Engineering Multi-Agent Systems through Statecharts-Based JADE Agents and Tools

Giancarlo Fortino, Francesco Rango, and Wilma Russo

Dept. of Electronics, Informatics and Systems (DEIS) – University of Calabria,
Via P. Bucci, cubo 41C, 87036 Rende (CS), Italy
{g.fortino,w.russo}@unical.it, frango@si.deis.unical.it

Abstract. The JADE framework, which is one of the most used in the AOSE community to program and execute multi-agent systems (MASs), still needs to be further supported by methods and tools for enabling a more effective modeling and prototyping of JADE-based MASs. In this paper we propose a framework and a related tool supporting a Statecharts-based development of JADE-based MAS with the purpose of providing an effective approach for engineering multi-agent systems and leveraging agent-oriented development methodologies and processes adopting JADE as target agent platform. In particular, a framework for programming JADE behaviors through a variant of the Statecharts, named Distilled StateCharts (DSCs), has been first developed by enhancing the JADE add-on HSMBehaviour. Then, to enable rapid prototyping of JADE agents, a visual tool for DSCs has been extended with translation rules based on the developed framework that allows to automatically translate DSC specifications into DSC-based JADE behaviors. The proposed approach is exemplified through a case study concerning an agent-based meeting organization system.

Keywords: Statecharts, Software agents, JADE, Visual programming, Automatic code generation, CASE tool.

1 Introduction

In the last decade the agent oriented software engineering (AOSE) research area has produced a rich set of methodologies and tools that can be actually exploited for the development of complex software systems in terms of multi-agent systems (MASs) [1]. In parallel with AOSE, the mainstream software engineering area has driven UML 2.0 [2] along with related methodologies and tools to become the *de facto* standard for the development of software systems. In particular, the UML state machines, derived from the Harel's Statecharts [3], are an effective and widely adopted formalism for the specification of active component behaviors and protocols in general-purpose and real-time systems. It is widely recognized that the benefits provided by Statecharts for engineering complex software systems are mainly visual programming, executable specifications, protocol-oriented specifications, and a set of CASE tools facilitating software development. In this context, to effectively develop

N.T. Nguyen (Ed.): Transactions on CCI VII, LNCS 7270, pp. 61–81, 2012.

multi-agent systems (MAS), models, frameworks and tools are needed to support flexible and rigorous specifications and subsequent implementations of agent behaviors and agent-to-agent interaction protocols [4]. Thus the use of Statecharts-based models, frameworks and tools for the development of MASs could provide the same benefits in the AOSE research area as those provided in the context of traditional software engineering. However, in the AOSE research area, Statecharts are still under-used to specify agent behaviors and protocols even though some proposed agent models founded on different types of state machines are available [5, 6, 7, 8, 9, 10, 12].

In this paper we propose programming frameworks and techniques supporting a Statecharts-based development of JADE-based MASs. The main contribution of this paper is twofold: (i) the integration of Statecharts and MASs to deliver the same important benefits provided by Statecharts for the engineering of traditional software systems; (ii) the definition of a Statecharts-driven development method for the JADE platform which is one of the most used agent platform in the agent community. Moreover, the proposed approach can be fruitfully exploited to leverage already existing agent-oriented development methodologies and processes adopting JADE as target agent platform (e.g. INGENIAS [16], PASSI [17], MESSAGE [18]). In particular, a framework for programming JADE behaviors through the Distilled StateCharts (DSCs) formalism, named DistilledStateChartBehaviour, has been developed by enhancing the JADE HSMBehaviour. To enable rapid prototyping of JADE agents, a CASE tool obtained by enhancing the ELDATool with a new component based on the DistilledStateChartBehaviour for automatic code generation of DSC-based behaviors into JADE code, is made available. The proposed approach is exemplified through a case study regarding an agent-based meeting organization system.

The rest of this paper is organized as follows. Section 2 discusses and compares related work. In section 3, after an introduction of the basic concepts of the Distilled StateCharts formalism, the JADE DistilledStateChartBehaviour is described. In section 4 a CASE tool-driven approach for engineering JADE-based MAS from modeling to implementation, is presented. Section 5 details a case study exemplifying the proposed Statecharts-based approach and provides an experimental evaluation of the scalability of the developed MAS. Finally, conclusions are drawn and on-going work delineated.

2 Related Work

To date several proposals are available which provide frameworks based on state machines to design and implement agent behaviors and interactions. Among such proposals, the most known and interesting ones are the JADE FSMBehaviour [5], the SmartAgent framework [6], the ELDA agent model [7], and the Bond agent framework [8]. In particular, the JADE framework [5], one of the most used agent-oriented framework in academy and industry, provides the FSMBehaviour [9] for the modeling of agent behaviors based on finite state machines (FSMs). However agent behavior programming is not flexible as it does not rely on ECA (Event-Condition-Action)-rule

based transitions, and does not provide important mechanisms for reducing behavior complexity such as well-structured OR-decomposition and history entrances. In particular, although states of the FSMBehaviour can be FSMBehaviours or other behaviors, mechanisms for handling this induced state hierarchy are not provided. The SmartAgent model [10, 6] extends the JADE CompositeBehaviour and provides a behavior based on hierarchical state machines driven by ECA rules, named HSMBehaviour. However, the HSMBehaviour does not even support shallow and deep history entrance mechanisms, useful for reducing behavior complexity even further and for transparently archiving agent states. In addition, although visual modeling and emulation of HSMBehaviour agents can be done with the provided HSMEditor [11], automatic translation of modeled agents into JADE code is not supported. The ELDA (Event-driven Lightweight Distilled Statecharts-based Agents) agent model [7] is based on a Statecharts-like machine, providing or-decomposition and history entrance mechanisms, named Distilled StateCharts [12] suitable for the modeling of lightweight agents for distributed computing. Moreover, they can be effectively modeled through the ELDATool, a graphical tool for visual specification, automatic code translation and simulation of ELDA-based systems [13]. However, an ELDA-based execution platform is not yet available so confining the use of ELDA agents in the MAS simulation domain. The behavior of the Bond agents [8] is based on a multi-plane state machine where each plane is modeled as an FSM. However, the Bond agent model does not offer the state hierarchy, history mechanisms, and tools for automating agent prototyping. Finally other previous agent frameworks are ZEUS [14], which provides an execution subsystem for non-hierarchical state machine-based agents, and the JACKAL conversation engine that also uses a state machine model [15]. In Table 1 a comparison in terms of behavioral, interaction and mobility models among the aforementioned frameworks is provided. In particular, the differences about behavioral models are those discussed above whereas, with respect to the interaction models, they are mainly based on messages apart from Bond and ELDA which rely on multiple coordination models (not only messages but also tuple spaces and publish/subscribe); moreover, the mobility model is of the weak type apart from ELDA which allows for coarse-grain strong mobility [7].

