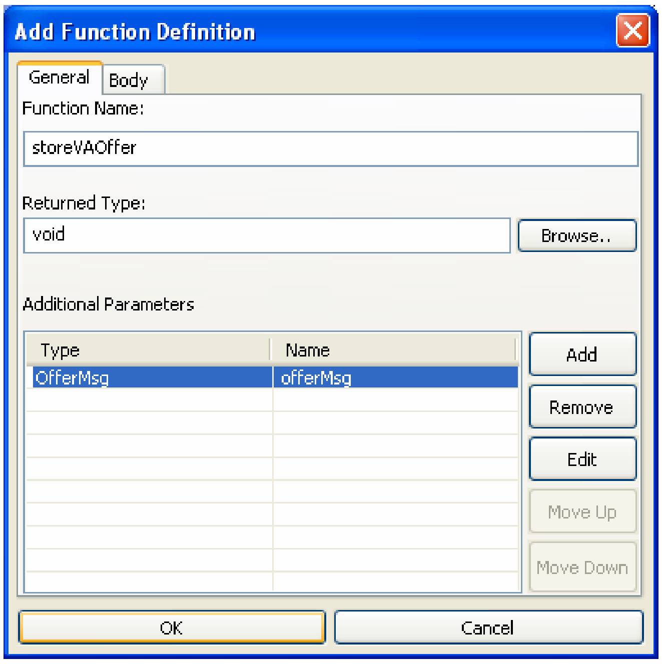


```
trials--;
 <--ac2--> ;
```

Figure 9: Definition of an action

E. *Definition of supporting functions*

The definition of supporting functions which can be used by actions and guards to improve design modularity, is constituted by the specification of the function name, of the type of the returned value, of the parameters and of the function body. Figure 10 shows the dialog for the definition of the supporting functions; in particular, the StoreVAOffer function is defined which returns void and has only the parameter offerMsg of the OfferMsg type.



Function Name: storeVAOffer

Returned Type: void

Additional Parameters

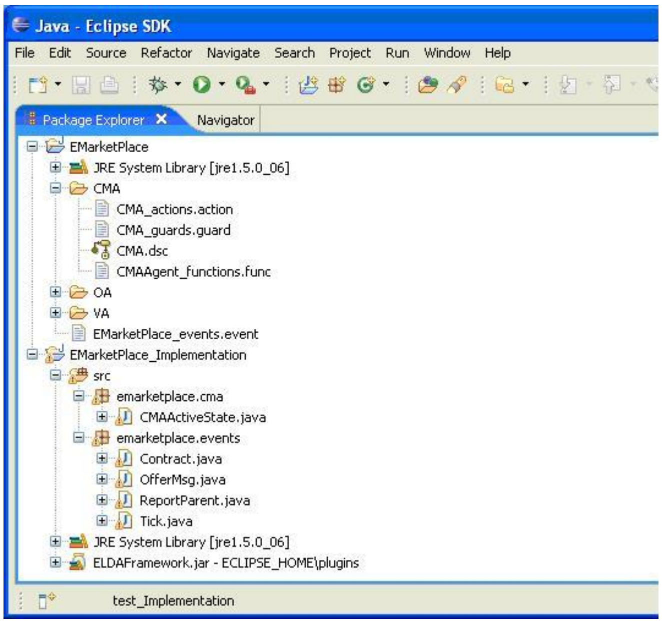
Type	Name
OfferMsg	offerMsg

Figure 10: Definition of a supporting function

F. *Code generation*

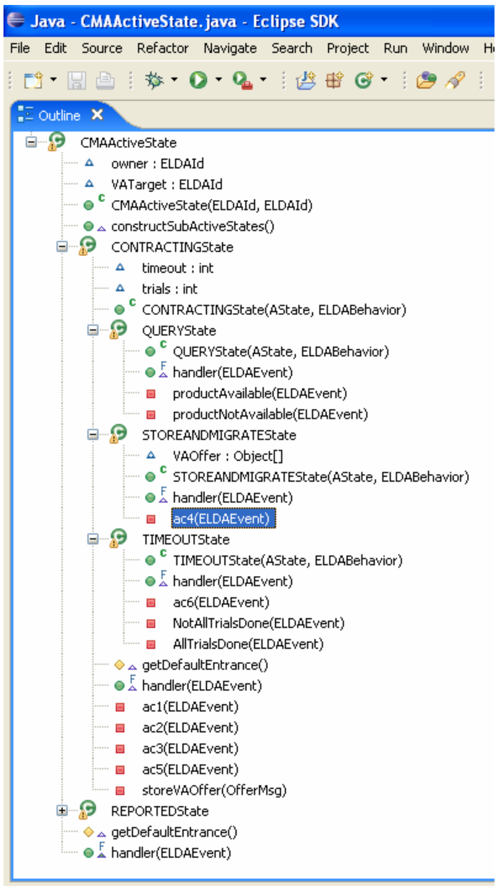
After defining the agent behavior, it is possible to generate Java code through the CodeGenerator component. The code generation activity creates a new project containing the translation of specified models into Java code according to the ELDAFramework.

