
**UNIVERSITÀ DELLA CALABRIA**

**Dipartimento di Elettronica,
Informatica e Sistemistica**

**Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXII ciclo**

***Tesi di Dottorato***

# A methodology for the simulation-based prototyping of distributed agent systems

**Samuele Mascillaro**

UNIVERSITÀ DELLA CALABRIA
Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica

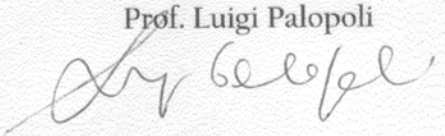
XXII ciclo

*Tesi di Dottorato*

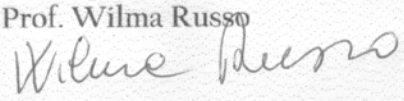
# A methodology for the simulation-based prototyping of distributed agent systems

Samuele Mascillaro

Coordinatore
Prof. Luigi Palopoli



Supervisore
Prof. Wilma Russo



DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Settore Scientifico Disciplinare: ING-INF/05

# Contents

# 1 Introduction

## 1.1 Motivation

The ubiquitous diffusion and usage of the Internet have promoted the development of new kinds of distributed applications characterized by a huge number of participants, high decentralization of software components and code mobility, which are typical of application domains such as distributed information retrieval, content management and distribution, and e-Commerce. In these application domains, the agent-based computing paradigm [72] has been demonstrated to be effective for the analysis, design and implementation of distributed software systems. In particular, in the context of the agent oriented software engineering (AOSE) [62], several agent-oriented methodologies based on suitable agent models, frameworks and tools, have been defined to support the development lifecycle of distributed agent systems (DAS). Their in-depth analysis has allowed to identify the key elements for the provision of an effective development of distributed agent systems: the agent model, the development methodology and the supporting CASE tool.

The agent models aim at providing abstractions for the modelling of the agent behavior and interactions. Basically they can be classified in two large groups: (i) models based on intelligent agent architectures [72, 81] 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 [11, 104]. 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 Internet computing, agent models and related frameworks based on lightweight architectures, asynchronous messages/events and state-based programming such as JADE [7], Bond [10], and Actors [2], have demonstrated great effectiveness for modeling and programming agent-based distributed applications. In particular, such models define suitable abstractions for the modelling of reactiveness and proactiveness of agent behavior and agent interactions. However, they mainly consider messages (and related message-based protocols and infrastructures) as a means of interaction among agents and mobility as an auxiliary feature of agents.

Considering the exploitation of coordination models and infrastructures based not only on messages but also on events, tuples, blackboards and other coordination abstractions [16] can provide more effectiveness in designing complex agent interactions and more efficiency in their actual implementation. Moreover, mobility, if considered a main feature of agents, can provide a powerful means for dynamic organization of distributed components modelled as mobile agents [11]. Thus mobility also enables and demands for new non-message-based coordination models.

The agent-oriented development methodologies aim at supporting the development lifecycle of agent-based systems from analysis to deployment and maintenance. They can be classified into general-purpose and domain-specific methodologies. The general-purpose methodologies such as Gaia [113], PASSI [23], Tropos [12], Ingenias [89] are suitable for the development of multi-agent systems in different application domains whereas the domain-specific methodologies can be more effectively exploited in a given, very specific application domain. Apart from their context of use, they are all based on a meta-model of multi-agent system which loosely or tightly depends on a reference agent model and a phase-based iterative development process. Agent oriented methodologies for Internet-based distributed agent systems should incorporate not only a MAS meta-model and related agent model suitable for distributed computation but also effective prototyping methods able to validate the design models before deployment in a large-scale distributed testbed. In particular, dynamic validation based on simulation is emerging as a powerful means for functional and non functional validation of designed agent systems in a large-scale controlled environment. To date a few agent-oriented development methodologies have been proposed in the literature, such as Electronic Institutions [103], DynDEVS/James [99], CaseLP [76], GAIA/MASSIMO [41], TuCSon/pi [54], Joint Measure [101], Ingenias/Repast [90]. They incorporate simulation to support the design phase of the MAS development lifecycle with the main focus on the validation and performance evaluation of the designed MAS model. Moreover, two important characteristics of agent-oriented methodologies are high degree of integration with other methodologies and availability of a CASE tool supporting the process phases. The former would allow for an easy integration with other methodologies for the purpose of enriching already existing methodologies or creating new and more effective ones. The latter would allow for automating the development process phases and their transitions so providing more robust development and rapid prototyping.

## *1.2 Thesis proposal and contributions*

The main objective of the proposed thesis is the definition of a methodology, also supported by a CASE tool, for the simulation-based prototyping of distributed agent systems. The proposed methodology, hereafter referred as ELDAMeth, is based on the key features enabling the development of DAS

delineated in the previous section. In particular, ELDAMeth relies on the ELDA (Event-driven Lightweight Distilled Statecharts Agents) agent model and related frameworks and tools, and on an iterative development process seamlessly covering the modeling, coding and simulation phases of DAS. ELDAMeth can be used both stand-alone and in conjunction/integration with other agent-oriented methodologies which provide support to the analysis, (high-level) design, and implementation phases. A simplified process schema of ELDAMeth is shown in Figure 1.1.

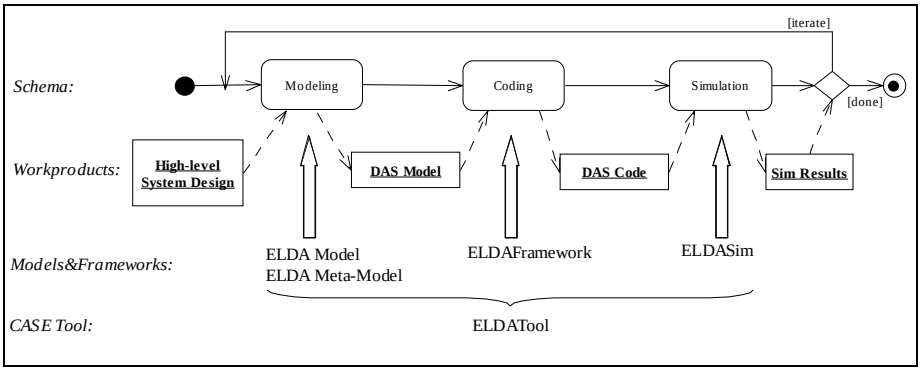



Figure 1.1: A basic ELDAMeth process schema.

The Modeling phase produces the DAS Model on the basis of the High-Level System Design which is a high-level design of the distributed system to be prototyped. In particular, the DAS Model is specified according to the ELDA MAS meta-model which provides the structure and the behavior of agent systems based on the ELDA model. Moreover, the High-Level System Design can be defined either ad-hoc or by means of another methodology supporting the analysis and high-level design phases.

The Coding phase receives the DAS Model and automatically produces the DAS code according to the ELDAFramework, which provides all the programming abstractions defined in the ELDA MAS meta-model.

Finally, the Simulation phase produces the Simulation Results in terms of execution traces of the simulated DAS and calculation of the defined performance indices which must be carefully evaluated with respect to the functional and non-functional requirements. Such evaluation could lead to a further iteration step which starts from a new (re)modeling activity. In particular, the Simulation Results come from the execution of the DAS Simulator carried out through ELDASim, a discrete-event simulation framework for ELDA agents. The DAS Simulator is obtained by synthesizing the DAS Code with the simulation parameters and performance indices, defined on the basis of the requirements.

All the described phases are fully supported by the ELDATool, a CASE tool completely developed in this thesis work to enable: (i) the visual modelling of the DAS under-development, (ii) the automatic translation into code of the

DAS Model, (iii) the execution of the DAS Simulator in a large-scale controlled environment.

The main contributions of the thesis to the AOSE research field are in the following areas:

- *Agent models*. The proposed ELDA (Event-driven Lightweight Distilled StateCharts-based Agents) model incorporates the three enabling features for distributed agent systems: lightweight reactive/proactive behavior, multi-coordination and mobility. In particular: (i) the agent behavior is based on the Distilled StateCharts formalism [50] which allows to effectively structure the behavior in hierarchies of states, transitions among states, and (re)actions attached to transitions; (ii) the agent interactions are based on high-level asynchronous events which enable multi-coordination among agents and between agents and non-agent components through the exploitation of multiple coordination structures; (iii) the agent mobility relies on a coarse grain strong mobility model which allows for agent transparent migration (both autonomous and passive). Moreover, the structure of ELDA-based DAS is specified according to the well-defined ELDA MAS meta-model which provides the modelling abstractions related to agents and their infrastructures in which agents execute and through which agents interact.
- *Agent frameworks*. The ELDAFramework is a Java-based implementation of the ELDA MAS meta-model and makes it available a rich set of programming abstractions enabling the implementation of distributed agent systems based on the ELDA model. ELDA-based agent systems developed through the ELDAFramework can be actually executed through simulation by the ELDASim framework. In particular, ELDASim is an ELDA-oriented discrete event simulation framework which provides simulation abstractions and components allowing functional validation and performance analysis of ELDA-based agent systems.
- *Agent methodologies*. In this thesis work, ELDAMeth is also exploited to define other two agent-oriented methodologies based on simulation: PASSIM and MCP. PASSIM is a simulation-based process for the development of multi-agent systems which is obtained by integrating the well-known and established Process for Agent Societies Specification and Implementation (PASSI) methodology with ELDAMeth. In particular, PASSI-based design models of the multi-agent system under-development are semi-automatically translated according to the ELDA MAS meta-model and then validated through simulation. The validated multi-agent system can be therefore implemented and deployed according to the PASSI phases. The Multi-Coordination based Process (MCP) is an iterative process for the design of mobile agent interactions based on two subsequent phases of modeling and evaluation. In particular, the modeling phase uses interaction patterns and coordination models to semi-automatically provide alternative coordination solutions whereas the evaluation phase relies on simulation to evaluate and

compare such solutions on the basis of ad-hoc defined performance indices. The evaluation phase is carried out through ELDAMeth.

- *Agent-oriented CASE tools*. The ELDATool is an integrated development environment implemented as a Java-based Eclipse plug-in, which aims at supporting developers during the modelling, coding and simulation phases of ELDAMeth. In particular, ELDATool provides in an integrated fashion: (i) a visual editor for the modelling of agent behavior in terms of Distilled StateCharts machines; (ii) an automatic translator which implements the translation rules from the ELDA MAS meta-model to the ELDAFramework so allowing to translate ELDA-based models into Java code; (iii) a visual editor to configure the simulation parameters and to control the execution of ELDA-based simulation programs.
- *Distributed agent systems*. The application of ELDAMeth for the prototyping of DAS in key Internet-based application domains such as e-Commerce, content delivery and distributed information retrieval has resulted in the definition of novel distributed agent systems in such domains. In the e-Commerce domain, the objective is the design and validation of an agent-based e-Marketplace (AeM) modeled as a multi-agent system. In particular, several new kinds of consumer agents characterized by mobility and related policies have been defined and their evaluation shows the effectiveness of the defined mobile consumer agents and their efficiency for searching, contracting and buying goods. In the context of Content Delivery Networks (CDN), the main goal is the design and evaluation of several distributed architectures for clustering surrogate. In particular, three novel architectures (master/slave, multicast-based, peer-to-peer) have been proposed. The obtained results show that the designed surrogate clustering architectures allow to improve performance with respect currently available CDN architectures. In the distributed information retrieval domain, the objective is the design and evaluation of novel agent-based solutions for searching information across a network of federated locations. In particular four solutions have been proposed in which agent interactions are designed by using single and multiple coordination models. The obtained results show that the use of the multi-coordination approach can improve efficiency of the provided agent solutions.

## *1.3 Thesis organization*

After providing some background concepts and a discussion about related work in chapter 2, the thesis is organized in three main parts. In the first part involving chapter 3, ELDAMeth is described in detail; in particular, all the process phases are explained along with related models (ELDA model and ELDA MAS meta-model), frameworks (ELDAFramework and ELDASim) and supporting tool (ELDATool). The second part of the thesis (chapter 4) details a complete case study of the application of ELDAMeth in the Content Delivery Networks domain from modelling to evaluation. The third part

presents the use of ELDAMeth and, in particular, PASSIM and its application in the e-Commerce application domain (chapter 5), and the Multi-Coordination-based Process and its application in the distributed information retrieval domain (chapter 6). Finally, conclusions summarizing the main contributions and results of this thesis are drawn and then the future work is briefly delineated.

# 2 Background and related work

The objective of this chapter is to provide fundamental background concepts and a presentation of the literature work strongly related to the thesis proposal. In particular, the chapter is organized as follows: (i) the first section introduces basic agent-based concepts in distributed systems engineering and, particularly, in the context of AOSE (Agent Oriented Software Engineering); (ii) the second section describes the main agent models for distributed computing; (iii) the third section presents coordination models among agents; (iv) the last section discusses simulation-based methodologies related to the methodology proposed in this thesis.

## *2.1 Distributed Computing and Agent Oriented Software Engineering*

Today's distributed software engineering approaches are increasingly adopting abstractions deriving from the agent-based computing: the majority of modern distributed systems are intrinsically proper to be developed in terms of agent-based systems, and the modern distributed systems (e.g. control systems, mobile and pervasive computing environments, internet-based applications) are *de facto* agent-based systems as they are indeed composed of autonomous, situated, and social components.

Very often computing systems integrate autonomous components: autonomy implies that a component integrates an autonomous thread of execution, and can execute in a proactive way (i.e. taking the initiative). This is the case of most modern control systems for physical domains, in which control is not simply reactive but also proactive, implemented through a set of cooperative autonomous processes or, as is often the case, via embedded computer-based systems interacting with each other or via distributed sensor networks. The integration in complex distributed applications and systems of (software running on) mobile devices can be tackled only by modeling them in terms of autonomous software components. Another example is represented by Internet-based distributed applications which are typically made up of autonomous processes, possibly executing on different nodes, and cooperating with each other.

Moreover, computing systems are also typically situated: they have an explicit notion of the environment where components are associated to and executed, and with which components explicitly interact. Control systems for

physical domains, as well as sensor networks, tend to be built by explicitly managing data from the surrounding physical environment, and by explicitly taking into account the unpredictable dynamics of the environment via specific event-handling policies. Other examples are Internet applications and web-based systems that to dive into the existing Internet environment, are typically engineered by clearly defining the boundaries of the system in terms of the “application”, including the new application components to be developed, and ‘‘middleware’’ level, as environmental substrate in which components are to be embedded. In addition, mobile and pervasive computing applications recognize (under the general term of context-awareness) the need for applications to model explicitly environmental characteristics rather than to model them implicitly in terms of internal object attributes.

Finally, in modern distributed systems we can recognize sociality aspects which come in different flavors: (i) the capability of components of supporting dynamic interactions; (ii) the somewhat higher interaction level, overcoming the traditional client-server scheme; (iii) the enforcement of some sorts of societal rules governing the interactions. Control systems for critical physical domains typically run forever, cannot be stopped, and sometimes cannot even be removed from the environment in which they are embedded. Nevertheless, these systems need to be continuously updated, and the environment in which they live is likely to change frequently, with the addition of new physical components and, consequently, of new software components and software systems. For all these systems, managing openness and the capability to automatically re-organize interaction patterns is crucial, as is the ability of a component to enter new execution contexts in respect of the rules that are expected to drive the whole execution of the system. With reference to pervasive computing systems, lack of resources, power, or simply communication un-reachability can make nodes come and go in unpredictable ways, calling for re-structuring of communication patterns, as well as for high-level negotiations for resource provision. Such issues are even exacerbated in mobile networking and P2P systems, where interactions must be made fruitful and controllable despite the lack of any intrinsic structure and dynamics of connectivity. Similar considerations apply to Internet-based and open distributed computing. There, software services must survive the dynamics and uncertainty of the Internet, must be able to serve any client component, and must also be able to enact security and resource control policy in their local context: E-marketplaces are the most typical examples of this class of open Internet applications.

Thus, the explicit adoption of agent-based concepts in distributed systems engineering would carry several advantages [84]:

- *autonomy of application components*, even if sometimes directly forced by the distributed characteristic of the operational environment, enforces a stronger notion of encapsulation (i.e., encapsulation of control rather than of data and algorithms), which reduces the complexity of managing systems with a high and dynamically varying number of components;

- *taking into account situatedness explicitly*, and modeling environmental resources and active computational entities in a differentiated way, rather than being the recognition of a matter of fact, provides for a better separation of concerns which, in turn, helps reduce complexity;
- *dealing with dynamic and high-level interactions* (i.e., with societal rather than with architectural concepts) enables to address in a more flexible and structured way the intrinsic dynamics and uncertainties of modern distributed scenarios.

The above considerations make more appealing the use of techniques based on the agent-paradigm to deal with the design of distributed applications but however there is a big concern on its applicability in an industrial context. In fact, industrial applicability implies the definition of repeatable, reusable, measurable and robust software process and techniques for the development of multi-agent systems (MASs) [8].

To manage multi-agent systems complexity, the research community has produced a number of methodologies that aim to structure agent development. However, even if practitioners follow such methodologies during the design phase, there are difficulties in the implementation phase, partly due to the lack of maturity in both methodologies and programming tools. There are also difficulties in understanding the nature of what is a new and distinct approach to systems development and in implementation due to:

- a lack of specialized debugging tools;
- skills needed to move from analysis and design to code;
- the problems associated with awareness of the specifics of different agent platforms;

For these reasons, the development multi-agent systems requires providing reasoning at appropriate levels of abstraction, automating the design and implementation process as much as possible, and allowing for the calibration of deployed multi-agent systems by simulation and run-time verification and control [73].

Despite a number of languages, frameworks, development environments, and platforms that have appeared in the literature, implementing multi-agent systems is still a complex task. For this reason, a lot of effort in the agent field has been devoted to the definition of techniques, methods and tools for supporting Agent Oriented Software Engineering (AOSE). The main goal of AOSE is to determine how agent qualities affect software engineering, and what additional tools and concepts are needed to apply software engineering processes and structures to agent systems. Specific areas of interest here include [9, 72, 84]:

- **methodologies for agent based systems**. Traditional methodologies of software development, driving engineers from analysis to design and development, must be tuned to match the abstractions of agent-oriented computing;
- **requirements engineering for agent based systems.** The agent-oriented community have introduced new concepts to cope with the needs arisen from these complex problems then requirements elicitation

techniques should provide a way to reason and model them since the early stages of agent-based software development process;

- **agent-oriented analysis and design**. Novel formal and practical approaches to analysis and modeling agent-based systems are required to deal with agent's features such as autonomy, situatedness, and sociality.
- **techniques for specification of (conceptual) designs of agent systems.** The development of specific notation techniques to express the outcome of the various phases of an agent-based system development process are needed, because traditional object- and component-oriented notation techniques cannot easily apply;
- **verification, validation and testing techniques**. Verification is normally based on formal theories that allow the analysis of a system in order to determine whether certain properties hold. When such properties consist on whether the application fulfils the requirements, usually verification is referred as validation. Testing is the activity of looking for errors in the final implementation;
- **agent design patterns**. There is by now a growing literature on the use of patterns to capture common design practices for agent systems which aim at increasing re-use and quality of code and at the same time reducing the effort of development of agent based systems;
- **agent models.** A variety of agent models are being investigated and each of them is suitable to model different types of agents or specific aspects of agents: purely reactive agents, logic agents, agents based on belief, desire and intentions, etc;
- **agent-based infrastructures**. To support the development and execution of agent-based systems, novel tools and novel software infrastructures are needed. Various tools are being proposed to transform specifications into actual agent code and a variety of middleware infrastructures have been deployed to provide proper services supporting the execution of distributed agent based systems;
- **agent-based systems architecture**. As it is necessary to develop new ways of modelling the agents, in the same way it is necessary to develop new ways of modelling an agent based systems as a whole. A variety of approaches are being investigated to model agent based systems such as, approaches inspired by societal, organisational, and biological metaphors.
- **tools to support the agent system development process**. There is a need to integrate existing tools into Integrated Development Environments (IDEs) rather than starting from scratch. At present there are many research tools, but few are integrated with generic development environments, such as Eclipse; such advances would boost agent development and reduce implementation costs.

## 2.2 Agent models for distributed computing

To date many agent models have been defined and proposed which share the fundamental notion of agent but significantly differ in terms of agent architectures and the problems they aim at solving. In particular, they can be roughly distinguished in two main categories:

- *Artificial Intelligence (AI)-oriented* models which are based on intelligent agent architectures [72, 81] ranging from reactive agents (e.g. Brook's subsumption architecture) to deliberative agents (e.g. BDI agents) and are mainly oriented to problem-solving, planning and reasoning systems;
- *Distributed Computing (DC)-oriented* models which are based on the mobile active object concept encompassing mobile agent architectures [11, 104] and are more oriented to distributed computation in open and dynamic environments like the Internet.

As features such as lightweight agent architectures, asynchronous agent interaction and state-based programming, which characterize many agent models belonging to the *DC-oriented* category, have demonstrated great effectiveness for modeling and programming agent-based distributed applications, only agent models of such category which share such basic features will be shown in the following. In particular, Jade [7], HSM/SmartAgent [66], Bond [10, 75] and Actors [2, 3] are briefly synthesized paying attention to the provided abstractions aimed at specifying behavior, interactions and mobility of agents.