Table 1. Comparison among state machine oriented frameworks

FRAMEWORKS/*MODELS*	*BEHAVIOURAL*	*INTERACTION*	*MOBILITY*
Jade	flat finite state machines (FSMBehaviour)	Message passing	Weak
SmartAgent	hierarchical finite state machines (HSMBehaviour)	Message passing	Weak
Bond	multi-plane state machine: each plane is an FSM	Message passing, TSpace and P/S	Weak
Actors	active objects with state variables and action methods	Message passing	Weak
ZEUS	flat finite state machines	Message passing	Weak
JACKAL	flat finite state machines	Message passing	Weak
ELDA	Distilled StateCharts	Multi-coordination	Coarse-grain Strong

3 Statecharts-Based JADE Agents

In this section, the DSC formalism, which provides a powerful and rich set of modeling concepts enabling an effective specification of agent behavior, is overviewed. Then, the proposed framework for programming DSC-based JADE agents, which enhances JADE with the benefits deriving from Statecharts, is described.

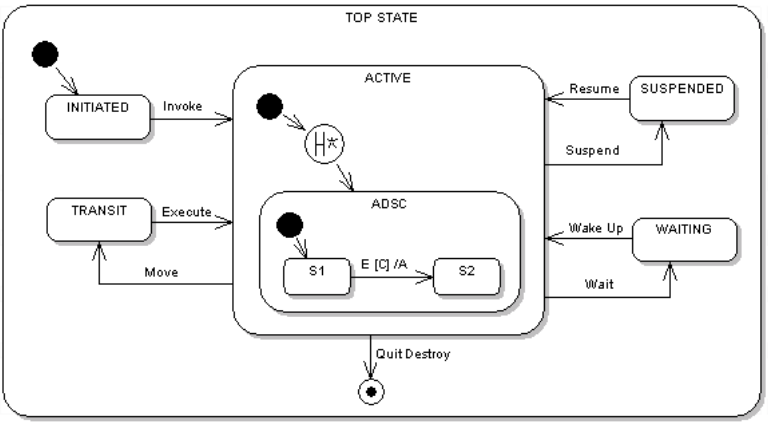

Fig. 1. A FIPA compliant DSC-based agent behavior

3.1 The Distilled StateCharts Model

The Distilled StateCharts (DSCs) formalism [12] is derived from the Harel's Statecharts through a distillation process, purposely carried out for the modeling of lightweight mobile agent behavior, which led to the following structural/semantics differences between Statecharts and DSCs:

- State entry and exit actions as well as activities are empty so actions can be only hooked to transitions;
- Each composite state has a pseudo initial state from which the default entrance of the composite state originates;
- Transitions (apart from default entrances and default history entrances) are always labeled by an event;
- Default entrance and default history entrances can only be labeled with an action;
- And-decomposition of states and related synchronization modeling constructs are not used as DSCs were introduced for supporting the behavioral modeling of single-threaded agents;
- Run-to-completion step semantics, defined according to the UML state machines semantics [19], are adopted.

A DSC-based agent behavior relies on an enhanced basic template built according to the FIPA agent lifecycle [20] which JADE agents are compliant with (see Figure 1). In particular, the ACTIVE state, in which an agent carries out its goal-oriented tasks, is always entered through a deep history entrance (H*) whose default history entrance

targets the active DSC (ADSC) state, which actually models the active agent behavior. The default entrance of ACTIVE targeting H* allows restoring the agent execution state after agent migration and, in general, after agent suspension.

3.2 A Framework for Programming DSC-Based JADE Agents

A new JADE behavior, named DistilledStateChartBehaviour, has been defined to program JADE agents through the DSC formalism. In particular, the DistilledStateChartBehaviour, which is defined by enhancing the HSMBehaviour [10, 6] with the DSC mechanisms, specifically implements the history mechanisms that allow a partial (through shallow history H) or full (through deep history H*) recovery of the state history when re-entering into any state previously exited.

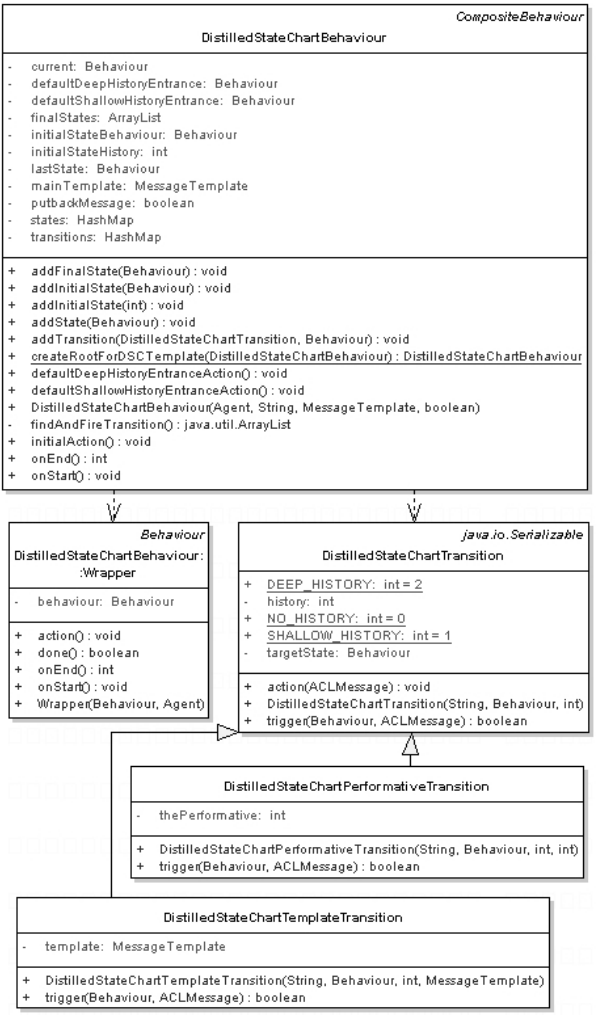

Fig. 2. Simplified class diagram of the JADE DistilledStateChartBehaviour

Figure 2 shows a simplified UML class diagram of the DistilledStateChartBehaviour. In particular, the DistilledStateChartBehaviour inherits from the JADE CompositeBehaviour and includes both a set of nested DistilledStateChartBehaviours and other Behaviours, which represent the *states* of the DSC. It maintains the list of *transitions*, represented by the DistilledStateChartTransition class, and handles the event-driven mechanism for transition firing which also determines the *current* state of the DSC state machine at run-time. As it is shown in Figure 3, an event E, instance of the ACLMessage class, is fetched from the JADE event queue by the dispatcher component of the DistilledStateChartBehaviour and delivered to the DSC current state (S1) so triggering a state transition to a new state (S2) if the guard C holds.

In the following a detailed description of the main mechanisms (state management, behavior scheduling, event handling, transition firing and history entrances) of the DistilledStateChartBehaviour is presented.