Figure 11 shows both the project containing the models of the whole MAS under-development (EMarketPlace) and the project structure (EMarketPlace_Implementation) generated only for the CMA which contains a package (emarketplace.cma) with the CMAActiveState class and a package (emarketplace.events) with event classes triggering the CMA (Contract, OfferMsg, ReportParent, Tick).


Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Navigator
test_Implementation

```
EMarketPlace
  JRE System Library [jre1.5.0_06]
  CMA
    CMA_actions.action
    CMA_guards.guard
    CMA.dsc
    CMAAgent_functions.func
  OA
  VA
  EMarketPlace_events.event
EMarketPlace_Implementation
  src
    emarketplace.cma
      CMAActiveState.java
    emarketplace.events
      Contract.java
      OfferMsg.java
      ReportParent.java
      Tick.java
  JRE System Library [jre1.5.0_06]
  ELDAFramework.jar - ECLIPSE_HOME\plugins
```

Figure 11: Structure of the generated project


Java - CMAActiveState.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window H
Outline

```
CMAActiveState
  owner : ELDAId
  VATarget : ELDAId
  CMAActiveState(ELDAId, ELDAId)
  constructSubActiveStates()
  CONTRACTINGState
    timeout : int
    trials : int
    CONTRACTINGState(AState, ELDABehavior)
    QUERYState
      QUERYState(AState, ELDABehavior)
      handler(ELDAEvent)
      productAvailable(ELDAEvent)
      productNotAvailable(ELDAEvent)
    STOREANDMIGRATEState
      VAOffer : Object[]
      STOREANDMIGRATEState(AState, ELDABehavior)
      handler(ELDAEvent)
      ac4(ELDAEvent)
    TIMEOUTState
      TIMEOUTState(AState, ELDABehavior)
      handler(ELDAEvent)
      ac6(ELDAEvent)
      NotAllTrialsDone(ELDAEvent)
      AllTrialsDone(ELDAEvent)
    getDefaultEntrance()
    handler(ELDAEvent)
    ac1(ELDAEvent)
    ac2(ELDAEvent)
    ac3(ELDAEvent)
    ac5(ELDAEvent)
    storeVAOffer(OfferMsg)
  REPORTEDState
  getDefaultEntrance()
  handler(ELDAEvent)
```

Figure 12: Active state of the CMA agent

Figure 12 shows the outline of the CMAActiveState which highlights the hierarchical dataspace of the agent and the location of guards, actions, and functions.

Figure 13 shows an excerpt of the CMA generated code related to the Timeout state.

```
// TIMEOUTState Inner Class
public class TIMEOUTState extends SimpleState implements Serializable
{

 public TIMEOUTState (AState parent, ELDABehavior ebeh) {
  super(parent,ebeh);
 }

 public final int handler(ELDAEvent evt){
 if (evt instanceof Tick){
   ac5(evt);
   return 0;
 }
 else if (evt instanceof Contract && NotAllTrialsDone(evt)){
  ac6(evt);
  ((CompositeState) parent).setActiveState(
    ((CompositeState) parent).getState("QUERY"));
  changeState(((CompositeState) parent).getActiveState());
  return 0;
 }
 else if (evt instanceof Contract && AllTrialsDone(evt)){
  ac3(evt);
  ((CompositeState) parent).setActiveState(
     ((CompositeState) parent).getState("STOREANDMIGRATE"));
  changeState(((CompositeState) parent).getActiveState());
  return 0;
 }
 else return parent.handler(evt);
 }
 // Actions Definitions Section
 private void ac6 (ELDAEvent e){
  trials--;
  ac2(e);
 }
 // Guards Definitions Section
 private boolean NotAllTrialsDone (ELDAEvent e){
  return ! AllTrialsDone(e);
 }
 private boolean AllTrialsDone (ELDAEvent e) {
  if(trials==0)
   return true;
  return false;
 }
}
```
Figure 13: The code of the Timeout state.