### 2.2.1 JADE

JADE (Java Agent Development Environment) [7] is a software framework aimed at programming agent applications in compliance with the FIPA specifications for inter-operable intelligent multi-agent systems. In particular, the purpose of JADE is to simplify development while ensuring standard compliance through a comprehensive set of system services and agent behaviors and protocols. To design agent behavior, JADE offers several types of supplied programming abstractions which are defined as direct subclasses of the Behaviour class; such class provides the skeleton of an elementary task to be carry out by an agent. A CompositeBehaviour which extends the Behaviour class, is one of the available agent behaviors and it can have an arbitrary number of sub-behaviors: each CompositeBehaviour implements a particular scheduling policy used to select which sub-behavior to fire at each round. A particular CompositeBehaviour is the FSMBehaviour one [6] that schedules its children according to a finite state machine whose states, which are behaviors themselves, correspond to the FSMBehaviour children: it provides methods to register sub-behaviors as FSM states and to register transitions, marked by integer label, between states. Note that transitions only serve to link states and do not encapsulate any action. A FSMBehaviour keeps a pointer to the current sub-behavior: as soon as this sub-behavior ends, the FSMBehaviour checks its internal transition table and, according to the returned value of *onEnd* method of the current child

(returning the label of a transition), selects the next behavior that has to be executed. As a consequence, the execution semantics of the FSMBehaviour is not driven by events but by action completions. With respect to the interaction among agents, the provided communication model is peer-to-peer though a multi-message context which is provided by interaction protocols and conversation identifiers. In particular, communication among agents is performed through asynchronous message passing and the language used to represent messages is FIPA ACL. Each agent has a mailbox (agent message queue) where the system posts messages sent by others agents: whenever a message is posted in the message queue, the receiving agent is notified. However, when, or if, the agent picks up the message from the queue for processing is a design choice of the agent programmer. Moreover, to design agents' interactions several interaction protocols are made available providing a sequence of acceptable messages and a semantic for those messages. With respect to agent mobility, JADE implements a *weak mobility* model: after migration the agent will continue its execution from the beginning of its behavioral code. It's worthy noting that, in order to save agent execution state, programmers have to explicitly capture the agent execution state. More in detail, agent migration is triggered when an agent calls the *doMove* method that causes the agent to cease its current activities and suspend itself (i.e. the agent state goes from the ACTIVE to the TRANSIT state) while the system relocates it. Moreover, agent modeling is not directly supported by an official visual toolkit for developing MAS according to the JADE model.

### 2.2.2 The HSM/SmartAgent model

To realize flexibility, partitioning, and control when programming JADE agents, an architecture supporting hierarchical state machine based programming of agent behaviors, augmented with several flexibility enhancing mechanisms (such as events and dispatcher chains) has been proposed. In particular, HSM (Hierarchical State Machine) extends JADE framework with uniform message and system events, a multi-level dispatching mechanism that matches and routes events, and a hierarchical state machine that is based on the UML state machine model [56, 66].
Using JADE FSMBehaviour as their starting point, to design agent behavior authors defined HSMBehaviour which inherits from JADE ComplexBehaviour and can manage a set of nested HSMBehaviour (states) or any type of behavior. In particular, HSMBehaviour inherits *onStart*, *onEnd* and *action* method from JADE ComplexBehaviour to represent entry, exit and activity actions of an UML State, respectively.
Like UML State Machine, transitions among states (HSMTransition), labeled by an ECA rule, are driven by an event (HSMEvent) and may have source and destination that are in any state within the entire hierarchy: when a transition happens across boundary, *onStart* and *onEnd* methods code are guaranteed to be executed in an appropriately hierarchical fashion. When an

HSMEvent is delivered to an agent (it is posted in the its message queue), HSMBehaviour searches for a valid transition in an hierarchical fashion through invoking trigger methods (passing the HSMEvent) and according to the returned value of the trigger method, the transition is evaluated if it should be fire: if the trigger returns true, then action associated to the transition is executed and the transition to the target state is performed whereas if any of transitions can fire, event is removed. It's worthy noting that events processing in HSM JADE was implemented according to a Run-to-Completion semantics: if a event that trigger a transition arrives in the middle of the execution of behavior, its execution is terminated and then the transition is taken. With respect to the interaction among agents, although HSM is explicitly event-driven (every action that an agent is subjected to is translated into an event), interaction among agents is mainly based on JADE's interaction model that is asynchronous message passing. In particular, ACL messages sent by other agents are wrapped in a MessageEvent objects for uniform handling and processing and delivered to agents through their message queue. Moreover, system events such as ExceptionEvent and TimerEvent (external events) and events signaled by other behaviors and activities of the same agent, such as SuccessEvent and FailureEvent (internal events) are wrapped in a MessageEvent object. With respect to agent mobility, features supplied by HSM are the same of those offered by JADE's mobility model. To develop agent based systems according to the HSM model, is made available an visual tool, HSMEditor, that allow for modeling an HSMBehaviour and generating Java code associated to the state machine modeled. Authors have also developed a tool that allows them to visualize the execution of an HSMBehaviour in order to check agent behaviors.

### 2.2.3 The Bond agent model

The major components of a Bond agent [10, 75] are: the model of the world, the agenda, strategies and the multi-plane state machine. The model of the world represents the information that an agent has about its environment; the agenda defines the goal of an agent; a strategy generate agent's actions (based upon model of the world and agenda); and the multi-plane state machine is a data structure in which each state has associated a strategy. Bond model defines the agent behavior as a multi-plane state machine in which each plane is modeled as a flat finite state machine: multi-plane state machine can be seen as a different way to expressing parallelism amongst activity (Statecharts machines express parallelism as concurrent sub-states). Authors to make multi-plane state machine independent on the model of the world, adopt a simpler state machine in which transitions are unconditional and only states can generate actions. In particular, each state is associated to a strategy that performs actions which are considered atomic from the agent point of view (events cannot interrupt them). Moreover, adopted state machine cannot include embedded sub-states.

As the current state of each plane of a multi-plane state machine is defined by the active state, the state of the agent is defined by a vector of states (the active one for each plane). It is worthy noting that there is interdependency amongst planes hence all of them share a common model of the world and the transition triggered by one plane are applied to the whole structure of planes.
Moreover, the behavior of an agent can be modified at run-time because it is possible to change the structure of the multi-plane state machine associated to an agent. Authors defined several operations on agent's behavior such as joining, splitting and trimming. *Joining* two agents produces a new agent characterized by (i) a multi-plane state machine which contains all the planes of the joined agents and (ii) a model which is created by merging the models of the joined agents. In case of *splitting* of an agent, two agents are obtained which inherit the full model of the original agent whereas the union of their planes gives the planes of the original agent. Finally, the *Trimming* is an operation which is performed when the multi-plane state machine of an agent contains states and transitions unreachable and they can be eliminate in order to make the agent smaller. With respect to the interaction among agents, the Bond agent system uses asynchronous message passing and KQML as communication language although authors assert that design principles are largely independent on the communication language. In addition to asynchronous message passing, Bond agent system offers [10]:

- support for synchronous communication;
- an implementation of the publish-subscribe model;
- an implementation of the tuples space model based on IBM TSpace

Moreover, authors argue that agent interaction can be also described in terms of knowledge sharing then they provide two ways to share the model of the world which are dependent upon the agent initiating of the process:

- *Push mode*: an agent copies part of its model to the model of the other agent
- *Pull mode*: an agent copies part of the model of the remote agent into its own model.

With respect to mobility, as authors considered agent migration a rare event in the life of agents, they deliberately choose to implement a weak migration model also reflecting the difficulties of migrating running Java threads. In particular, agents are only allowed to migrate when all their active strategies (at the time of the request of migration) complete their execution. More in detail, agent migration is performed when an agent requests migration through a specific message. Then, the Bond agent system (i) pauses the agent as soon as its current active strategies are completed; (ii) serializes the agent; (iii) sends to the new host the agent serialization and the model of the world held by the agent; (iv) creates a new agent and a new model at destination; (v) starts the new agent after deletes both the old agent and the old model in the initial location. To develop agent based systems according to Bond model several ways have been made available: the multi-plane state

machine can be constructed as a program but a more flexible approach is a declarative approach (Python-based) which is interpreted by an agent factory; moreover, Bond objects can be visualized and edited through a visual editor which allows for editing fields and dynamic properties of an agent.

### 2.2.4 The Actor model

Authors use the Actor model [2, 3] as a basis for modeling distributed software architectures since it provides a general and flexible model of concurrency: Actors may be used to build typical architectural elements including procedural, functional and object-oriented components. Conceptually, an Actor encapsulates a state, a single-thread of control and a set of procedure to manipulate its state. At design level, an Actor is an active object which consists of a private local state, a set of methods and a globally unique name; moreover, each Actor is associated to a mail buffer in which messages sent to it are queued. Actor computation step is defined in terms of processing messages and consists of removing a message from its mail buffer, processing it, and, eventually, changing the computational environment through three (abstract) basic actions:

- *Send* messages to other Actors;
- *Create* Actors with specified behaviors;
- become *Ready* to process the next message.

Such actions are factored into signal-notification pairs: an Actor generates *signal* events (which request the system to perform some actions) and then it blocks itself; as soon as the system sends back a *notification* event (which alerts the actor that its request has been performed) the requester Actor resumes its behavior. In particular, an actor blocked on a:

- **ready** may be resumed by receiving a **deliver** notification;
- **transmit** may be resumed by receiving a **continue** notification;
- **create** may be resumed by receiving a **newActor** notification.

Moreover, in order to enhance the Actor base model, the authors introduced: (i) the Meta-Actor concept aimed at customize the content of generated signal, which is an entity capable of processing signals generated by Actors; (ii) the Actor group concept aimed at model parallel computation, which represents a multi-thread component of the architecture. With respect to interaction among agents, Actors interact by asynchronously exchanging messages to one other: each Actor can generate messages and receive messages which are queued into its mail buffer. There isn't any information about Actors mobility in their model but *Actor Foundry*, which is a Java-based programming environment for developing Actor systems, should implement the weak mobility notion. Currently, Actors modeling has to be performed by coding because there isn't any official visual toolkit for developing agent base systems according to the Actor model.

### 2.2.5 A comparison

With reference to the agent behavior model, Jade offers, among different agent behavior types, an agent behavior (called FSMBehaviour) based on flat finite state machines (FSMs), SmartAgent 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. More in detail, the execution semantics of the HSMBehaviour, Bond 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.
With reference to the agent interaction model, all the models 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.
With reference to the agent mobility all the models, Jade, SmartAgent, Bond and Actors (in particular the implementation of Actors carried out in the ActorFoundry framework [3]) are based on a weak mobility model [52].
In table 2.1 the main features of the above introduced related models with respect to the three main dimensions of agent modeling are synthesized.

Table 2.1: Comparison of related Agent models.

| **MODELS**/*DIMENSIONS* | *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 |

## *2.3 Agent interaction design*

Agent interaction design represents a very important stage during the design process of an agent-based distributed system as it influences both the efficacy and the efficiency of the developed agent system.
As it initially happened for the agents behavior design, the use of patterns to drive the agent interaction design is notably increased and a lot of contributes have been provided in literature [1, 29, 64, 65, 106, 111] which will be shown in Section 2.4.1. On the other hand, several coordination models [22] have been introduced in literature to allow the agents interaction

design according to the interaction scenarios features which will be shown in Section 2.4.2.

Table 2.2: Interaction and coordination patterns.

| AUTHORS | PATTERN | DESCRIPTION |
|---|---|---|
| Aridor and Lange [1] | Meeting | Provides a way for two or more agents to initiate local interaction at a given host. |
| | Locker | Defines a private storage space for data left by an agent before it is temporarily dispatched (send) to another destination. |
| | Messenger | Defines a surrogate agent to carry a remote message from one agent to another. |
| | Facilitator | Defines an agent that provides services for naming and locating agents with specific capabilities. |
| | Organized Group | Composes agents into groups in which all members of a group travel together. |
| Kendall et al. [64, 65] | Conversation | Concerns with a sequence of messages between two agents, taking place over a period of time: agent messaging may occur within a context established by previous messages. |
| | Facilitator | Allows for interaction among agents which do not have to have direct knowledge of one another as it is based on a Mediator agent which provides a gateway or clearinghouse for agent collaboration. |
| | Agent Proxy | Enables agents to collaborate directly with one another through a proxy agent which provides distinct interfaces and allows agent to be engaged in multiple conversations. |
| | Protocol | Establishes conversation policies that explicitly characterize communication sequences. |
| | Emergent Society | Enables reactive agents to collaborate without known protocols as actions performed by agents can stimulate behaviour of neighbour agents. |
| Deugo et al. [29] | Blackboard | Decouples interacting agents from each other as instead of communicating directly, agents interact through an intermediary which provides both time and location transparency to the interacting agents. |
| | Meeting | Allows for interaction among agents without the need for explicitly naming among them as they know a meeting point in which agent can coordinate themselves through a statically located agent. |
| | Market Maker | Allows for interaction among agents through a third party agent which takes an active role in the coordination process enforcing the house rules of agent interaction. |
| | Master/Slave | Allows for vertical coordination which is used to coordinate the activity of a delegating agent and two or more delegated agents in which delegated agents carry out a subtask for delegating agent. |
| | Negotiating Agents | Deals with the situation where the interacting agents appear as peers to each other, but need to align their actions for some reason. |

### 2.3.1 Coordination and interaction patterns

Patterns are reusable solutions to recurring design problems, and provide a vocabulary for communicating these solutions to others [53]. The purpose is to increase re-use and quality of code and at the same time reduce the effort of development of software systems. Selecting patterns as a methodology for agent development is being justified by referring to the previous successes of applying patterns in traditional software technology. There is by now a growing literature on the use of patterns to capture common design practices for agent systems [72, 107, 111]. In the following, some pattern-based agent design approaches, which also cover issues related to the design of interaction among agents, are summarized (see Table 2.2 for a brief description of each proposed patterns).

Aridor and Lange [1] describe a set of domain-independent patterns for the design of mobile agent systems. They classify mobile agent patterns into *travelling, task,* and *interaction patterns* and propose some patterns belonging to each the classes. Patterns in the *travelling class* specify features for agents that move between various environments, patterns of the *task class* specify how agents can perform tasks and patterns of the *interaction class* specify how agents can communicate and cooperate. In particular, with reference to the interaction patterns authors present the following ones: Meeting, Locker, Messenger, Facilitator, and Organized Group which concern with locating agents and facilitating their interactions.

Kendall et al. [64] capture common building blocks for the internal architecture of agents in patterns. Authors suggest a seven-layer architecture pattern for agents, and sets of patterns belonging to each of the layers. The presented seven layers are: mobility, translation, collaboration, actions, reasoning, beliefs and sensory but the exact number of layer may vary. Compared to the previously mentioned pattern classification scheme in the work by Aridor and Lange, the layered architecture has a similar logical grouping of patterns. The mobility layer together with the translation layer corresponds to the class of traveling, the collaboration layer corresponds to the class of interaction, and the actions layer corresponds to the class of task. In particular, with reference to the interaction patterns authors present the following ones: Conversation, Facilitator, Agent Proxy, Protocol and Emergent Society which concern with how agents cooperate and work with other agents. The main difference between this and the previously mentioned approaches for mobile agents, is that this one aims to cover all main types of agent design patterns.

Deugo et al. [29] identify a set of patterns for agent coordination, which are, again, domain-independent. Authors classify agent patterns into architectural, communication, traveling, and coordination patterns. Moreover, they identify an initial set of global forces (Mobility and Communication, Standardization, Temporal and Spatial Coupling, Problem Partitioning and Failures) which are different types of criteria that engineers use to justify their designs and implementations. In particular, with reference to the coordination patterns authors present the following ones: Blackboard, Meeting, Market Maker, Master/Slave and Negotiating Agents which are well-documented solutions to recurrent problems related to the coordination among agents.

Kolp et al. [67] propose a catalogue of architectural styles and agent patterns for designing MAS architectures at a macro- and micro- level adopting concepts from organization theory and strategic alliances literature. Although interesting, these patterns define how goals assigned to actors participating in an organizational architecture will be fulfilled by agents without focus on coordination issues.

### 2.3.2 Coordination models

Coordination basically implies the definition of a coordination model and related coordination architecture or related coordination language. In particular, in the context of Agents, an agent coordination model [22] is a conceptual framework which should cover the issues of creation and destruction of agents, communications among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time. In this framework, the *coordinables* are the coordinated entities (or agents) whose mutual interaction is ruled by the model, the *coordination media* are the abstractions enabling the interaction among the agents, and the *coordination laws* are the rules governing the interaction among agents through the coordination media as well as the behavior of the coordination media itself.

To date, agent coordination has been classified by using several taxonomies [16, *75,* 88]. Agent coordination can be classified in control-driven and data-driven according to the taxonomy proposed in [88]. In control-driven models, entities receive command and react to them whereas in data-driven models the entities receive data items, interpret and react to them. Another interesting coordination models taxonomy is that proposed in [75] in which coordination models have been classified in endogeneous and exogeneous. In coordination models belonging to the first category, entities are responsible for receiving and delivering coordination information whereas in models belonging to the latter category, the actual coordination is outside of their scope. However, in the context of Internet-based computing a reference taxonomy for agent coordination is proposed in [16]; here, the focus is on agents strongly characterized by mobility. It is worth noting that, although mobility can be an enabling feature for improving efficiency and effectiveness in distributed systems, mobility poses further issues on agent coordination as mobile entities demand for more complex coordination frameworks. The reference taxonomy for Internet-based mobile agent coordination takes these issues into consideration and, in particular, classifies coordination models on the basis of the degrees of spatial and temporal coupling induced by the coordination models themselves. *Spatial coupling* requires that the entities to be coordinated share a common name space or, at least, know the identity of their interaction partners; conversely, *spatial decoupling* allows for anonymous interaction, i.e. there is no need for an acquaintance relationship. *Temporal coupling* implies synchronization of the interacting entities whereas *temporal decoupling* allows for asynchronous interactions.

On the basis of the reference taxonomy (see Table 2.3), the following coordination models have been classified: Direct, Meeting-oriented, Blackboard-based and Linda-like.

In *Direct* coordination models, agents usually coordinate using RPC-like primitives or asynchronous message passing. The former coordination method implies temporal and spatial coupling whereas the latter implies only spatial coupling as temporal decoupling can be obtained by adopting message reception queues [115]. The majority of the Java-based mobile

agent systems [104], particularly the most famous ones, namely Aglets, Voyager, Ajanta and Grasshoppers, rely on this model.

In Meeting-*oriented* models, agents coordinate using implicit or known meeting points (places where meeting can occour) which allow them to communicate and synchronize with other participating agents. In particular, this model solves the problem of locating agents, found in direct coordination, but requires agents to know the meeting point. Moreover, it requires synchronization among the agents, e.g. they must be co-located at the meeting point during at a certain period of time in order to be able to interact with each other. Examples of systems based on this coordination model are Ara [91] and MOLE [5].

In Blackboard-*based* models, agents interact through shared message repositories at each place, called blackboards, in which agents can store and retrieve information under the form of messages. The main advantage of this model is the temporal decoupling: messages are left on the blackboard no matter where the corresponding receivers are or when they will read the message. The drawback of backboard systems is the spatial coupling: the agents have to visit the correct place and agree on common messages types and formats. Ambit [18] is an example of a system using the blackboard-based coordination.

In *Linda*-like models, coordination is also based on a shared namespace, but unlike blackboards, it uses an associative tuple space which allow for insertion and retrieval of tuples; such models organize information as tuples which can be accessed and retrieved through associative pattern-matching. The main advantage of the Linda-like coordination is its temporal uncoupling and partial spatial uncoupling. Although it does not require agent synchronization, the patterns used to access the tuple space embody some implicit knowledge of the peer agent's interaction requirements.

Recently new coordination models which can be classified as spatially and temporally decoupled have emerged in the context of Internet applications: (i) the *reactive tuple space* models which enable programmable coordination spaces [15**,** 85], (ii) *transiently shared tuple space* models which handle interactions in the presence of active mobile entities [93], and (iii) the *publish/subscribe event-based* models [19, 27, 87].

The *reactive tuple space* model extends the simple tuple space model by introducing computational capability inside the coordination media under the form of programmable reactions, triggered by operations on the tuple space or by other reactions, which can influence the behavior of agents. This model also allows for the separation of concerns between agent computation and coordination issues.

The transiently *shared tuple space* [93] is another Linda-like coordination model. As Linda offers a static, persistent and globally accessible tuple space, which is scarcely usable in presence of (physical or logical) mobility, the transiently shared tuple space model attempts to deal with these issues. In particular, each mobile agent owns a personal tuple space, named ITS (Interface Tuple Space). Whenever a mobile agent migrates, its ITS is

carried with it and merged to the other co-located agent's ITS making a transiently shared tuple space. *Shared* means that co-located agents can interact through the merged tuple space and *transient* means that its content changes according to agent migrations.
In the *Publish/Subscribe event-based model*, agents coordinate through asynchronous publication and notification of events so enabling temporal and spatial decoupling [87]. In particular, to be notified about a published event an agent has to previously subscribe to the topic/type/context of the published event.

Table 2.3: A spatial/temporal taxonomy for coordination models.

| | | **Temporal** | |
|---|---|---|---|
| | | *Coupled* | *Uncoupled* |
| **Spatial** | *Coupled* | Direct | Black-board |
| | *Uncoupled* | Meeting | Linda-like<br>Reactive tuple space<br>Transiently shared tuple space<br>Publish/Subscribe |

## 2.4 Simulation-based agent-oriented methodologies

The validation phase of software systems can be founded on several techniques such as formal methods, testing and simulation. In particular, testing requires the real deployment of the software system under-development whereas formal methods and simulation don't require the real deployment. Such considerations can drive the choice of the adopted validation technique especially if the target execution environment is distributed as it occurs in the case of agent-based systems executing in Internet-like environments. In fact, agent-based systems are typically constituted by a huge number of agents executing in a large scale system so testing could be a very inefficient validation technique; conversely, simulation and formal methods validation techniques can be effectively used to validate functional and not functional requirements of agent-oriented designs before their implementations and deployments. Although formal methods are recognized as suitable validation techniques of agent-based systems [84], simulation-based ones have recently emerged.
In fact, to date a few multi agent-based systems development processes have been proposed in the literature that incorporate simulation to support the agent-based system development lifecycle with the main focus on the validation and performance evaluation of the designed solution. In the following, we briefly describe some interesting approaches for the development of agent-based systems which explicitly incorporate simulation

such as Electronic Institutions [103], DynDEVS/James [99], CaseLP [76], GAIA/MASSIMO [41], TuCSon/pi [54], Joint Measure [101], and Ingenias/RePast [90].

### 2.4.1 Electronic Institutions

In [103] an integrated development environment for the engineering of multi-agent systems (MASs) as Electronic Institutions (EI) is presented. EIs provide a computational analogue of human organizations in which intelligent agents playing different organizational roles and interact to accomplish individual and organizational goals; authors define an EI as a performative structure of multi agent protocols along with a collection of normative rules that can be triggered off by agents' actions performed through speech acts.
The development environment, aimed at facilitating the iterated and progressive refinement of the development cycle of EIs, is composed of a set of tools supporting the design, validation through simulation, development, deployment and the execution of EIs. In particular, the simulation tool SIMDEI, allows for the animation and analysis of the rules and protocols specification in an EI. Moreover, SIMDEI supports simulations of EIs with varying populations of agents to conduct what-if analysis. The institution designer is in charge of analyzing the results of the simulations and returning to the design stage if they differ from the expected ones.