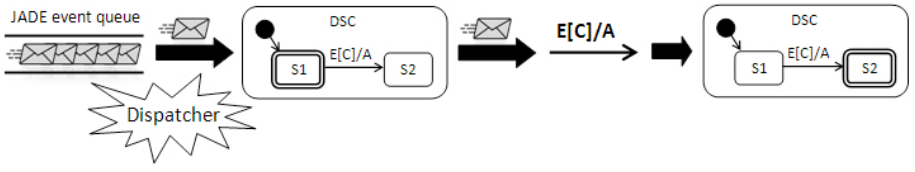

Fig. 3. The event handling scheme

State Management. Any type of JADE behavior can be added as simple, initial or final state to the DistilledStatechartsBehaviour through the methods *addState*, *addInitialState* and *addFinalState*, respectively. The *createRootForDSCTemplate* method automatically builds the root of the DistilledStatechartsBehaviour that allows entering into the active state through the deep history entrance (see Figure 1). The *initialAction* method allows inserting an initial action on the default entrance. The methods *onEnd*, *onStart* and *action* should be kept empty.

Behaviour Scheduling. The DistilledStateChartBehaviour receives the thread of control from the JADE run-time system through the invocation of the *action* method according to the cooperative concurrency mechanism of JADE. The *action* method of the DistilledStateChartBehaviour, in turn, invokes the *action* method of the current state; the DistilledStateChartBehaviour starts executing the initial state, activates other states by following the fired transitions and, finally, terminates when enters into one of its final states. On the invocation of the *action* method of the current behavior, the Wrapper object, which encapsulates each simple state, allows checking all transitions outcoming from the current state and executing the fireable transitions (through the *findAndFireTransition* method). This mechanism allows implementing the UML state machine rule: “as soon as a transition is able to fire, it does”. Indeed, the actual implementation is based on the single-threaded model of JADE, which does not support preemption of an action execution.

Event Handling. An important feature of the DSC state machines is the event driven mechanism for triggering transitions. An event can be represented as a regular JADE ACLMessage so enabling the reuse of the message queuing mechanism of JADE (see Figure 3): when the DistilledStateChartBehaviour is checking for a transition firing, the *receive* method of JADE is invoked to fetch the first message in the queue, which is then passed to the transitions to check if one of them can be fired. The main issue of such mechanism is the integration of behaviors as states. In particular, as an event message in queue is fetched through the *receive* method, if this method is invoked inside the *action* method, it can interfere with the transition firing mechanism. Moreover, if a message/event is received in a state in which the event is not expected, the two following options, which can be set in the DistilledStateChartBehaviour constructor are possible: the event is re-inserted into the queue (*putbackMessage*=true) so that it could be fetched by another state that is able to handle it, or it is discharged (*putbackMessage*=false). The same event handling mechanism can be also used when an agent has multiple behaviors for the purpose of avoiding important event losses. In this case, the message template mechanism based on selective filters for events can be used. In particular, each behavior performs a *receive* operation with a different message template so as to fetch only the events it is able to handle.

Transition Firing. A transition is represented by the DistilledStateChartTransition class and is added through the *addTransition* method which takes as parameters the transition to be added and the source state. The target state is defined at DistilledStateChartTransition creation and can be at any level of the hierarchy so supporting the specification of inter-level state transitions. The DistilledStateChartTransition unifies the mechanisms of trigger event and guard into the *trigger(Behaviour source, ACLMessage event)* method, where *source* is the transition source state and *event* is the transition triggering event. The *trigger* method checks for the transition firing and, if the check is successful, the *action* method of DistilledStateChartTransition, which can contain the action hooked to the transition, is invoked. The check based on both the *trigger* and *findAndFireTransition* methods not only involves the current state but also all the states, from the inner to the outer, encapsulating it. The DistilledStateChartPerformativeTransition and DistilledStateChartTemplateTransition classes extend DistilledStateChartTransition providing a new version of the *trigger* method that allows to check respectively if the received event respects a specific performative or MessageTemplate.

History Entrances. The DistilledStateChartBehaviour includes the *defaultDeepHistoryEntrance* and the *defaultShallowHistoryEntrance* referring to the states (or behaviors) associated to the deep and shallow history entrances, respectively. To restore the state history, the *lastState* variable of a composite state of the DistilledStateChartBehaviour type, which stores a reference to the last visited state before exiting the composite state, is used. Moreover, the DistilledStateChartTransition includes the two constants *DEEP_HISTORY* and *SHALLOW_HISTORY* that indicate that the target composite state is to be entered through the deep or shallow history.

4 CASE Tool-Driven Development of DSC-Based JADE Agents

The development of DSC-based JADE agents relies on the process reported in Figure 4 which is organized in the following three phases:

- The *Modeling* phase produces the DSC-based MAS Model on the basis of the High-Level System Design which can be defined either ad-hoc or by means of other methodologies which also support the analysis and high-level design phases [17, 18, 16]. In particular, the DSC-based MAS Model is specified through the DSC formalism and the JADE API.
- The *Coding* phase works out the DSC-based MAS Model and automatically produces the JADE MAS code according to the DistilledStateChartBehaviour.
- The *Deployment and Execution* phase is fully supported by the JADE Platform to run the developed MAS. A careful evaluation of the obtained Testing Results (e.g. execution traces, performance indices, etc) with respect to the functional and non-functional requirements could lead to a further iteration step which starts from a new (re)modeling activity.

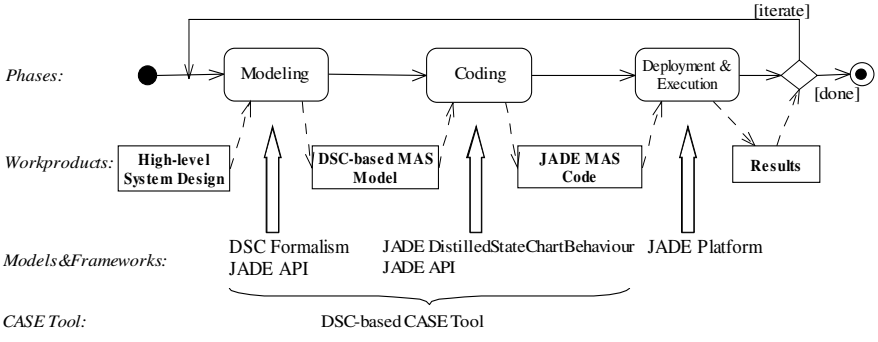

Fig. 4. The CASE-driven development process

The first two phases are fully supported by the DSC-based CASE tool that makes it available (i) the visual modeling of the DSC-based behavior of the agents composing the MAS under-development and (ii) the automatic translation of the modeled agent behaviors into ready-to-be-executed JADE code according to the DistilledStatechartsBehaviour framework.

The CASE tool is obtained by enhancing the ELDATool [7], a graphical tool for visual specification, automatic code translation and simulation of ELDA-based systems, with a new component named CodeGeneratorForJADE embedded into the ELDAEditor plug-in. This important facility, which is not offered by the HSMBehaviour graphical tools [11], makes the programming of Statecharts-based JADE agents easier than manual programming of the HSMBehaviour and DistilledStateChartBehaviour based on complex programming patterns.