VI. CONCLUSIONS AND FUTURE WORKS

This paper has presented the ELDATool by describing its system requirements, design, implementation, and use through a simple example. The ELDATool represents a research effort aiming at supporting the rapid prototyping of MASs which is contextualized in the active research area on agent-oriented software engineering. In particular, the ELDATool gives support to a DSC-based agent-oriented methodology seamlessly covering the phases of the MAS development lifecycle from modeling to implementation.

Visual modeling and programming, and automatic code generation are very important features that any tool supporting an agent-oriented methodology should have to ease the designer tasks. The ELDATool fully provides such features and, furthermore, being based on the Eclipse platform, can be easily distributed and used by the community.

Currently, efforts are underway for (i) completing the Simulation component so allowing validation and performance evaluation of the MAS under-development; (ii) designing and implementing new components for the implementation and deployment of the prototyped MAS for a target agent platform.

REFERENCES

[1] F. Zambonelli and A. Omicini, “Challenges and research directions in agent-oriented software engineering”, *Autonomous Agents and Multi-Agent Systems*, 9(3), pp. 253-283, Nov. 2004.

[2] M. Cossentino, “From Requirements to Code with the PASSI Methodology,” In *Agent-Oriented Methodologies*, Eds. B. Henderson-Sellers and P. Giorgini, Idea Group Inc., Hershey, PA, USA, 2005, pp. 79–106.

[3] F. Zambonelli, N. Jennings, and M. Wooldridge, "Developing multiagent systems: The Gaia methodology," *ACM Trans. Software Eng. Meth.*, vol. 12, no. 3, pp.417-470, 2003.

[4] A. Molesini, A. Omicini, E. Denti, and A. Ricci, “SODA: A Roadmap to Artefacts,” *6th International Workshop on Engineering Societies in the Agents World VI*, (ESAW 2005), Kusadasi, Aydin, Turkey, October 2005. *LNAI 3963*, Springer, 2006.

[5] J. Pavón, J. Gómez-Sanz, and R. Fuentes, “The INGENIAS Methodology and Tools,” In *Agent-Oriented Methodologies,* Eds. B. Henderson-Sellers and P. Giorgini, Idea Group Publishing, 2005, pp. 236-276.

[6] G. Fortino, W. Russo, and E. Zimeo, “A Statecharts-based Software Development Process for Mobile Agents”, *In Information and Software Technology*, 46(13), pp.907-921, Elsevier, Amsterdam, The Netherland, 2004.

[7] G. Fortino, A. Garro, and W. Russo, “An Integrated Approach for the Development and Validation of Multi Agent Systems”, *In Computer Systems Science & Engineering*, 20(4), pp. 94-107, CRL Publishing Ltd., Leicester (UK), Jul. 2005a.

[8] R. Caico, M. Cossentino, G. Fortino, A. Garro, W. Russo, and F. Termine, "Simulation-driven Development of Multi-Agent Systems", *Proceedings of the EUROSIS Workshop on Multi-Agent Systems and Simulation (MAS&S’06)*, Palermo, Italy, 2006, pp. 17-24.

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

[10] D. Harel and E. Gery, “Executable Object Modelling with Statecharts”, *IEEE Computer*, 30(7), pp. 31-42, 1997.

[11] FIPA (Foundation for Intelligent Physical Agents). 2002. FIPA Agent Management Support for Mobility Specification, Document FIPA DC00087C (2002/05/10).

[12] Eclipse - an open development platform, documentation and software, available at the World Wide Web: http://www.eclipse.org.

[13] G. Fortino, A. Garro, and Russo, W. (2005b) ‘A Discrete-Event Simulation Framework for the Validation of Agent-based and Multi-Agent Systems’, *Proceedings of the Workshop on Objects and Agents (WOA’05)*, Camerino, Italy, Nov 14-16.

[14] The Graphical Editing Framework (GEF), documentation and software, available at the World Wide Web: http://www.eclipse.org/gef/.

[15] Eclipse Modeling Framework Project (EMF), documentation and software, available at the World Wide Web: http://www.eclipse.org/modeling/emf/.