### 2.4.2 DynDEVS

In [99] a modeling and simulation framework (DynDEVS) for supporting the development process of MAS from specification to implementation is proposed. Authors advocate the use of controlled experimentation in order to allow for the incremental refinement of agents while providing rigorous observation facilities. The exploited framework is JAMES (Java Based Agent Modeling Environment for DEVS-based Simulation) which is aimed at an agent-oriented model design and execution supporting a modular and flexible construction of experimental frames for MASs [108]. In particular, JAMES aims at exploring the integration of the agents paradigm within a general modeling and simulation formalism for discrete-event systems, DEVS (Discrete Event Systems Specification). Such formalism lends itself to an object-oriented model design and execution, facilitating the construction of experimental frames.

### 2.4.3 CaseLP

In [76] a logic based prototyping environment for multi-agent systems, CaseLP (Complex Application Specification Environment Based on Logic Programming) is presented. Authors propose an architectural description language which can be adopted to describe the prototype at the system specification level, in terms of agent classes, instances, their provided and requested services and communication links. Moreover, at the agent

specification level, authors propose a rule-based not executable language which can easily define the behavior of reactive and proactive agents; with respect to the implementation of prototype, authors propose a platform-independent prolog-based language in which new primitives have been defined such as communication capabilities and safe state updates.
It is worth pointing out that from a simulation point of view, CaseLP is a time-driven centralized simulator with a global time known from all the agents in the system. Moreover, CaseLP integrates simulation tools for visualizing the prototype execution and for collecting the related statistics; more in detail, the CaseLP visualizer tool provides documentation about events that happen at the agent level during the MAS execution. Developers according to their needs can instrument the code of some agents after it has been loaded by adding probes to the code of agents. In this way, events related to state changes and /or exchanged messages can be recorded and collected for on-line and/or off-line visualization.

### 2.4.4 TuCSoN/ π-calculus

In [54] authors promote the use of formal tools, such as Stochastic π-Calculus process algebra, for simulating the dynamics of self-organizing multi agent systems through higher-level models defined at the early stages of design. In particular, authors assert that this approach appears to be almost unavoidable in order to foster evolving ideas and design choices, and to effectively tune parameters of the final system. Moreover, authors promote the use of SpiM (Stochastic PI-calculus Machine) which can be effectively used to simulate the Stochastic π-Calculus specifications and to track the dynamics of global system properties in stochastic simulations, validating design directions, inspiring new solutions, and determining suitable system parameters.

### 2.4.5 Joint Measure

In [101] a layered architectural framework to support agent-based system development in a collaborative, multidisciplinary engineering setting is proposed. In particular, such framework supports incremental specification, design, implementation, and simulation of agent-based systems. As authors distinguish between performing agents (executing in real-world settings) and simulated agents (simulated in a virtual environment), the proposed framework is intended to form the basis for environments that support development of agents, in both performance and simulation modes, as well as in hybrid combination (both performing and simulated agents interacting at the same time). Authors assert that for complex systems (e.g., distributed agent-oriented systems) adopting a single architectural framework, both for simulation and system development, can offer key advantages such as enable designers to study competing/alternative designs by employing a mixture of “simulated” and “performing” agents and, therefore, support migration from simulation design to operational design. The simulation is

enabled by Joint MEASURE (Mission Effectiveness Analysis Simulator for Utility, Research and Evaluation) which is built upon DEVS/HLA, a generic HLA-compliant distributed simulation environment. It's worthy noting that although Joint MEASURE affords a baseline to consider the requirements for agent development simulation environments, it is not intended to focus on agents per se.

### 2.4.6 INGENIAS/RePast

In [90] authors propose an agent-oriented methodology aimed to support modeling and simulation of social systems based on MASs. In particular, with respect to the modeling, they propose the use of an agent-oriented modeling language to specify MAS models representing complex social systems. Whereas, with respect to the simulation, authors propose the use of simulation toolkits (RePast, MASON, etc) that execute code obtained through MAS models transformation according with Model Driven Engineering (MDE) practices. This methodology is supported by a set of tools belonging to the IDK (INGENIAS Development Kit), which facilitates the edition of models and the definition of transformations for automatic code generation. Currently, the defined software component of IDK aimed at automatic code generation implements the mapping from INGENIAS models to RePast simulation toolkit.

### 2.4.7 GAIA/MASSIMO

In [41] an integrated approach for the development and validation through simulation of MASs is proposed. Such approach centers on the instantiation of a software development process which specifically includes a simulation phase that makes it possible the validation of a MAS before its actual deployment and execution. In particular, the requirements capture is supported by a goal-oriented approach based on TROPOS methodology, the analysis and design phases are supported by the Gaia methodology, the detailed design phase is supported by the Agent-UML and the Distilled State-Charts formalisms, the implementation phase is supported by the MAO Framework, the simulation phase is enabled by a Java-based event-driven simulation framework, and the deployment phase is supported by the MAAF framework which allows for the adaptation of the MAO Framework to programming abstraction provided by specific Java-based agent platforms.

### 2.4.8 A comparison

Although all the overviewed methodologies offer different approaches to the modeling and simulation of MAS, they are centered on the use of simulation to validate and evaluate the design of the MAS under-development. Actually, few of them represent a full-fledged methodology (i.e. covering all the MAS development lifecycle) for the simulation-driven development of general-purpose MAS. Moreover, only INGENIAS and EI offer visual modeling tools

for supporting the development process. In table 2.4 the methodologies are compared with respect to the key features for the provision of an effective development of distributed agent systems: (i) Agent model (Mobility, Light-weight Reactive/Proactive agent behavior, Multi-coordination); (ii) Methodology (Dynamic validation methods before implementation, Integrability with other methodologies); (iii) CASE Tool. In particular, the methodology proposed in this thesis supports all the mentioned features as it will be described in detail in the next section.

Table 2.4: Comparison among simulation-based methodologies for MAS development.

| | Agent Model | | | Methodology | | CASE Tool |
|---|---|---|---|---|---|---|
| | Multi-Coord. | Light-Weight R/P Behavior | Mobility | **Dynamic validation through simulation** | Integrability | |
| TuCSon/Pi | | | | **X** | | |
| EI | | | | **X** | | **X** |
| DynDEVS | | **X** | **X** | **X** | | **partial** |
| GAIA/Massimo | | **X** | **X** | **X** | **X** | |
| JM-DEVS/HLA | | | | **X** | | |
| Ingenias/Repast | | | | **X** | | **X** |
| CaseLP | | **partial** | | **X** | | **X** |
| ELDAMeth | **X** | **X** | **X** | **X** | **X** | **X** |

# 3 ELDAMeth: A Methodology for the Simulation-based Prototyping of DAS

ELDAMeth is a methodology specifically designed for the simulation-based prototyping of distributed agent systems (DAS). It is based on the ELDA agent model and related frameworks and tools [44, 46], and on an iterative development process covering the modeling, coding and simulation phases of DAS. ELDAMeth can be used both stand-alone and in conjunction/integration with other agent-oriented methodologies which fully support the analysis and (high-level) design phases. In particular, the development process of ELDAMeth (see Figure 3.1) consists of the following three phases:

- The *Modeling* phase (see section 3.1) produces an ELDA-based MAS design object which is a specification of a MAS fully compliant with the ELDA MAS meta-model (MMM). This design object can be produced either by (i) the ELDA-based modeler which uses the ELDA MMM and the ELDATool [32], a CASE tool supporting visual modeling and coding of ELDA-based MAS, or by (ii) translation and refinement of design objects produced by other agent-oriented methodologies such as PASSI [23], GAIA [113], MCP [42], and others [12, etc]. In particular, while the translation process centers on (semi) automatic model transformations based on the MMM of the employed methodology and the ELDA MMM, the refinement process is usually carried out manually by the ELDA-based Modeler by using the ELDATool.
- The *Coding* phase (see section 3.2) produces an ELDA-based MAS code object which is a translation of the ELDA-based MAS design object carried out manually or automatically (by means of the ELDATool) according to the ELDAFramework, which is a set of Java classes formalizing all the modeling abstractions of the ELDA MMM.
- The *Simulation* phase (see section 3.3) produces the Simulation Results in terms of MAS execution traces and calculation of the defined performance indices which must be carefully evaluated with respect to the functional and non-functional requirements. Such evaluation can lead to a further iteration step which starts from a new (re)modeling activity. In particular, the Simulation Results come from the execution of the ELDA-based MAS simulation object carried out through the ELDASim engine

(ELDASim is a Java-based event-driven simulation framework for ELDA agents). The ELDA-based MAS simulation object is obtained by synthesizing the ELDA-based MAS code object with the simulation parameters and performance indices, defined on the basis of the requirements, by means of the ELDASim framework.

In the following sections (3.1-3.3) each phase of the ELDAMeth process is described in detail.

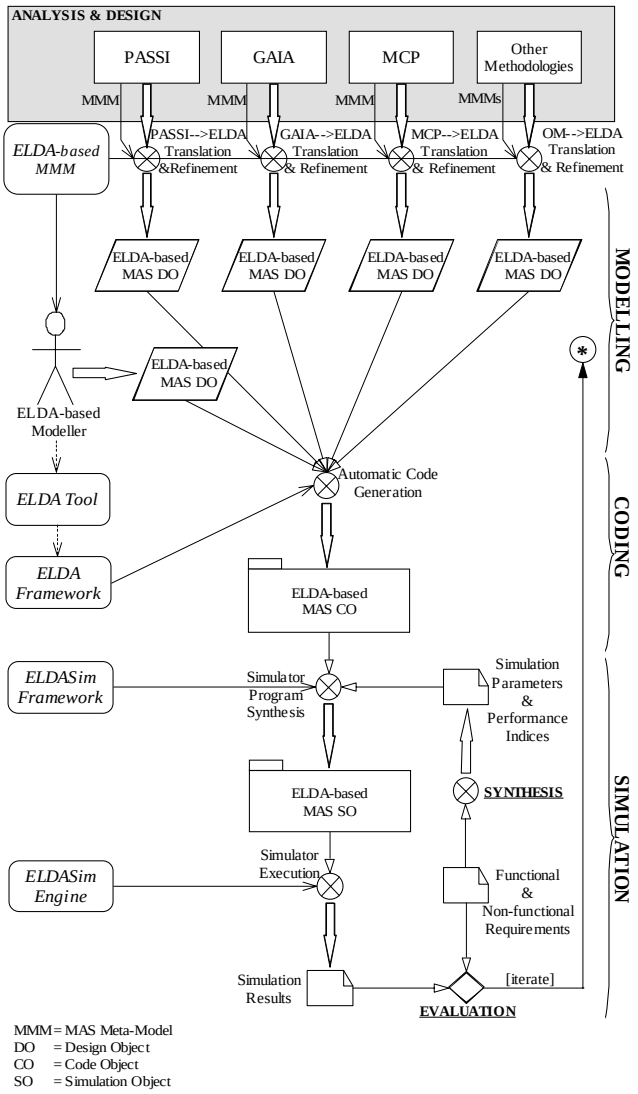



Figure 3.1: Iterative process for prototyping ELDA-based MASs.

## *3.1 Modeling phase*

The aim of the modelling phase is to fully specify the DAS with respect both to micro-level and macro-level aspects [84]. In particular, to model micro-level aspects of DAS, agent models based on lightweight architecture, asynchronous messages/events and state-based programming such as Jade [7], Bond [10], and Actors [2], have demonstrated great effectiveness. The modeling phase relies on the ELDA model that is based on the characteristics of the aforementioned models and, additionally, offers new abstractions suitable for distributed applications. In particular, the ELDA model is founded on the following basic concepts:

- **Behavior:** agent behavior can be specified both in terms of environment reactions in which an agent executes (fluctuations of available resources, network topology modifications, actions of others agents, etc. ) and in terms of pro-active actions aimed to pursuing a specific objective;
- **Interaction:** agents interactions can be specified through different coordination models according to the required degree of temporal and spatial coupling;
- **Mobility:** both autonomous agent migration and passive agent migration are performed according to a strong mobility model in which code, data and execution state are transparently restored;

Moreover, to easily model macro-level aspects of DAS, the ELDA MAS meta-model is defined which, according to the distinctive ELDA modelling concepts, provides a structured representation of the system under-development.

In the following sections, both the ELDA model and the ELDA MAS meta-model will be described in detail.

### 3.1.1 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 (how the agent reacts to a specific set of events) through the definition of agent states, transitions among states, and agent reactions (i.e. atomic actions attached to transitions); in particular, an agent 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 [50] which is derived from the Statecharts formalism [59], a visual formalism that has gained notable 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. In the following sections, the DSC formalism is briefly shown and then, the Behavioral, Interaction and Mobility models are presented in detail.

#### 3.1.1.1 Distilled StateCharts formalism

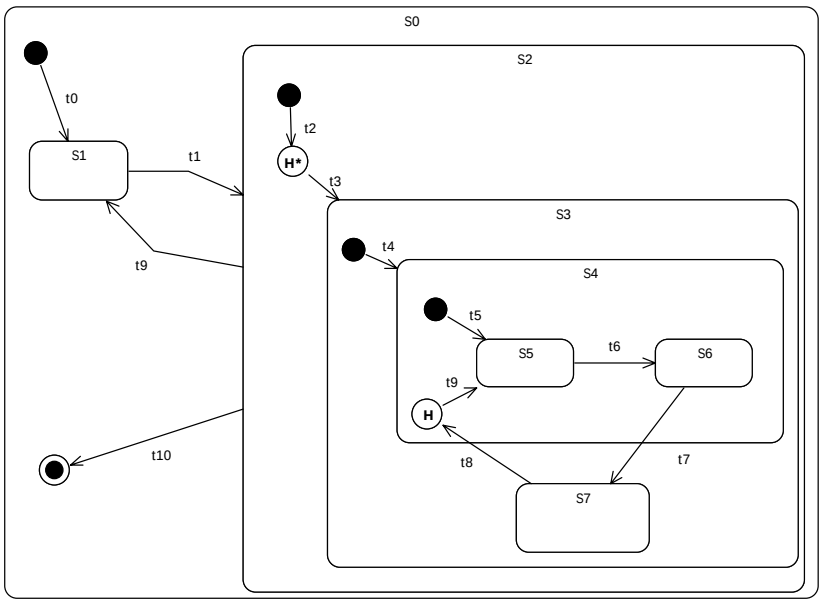
The Distilled StateCharts (DSCs) formalism [50] is derived from the Statecharts formalism which, formerly introduced by Harel, was included in UML [82] and currently is the most used formalism for modeling the behavior of object-oriented reactive systems [58]. DSCs are obtained from Statecharts as follows: (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 such processes are described in the following and exemplified with respect to the example DSC shown in Figure 3.2:

- *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 ECA rules defined as 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 among which the enclosing states are called composite state (see States S0, S2, S3, S4), the nested states are called substates and states without nested states are called simple states (see States S1, S5, S6, S7).
  - Inter-level state transitions that can originate from or lead to nested states on any level of the hierarchy (see Transition t7).
  - History entrance pseudostates (shallow and deep) allow entering substates which were most recently visited. With respect to the composite state on which the pseudostate appears, shallow history indicates that history is applied only at the level of the composite state (see State S4) whereas deep history applies the same rule recursively to all levels of the state hierarchy of the composite state (see State S2).

  - Default entrances indicate the substate of a composite state to be entered when a transition targets its border (see Transitions t0, t2, t4, t5).
  - Default history entrances indicate the substate of a composite state to be entered in the absence of any history (see Transition t3, t9).
- *Constraining process*. DSCs impose the following constrains:
  - Each DSC has an enclosing top state (see State S0).
  - 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 (see State S0, S2, S3, S4) from which the default entrance originates, which can only be labeled by an action (see Transitions t0, t2, t4, t5).
  - 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 (RTC) 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 are its formal parameters including the sender, the target, and (possibly) a list of specific event parameters (see Section 3.1.1.3).
  - Variables can be declared in each state and inside the actions so forming a hierarchical data space.

### 3.1.1.2 Behavioral model

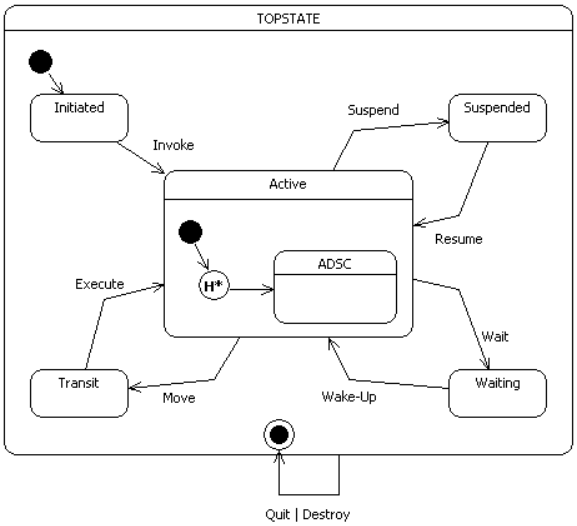
The Behavioral model allows for the specification of the agent behavior using the DSC formalism that is through the definition of agent states, hierarchical data-space, transitions among states, and agent reactions (i.e. atomic actions attached to transitions). In this way, each ELDA behavior is forged according to an extended version of the FIPA agent lifecycle template [35] 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. In particular, such ACTIVE state contains the active DSC (ADSC) composite state to which the default entrance of the DHS points: agent modelers can only refine the ADSC state to customize agent's behavior leaving it compliant with the FIPA agent lifecycle specifications. The resulting FIPA template of an ELDA agent is shown in Figure 3.3 by using both the DSC formalism and a term-rewriting formalism [70].




| **DSC ABSTRACTION** | **EXAMPLE** |
|---|---|
| Simple State | S1, S5, S6, S7 |
| Composite State | S0, S2, S3, S4 |
| Default entrances | t0, t2, t4, t5 |
| Default history entrances | t3, t9 |
| History entrance pseudostates | H, H* |
| Final Transition | t10 |
| Inter-level state transition | t7, t8 |

Figure 3.2: An example of Distilled StateCharts.

As previously mentioned, in addition to computations, and/or migration, agent reactions can generate one or more events (see Section 3.1). Such feature can be used to model pro-activity in agents behavior in terms of events generated by agent itself driving new agent reactions. To exemplify such kind of pro-activity, the ADSC of an agent is shown in Figure 3.4. In particular, when the agent is in state A and the event E1 is handled, the action Ac1 is atomically executed and, at the end of its execution, the agent goes into state B and the RTC step is completed; as the action Ac1 generates the event E2, the agent will change its current state into state B and will execute the action Ac2. The state change from A to B is driven by the agent itself in a proactive way.




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 3.3: The FIPA-based template of the agent behavior.

STATES:
Simple States = {A, B, C}

TRANSITIONS:

$t_{10}$ : TopState(ACTIVE(ADSC(A))) $\xrightarrow{\text{E1/Ac1}}$ TopState(ACTIVE(ADSC(B)))

$t_{11}$ : TopState(ACTIVE(ADSC(B))) $\xrightarrow{\text{E2/Ac2}}$ TopState(ACTIVE(ADSC(C)))

Default Entrance of ADSC=A;

ACTIONS:

$Ac_1$ : generate(E2)

Figure 3.4: An example of agent proactiveness.

### 3.1.1.3 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 the local agent server and delivered to target agents. In particular, an agent can generate through an OUT-event a service request (management, coordination, resource access, timer request, etc) and, if both the agent holds privileges and request is correct, the service will be supplied by the execution infrastructure. Moreover, depending on the type of the requested service, the requester agent will receive a notification through an IN-event which contains information about the service accomplishment or failure.

#### *3.1.1.3.1 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.

#### *3.1.1.3.2 Management Events*

Management events (see Table 3.1) 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 3.1.1.2);
- 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.

#### *3.1.1.3.3 Coordination Events*

Coordination events (see Table 3.1) enable coordination acts between agents and between agents and non-agent components (e.g. remote objects, web services) according to specific coordination models. The inter-agent coordination models considered are the Direct (synchronous and asynchronous), the Tuple-based, and the Publish/Subscribe event-based models, whereas the considered interactions between agent/non-agent components 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.

In addition, the direct model was purposely extended to enable agents to communicate using ACL messages [34] by means of the IN-event ACLMSG (extending the IN-event MSG) which formalizes a message according to the ACL structure (performative, sender, receiver, reply-to, content, language, encoding, ontology, protocol, conversation-id, reply-with, in-reply-to, reply-by).

Table 3.1: Classification of Management and Coordination 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 | |

| **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 |

### *3.1.1.3.4 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 services which were requested through the generation of corresponding OUT-events. An exception is defined per each OUT-event and includes the description of the raised exception and its typology. An exception also contains the causing Event, 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.

### 3.1.1.4 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 RTC step (see Section 3.1.1.1) and represent the only agent execution points in which MOVE events can be processed.

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) [114]. Specifically, in case of autonomous migration, migration points are known by the agent as they are specified in the agent 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 passive 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 3.3 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 retention is intrinsic due to (i) the properties of the DSCs, particularly empty states and run-to-completion semantics, and (ii) the structure of the FIPA-based template, specifically the entrance with deep history in the ACTIVE state (see Section 3.1.1.2). 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.

To exemplify the migration mechanism defined above, the ADSC of an example agent is shown in Figure 3.5. In particular, when the agent is in state A and the event E is fetched and guard G holds, the action Ac is atomically executed. At the end of its execution the agent goes into state B and the RTC step is completed.

STATES:
Simple States = {A, B}
TRANSITIONS:

$$t_{10} : \text{TopState}(\text{ACTIVE}(\text{ADSC}(A))) \xrightarrow{E[G]/Ac} \text{TopState}(\text{ACTIVE}(\text{ADSC}(B)))$$

Default Entrance of ADSC=A;

Figure 3.5: ADSC of an ELDA.

If a MOVE event arrives during the execution of Ac it is enqueued as it cannot be processed until the end of RTC step; after the completion of the RTC step, the agent is ready to process the MOVE event and then transition t2 (see Figure 3.3) is fired as follows:

$$t_2 : \text{TopState}(\text{ACTIVE}(\text{ADSC}(B))) \xrightarrow{\text{Move}} \text{TopState}(\text{TRANSIT}) ;$$

where the variable X is replaced by the current state (B) of the agent. This firing causes the update of the DHS to the simple state ADSC(B) to keep memory of the left state.
After migration the EXECUTE event is delivered and transition t3 is fired as follows:

$$t_3 : \text{TopState}(\text{TRANSIT}) \xrightarrow{\text{Execute}} \text{TopState}(\text{ACTIVE}(\text{ADSC}(B))) ;$$

where DHS is replaced with its current value.
The agent is therefore reactivated in the same execution state it was before migration as the agent execution state is represented by the current state and the values of the history pseudostates of the agent behavior. The agent data are automatically preserved due to the absence of entry and exit actions which could be executed during agent migration which makes the agent behavior pass from the Active to the Transit states and vice versa.