As the ELDATool is based on the ELDA agent model [7], the specific event types, exploitable for the modeling phase, are: (i) the ELDAEventMSG, which represents

asynchronous messages; (ii) the ELDAEventInternal, which represents self-triggering events. Both kinds of events derive from the ELDAEvent class and are inserted into an ACLMessage as message content. Moreover it is worth noting that the specification of state variables, actions, guards, events and functions is based on the Java language and the JADE API.

5 A Case Study: An Agent-Based Meeting Organization System

In this section the DSC-based development of an agent-based meeting organization system, in which agents coordinate to arrange meetings, is proposed. The developed MAS is derived from a case study based on a meeting participant protocol proposed in [21, 11]. In particular, the MAS is based on three types of agents (see Figure 5): (i) MeetingRequester (MRA), which is the meeting organizer; (ii) MeetingBroker (MBA), which arranges meetings on the basis of the MRA requests; (iii) MeetingParticipant (MPA), which represents a meeting participant.

Fig. 5. Class diagram of multi-agent meeting system

In the following subsections we first describe the agent interactions for the meeting arrangement and detail the agent behaviors and, then, provide some implementation details of the agent-based system along with an experimental performance evaluation aiming at analyzing the MAS scalability.

5.1 Agent Interactions

The defined agents interact with each other to fulfill a meeting arrangement that can be constituted by one or more iterations (i.e. an iteration is an attempt to arrange a given meeting driven by the MRA requests). The interaction protocol is defined through the sequence diagrams reported in Figures 6-8 that show successful and unsuccessful cases. Figure 6 shows the 1-iteration successful interaction scenario in which a meeting is arranged with two participants (even though it can be generalized to n-participants). In particular, after the Request sent by the MRA to the MBA, the successful event flow is: the Propose event is sent by the MBA to the two MPAs that, in turns, accept it and send the AcceptProposal event to the MBA that finalizes the meeting and sends out the Confirm event to the accepting MPAs and MRA.

In Figure 7, the 2-iteration successful interaction scenario, in which a meeting is arranged with three participants, is reported. Differently from the previous interaction scenario, here the MPA_1 refuses the proposal by sending the RejectProposal event to the MBA that, in turn, send the AskForRequest event to the MRA to have information about new potential participants. After receiving such information the MBA therefore sends out a Propose event to MPA_3 that accepts it.

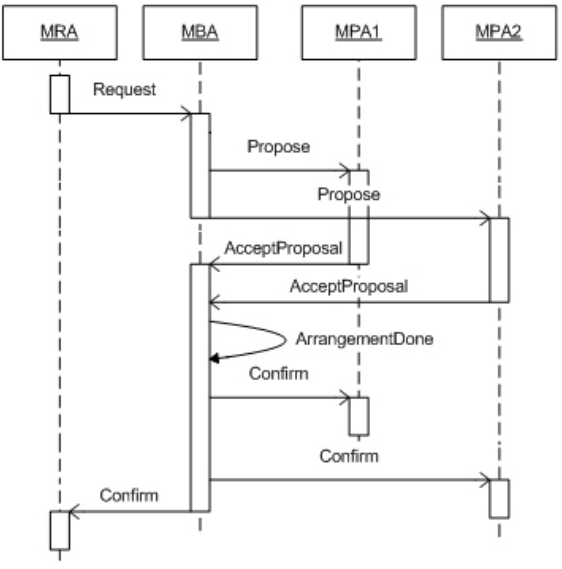

Fig. 6. Sequence diagram of agent interactions: successful case after 1-iteration

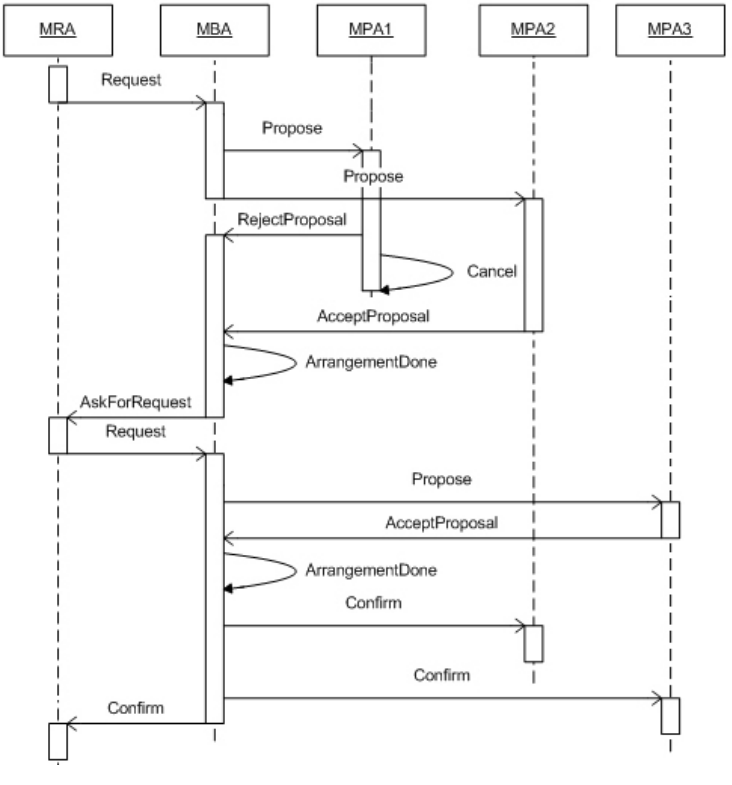

Fig. 7. Sequence diagram of agent interactions: successful case after 2-iterations

Finally, in Figure 8, the unsuccessful interaction scenario, in which a meeting is being arranged with three participants, is shown. MPA_2, MPA_3 and MPA_4 refuse the proposal so that after three additional requests (the maximum fixed number of attempts) the arrangement of the meeting fails.