### 3.1.2 ELDA MAS Meta-Model

The modeling of agent-based systems based on the ELDA model is carried out through the ELDA MAS meta-model (ELDA MMM) which was specifically defined to provide design abstractions concerning with both agent modeling and environment modeling. However, as agent-system domains could require specific design abstractions which haven't been originally included within the ELDA MMM, the structure of the ELDA MMM was suitably designed to be extensible; in fact, to increase the degree of environment modeling, ELDA MMM makes it possible to introduce new design abstractions (such as new services providers or new coordination spaces) which characterize a specific execution environment. An example of such extension mechanism will be shown in Section 3.1.2.1.
As the ELDA MMM contains abstractions which concern very different aspects of agent-system modelling, its presentation is organized in six views correlated as shown in Figure 3.6:

- *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.

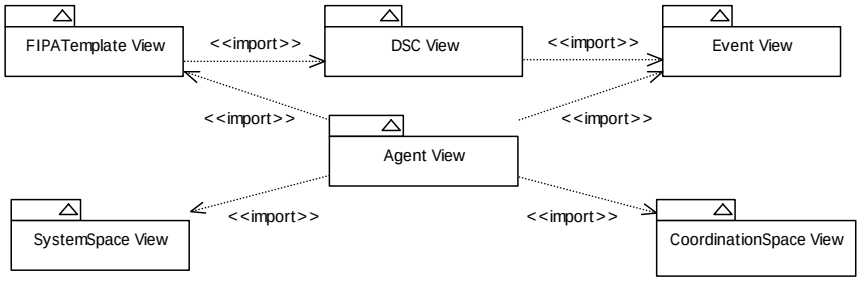



Figure 3.6: ELDA meta-model: Top-Level View.

As shown in the Agent View (see Figure 3.7) an ELDA agent is composed of a single behavior which is specified through a refined version of the FIPA template (see Figure 3.3) whose structure is shown in the FIPATemplate View (see Figure 3.10): 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 3.8).
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 the Internal and Exception events, defined in Section 3.1.1.3, are included in the Event View (see Figure 3.10).

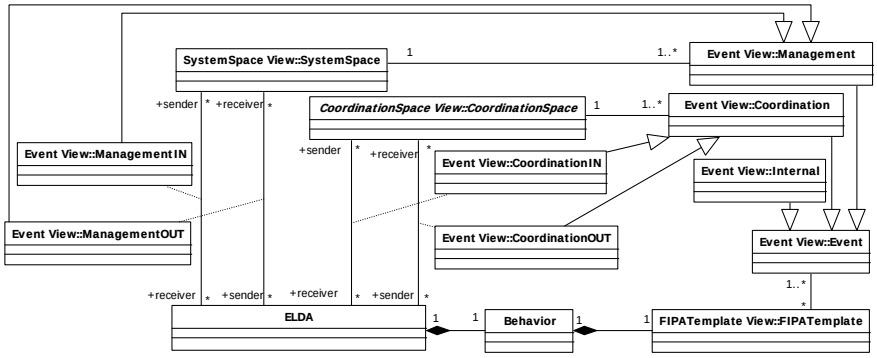



Figure 3.7: ELDA meta-model: Agent View.

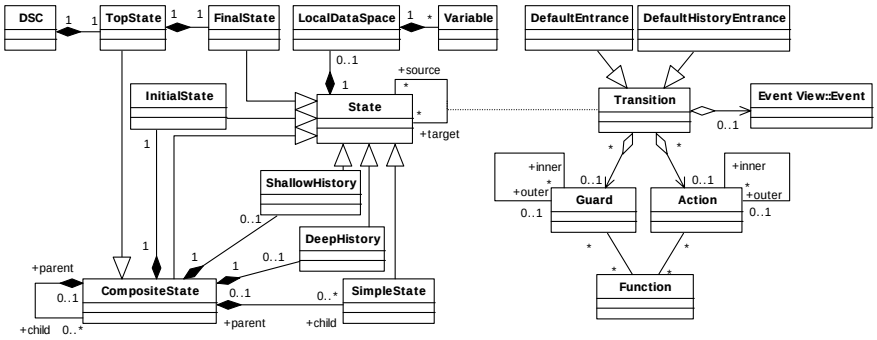



Figure 3.8: ELDA meta-model: DSC View.

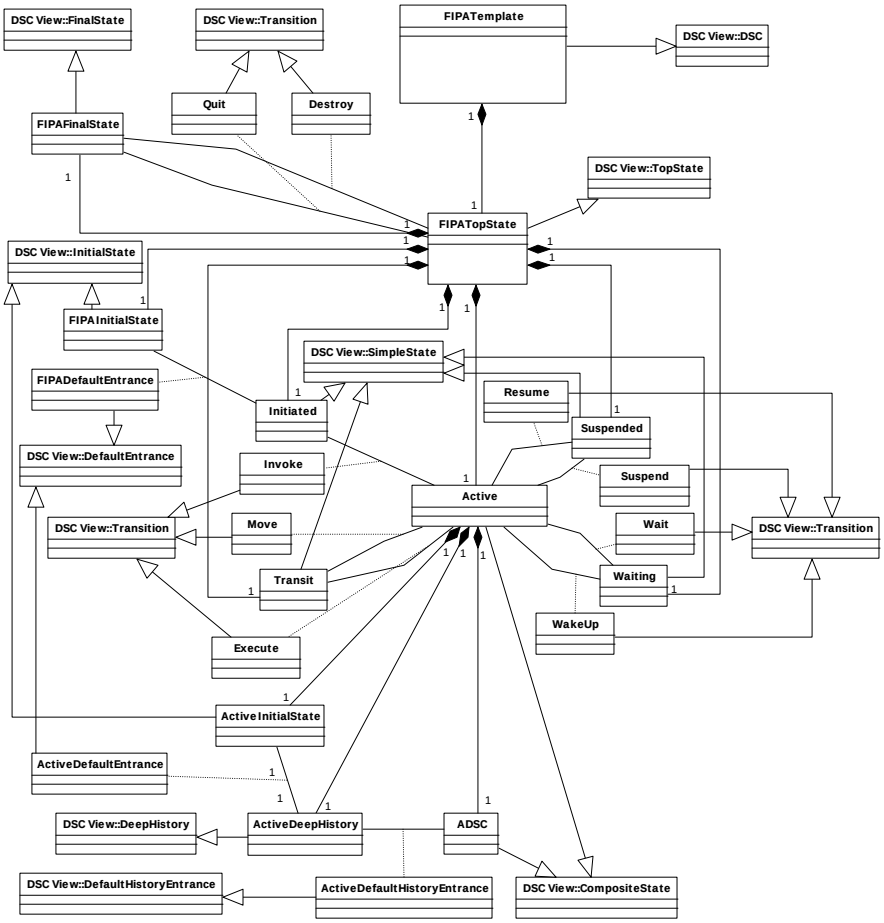



Figure 3.9: ELDA meta-model: FIPA template View.

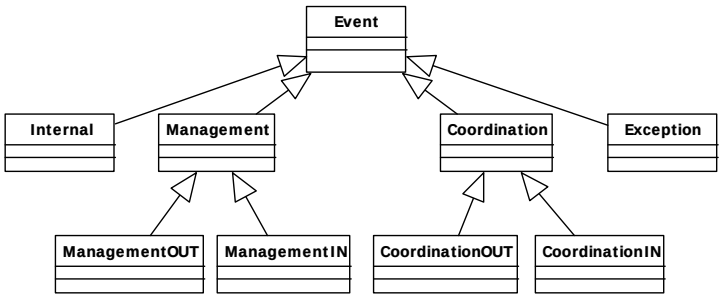


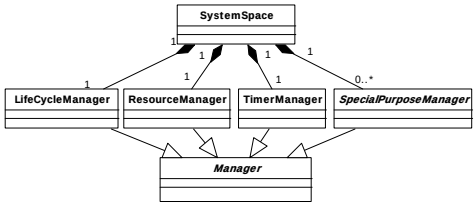

Figure 3.10: ELDA meta-model: Event View (partial).

As shown in the SystemSpace View (see Figure 3.11a), the System Space which provides system services is composed of three basic managers, LifeCycleManager, TimerManager, and ResourceManager which handle the *Management* events of the *Lifecycle*, *Timer*, and *Resource* classes, respectively. 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 (see Section 3.1.2.1).

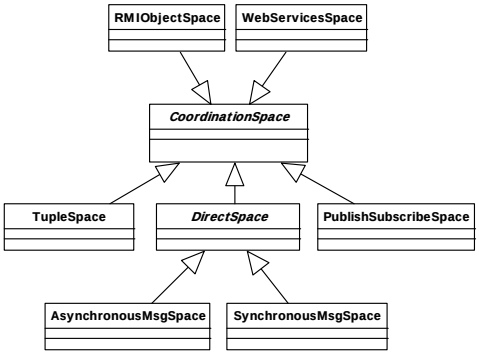
A Coordination Space represents a local or global coordination structure based on a given coordination model through which agents can interact. As shown in the CoordinationSpace View (see Figure 3.11b), six coordination spaces are currently defined: DirectSpace (AsynchronousMsgSpace and SynchronousMsgSpace), TupleSpace, PublishSubscribeSpace, RMIObjectSpace, and WebServicesSpace. The interaction with these spaces is regulated by the *Coordination* events reported in Table 3.1 and described in Section 3.1.1.3. Moreover, new coordination spaces can be easily introduced by defining new coordination space structures along with their related OUT/IN events (see Section 3.1.2.1).

### 3.1.2.1 ELDA MAS Meta-Model extensions

In this section, it is shown how the ELDA meta-model can be extended to include new logical and physical interaction models by appropriately introducing new components in the system space and/or in the coordination space along with the associated events needed to the agents to interact with them. The extension mechanism of the ELDA meta-model is exemplified through the inclusion of the PACO model abstractions [63] into our model. In particular, the PACO model focuses on purely reactive agents situated in an environment. Each agent according to the PACO model is defined by three fields which determine what the agent can perceive about its environment (Perception field), which agents an agent can interact with (Communication field), the space in which an agent can perform its actions (Action field).




(a) System Space view




(b) Coordination Space view

Figure 3.11: ELDA meta-model: SystemSpace and CoordinationSpace Views.

From a system point of view the PACO model splits the multi-agent system domain into some conceptual parts according to the VOWELS formalism [28] and thus decomposing the problem into four components: Agent, Environment, Interaction and Organization. To include the PACO model abstractions the ELDA meta-model was extended as follows:

- to model the environment (in which PACO agents are situated) the ELDA System Space was first generalized into a *LogicalSystemSpace* (formerly *SystemSpa*ce, see Figure 3.11a) and a *PhysicalSystemSpace* which was appositely extended into an *EnvironmentSpace*. The *EnvironmentSpace* handles the position of the agents and applies them (through apposite OUT-events named FORCEEVENTS) the repulsion forces resulting from co-located agents.
- to model the social laws defined in the PACO's Organization component, a new coordination space (see Figure 3.11b), named *PACOCordinationSpace,* was introduced. In particular, the *PACOCordinationSpace* monitors the environment and the agents' status (position, goals, etc.) and, if any of the pre-defined conditions are triggered, it informs the correct agent(s) on the actions to be performed through ad-hoc defined OUT-events named RULEEVENTS.

## *3.2 Coding phase*

The aim of the coding phase is to translate DAS models which have been obtained in the modelling phase according to the ELDA MAS meta-model

into executable code. To meet such objective, an object-oriented agent implementation framework named ELDAFramework (see Section 3.2.1) was designed which consists of a set of abstractions which enable the actual implementation of the DAS. Moreover, to automate such phase a set of translation rules (which map concepts belonging to the ELDA meta-models to implementation abstractions belonging to ELDAFramework) were defined.
In the following section, the structure of the ELDAFramework will be described (for technical details refer to the online documentation at the ELDATool site [32]).

### 3.2.1 ELDAFramework: a framework for the coding of ELDA-based MAS

The ELDAFramework is an object-oriented framework which allows developers to implement an ELDA-based application as it offers the implementation abstractions representing the modeling concepts offered by the ELDA MAS meta-model (ELDA MMM). Such abstractions are organized in classes inside the *dsc*, *agent* and *eldaevent* packages (see Figure 3.12). In particular, as the *eldaevent* package contains classes which include events (see Section 3.1.1.3) it is structured in *internal*, *coordination*, *management* and *exception* subpackages. In order to mirror events hierarchy, the *coordination* package has been further organized in *direct*, *tuples*, *publish_subscribe*, *services* and *rmi* subpackages whereas the management package has been further organized in *lifecycle*, *timer* and *resource* subpackages.

#### 3.2.1.1 DSC

Classes grouped in the *dsc* package allow translating a DSC-based state machine (see Section 3.1.1.1) into executable code. In particular, such package (see Figure 3.13) consists of a class hierarchy (AState, SimpleState, CompositeState e TopState) which covers modelling abstractions related to DSC states and a Context class which enables the storing of the current state of a DSC-based state machine.

#### 3.2.1.2 Agent

Classes grouped in the *agent* package (see Figure 3.14) allow translating agent-related modelling abstractions into executable code. In particular, the ELDABehaviour class uses: (i) the ELDAId class to uniquely identify an agent; (ii) the ELDAFIPATemplate class to specify the FIPA compliant agent behavior. Moreover, the ELDAFIPATemplate class is associated with the ELDAActiveState to specify the behavior of an agent which entered the Active state of the FIPA template.

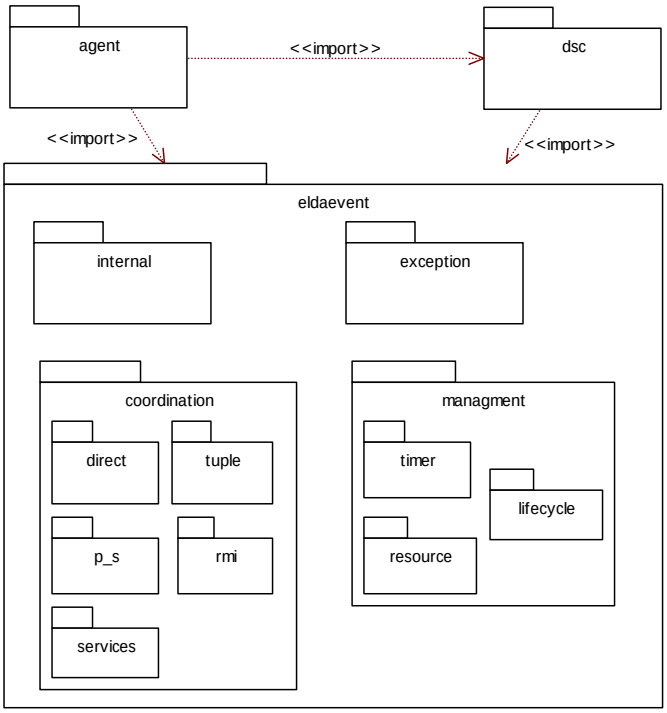



Figure 3.12: Packages of the ELDAFramework.

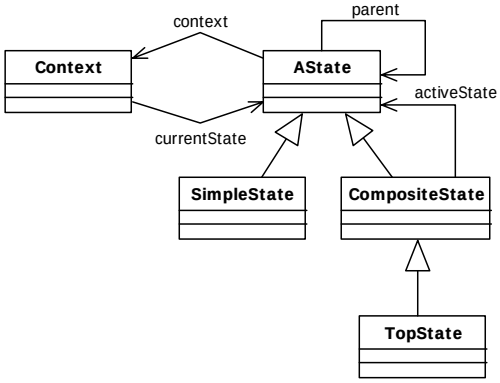



Figure 3.13: The dsc package of the ELDAFramework.

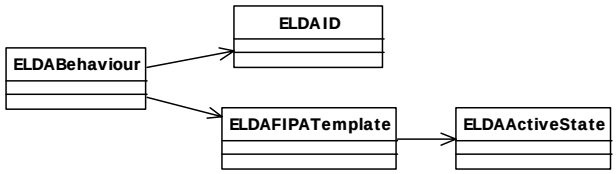



Figure 3.14: The agent package of the ELDAFramework.

### 3.2.1.3 ELDA events

The *eldavent* package and its sub-packages (see Figure 3.15) contains classes which translate into code all the events defined in the events taxonomy shown in the Interaction model (see Section 3.1.1.3).

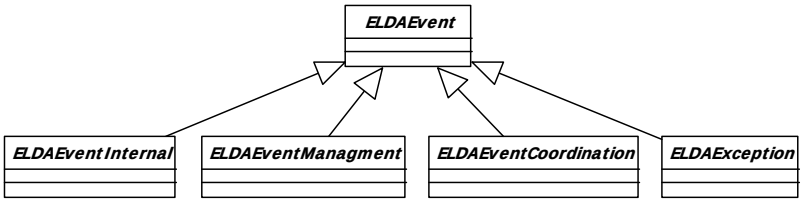



Figure 3.15: The eldaevent package of the ELDAFramework.

It is worth noting that classes representing functionalities requests sent by agents to the agent server (e.g. ELDAEventCreateRequest, ELDAEventCloneRequest, etc.) are implemented as final classes (to force developers to directly use them); whereas classes representing responses sent back by the agent server to the agents are implemented as abstract classes (to force developers to extend them); in this way, two or more transitions outgoing from the same state can be triggered by events which extend the same event (see Figure 3.16).

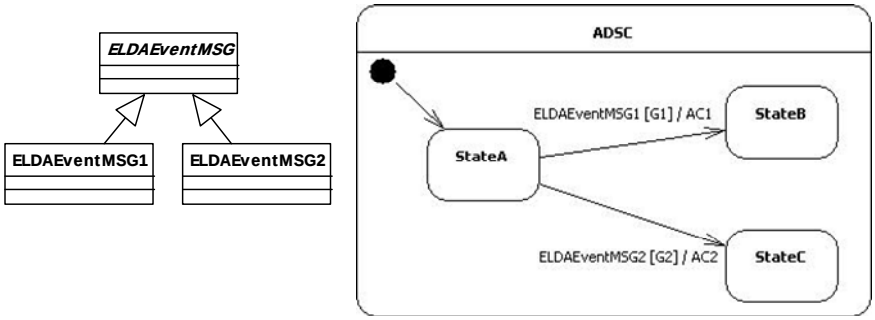



Figure 3.16: An example of transitions labeling using the event hierarchy.

#### *3.2.1.3.1 Internal ELDA events*

This package offers an abstract class, which opportunely extended, allows translating internal events into executable code.

#### *3.2.1.3.2 Management ELDA events*

Classes grouped into the *management* package (see Figure 3.17) and its sub-packages allow translating management events into executable code; in particular, according to the management events hierarchy, management package is furthermore subdivided into *lifecycle*, *resource* and *timer* packages which are detailed in the following.

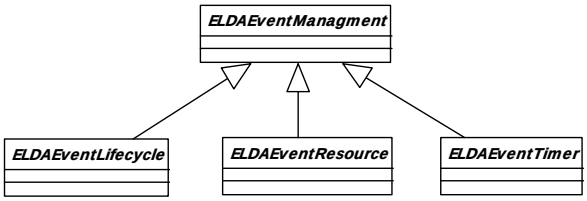



Figure 3.17: The management package of the ELDAFramework.

*Lifecycle management ELDA events*

Classes grouped in the *lifecycle* management package (see Figure 3.18) allow translating lifecycle management events into executable code. In particular, classes mirroring events belonging to FIPA template, are implemented as final classes to force developers to directly use them; remaining classes have to be extended before their use.

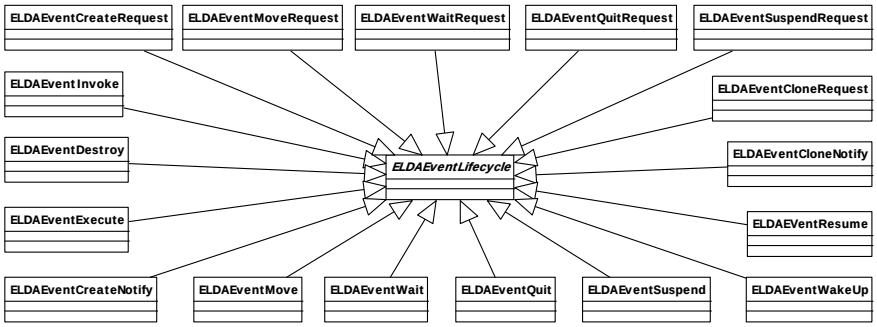



Figure 3.18: The lifecycle-managment package of the ELDAFramework.

*Resource management ELDA events*

Classes grouped in the *resource* management package (see Figure 3.19) allow translating resource management events into executable code. In particular, classes mirroring services requests (ELDAEVENTRESOURCEREQUEST, ELDAEVENTRESOURCEINPUT, ELDAEVENTRESOURCEOUTPUT, ELDAEVENTRESOURCERELEASE) are implemented as final classes to force developers to directly use them; moreover, classes mirroring responses sent back to the agents (ELDAEVENTRESOURCENOTIFY, ELDAEVENTRESOURCEINPUT) have to be extended before their use.

*Timer management ELDA events*

Classes grouped in the *timer* management package (see Figure 3.20) allow translating timer management events into executable code. In particular, ELDAEVENTCREATETIMER and ELDAEVENTRESETTIMER which mirror services requests are implemented as final classes to force developers to directly use them; ELDAEVENTCREATETIMERNOTIFY and ELDAEVENTTIMEOUTNOTIFY classes mirroring responses sent back to the agents have to be extended before their use.

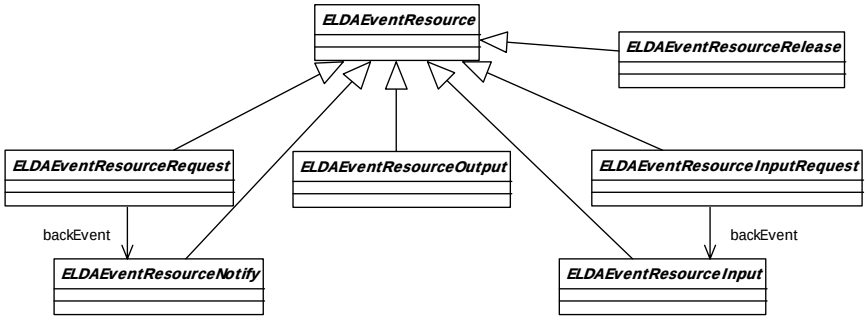



Figure 3.19: The resource-managment package of the ELDAFramework.

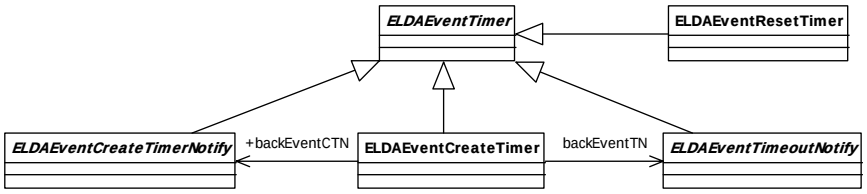



Figure 3.20: The timer-managment package of the ELDAFramework.

### *3.2.1.3.3 Coordination ELDA events*

Classes grouped in the *coordination* package (see Figure 3.21) allow translating coordination events into executable code; in particular, according to coordination events hierarchy, coordination packages is furthermore subdivided into *direct, tuples, rmi, services* and *publish/subscribe* packages which are detailed in the following.

#### *Direct coordination ELDA events*

Classes grouped into *direct* coordination package (see Figure 3.22) allow translating direct coordination events into executable code thus enable implementation of asynchronous/synchronous message passing. In particular, ELDAEVENTMSGREQUEST and ELDAEVENTRPCREQUEST classes which implement asynchronous and synchronous communication requests respectively, are implemented as final classes to force developers to directly use them; moreover, classes mirroring messages exchanged among agents (ELDAEVENTMSG, ELDAEVENTRPCRESULT) have to be extended before their use.

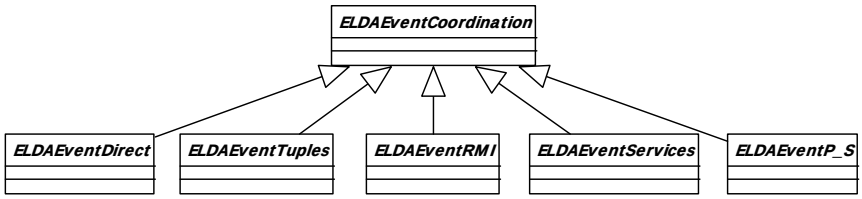



Figure 3.21: The coordination package of the ELDAFramework.

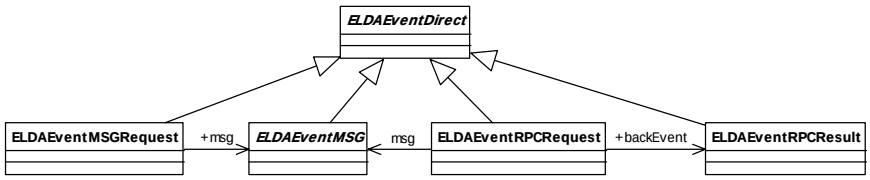



Figure 3.22: The direct-coordination package of the ELDAFramework.

### *Tuple coordination ELDA events*

Classes grouped into the *tuple* coordination package (see Figure 3.23) allow translating tuples coordination events into executable code; in particular, classes mirroring services requests (ELDAEVENTIN, ELDAEVENTOUT and ELDAEVENTRD) are implemented as final classes to force developers to directly use them; only the ELDAEVENTRETURNTUPLE class which contains tuples sent back to the agents has to be extended before its use.

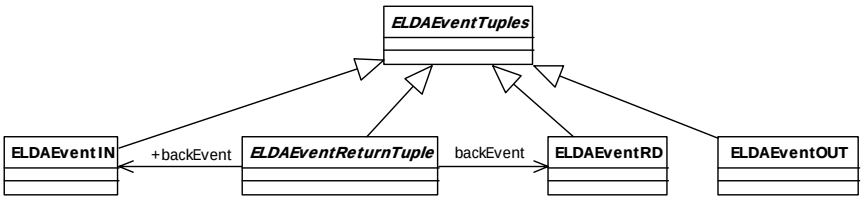



Figure 3.23: The tuple-coordination package of the ELDAFramework.

### *RMI coordination ELDA events*

Classes grouped in the *rmi* coordination package (see Figure 3.24) allow translating RMI coordination events into executable code; in particular, ELDAEVENTRMIINVOKE class which implements the RMI request is implemented as a final class to force developers to directly use it whereas the ELDAEVENTRMIRETURN has to be extended before its use.

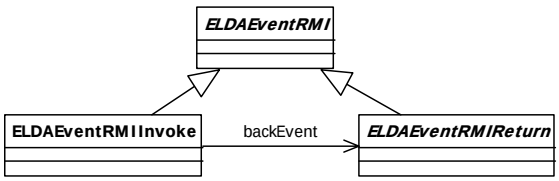



Figure 3.24: The rmi-coordination package of the ELDAFramework.

### *Services coordination ELDA events*

Classes grouped in the *services* coordination package (see Figure 3.25) allow translating web services coordinations events into executable code; in particular, classes mirroring requests (ELDAEVENTSERVICEDISCOVERY, ELDAEVENTWSDLREQUEST, ELDAEVENTSERVICEINVOKE) are implemented as final classes to force developers to directly use them; remaining classes classes mirroring responses sent back to the agents (ELDAEVENTDISCOVERYRESULT, ELDAEVENTWSDLRESULT e ELDAEVENTSERVICERESULT) have to be extended before their use.

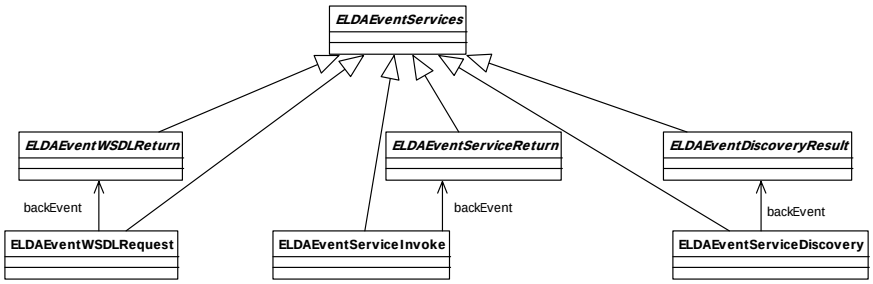



Figure 3.25: The service-coordination package of the ELDAFramework.

*Publish/subscribe coordination ELDA events*

Classes grouped in the *publish/subscribe* coordination package (see Figure 3.26) allow translating publish/subscribe coordination events into executable code; in particular, classes mirroring requests (ELDAEVENTSUBSCRIBE, ELDAEVENTUNSUBSCRIBE, ELDAEVENTPUBLISH) are implemented as final classes to force developers to directly use them; only the ELDAEVENTEVTNOTIFICATION class which contains notifications sent back to the agents has to be extended before its use.

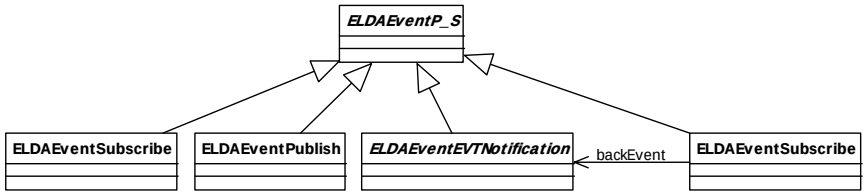



Figure 3.26: The publish/subscribe-coordination package of the ELDAFramework.

## 3.3 Simulation phase

The development process of ELDAMeth (see Figure 3.1) includes a simulation phase (see Figure 3.27) which consists of the following three subphases:

1. *Performance Indices Definition*. On the basis of functional and non functional requirements, it produces the definition of the performance indices which will be evaluated during the simulation;
2. *Simulation Implementation*. This subphase aims at the realization of a simulation program which takes into account the previously identified indices, the definition of the controlled environment and the ELDAFramework-based DAS implementation. In particular, such program uses abstractions provided by the ELDASim (see Section 3.3.1), a discrete-event simulation framework able to execute ELDA models, to define:
   - the controlled execution environment (both features characterizing the computational nodes and the network) which mirrors the real execution environment;
   - the initial DAS configuration (agents and related locations);

3. *Simulation Execution*. It consists of the execution of the DAS within the controlled execution environment and the collection of the defined performance indices which allow the analysis and the validation of the DAS under-development.

The simulation phase can be iteratively executed to modify, according to obtained simulation results, the modelling choices taken in a former iteration. In the following sub-section, the architecture of ELDASim will be explained in detail.

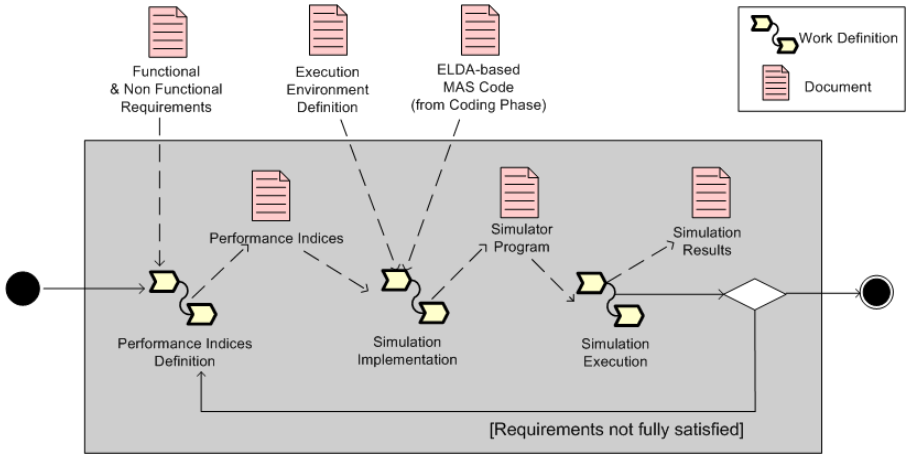



Figure 3.27: Schema of the Simulation phase.

### 3.3.1 ELDASim: a discrete-event simulation framework

The ELDA simulation environment (ELDASim) is a Java-based execution environment for ELDA agents that has been obtained as an extension of MASSIMO simulation framework [38] and aims to validate and evaluate through simulation an ELDA model based solution with respect to efficacy and efficiency aspects.

To accomplish this, ELDASim is equipped with:

- The basics mechanisms of the distributed architectures supporting ELDA agents. In particular, agent servers, the network interconnecting agent servers, and several kinds of coordination infrastructures for fully supporting the distinctive multi-coordination feature of the ELDA model.
- The simulation of accomplishment time of time-consuming operations such as agent actions, agent management operations, coordination acts, and agent migrations.
- The capture of the traces of interactions (among agents and between agents and agent servers) in terms of exchanged events, filtered in an application-specific fashion.

On the basis of the aforementioned features, the architecture of ELDASim (Figure 3.28) has been structured as following:

a. *Engine layer*, which provides the basic mechanisms and classes to simulate general purpose systems;

b. *Platform layer*, which provides a distributed infrastructure formed by a network of interconnected agent servers;
c. *Agent layer*, which provides abstractions needed to execute ELDA agents in the simulated environment;
d. *Setup layer*, which provides abstractions needed to setup the simulation such as agent server available services, virtual network configuration, initial agent locations.

*Engine layer*

To support the ELDA model features, ELDASim was developed atop an general-purpose event-discrete simulation engine that allows simulating discrete-event systems. In particular, this framework consists of computational components, named *ActiveEntity*, interacting through asynchronous message passing. *ActiveEntity* can queue *Messages* into a global queue which is handled by the *SimulationEngine* as described in the following: the *SimulationEngine* extracts the forthcoming *Message* into the queue and delivered it to the correct *ActiveEntity* for the handling process. Moreover, it is possible to program the deferring of the delivering process through the *Timer* abstraction which encapsulates a *Message* with a timeout: such message will be delivered as soon as the timeout is expired.

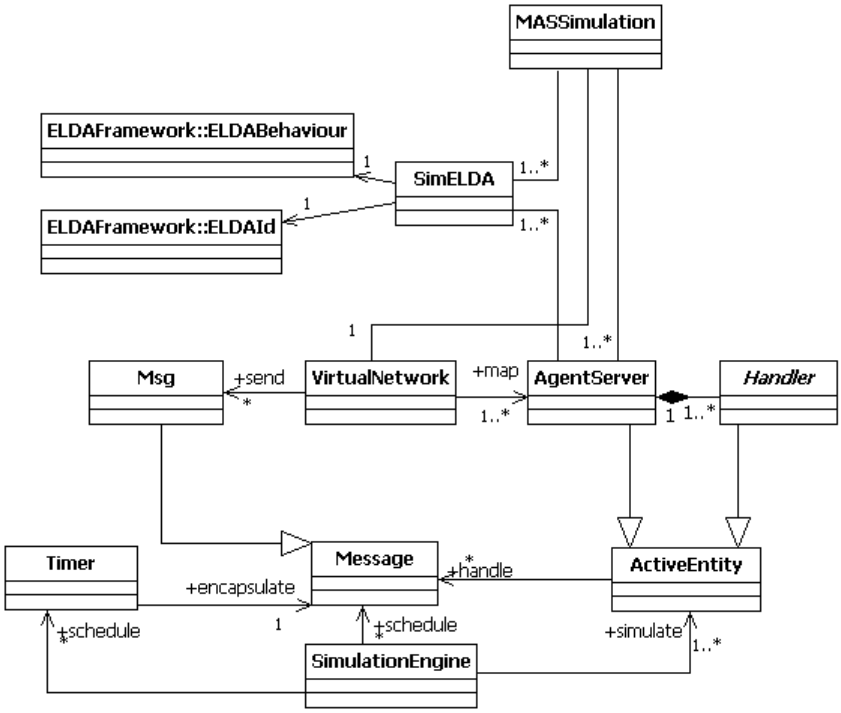



Figure 3.28: The layered architecture of ELDASim.

*Platform layer*
The platform layer provides functionalities related to both the agent lifecycle (execution, suspensions, migration, etc.) [35] and agent interactions (several coordination spaces are available to enable interactions among agents).
In particular, this layer supplies the agents execution environment though a set of *AgentServer* (in which ELDAs run) linked by the *VirtualNetwork*: each *AgentServer* can be differently customized in order to allow the simulation of a heterogeneous execution environment and the *VirtualNetwork* represents a network of *AgentServer* components.
Moreover, the following coordination spaces are implemented:

- The asynchronous Message-based coordination space which is based on proxies [115]. In particular, a message is delivered at the agent home location and, from here, forwarded to the actual agent location by following the chain of proxies left during agent migration.
- The Publish/Subscribe coordination space which behaves like a state-full ELVIN event notification system [71]. In particular, before agent migration the system removes all existing subscription of the migrating agent and re-subscribes the agent to the same notifications after the agent arrives at the new location.
- The Tuple coordination space which is based on TuCSoN [83]. In particular, each location has its own local tuple space, an instance of a TuCSoN tuple space which relies on text-based tuples.

***Agent Server.*** To take into account the extensible architecture feature of the ELDA Meta-Model (see Section 3.2.1) an *AgentServer* was designed through a component-based architecture (see Figure 3.29) in which a set of independent components named *handlers* offer specific services to the agents. In particular, using *handlers* in a cooperative fashion, an *AgentServer* provides the following functionalities:

1. agent management lifecycle, which supports registration and execution of ELDAs;
2. agent migration, which supports the migration of an ELDA from one *AgentServer* to another;
3. agent interaction, which supports the event-based interaction among ELDAs;
4. inter-AgentServer service signaling.

*Handlers* were organized in a hierarchical fashion mirroring the events taxonomy (see Section 3.1.1.3):
At the top level of the handler hierarchy the TopLevelHandler (TLH) is located which routes all the IN events targeting ELDA agents and the OUT events generated by ELDA agents. In particular, the TLH component (i) encapsulates the OUT events into MSG objects to route them within the AgentServer architecture and (ii) unwraps MSG objects including IN events to be delivered to ELDA agents. In addition to the TLH, the hierarchy of handlers consists of PrimaryHandlers, IntermediateHandlers and FinalHandlers. PrimaryHandlers and IntermediateHandlers aim to effectively route events, according to their

type, to the correct target whereas the FinalHandlers aim to provide the requested services to ELDA agents.
Interaction among handlers (belonging both to the same AgentServer and to different AgentServers) takes place through exchange of MSG objects.
Moreover, an AgentServer includes a WhitePage component which stores the ELDA agents running in the AgentServer. An entry of the WhitePage consists of pairs <ELDAId, ELDARef>, where ELDAId is the ELDA agent identifier and ELDARef is either (i) the reference to the ELDA agentidentified by ELDAId or (ii) the proxy of the ELDA agent identified by ELDAId and migrated to another AgentServer.

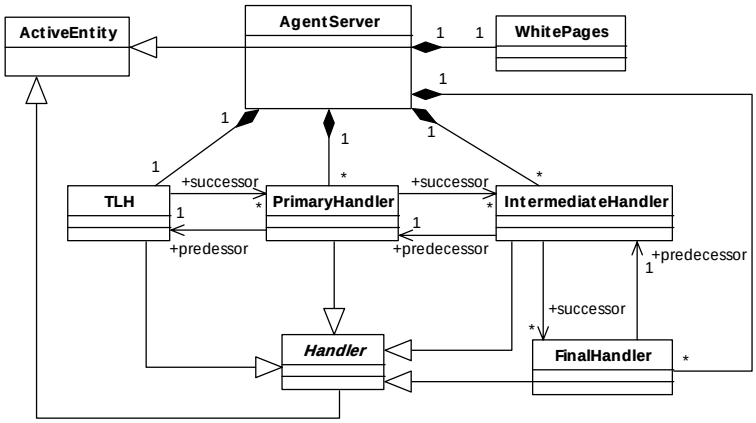



Figure 3.29: Agent Server Architecture.

***Virtual Network*.** *AgentServers* are mapped on computational nodes which are linked together by the *VirtualNetwork* component which relies on a graph-based network structure in which a network link is completely reliable and based on an end-to-end delay model (based on bandwidth, latency time and weight of the *MSG*). In particular, the calculated delay of a message transmission (which can contain both events and migrating agents), is used as timeout value of a *Timer* containing the *MSG*. Moreover, the *VirtualNetwork* aims to supply the hosts names resolving mechanism which allows for determine the host in which the message's target is located.

*Agent layer*
The **agent layer** of the ELDASim architecture provides the *SimELDA* abstraction which allows to execute an ELDA agent into the simulation framework. In particular, a *SimELDA* (unambiguously identified by an ELDAId), for each simulation step, has (i) a location depending on the specific *AgentServer* in which it runs and (ii) has a specific state of the state machine which represents its behavior.

*Simulation setup layer*
The **simulation setup layer** provides the MASSimulation abstraction which allows to setup the simulation. In particular, it allows for setup agent 's initial locations, the *AgentServer*'s features (available both system spaces and coordination spaces) and the *VirtualNetwork* configuration (network topology, nodes number, band and latency time of the links, transmission policy).

## 3.4 ELDATool: An integrated development environment for prototyping ELDA-based MAS

To facilitate the use of ELDAMeth, an integrated development environment, named ELDATool [32, 37], has been fully developed in this thesis work. It aims to support developers during the modelling, coding and simulation phases. In particular, ELDATool provides in an integrated fashion:

- a visual editor which allows to model behavior, interaction and mobility aspects of an agent-system according to the ELDA model (see Section 3.1.1);
- an automatic translator which implements the translation rules from ELDA meta-model to ELDAFramework (see Section 3.2.2);
- a visual editor to configure simulation parameters used to generate a simulation program based on ELDASim framework (see Section 3.3.1)

To support the *Modelling* phase (see Section 3.1), the tool offers the basic functionality of visual modelling of the active state of the agent behavior according to DSC formalism. 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 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 (see Section 3.2), 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 offers a visual editor to configure simulation parameters which are used to generate the simulation program according to the ELDASim framework (see Section 3.3).
Currently, the ELDATool is implemented in Java as a collection of 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 therefore quicker.

### 3.4.1 Architecture

In order to keep ELDATool architecture as modular as possible, a component-based design approach was adopted: each component is responsible of the specific aspects of the *Modelling, Coding* and *Simulation* phases. In fact, for each different agents behavior modelling aspect have been identified several editors and each of them, is capable of visually handling the related elements of the ELDA MAS meta-model and producing an instance of such meta-model (or specific model) as output. In particular, 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.

The CodeGenerator component uses the models produced by the aforementioned editors as input to offer the functionalities needed for the code generation according to the classes constituting the ELDAFramework.
The SimulatorEditor component allows to visually configure simulation parameters which are used as input by the SimulationCodeGenerator to produce a simulator program according to the ELDASim framework.
Figure 3.30 shows the components, the dependence relationships among them, and their contextualization with respect to the development process phases.

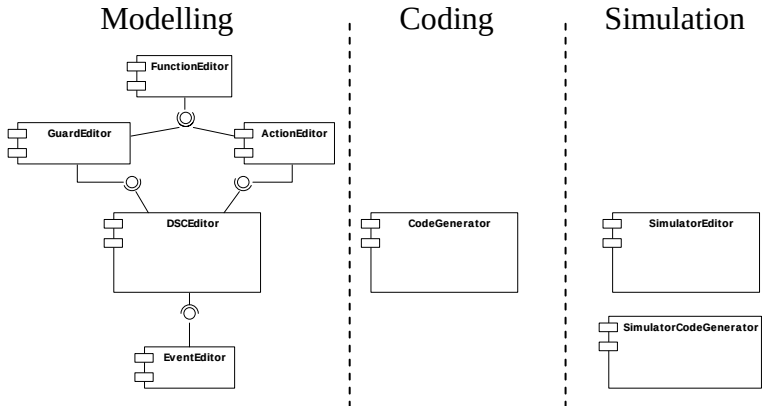



Figure 3.30: The ELDATool components.

### 3.4.2 Implementation

The ELDATool fully supports the three phases of the process: Modelling, Coding and Simulation. The architectural components described in the previous section are implemented in Java by exploiting:

- the Eclipse platform [30], 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) [55] 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) [31] which supports the modelling phase of a structural model and the automatic generation and manipulation of its Java implementation.