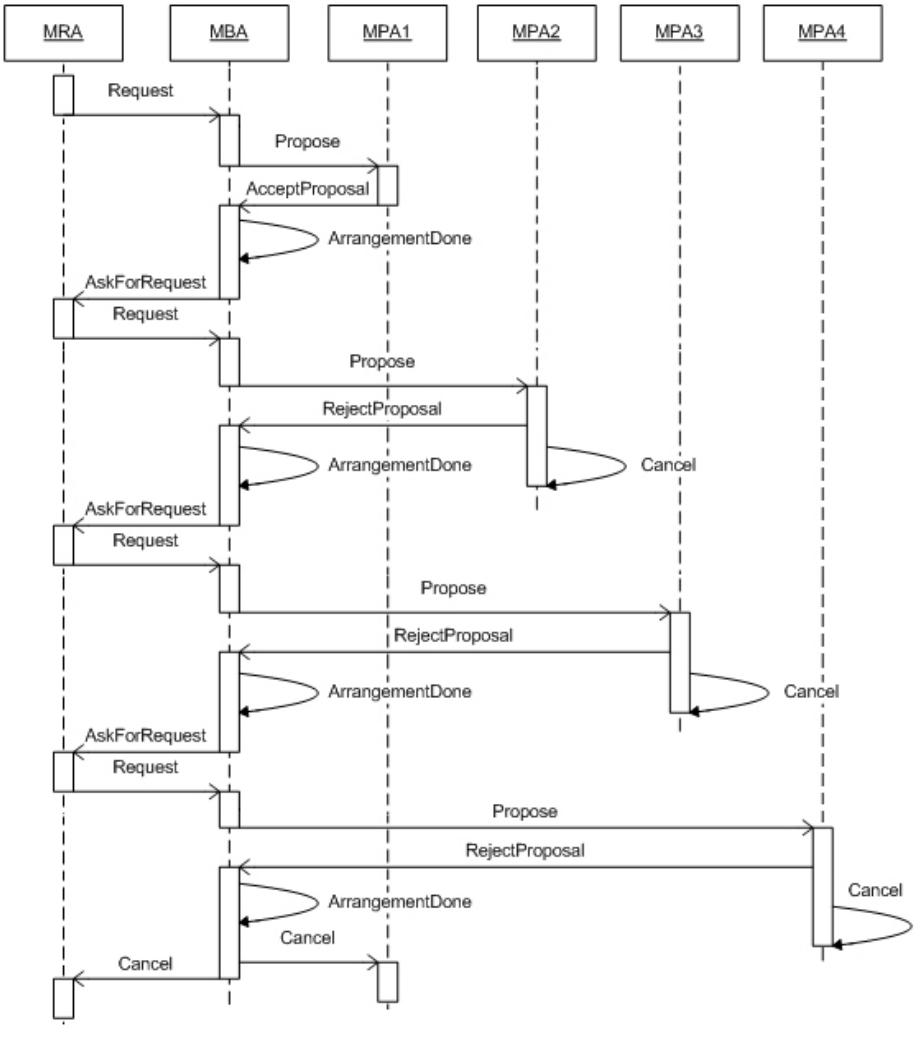

Fig. 8. Sequence diagram of agent interactions: unsuccessful case

5.2 Agent Behaviors

While the behaviors of the MRA (see Figure 9) and MPA (see Figure 10) are straightforward, more complexity is retained by the MBA behavior (see Figure 11). In particular, each behavior is described in terms of a DSC diagram, state variables and actions. Moreover, Table 2 summarizes the event-based interaction relationships

Table 2. Event-based interaction relationships among agents

SOURCE \ TARGET	MRA	MBA	MPA
MRA		Request	
MBA	Confirm, Cancel, AskForRequest	ArrangementDone	Propose, Confirm, Cancel
MPA		AcceptProposal, RejectProposal	

among agents, specifying the event source agent, which generates the event, and the event target agent, which receives and handles the event.

According to the MRA behavior (see Figure 9 and Tables 3-4), the MRA sends a Request event to the MBA (see action `sendRequest`) containing all needed information (potential participants, minimum number of participants, meeting topic, and chosen date) related to the appointment to arrange and waits for the meeting confirmation. As soon as the MRA receives the AskForRequest event, it will send out a new or modified Request (see action `sendRequest`). The reception of the Confirm event signals an arranged meeting (action `meetingDone`) whereas the Cancel event signals a failure in organizing a meeting (action `meetingCanceled`).

According to the MPA behavior (see Figure 10 and Tables 5-6), in the Started state, the MPA can receive the Propose even to check an appointment (see action `checkAppointment`) or to refuse it. As soon as it receives the Confirm event, the MPA finalizes the appointment set-up (see action `fixAppointment`).

As described above, the MBA manages the meeting arrangement requests sent by the MRA, and coordinates the MPAs. The MBA behavior (see Figure 11 and Tables 7-8) starts in the Negotiation composite state and acts as follows: upon the reception of the Request event, the MBA sends all the MPAs a Propose event containing the appointment to schedule (action `sendPropose`), starts a timer (action `initializeTimer`) and finally goes into the Arrange composite state. The MPAs send the MBA an AcceptProposal event to accept the appointment or a RejectProposal event to refuse it (see Figure 10). On the basis of the received responses, the MBA accepts (action `acceptParticipant`) or excludes (action `excludeParticipant`) the participants and, when it receives all the responses or when the timeout associated to the set timer expires (action `sendArrangementDone`), sends an ArrangementDone event to itself to carry out the final operations (see action `completeArrangement`) for the current appointment as follows:

- If at least M MPAs have accepted the appointment, the meeting organization is successfully done; then, the MBA sends a Confirm event to the MRA and to the accepting MPAs, which schedule the appointment in their rosters (see Figure 10).
- If the appointment has been accepted by less than M MPA and it is not yet reached the maximum limit of N requests of new participants sent to the MRA, the MBA issues a request of new participants to the MRA by sending it an AskForRequest event. Then, the MRA sends a new Request event to the MBA indicating new participants for the same appointment (see Figure 9). This way, the MBA can retry to schedule the appointment involving the new provided participants.

- If the appointment has been accepted by less than M MPA and it is reached the maximum limit of N requests of new participants sent to the MRA, the appointment is canceled and a Cancel event is sent to the accepting MPAs and MRA.

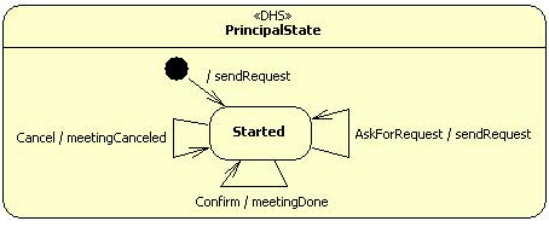

Fig. 9. The state diagram of the DSC-based behavior of the MRA

Table 3. Variables of the DSC-based behavior of the MRA

STATE	VARIABLES
ROOT	String meetingBroker
PrincipalState	Appointment currentAppointment

Table 4. Actions and functions of the DSC-based behavior of the MRA

ACTIONS

sendRequest

```
if (currentAppointment == null) {
 String description = getDescription();
 Calendar date = getDate();
 int n = getNumberOfParticipants();
 java.util.ArrayList<AID> participantsList = new java.util.ArrayList<AID>();
 for(int i = 1; i <= n; i++){
  String nickname = getNickname(i);
  participantsList.add(new AID(nickname, AID.ISLOCALNAME));
 }
 currentAppointment = new Appointment(participantsList, date, description);
 java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();
 target.add(new AID(meetingBroker, AID.ISLOCALNAME));
 Request msg = new Request(self(), target, currentAppointment);
 generate(msg);
}
else {
 int n = getNumberOfParticipants();
 java.util.ArrayList<AID> participantsList = new java.util.ArrayList<AID>();
 for(int i = 1; i <= n; i++){
  String nickname = getNickname(i);
  participantsList.add(new AID(nickname, AID.ISLOCALNAME));
 }
 currentAppointment = new Appointment(participantsList,
  currentAppointment.getDate(), currentAppointment.getDescription());
 java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();
 target.add(new AID(meetingBroker, AID.ISLOCALNAME));
 Request msg = new Request(self(), target, currentAppointment);
 generate(msg);
}
```

meetingDone (omissis)

meetingCancelled (omissis)

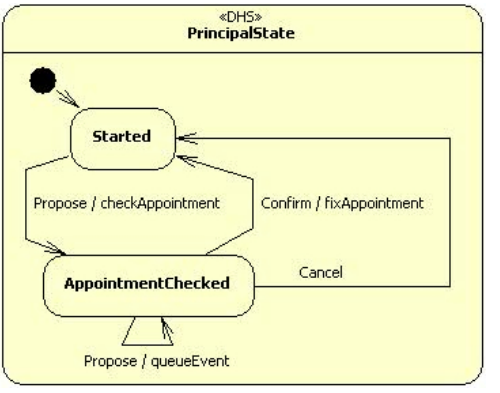

Fig. 10. The state diagram of the DSC-based behavior of the MPA

Table 5. Variables of the DSC-based behavior of the MPA

STATE	VARIABLES
PrincipalState	java.util.Hashtable myCalendar = new java.util.Hashtable() Appointment currentAppointment

Table 6. Actions and functions of the DSC-based behavior of the MPA

ACTIONS

checkAppointment

```
Propose p = (Propose) e;
currentAppointment = (Appointment) p.getData();
AID meetingBroker = p.getSource();
if(myCalendar.containsKey(getKey(currentAppointment.getDate()))){
 java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();
 target.add(meetingBroker);
 RejectProposal msg = new RejectProposal(self(), target, null);
 generate(msg);
 java.util.ArrayList<AID> target2 = new java.util.ArrayList<AID>();
 target2.add(self());
 Cancel msg2 = new Cancel(self(), target2, null);
 generate(msg2);
} else{
 java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();
 target.add(meetingBroker);
 AcceptProposal msg = new AcceptProposal(self(), target, null);
 generate(msg);
}
```

fixAppointment

```
myCalendar.put(getKey(currentAppointment.getDate()), currentAppointment);
```

queueEvent

```
java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();
target.add(self());
e.setTarget(target);
generate(e);
```

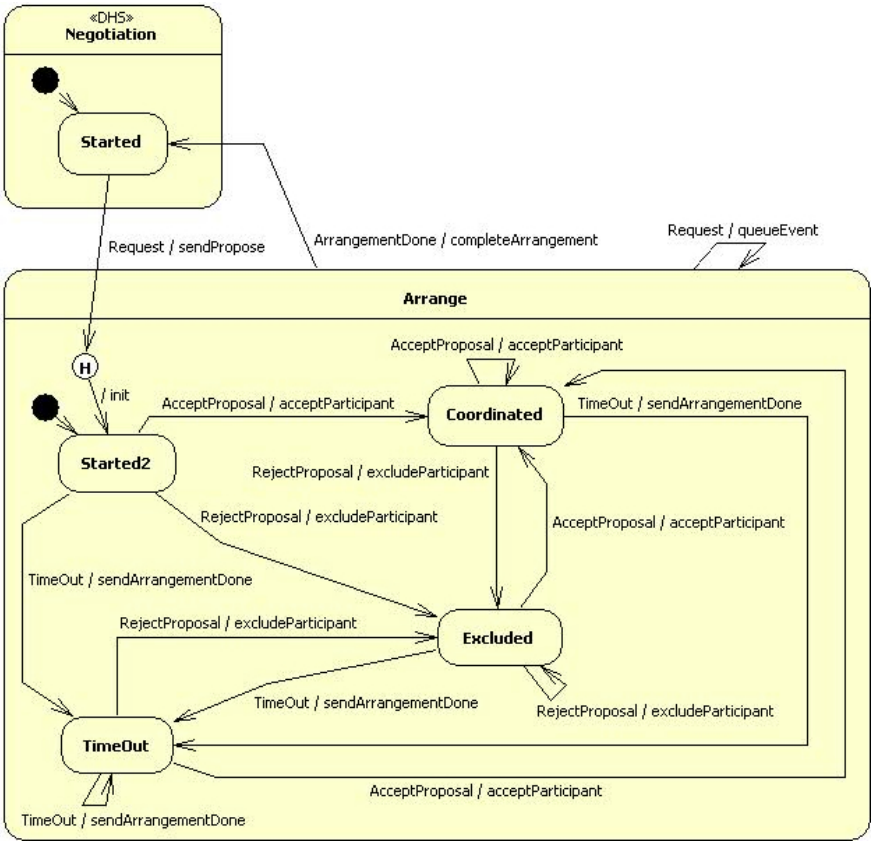

Fig. 11. The state diagram of the DSC-based behavior of the MBA

Table 7. Variables of the DSC-based behavior of the MBA

STATE	VARIABLES

ROOT	`int contResponses`
	`int contRequestsToMeetingRequester`
	`WakerBehaviour timer`
	`AID meetingRequester`
	`int M, N`

Arrange	`ArrayList<AID> acceptedParticipants`

After the completion of the `completeArrangement` action, the MBA goes back into the Negotiation composite state. The shallow history entrance (H) provides a powerful modeling solution when the Arrange composite state is to be re-entered due to a new Request related to the same appointment. In particular, when a new Request event is received, the MBA goes into the most recently left simple state of the

Table 8. Actions and functions of the DSC-based behavior of the MBA

ACTIONS
sendPropose

`Request r = (Request) e; Appointment app = (Appointment) r.getData();`
`contResponses = app.getParticipantsList().size();meetingRequester = r.getSource();`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`for(int i=0; i < app.getParticipantsList().size(); i++)`
`target.add(app.getParticipantsList().get(i));`
`Propose msg = new Propose(self(), target, app); generate(msg);`
`initializeTimer(e);`

initializeTimer

`timer = new WakerBehaviour(myAgent, 30000){`
`protected void onWake() {`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.add(self());`
`TimeOut msg = new TimeOut(self(), target, null); generate(msg);`
`}}; myAgent.addBehaviour(timer);`

acceptParticipant

`acceptedParticipants.add(e.getSource()); contResponses--;`
`if(contResponses == 0){`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.add(self());`
`ArrangementDone msg = new ArrangementDone(self(), target, null); generate(msg);}`

excludeParticipant

`contResponses--;`
`if(contResponses == 0){`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.add(self());`
`ArrangementDone msg = new ArrangementDone(self(), target, null); generate(msg);}`

sendArrangementDone

`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.add(self());`
`ArrangementDone msg = new ArrangementDone(self(), target, null); generate(msg);`

completeArrangement

`myAgent.removeBehaviour(timer);`
`if(acceptedParticipants.size() >= M){`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.addAll(acceptedParticipants); target.add(meetingRequester);`
`Confirm msg = new Confirm(self(), target, null); generate(msg);`
`contRequestsToMeetingRequester = 0;`
`acceptedParticipants = new java.util.ArrayList<AID>();`
`} else{`
`if(contRequestsToMeetingRequester > N){`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.addAll(acceptedParticipants); target.add(meetingRequester);`
`Cancel msg = new Cancel(self(), target, null); generate(msg);`
`contRequestsToMeetingRequester = 0;`
`acceptedParticipants = new java.util.ArrayList<AID>();`
`} else{`
`contRequestsToMeetingRequester++;`
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();`
`target.add(meetingRequester);`
`AskForRequest msg = new AskForRequest(self(), target, null); generate(msg);}}`

init
`contRequestsToMeetingRequester = 0;` `acceptedParticipants = new java.util.ArrayList<AID>();`
queueEvent
`java.util.ArrayList<AID> target = new java.util.ArrayList<AID>();` `target.add(self()); e.setTarget(target); generate(e);`

Arrange state, recovering exactly the same state variables and DSC status so continuing from the previous arrangement state without discontinuity. Moreover, if the Request event is received in the Arrange state, i.e. a Request from a different MRA is received, the MBA enqueues the Request.