The editor components (see Section 3.4.1) 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 concepts of the ELDA MAS meta-model;
- EventModel, which contains the implementation of the Event concepts of the ELDA MAS meta-model;
- ActionGuardModel, which contains the implementation of the Action and Guard concepts of the ELDA MAS meta-model;
- FunctionModel, which contains the implementation of the Supporting Function concept of the ELDA MAS meta-model;
- ELDAEditor, which supplies the visual editor and the code generator components;
- SimulatorEditor and SimulatorCodeGenerator, which allow us for the visual setup of the simulation parameters and the code generation of the simulator program;

Figure 3.31 highlights and clarifies the dependence relationships among the implemented plug-ins: the ELDATool is 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.
Figure 3.32 shows the ELDATool during the visual modeling of the ADSC of an ELDA agent and Figure 3.33 shows the dialogs which allow for simulation parameters setup and simulation execution control.

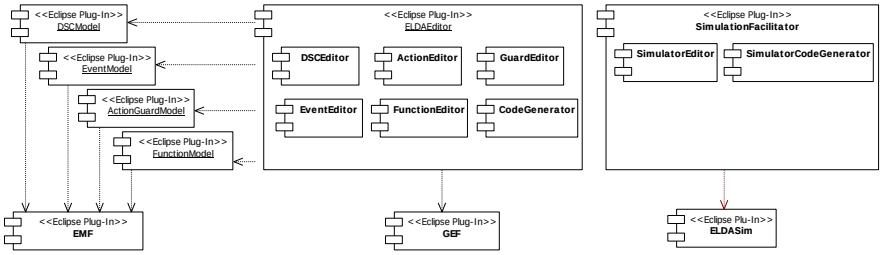



Figure 3.31: The ELDATool plug-ins.

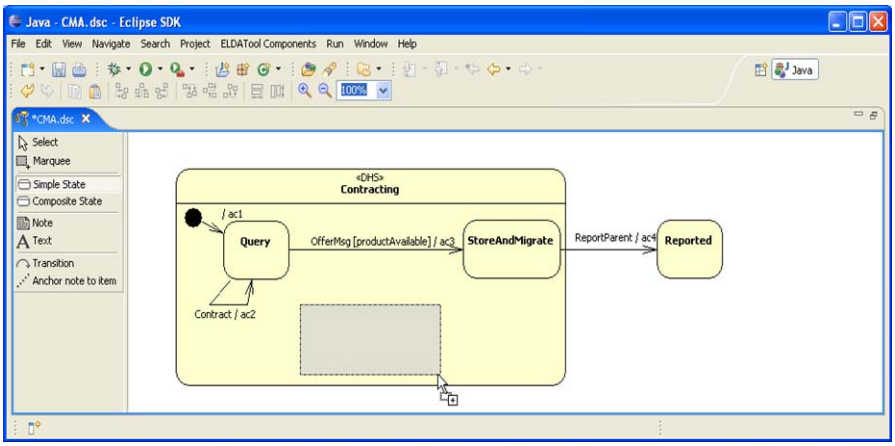
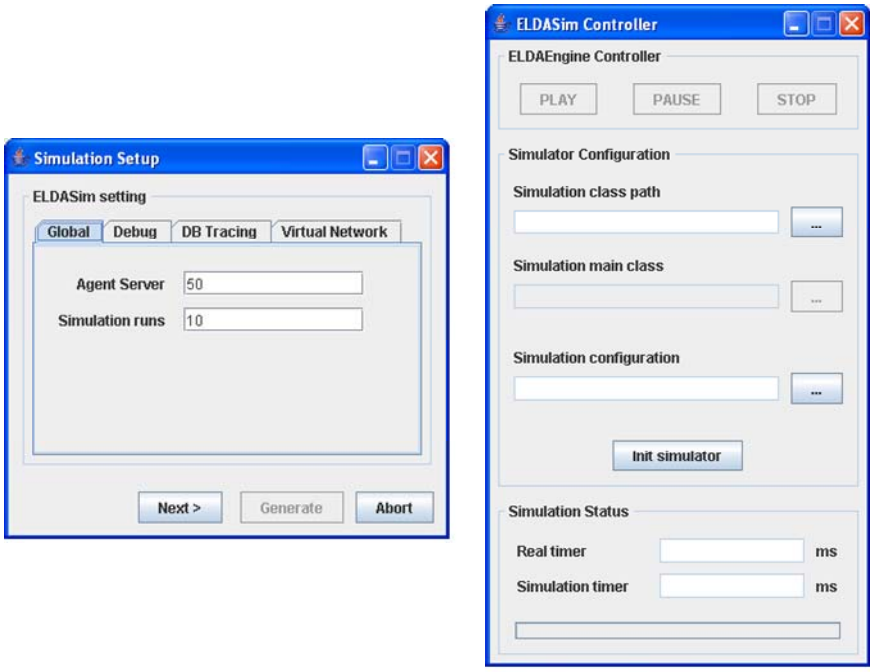



Figure 3.32: Snapshot of the ELDATool during the modeling phase.




(a) (b)

Figure 3.33: Simulation dialogs: (a) simulation setup and (b) simulation execution control.

# 4 Modeling and Validation of Distributed Architectures for Surrogate Clustering in CDNs: a case study

Content Distribution Networks (CDNs) have been introduced and extensively used in the Internet as effective solution for improving the performance of content delivery by means of coordinated content replication [14]. In particular, a CDN manages a geographically distributed set of surrogate servers, located at the network edge, that archive copies of identical content, so that users' requests can be fulfilled by the optimal surrogate servers. In conventional CDN architectures, when a user request is redirected to a surrogate server which is not able to fulfill it by providing the requested content, the surrogate server fetches the requested content from the origin server, which stores all the offered content, and delivers it to the requesting user. As the origin server is usually far way from each surrogate server, this basic scheme to deal with missing content causes a high average user perceived latency.

To overcome this issue, several cooperative caching mechanisms and architectures for CDNs have been proposed [39, 79, 105]. In this chapter, three kinds of distributed architectures (master/slave, multicast-based, and peer-to-peer) for surrogate clustering in CDNs have been modelled and analysed through ELDAMeth. In these architectures, which are an extension of the cooperative architectures proposed in [39, 43], surrogates are grouped into clusters of neighbour surrogates and cooperate to provide the requested content. In particular, a surrogate which is not able to provide the requested content checks for a surrogate of the same cluster having the content so as to forward the unfulfilled user request to it; otherwise, if no other surrogate has the content, the surrogate contacts the origin server as in the basic schema. As surrogates in the same cluster are much closer to each other than to the origin server, the average user perceived latency could be reduced. Moreover, as the cache of a cluster is larger than the cache of a single surrogate hit ratio of the cluster cache could be higher than hit ratio of a single surrogate.

## 4.1 CDN working principles

The general architecture of a CDN, shown in Figure 4.1, consists of seven components: client, replica server (or surrogate), origin server, request routing system, distribution system, accounting system and billing organization [92]. The interactions among these components, represented with numbered lines in Figure 4.1, are described as follows:

1. The origin server delegates the URI name space of the content (web objects, rich media, etc.) to be distributed and delivered by the CDN to the request routing system (or request redirection system).
2. The origin server publishes the content, which is to be distributed and delivered by the CDN through the distribution system.
3. The distribution system "intelligently" moves content to surrogates. This system also interacts with the request routing system and supports it during the surrogate selection phase triggered by client requests.
4. The client requests content from what it perceives to be the origin server. However, due to URI name space delegation, requests are actually directed to the request routing system.
5. The request routing system routes the client request to a suitable surrogate of the CDN.
6. The selected surrogate delivers the requested content to the client. In addition, the surrogate sends accounting information about delivered content to the accounting system.
7. The accounting system aggregates and distills the accounting information into statistics and the records of content detail and sends them to the origin servers and the billing organization. The billing organization uses the records of content detail to settle with each of the parties involved in the content distribution and delivery process. Statistics are also used as feedback to the request routing system and distribution system.

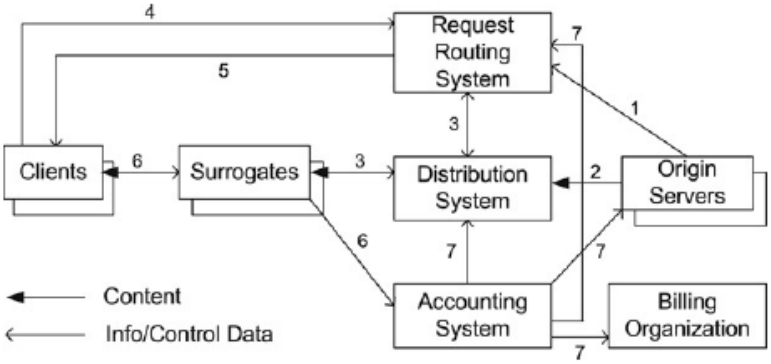



Figure 4.1: A basic CDN Architecture.

## 4.2 Distributed architectures for surrogate clustering

When a client issues a content request, the redirection system selects the most appropriate surrogate and routes the request to it. This surrogate serves the request if it has the requested content, otherwise it asks the origin server for the content and, once retrieved, sends it to the requesting client. A missing content (or miss) in the selected surrogate causes a high response time as the origin server is usually located far away from surrogates. Figure 4.2 summarizes the service dynamics of a client's content request in a stand-alone surrogate-based (SA) architecture. Moreover, surrogates usually adopt a least recently used (LRU) strategy to evict content from their cache when storage space is needed.

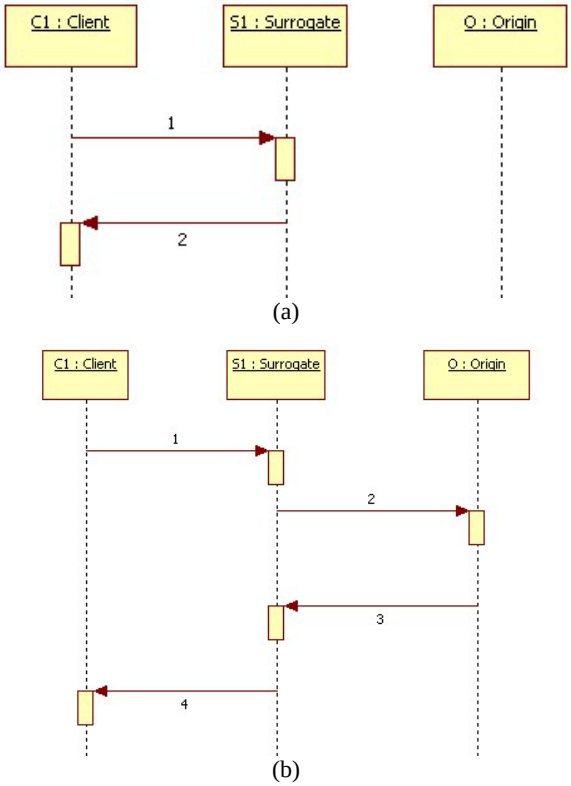



Figure 4.2: Content served by a stand-alone surrogate-based architecture. (a) the requested content is in the surrogate cache; (b) the requested content is not in the surrogate cache and must be therefore requested to the origin server.

Although the SA architecture is simple to develop and maintain, it suffers of two main drawbacks: limited dimension of the cache and high response time

when content is not present in cache and must be fetched from the origin server.
To deal with these drawbacks, surrogates can be grouped into clusters according to their proximity (e.g. neighboring surrogates belong to the same cluster). Surrogates in the same cluster (hereafter called peer surrogates or simply peers) cooperate to provide a requested content. A surrogate receiving a content request which cannot serve forwards such request to a peer. Only if none of the peers can provide the requested content, the request is forwarded to the origin server. Cooperation among peers can therefore enable:

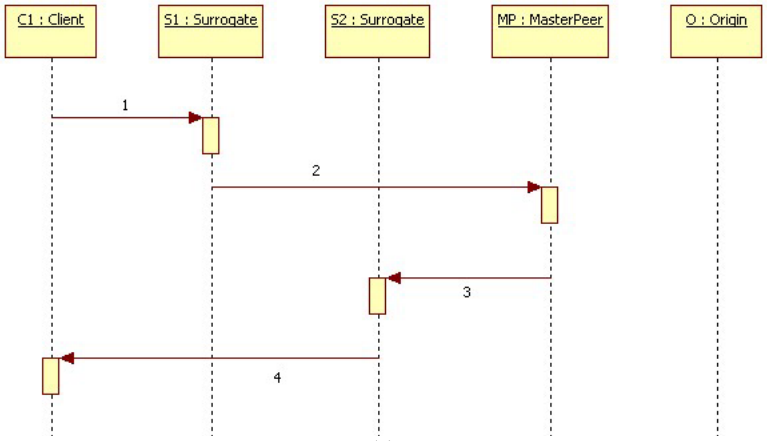
- *higher hit rate* as the available content is not only the content of a single surrogate but the total content of all peers;
- *shorter response time* as distance between peers is much shorter than the distance between a surrogate and its origin server.

However, cooperation among peers demands for the design and implementation of additional mechanisms for surrogate clustering to support cluster formation for grouping surrogates in clusters and cluster maintenance for handling join/leave of peers.
Three different kinds of distributed architectures for surrogate clustering have been designed: *master/slave*, *multicast-based* and *peer-to-peer*. In the multicast-based architecture content duplication can occur whereas in the master-slave and peer-to-peer architectures the content stored in a given peer cannot be duplicated in the other peers of the same cluster. For all the defined architectures, each peer has a content location hash table (CLHT) to maintain the content location information for content lookup. Each content has a content identifier (or CId) generated by a collision-free hash function-based algorithm (e.g. SHA-1 [77]). In addition, each surrogate has a surrogate identifier (SId) generated with the same algorithm. In the following subsections the designed distributed architectures are described.

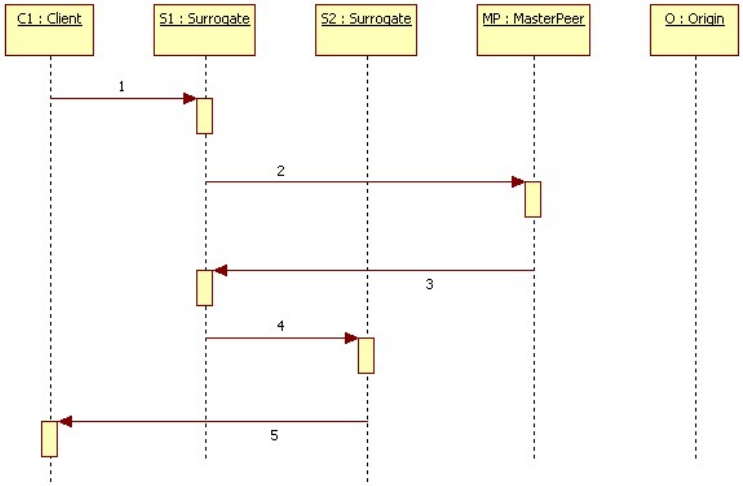
### 4.2.1 Master/Slave

In the master/slave (M/S) architecture, a master/slave approach is exploited which is based on a master peer to manage the cluster CLHT whereas the other peers only manage a CLHT of their own content. When a request arrives, a peer first looks up its CLHT and then, if it does not find the content, forwards the request to the master peer that, in turn, forwards it either to the peer (which could also be the master itself) with that content or to the origin server. It is worth noting that every time a peer chooses to evict a content, it notifies the master that consequently updates the global CLHT. In this way consistency of the cluster is guaranteed by the master even though it could become a bottleneck.
Three variants of the MS architecture (M/S_1, M/S_2, and M/S_3) have been defined. The dynamics of the scenarios related to *content found*/*content not found* in the cluster for M/S_1, M/S_2, and M/S_3 are reported in Figures 4.3a/4.4a, 4.3a/4.4b and 4.3b/4.4a, respectively. The three variants consider

the architectures which averagely involve the lowest number of exchanged messages (M/S_2), the highest number of exchanged messages (M/S_3), and a number of exchanged messages between the highest and lowest ones (M/S_1).
With reference to the two schemas for *content found* in the cluster (see Figure 4.3), the main difference is that in M/S_1 and M/S_2 the master peer forwards the request directly to the surrogate (S2) which has the content (see Figure 4.3a), whereas in M/S_3 the master peer notifies the address of S2 to the selected surrogate (S1) which, in turn, forwards the request to S2 (see Figure 4.3b).



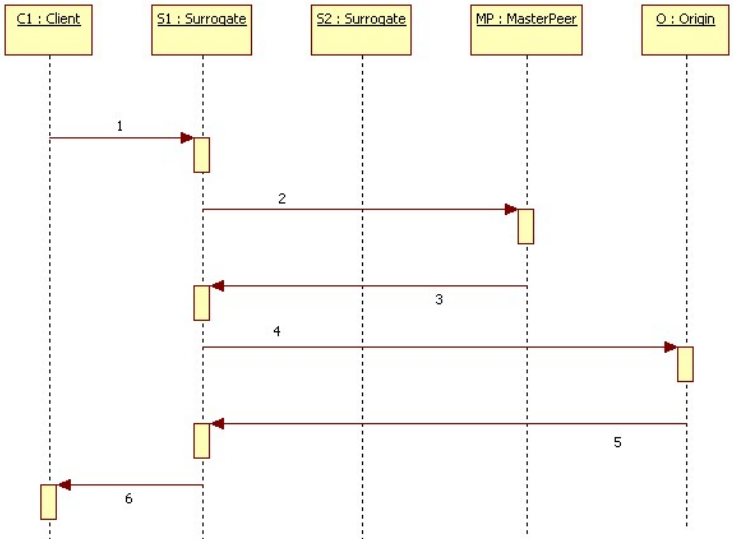

(a)



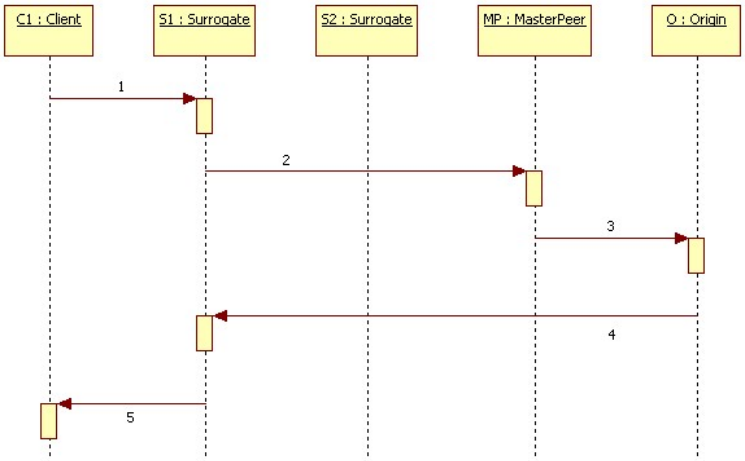

(b)

Figure 4.3: Master/slave architectures: content found in the cluster. (a) schema for M/S_1 and M/S_2; (b) schema for M/S_3.

With reference to the two schemas for *content not found* in the cluster (see Figure 4.4), the main difference is that in M/S_1 and M/S_3 the master peer replies to the selected surrogate (S1) which, in turns, downloads the content from the origin to serve the client (see Figure 4.4a); in M/S_2 the master peer contacts the origin which sends the missing content to the selected surrogate (see Figure 4.4b).




(a)




(b)

Figure 4.4: Master/slave architectures: content not found in the cluster. (a) schema for M/S_1 and M/S_3 and; (b) schema for M/S_2.

### 4.2.2 Multicast-based

In the multicast-based (MC) architecture, each peer surrogate manages a CLHT in which stores the content location information of all peer surrogates. In particular it is adopted a soft-state multicast-based communication paradigm [95] with BASE (Basically Available, Soft State, Eventual Consistency) semantics [51] for which the CLHT could be different in each peer surrogate at any given time. As shown in Figure 4.5, a missing content in the selected peer is handled as follows: if the CLHT has an entry for that content, the request is forwarded to the peer that has the requested content and will then serve the client request; otherwise, the request is forwarded to the origin server. Every update of the CLHT is multicast from the peer that updated its content to all the others without an ACID (Atomicity, Consistency, Isolation, e Durability) coordination mechanism. Each peer uses update messages to update its own CLHT; this implies that:

- according to its CLHT a peer could forward a request to another peer that may not have the requested content;
- duplicated copies of the same content could be present in a cluster.

Thus in this architecture the consistency of the CLHT is not guaranteed.

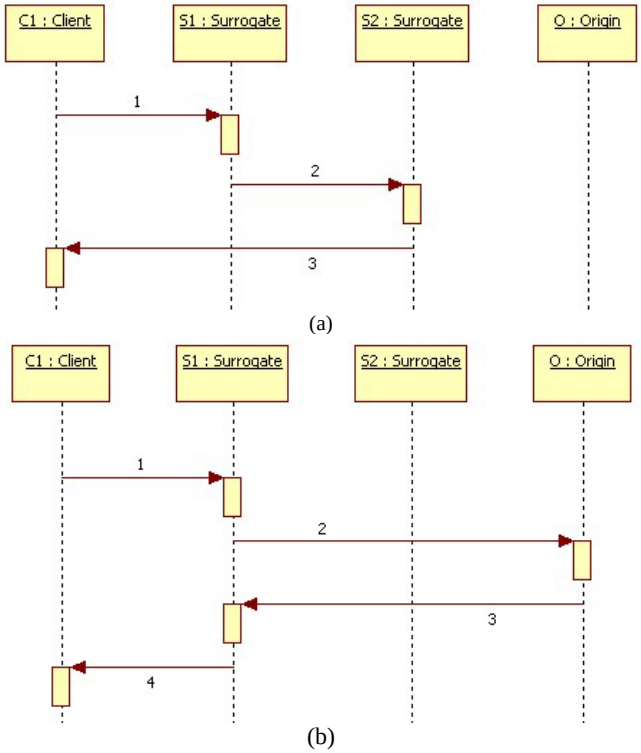



Figure 4.5: Multicast-based architecture: (a) content not found in the selected surrogate but found in the cluster; (b) content not found neither in the selected surrogate nor in the cluster.