5.3 MAS Implementation

The implementation of the meeting organization MAS is completely supported by the enhanced ELDATool features of visual modeling and automatic code generation. Figure 12 reports a screenshot of ELDATool containing the fully developed system described above. In particular, in the package explorer there are two folders: (i) *Meeting DSC* containing the set of graphical DSC agent behaviors (MeetingBroker.dsc, MeetingParticipant.dsc, MeetingRequester.dsc) and their related actions, events, functions, and guards; (ii) *Meeting_DSC_JADE_Implementation* containing the generated source code (src package). In the central panel, the MeetingBroker.dsc is visualized (the complete diagram is reported in Figure 11). Finally in the bottom panel, an excerpt of the generated code of the MeetingBroker is reported. The code of the DistilledStateChartBehaviour framework along with the generated source code of the meeting organization MAS is downloadable as (official) Jade add-on from [22].

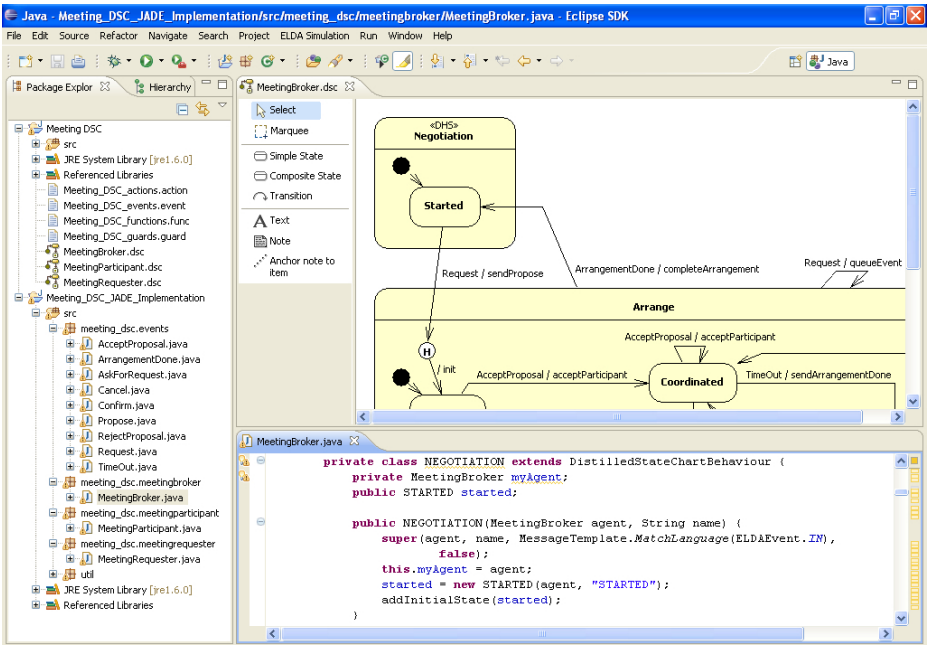

Fig. 12. A screenshot of the CASE tool showing the developed system

5.4 MAS Evaluation

The developed system was evaluated on a real experimental testbed composed of 50 workstations with the same hardware/software configuration (Windows XP Professional SP2 32-bit, CPU MD Athlon 64x2 dual-core 2.90GHz, RAM 4GB, JRE 1.6.0) interconnected by a 100Mbps switched Fast Ethernet. In particular, the goal of the evaluation was to compute the main application performance index, namely Meeting Arrangement Time (MAT), characterizing the speed with which the system replies to a user request, and analyze it by increasing the scale of the system. To this purpose, a supplemental monitoring agent-based architecture, which is able to collect statistical data about the application execution, was also developed and deployed atop the experimental testbed. The test runs were executed by varying the number of MRAs (and consequently the number of MBAs as there is a mapping 1-to-1 between MRAs and MBAs) and the number of MPAs. The number of MRAs was varied in the range [1..1000], whereas the number of MPAs was in the range [1..50]. In particular, each MPA was launched in its own JADE container, whereas all MRAs were launched in one different JADE container as well as all MBAs. Moreover, to avoid unbalance in the MPA behavior, only the successful case after 1-iteration was considered (see Section 5.1), so MPAs always agree to a meeting participation proposal as soon as they receive it.

The obtained results for the MAT index, averaged over 30 execution runs, are reported in Figure 13. As expected, MAT increases by increasing the number of MRAs and MPAs. In particular, the system with 1000 MRAs in parallel degrades its performance quadratically with the number of MPAs.

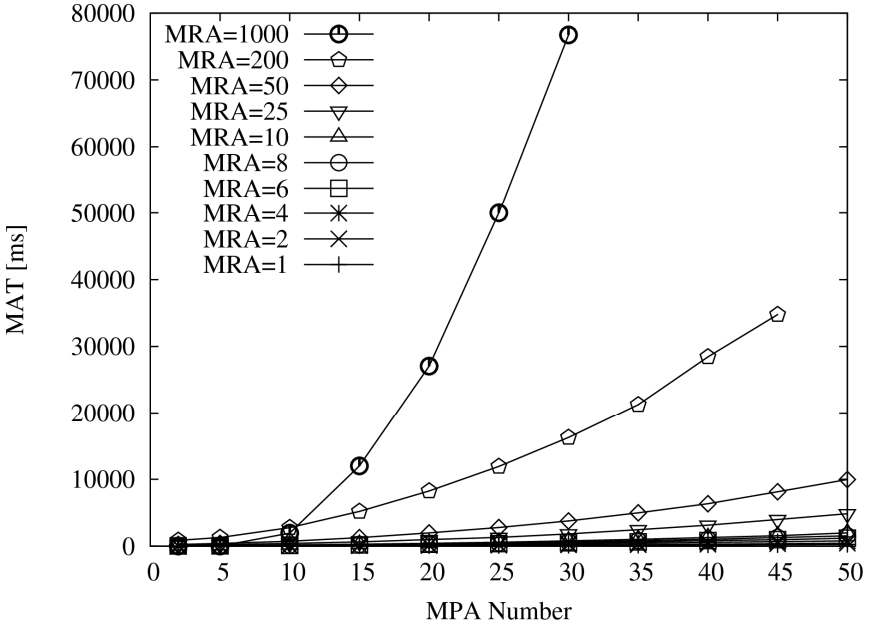

Fig. 13. Scalability evaluation of the system: meeting arrangement time by increasing the scale of the system

The MAS was also developed by using only the basic JADE framework without using the DistilledStatechartsBehaviour framework and evaluated on the same testbed with the same parameter setting. Performance evaluation results show an overlap of the performances of the DSC-based MAS and the JADE-based MAS so that the proposed framework does not introduce further overhead onto the system and system performances only rely on the JADE run-time infrastructure.