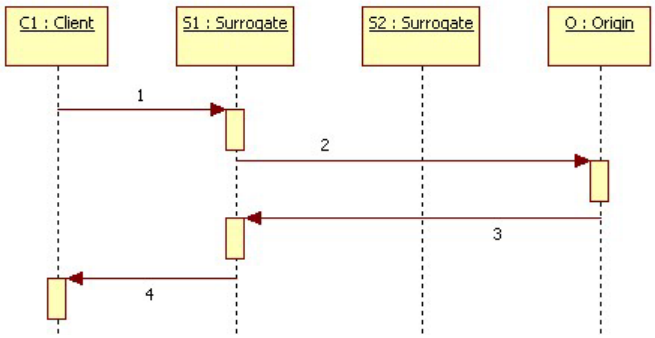
### 4.2.3 Peer-to-peer

In the peer-to-peer architecture (P2P), each peer has an SLT (Surrogate Location Table) which contains the location information of all the peers and their respective contents. In particular, for each peer surrogate an SLT has an entry formalized by the pair <SId, CZ>, where CZ (Content Zone) is the space of the identifiers of the contents potentially stored in the peer identified by SId. Such an organization implies that a given content can only be stored in the peer responsible for such content, i.e. its CId is in the CZ of the peer. According to the peer-to-peer model, a content request issued by a client is served by the selected surrogate as follows:

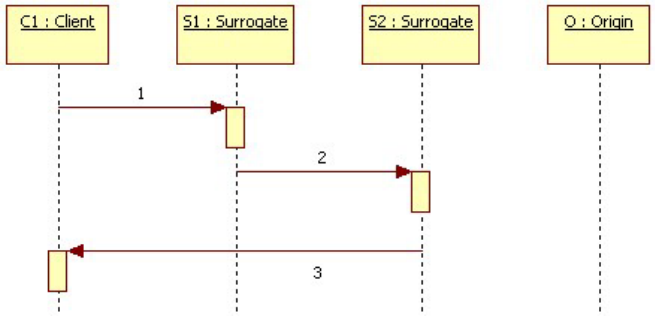
1. If the CId of the requested content belongs to its CZ, the content is looked up in the CLHT: if the content is present, it is sent to the client; otherwise, the content is retrieved from the origin, sent to the client and finally stored (see Figure 4.6a).
2. If the CId of the requested content does not belong to its CZ, the request is forwarded to the peer responsible for CId which operates as follows: if the requested content is present, it sent it to the requesting client (see Figure 4.6b); otherwise, the content is fetched from the origin before sending it to the requesting client (see Figure 4.6c).

As in the M/S architecture the P2P architecture provides consistency of the content in the cluster. Moreover it overcomes the main drawback of the M/S and MC architectures as a peer does not need to maintain content information belonging to the other peer surrogates. However this model has the following drawbacks: (i) since a given content can only be served by a given peer, a highly requested content can only be provided by a given peer leading to a service hotspot; (ii) since the mapping between content and surrogate is fixed, when content is added new CIds must be generated and mapped to given zones. This can lead to the problem of zone saturation which demands for a rearrangement of the zones in the cluster. However, if the rate of addition of new content is slow this does not introduce significant overhead.

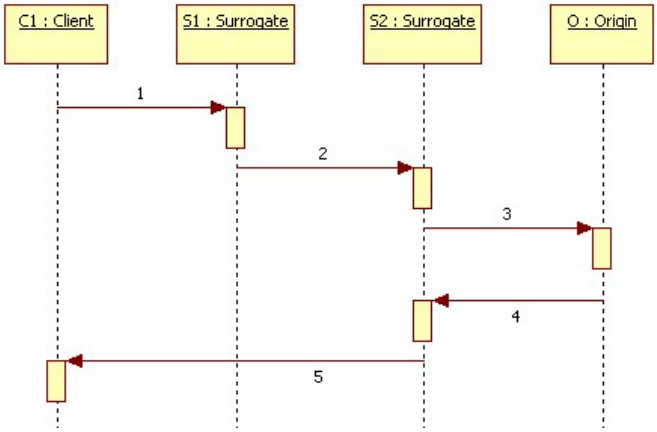
It is worth noting that the other two architectures can also suffer of the first drawback whereas the second drawback for such architectures does not subsist as content can be requested and stored by any peer.




(a)




(b)




(c)

Figure 4.6: Peer-to-peer architecture: (a) content not found belonging to the selected surrogate CZ; (b) content not found not belonging to the selected surrogate CZ but present in the cluster; (c) content not found not belonging to the selected surrogate CZ and not present in the cluster.

## 4.3 ELDA-based modelling

This section shows how the identified actors in a typical CDN scenario have been agentified and then modelled according to the ELDA model. As shown in Figure 4.7, Client and Origin servers have been agentified through a ClientAgent and an OriginAgent, respectively; moreover, in order to enable the concurrent processing of received content requests, each Surrogate server has been agentified through two agents: a SurrogateManagerAgent, which spawns a worker agent for each received request and a SurrogateAgent which actually fulfils the request.

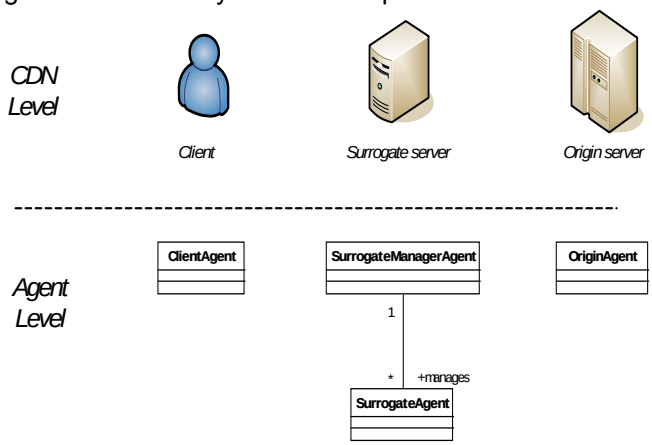



Figure 4.7: CDN actors and identified agent types.

As shows the class diagram of the defined agent types for the aforementioned architectures (see Figure 4.8), the ClientAgent and OriginAgent types are shared among them whereas SurrogateManagerAgent and SurrogateAgent have been differently designed depending on the specific architecture for surrogate clustering.

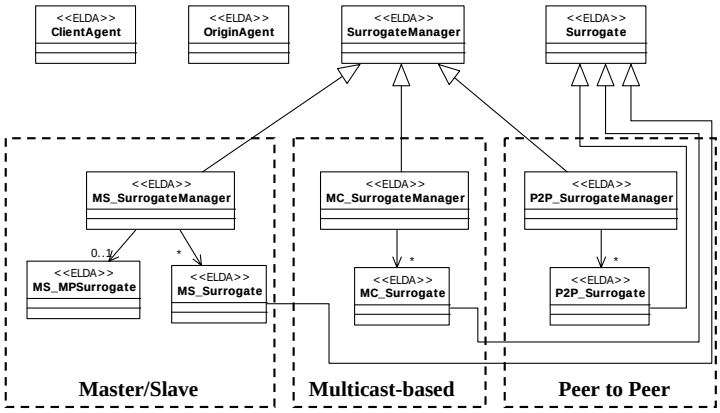



Figure 4.8: Class diagram of the defined agent types.

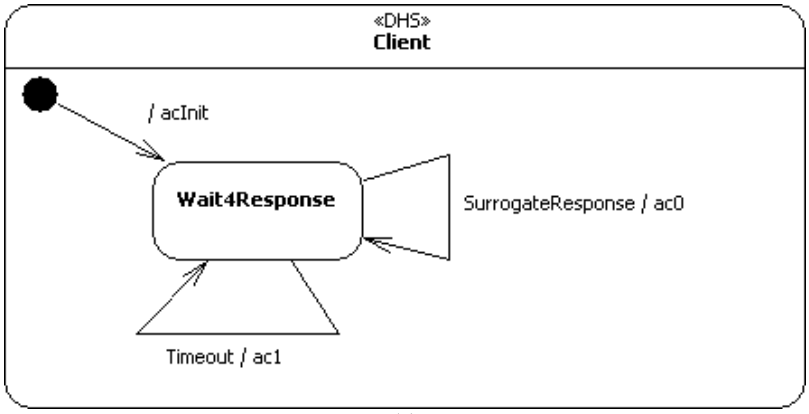
However, for sake of brevity, only the ELDA-based modelling of the agents of the M/S_1 variant of the master/slave architecture (see Section 4.2.1) will be explained in details. Events referred in the DSC diagrams are summarized in table 4.1.

Table 4.1: Events exchanged among agents of the M/S_1 variant of the master/slave architecture.

| EVENT | DESCRIPTION | SOURCE | TARGET |
|---|---|---|---|
| CLIENTREQUEST | It contains a content request. | Client | Surrogate |
| SURROGATERESPONSE | It contains the requested content. | Surrogate | Client |
| SURROGATEREQUEST | It contains a content request. | Surrogate | Origin |
| ORIGINRESPONSE | It contains the requested content. | Origin | Surrogate |
| SURROGATEMASTERPEERREQUEST | It contains the request of availability of a content within the cluster. | Surrogate | MasterPeer |
| SURROGATEMASTERPEERRESPONSE | It informs the Surrogate that a content is available within the cluster. | MasterPeer | Surrogate |
| SURROGATEINFORMCONTENT | It informs the MasterPeer that new contents have been downloaded from the OriginServer | Surrogate | MasterPeer |
| SENDTOCLIENT, WORKCOMPLETED, CONTENTNOTFOUND, UPDATECLHT, FORWARDINGNOTFOUND | Internal events. | Surrogate | Surrogate |

***ClientAgent***

A ClientAgent (see its behavior in Figure 4.9) requests contents to the CDN according to an exponential probability density function which is used to set-up the timeout value of the timer driving the agent activity: as soon as a timer expires, a TIMEOUT event is delivered to the ClientAgent which sends the content request and creates a next timer (see action ac1); instead, as soon as a previously requested content is delivered to the ClientAgent through the SURROGATERESPONSE event, performance indices are updated (see action ac0). It is worthy noting that target surrogates are chosen by ClientAgents according to a uniform distribution whereas the requested content is chosen according to a contents popularity distribution (CPD) which can be either uniform or Zipf [102].




(a)

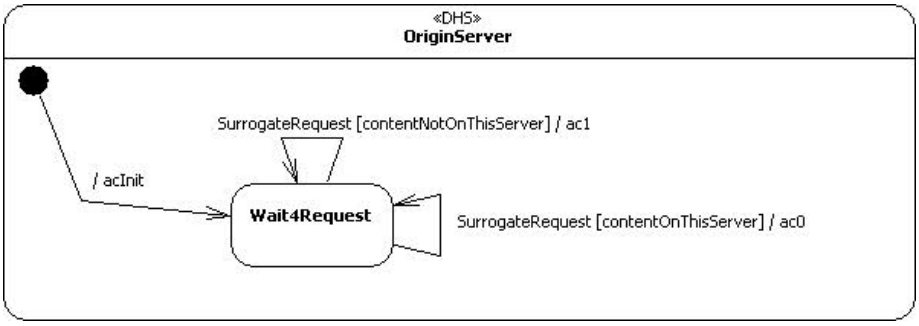
```
acInit:
  tableRichieste = new Hashtable<Integer,Long>();
  tableRichiesteContent = new Hashtable<Integer,String>();
  genTimout4Request();
ac0:
  SurrogateResponse surrogateResponse = (SurrogateResponse)e;
  String contentResponse = (String)surrogateResponse.getData();
  processContent(contentResponse);
ac1:
  int idRichiesta = pkgStatic.Variables.clientIdRequest++;
  int indiceContenuto = -1;
  if(distribution == 0){
    indiceContenuto = (int)(rnd.uniform(0, contents.length));
  }else if(distribution == 1){
    indiceContenuto = zipfDistribution.getElement();
  }
  int indiceSurrogato = (int)(rnd.uniform(0, surrogati.length));
  String request = contents[indiceContenuto];
  ClientRequest clientRequest = new ClientRequest(self(),
                                  surrogati[indiceSurrogato],request,idRichiesta);
  generate(new ELDAEventMSGRequest(self(),clientRequest));
  genTimout4Request() ;
genTimout4Request:
  long timeout = getRandomTimeout();
  generate(new ELDAEventCreateTimer(self(),timeout,new Timeout(self())));
```

(b)

Figure 4.9: ClientAgent behavior: (a) DSC and (b) related actions.

***OriginAgent***

OriginAgent (see its behavior in Figure 4.10) has to respond to content requests performed by agents representing Surrogate servers: as soon as a SURROGATEREQUEST event is delivered to the OriginAgent, it checks if the requested content is available and in this case sends back such content (see action ac0); otherwise, the SurrogateAgent is notified about the request failure (see action ac1).




(a)

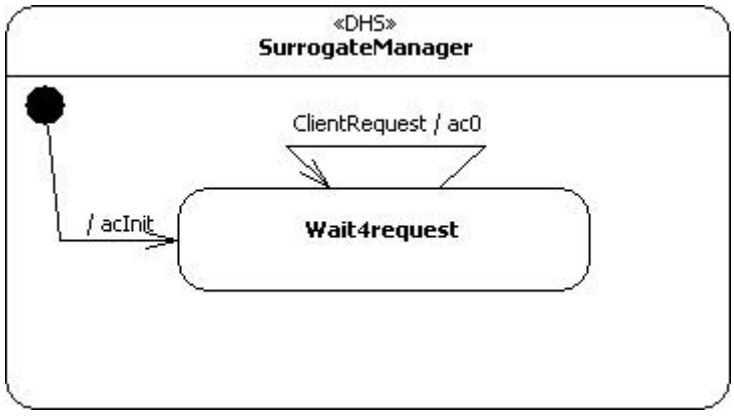
```
acinit:
  contentsInitiating();
ac0:
  SurrogateRequest surrogateRequest = (SurrogateRequest)e;
  String contentRequest = (String)surrogateRequest.getData();
  String contentValue = (String)tableContent.get(contentRequest);
  OriginResponse originResponse = new OriginResponse(self(),
                                                     surrogateRequest.getSource(),
                                                     contentValue,
                                                     contentRequest,
                                                     true,
                                                     surrogateRequest.getIdRequest());
  generate( new ELDAEventMSGRequest(self(),originResponse));
ac1:
  SurrogateRequest surrogateRequest = (SurrogateRequest)e;
  String contentRequest = (String)surrogateRequest.getData();
  OriginResponse originResponse = new OriginResponse(self(),
                                                     surrogateRequest.getSource(),
                                                     "Content not found!",
                                                     contentRequest,
                                                     false,
                                                     surrogateRequest.getIdRequest());
generate(new ELDAEventMSGRequest(self(),originResponse));
```

(b)

Figure 4.10: The OriginAgent behavor: (a) DSC and (b) the related actions.

***SurrogateManager***

A SurrogateManager agent (see its behavior in Figure 4.11) aims to manage the surrogate server in which it runs. Among the SurrogateManagers that belong to a cluster, only one of them has to create the MasterPeer agent which holds and maintains the cluster CLHT (see action acInit). Once initialized, a SurrogateManager waits for content requests sent by client or surrogate agents: as soon as a CLIENTREQUEST event is delivered, a SurrogateManager (see action ac0) creates a Surrogate agent which handles the received request.




(a)

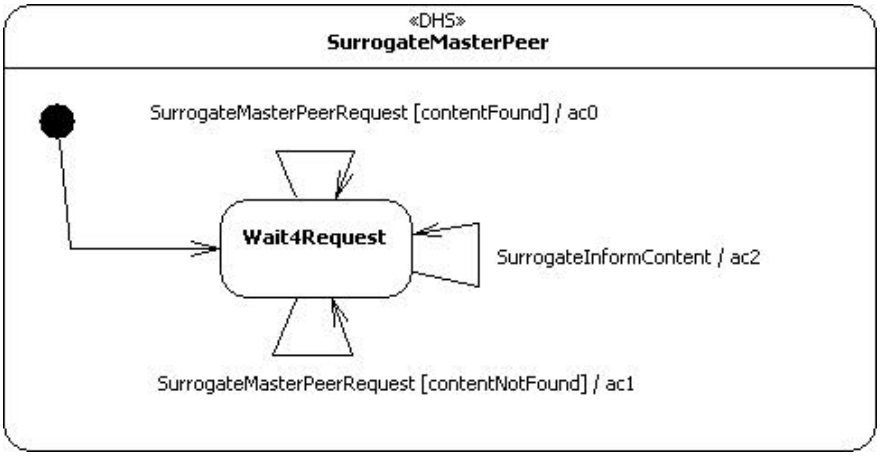
```
acInit:
  if(detainsMasterPeer){
     ELDAEventCreate createEvent = new ELDAEventCreate(self(),"SurrogateMPActiveState"
                                                          params,masterPeer);
     generate(createEvent);
  }
ac0:
  ClientRequest clientRequest = (ClientRequest)e;
  Object[] params = {originServer,self(),masterPeer,clientRequest.getSource(),
                   clhtContenuti, String)clientRequest.getData(),clientRequest.getIdRequest()
                     };
  ELDAEventCreate createEvent = new ELDAEventCreate(self(),"MS1_SurrogateActiveState",
                                                       params,new ELDAId("Surrogate"+(agentID++)));
  generate(createEvent);
```

(b)

Figure 4.11: SurrogateManager: DSC (a) and actions (b).

***Master Peer***

The MasterPeer agent (see its behavior in Figure 4.12) has to reply to content requests sent by other Surrogates through the SURROGATEMASTERPEERREQUEST event: if the requested content is available within the cluster (that is the contentFound guard holds), the MasterPeer agent forwards the client request to the target SurrogateManager (see action ac0); otherwise (that is the contentNotFound guard holds) the MasterPeer agent replies to the asking Surrogate (see action ac1) which directly downloads the requested content from the OriginServer. Moreover, a MasterPeer agent has to maintain the cluster CLHT: as soon as a SURROGATEINFORMCONTENT event is delivered to it (because new contents have been downloaded from the OriginServer into the cluster) it performs the cluster CLHT updating (see action ac2) in which new content entries are added and, eventually, obsolete content entries are removed from the cluster CLHT.




(a)

```
ac0:
  SurrogateMasterPeerRequest request = (SurrogateMasterPeerRequest)e;
  String contentRequest = (String)request.getData();
  ELDAId entry = clht.getEntry(contentRequest);
  SurrogateMasterPeerResponse response = new SurrogateMasterPeerResponse(self(),
                                                        request.getSource(),
                                                        entry,
                                                        entry!=null);
  generate( new ELDAEventMSGRequest(self(),response) );
  ClientRequest clRequest = new ForwardedRequest(request.getClient(),entry,
                                                 contentRequest,request.getIdRequest());
  generate( new ELDAEventMSGRequest(self(),clRequest));
ac1:
  SurrogateMasterPeerRequest request = (SurrogateMasterPeerRequest)e;
  SurrogateMasterPeerResponse response = new SurrogateMasterPeerResponse(self(),
                                                                request.getSource(),
                                                                null, false);
  generate( new ELDAEventMSGRequest(self(),response));
ac2:
  SurrogateInformContent inform = (SurrogateInformContent)e;
  String content = (String)inform.getData();
  if(inform.getContentEvicted()!=null)
     clht.removeEntry(inform.getContentEvicted());
  clht.putEntry(content, inform.getSource());
```

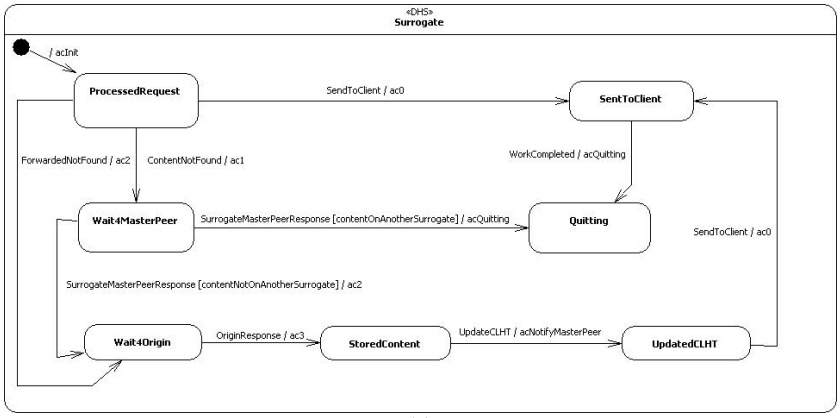
(b)

Figure 4.12: MasterPeer: DSC (a) and actions (b).

***Surrogate***

A Surrogate agent (see its behavior in Figure 4.13) which receives a client request checks if the content is locally available (see action acInit): in this case, the content is sent to the ClientAgent (see action ac0) and then the Surrogate agent quits itself (see action acQuitting); otherwise, the content request is forwarded to the MasterPeer agent of the cluster (see action ac1).
As soon as the MasterPeer agent sends back its response though the SURROGATEMASTERPEERRESPONSE event, the Surrogate agent verifies if the requested content is on another surrogate (that is the contentOnAnotherSurrogate guard holds) and then quits itself (see action acQuitting); otherwise (that is the contentNotOnAnotherSurrogate guard holds) the Surrogate agent: (i) downloads the requested content from the OriginServer (see action ac2); (ii) stores locally the content sent back by the

OriginServer through the ORIGINRESPONSE event (see action ac3); (iii) notifies the MasterPeer agent about CLHT updating (see action acNotifyMasterPeer); (iv) sends the stored content to the ClientAgent (see action ac0); and finally quits itself (see action acQuitting).




(a)

```
acInit:
  if(clht.containsContent(content)){
    generate(new SendToClient(self()));
  }else if(e instanceof ForwardedRequest){
    generate(new ForwardedNotFound(self()));
  }else generate(new ContentNotFound(self()));
ac0:
  String value = (String)clht.getContentValue(content);
  SurrogateResponse surrogateResponse = new SurrogateResponse(self(),
                                                client,value,idRequest);
  generate(new ELDAEventMSGRequest(self(), surrogateResponse ));
  generate(new WorkCompleted(self()));
ac1:
  SurrogateMasterPeerRequest smpRequest = new SurrogateMasterPeerRequest(self(),
                                          masterPeer,content,
                                          client, idRequest);
   generate(new ELDAEventMSGRequest(self(),smpRequest));
ac2:
  SurrogateRequest surrogateRequest = new SurrogateRequest(self(),originServer,
                                                    content,idRequest);
  generate(new ELDAEventMSGRequest(self(), surrogateRequest ));
ac3:
  OriginResponse originResponse = (OriginResponse)e;
  if(originResponse.getFound()){
       String contentValue = (String)originResponse.getData();
        clht.putContent(content,contentValue);
   }
   generate(new UpdateCLHT(self()));
acNotifyMasterPeer:
  String contentEvicted = (String)clht.getLastContentEvicted();
  SurrogateInformContent inform = new SurrogateInformContent(manager,
                                          masterPeer,content,
                                          contentEvicted);
  generate( new ELDAEventMSGRequest(self(), inform));
  generate(new SendToClient(self()));
acquitting:
  generate(new ELDAEventQuitRequest(self()));
```

(b)

Figure 4.13: Surrogate: DSC (a) and actions (b).