6 Conclusion

This paper has proposed programming techniques and tools based on Statecharts for the rapid development of JADE MASs. In particular, a new JADE behavior, named DistilledStateChartBehaviour, has been defined which is based on the Distilled StateCharts formalism providing hierarchical state machines including history mechanisms and features for enabling an automatic restoring of the agent execution state. The proposed DistilledStateChartBehaviour JADE add-on has been obtained on the basis of the HSMBehaviour that was purposely debugged and optimized. Moreover, the availability of a CASE tool, which supports the specification phase of JADE agent behaviors based on the DistilledStateChartBehaviour and their automatic translation into code, facilitates programming and enables rapid prototyping. As the JADE platform is one of the most used agent platform in the AOSE community to program and execute distributed agent systems, the paper proposal contributes to (i) enrich already existing agent-oriented methodologies having JADE as target platform with tools for further automating MAS development and (ii) foster a wider introduction and exploitation of Statecharts-based techniques for agents.

The effectiveness of the proposed approach for the development of MAS has been demonstrated through a case study concerning with a well-known agent-based meeting arrangement application. Specifically, the DSC-based modeling allows for a simplification of the MAS design and the availability of a visual tool supporting the development lifecycle of MAS allows for the automatic code generation so enabling rapid prototyping. Moreover, the exploitation of the DistilledStatechartsBehaviour facilitates the development of MAS in which agents interact through well-defined protocols as DSCs are a formalism well suited for defining agent protocols. This claimed effectiveness was directly experimented by also developing the MAS for meeting arrangement by means of the basic JADE framework. The developed DSC-based MAS and the basic JADE-based MAS have been also deployed and executed on an experimental testbed to analyze the system scalability. The obtained results show that scalability is only affected by the JADE run-time architecture as performances of the two developed systems overlap. Thus, the DistilledStatechartsBehaviour framework does not introduce any performance penalty.

Future work is geared at (i) integrating Statecharts-based modeling and the defined techniques within an MDD-driven agent-oriented methodology such as INGENIAS; (ii) defining a reverse engineering technique to obtain the DSC-based agent visual model from the agent source code compliant to the DistilledStateChartBehaviour.

References

1. Zambonelli, F., Omicini, A.: Challenges and Research Directions in Agent-Oriented Software Engineering. Autonomous Agents and Multi-Agent Systems 9(3), 253–283 (2004)
2. Ambler, S.W.: The Elements of UML 2.0 Style. Cambridge University Press (2005)
3. Harel, D., Gery, E.: Executable Object Modeling with Statecharts. IEEE Computer 30(7), 31–42 (1997)
4. Luck, M., McBurney, P., Preist, C.: A manifesto for agent technology: towards next generation computing. Autonomous Agents and Multi-Agent Systems 9(3), 203–252 (2004)
5. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (2007)
6. Griss, M., Fonseca, S., Cowan, D., Kessler, R.: SmartAgent: Extending the JADE agent behavior model. In: Proc. of the Agent Oriented Software Engineering Workshop, Conference in Systems, Cybernetics and Informatics, Orlando, Florida (July 2002)
7. Fortino, G., Garro, A., Mascillaro, S., Russo, W.: Using Event-driven Lightweight DSC-based Agents for MAS Modeling. International Journal on Agent Oriented Software Engineering 4(2) (2010)
8. Boloni, L., Marinescu, D.C.: A component agent model – from theory to implementation. In: Müller, J., Petta, P. (eds.) Proc. of the Second International Symposium from Agent Theory to Agent Implementation (2000); Trappl, R. (ed.): Proc. of Cybernetics and Systems, Austrian Society of Cybernetic Studies, Vienna, pp. 663–639 (March 2000)
9. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi agent systems with a FIPA-compliant agent framework. Software Practice and Experience 31, 103–128 (2001)
10. Griss, M., Fonseca, S., Cowan, D., Kessler, R.: Using UML State Machines Models for More Precise and Flexible JADE Agent Behaviors. In: AAMAS AOSE Workshop, Bologna, Italy (July 2002)
11. Kessler, R., Griss, M., Remick, B., Delucchi, R.: A Hierarchical State Machine using JADE Behaviours with Animation Visualization. Technical report, University of Utah (2004)
12. Fortino, G., Russo, W., Zimeo, E.: A statecharts-based software development process for mobile agents. Information and Software Technology 46(13), 907–921 (2004)
13. Fortino, G., Garro, A., Mascillaro, S., Russo, W.: ELDATool: A Statecharts-based Tool for Prototyping Multi-Agent Systems. In: Proc. of the Workshop on Objects and Agents (WOA 2007), Genova, Italy, September 24-25 (2007)
14. Nwana, H., Nduma, D., Lee, L., Collis, J.: ZEUS: a toolkit for building distributed multi-agent systems. Artificial Intelligence Journal 13(1), 129–186 (1999)
15. Cost, R.S., Finin, T., Labrou, Y., Luan, X., Peng, Y., Soboroff, I., Mayfield, J., Boughannam, A.: Jackal: A Java-Based Tool for Agent Development. In: Working Notes of the Workshop on Tools for Developing Agents, AAAI 1998 (1998)
16. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: Model transformations for improving multi-agent system development in INGENIAS. In: Gomez-Sanz, J.J. (ed.) AOSE 2009. LNCS, vol. 6038, pp. 51–65. Springer, Heidelberg (2011)
17. Cossentino, M.: From Requirements to Code with the PASSI Methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies. Idea Group Inc., Hershey (2005)

18. Caire, G., Coulier, W., Garijo, F., Gómez-Sanz, J., Pavón, J., Kearney, P., Massonet, P.: The Message Methodology. In: Bergenti, F., Gleizes, M.-P., Zambonelli, F. (eds.) Methodologies and Software Engineering for Agent Systems The Agent-Oriented Software Engineering Handbook, vol. 11, pp. 177–194. Springer (2006)
19. Eshuis, R.: Reconciling statechart semantics. Science of Computer Programming 74(3), 65–99 (2009)
20. FIPA (Foundation for Intelligent Physical Agents), FIPA Agent Management Support for Mobility Specification, Document FIPA DC00087C (2002/05/10) (2002), http://www.fipa.org/
21. Fonseca, S., Griss, M., Letsinger, R.: Agent Behavior Architectures A MAS Framework Comparison, Technical report, N. HPL-2001-332, University of Utah (2001)
22. The JADE DistilledStateChartBehaviour add-on, documentation and software (2010), http://jade.tilab.com/