## 4.4 Performance Evaluation

### 4.4.1 Simulation parameters

The simulation parameters are presented in Table 4.2. $C_{\%}$ is varied from 1% to $1/N_S$ with a step of 1%. $N_S$ is set to {2, 3, 5, 10} to consider small and large surrogate clusters. The average latency times among architecture components ($T_{CS}$, $T_{SS}$, and $T_{SO}$) are set according to the following model [49]:

$$\delta_i = K_f\delta_m + N(K_v\delta_m, \sqrt{K_v\delta_m}) \qquad \text{(Eq. 4.1)}$$

$$K_f + K_v = 1 \quad K_f, K_v \geq 0 \qquad \text{(Eq. 4.2)}$$

where $\delta_m$ is the mean delay and $\delta_i$ is the instantaneous delay for a given message. $\delta_i$ is the sum of a fixed part and a variable part. Eq. 4.2 guarantees that the mean of $\delta_i$ is equal to $\delta_m$. The variable part of $\delta_i$ is generated by a normal random variable whose mean and variance are set to $K_v\delta_m$. The distribution of the normal variable is truncated to $-K_f\delta_m$ in order to assure that $\delta_i$ cannot assume negative values. To limit the delay variability $K_f$ is set to 0.7. As clients are very close to surrogates, surrogates of the same cluster are close to each other, and the origin is usually far away from surrogates and clients, the following relationship among the average latency times are established: $T_{SO}$=3* $T_{SS}$=9* $T_{CS}$. In the simulation runs average latency times are set as follows: $T_{SO}$=90, $T_{SS}$=30, $T_{CS}$=10.

The eviction policy (EP) can be of the following types:

- *Random*. The object to be evicted is randomly chosen.
- *Last access*. The evicted object is the one that has not been requested for longest time.
- *Rank*. The evicted object is the one less requested.

Client requests are issued according to an exponential probability density function with $\lambda_C$ average rate (ranging from 0.1 to 0.01 requests per time unit). Objects are requested by clients according to a content popularity distribution (CPD) which can be uniform (i.e. all the $N_O$ objects have the same popularity) or Zipf (i.e. the $N_O$ objects are requested considering the object popularity distributed according to a Zipf). In particular, popularity of most popular and less popular objects is defined according to a variant of the algorithm proposed in [102] which has been developed for static Web objects.

Table 4.2: Simulation parameters.

| PARAMETER | DESCRIPTION |
|---|---|
| $N_O$ | The number of objects which are contained in the origin server |
| $C_{\%}$ | The percentage of objects that are stored in a surrogate with respect to the objects stored in the origin server |
| $N_S$ | The number of surrogates in the cluster |
| $T_{CS}$ | The average latency time between clients and surrogates of the same cluster |
| $T_{SS}$ | The average latency time among surrogates of the same cluster |
| $T_{SO}$ | The average latency time between surrogates and origin server |
| EP | The type of policy for the eviction of content in surrogates |
| $\lambda_C$ | Average rate of client requests |
| CPD | The type of distribution of the content popularity |

## 4.4.2 Simulation results

Simulation results are obtained with reference to the simulated CDN architecture shown in Figure 4.14. When a client request is generated it is forwarded to a surrogate randomly selected to simulate the request redirection system (RS).

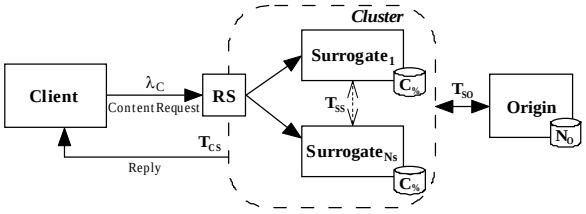



Figure 4.14: Reference simulated CDN architecture.

The simulation results analyzed in the following are obtained with reference to the following architectures (see Section 4.1): SA, M/S_2, MC, MC_PS, and P2P. In particular, as the M/S architectures have almost equal performances, M/S_2 has been selected as representative M/S architecture. Moreover, MC_PS refers to an implementation of the multicasting based on the Publish/Subscribe model whereas MC is based on message passing. Figure 4.15 shows the AUPL and CHR performance indices obtained by setting NS=2, CPD=uniform, and EP=Rank. The distributed architectures outperform the SA architecture for both performance indices. In particular, with reference to the AUPL, an actual performance improvement can be obtained with C%>=30. The CHR of the distributed architectures is almost the same whereas the AUPL for C%<30 is better for the multicast-based architectures.
By increasing NS the AUPL and CHR of the SA architecture does not change (see Figure 4.16) whereas the AUPL and CHR really improve for all the distributed architectures (as an example, see Figure 4.17 for the M/S_2

architecture). Moreover performance differences cannot be observed by changing EP.

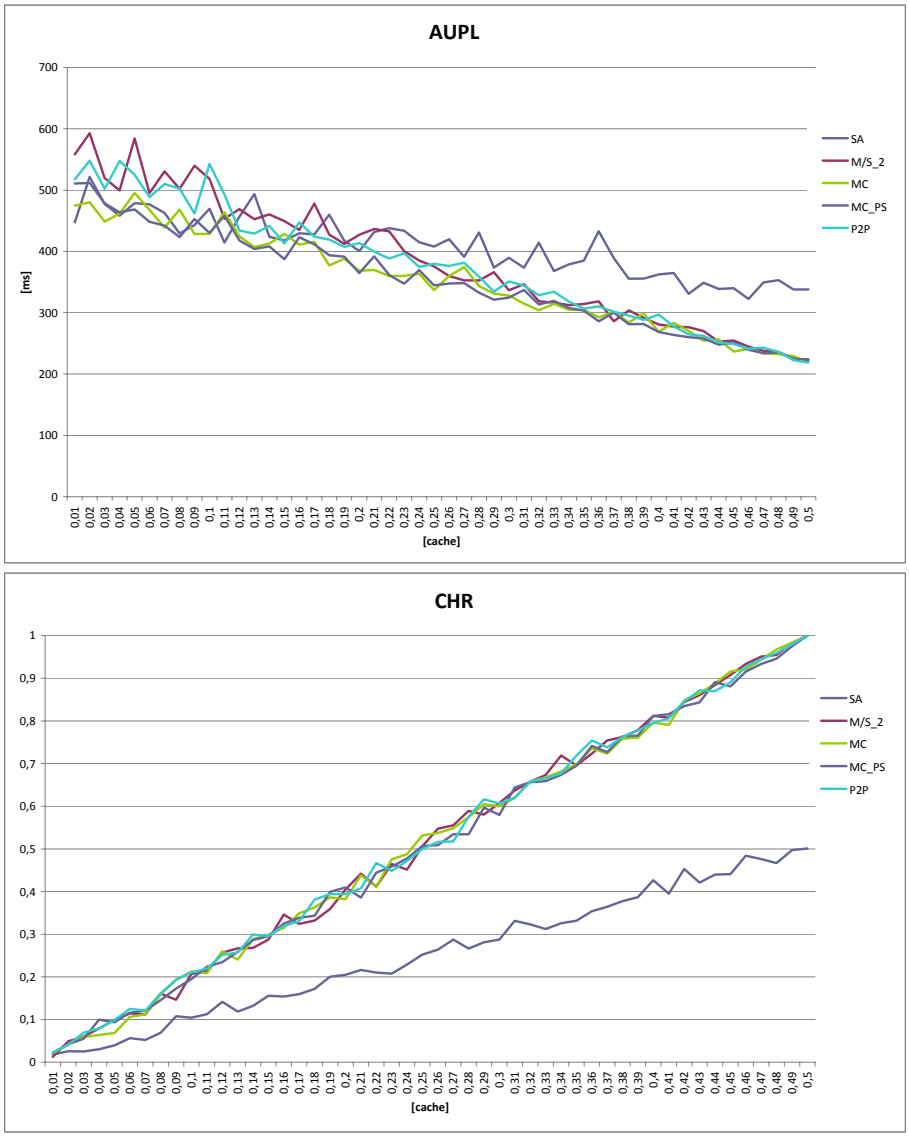



Figure 4.15: Performance indices for two surrogates, uniform content request distribution, and rank eviction policy.

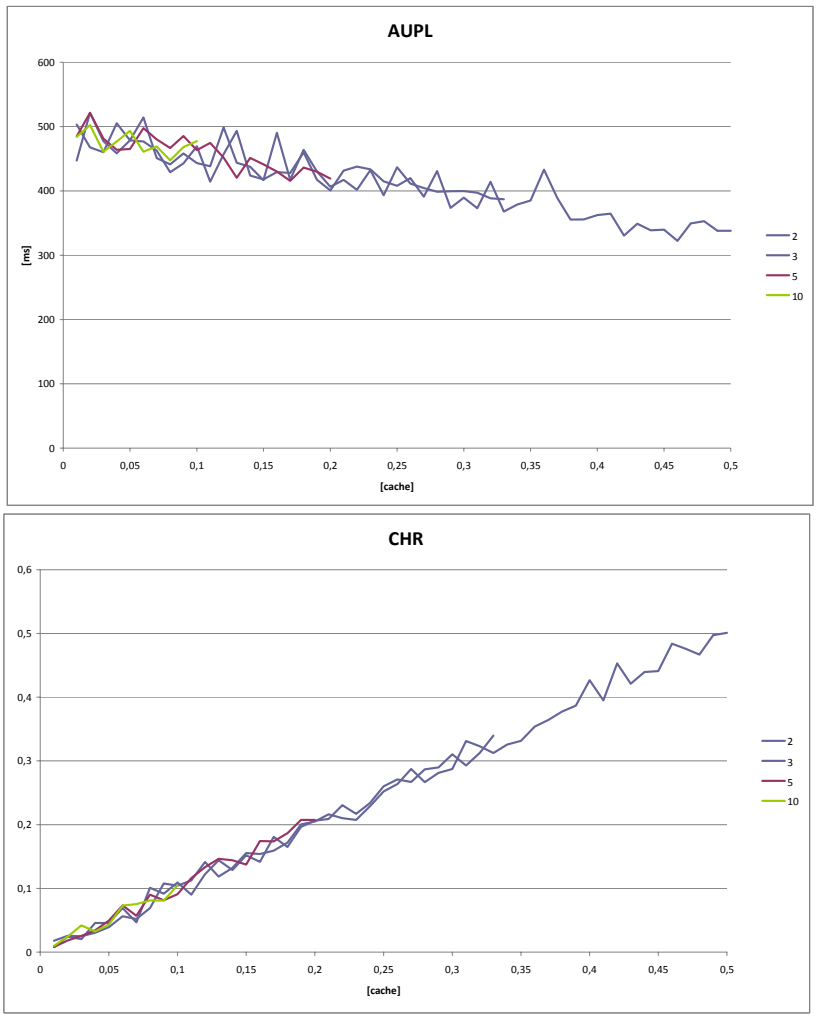
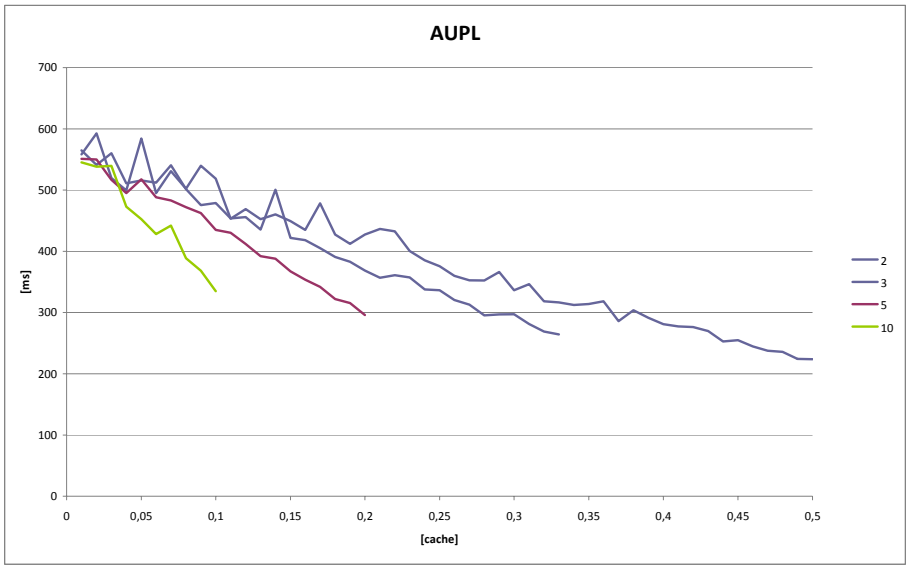



Figure 4.16: Performance indices for the SA architecture by varying the number of surrogates and with uniform content request distribution and Rank eviction policy.

Figure 4.18 shows the AUPL and CHR obtained by setting NS=2, CPD=Zipf, and EP=Rank. The distributed architectures outperform the SA architecture for both performance indices. In particular, with reference to the AUPL, an actual performance improvement can be obtained with C%>=25. The CHR of the distributed architectures is almost the same whereas the AUPL for C%<30 is better for the multicast-based architectures as happed for CPD=uniform. In this case performance differences can be observed by changing EP for all the architectures (see Figure 4.19 for the M/S_2 architecture). The best EP is the Rank whereas the worst one is the Last

access. By increasing NS both AUPL and CHR improve (as an example, see Figure 4.20 for the M/S_2 architecture) for the distributed architectures whereas they do not change for the SA architecture as observed in case of CPD=uniform.



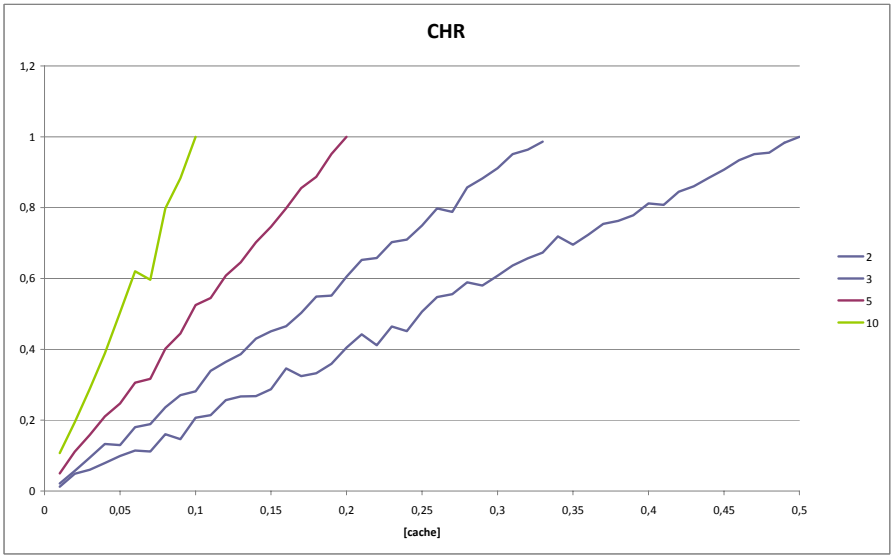




Figure 4.17: Performance indices for the M/S_2 architecture by varying the number of surrogates and with uniform content request distribution and Rank eviction policy.

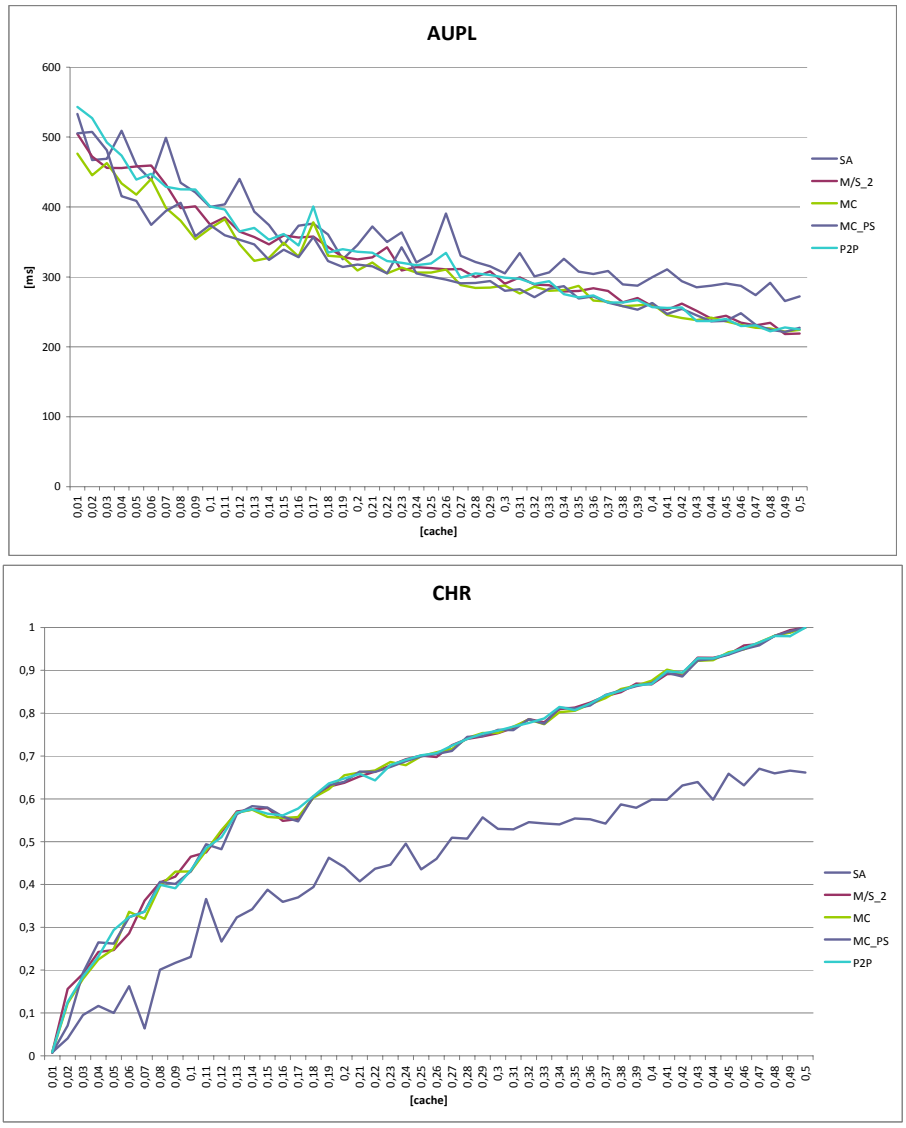



Figure 4.18: Performance indices for two surrogates, Zipf content request distribution, and Rank eviction policy.

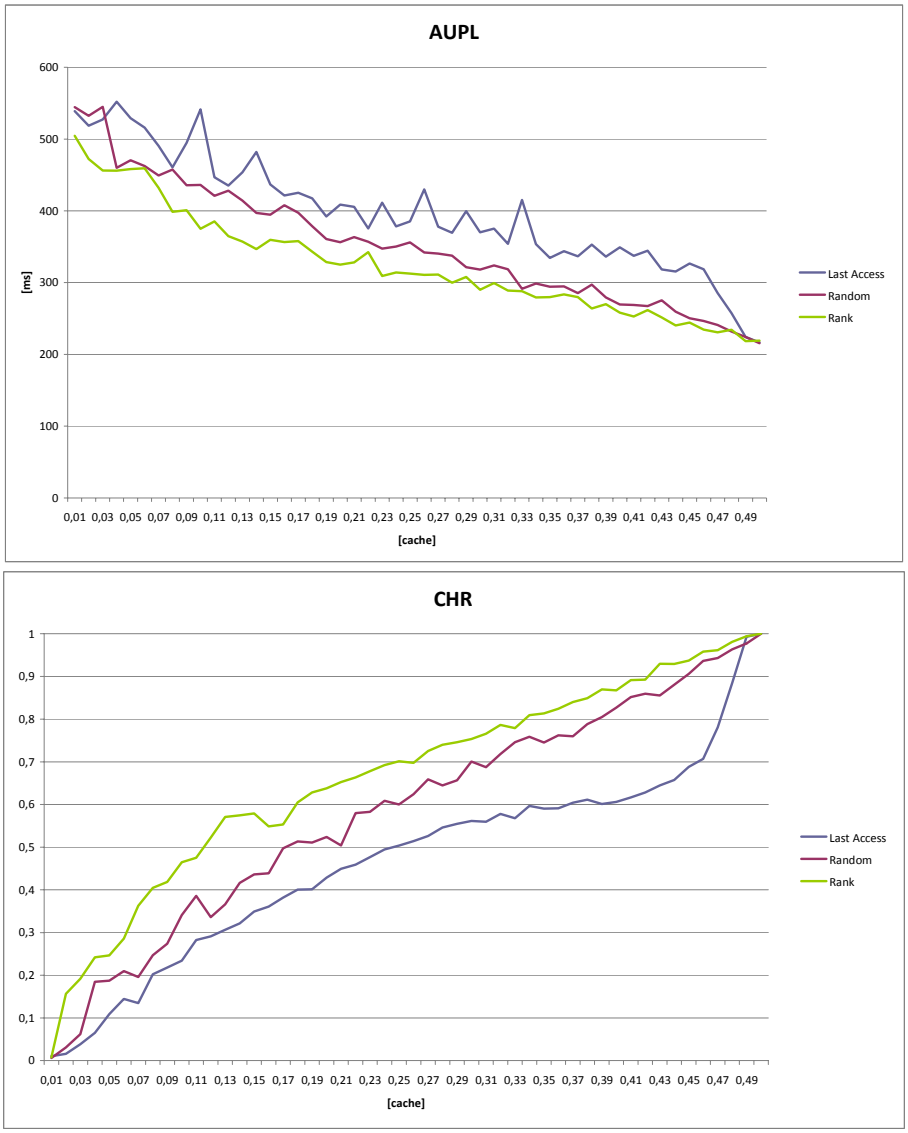



Figure 4.19: Performance indices for the M/S_2 architecture by varying the eviction policies and with Zipf content request distribution.

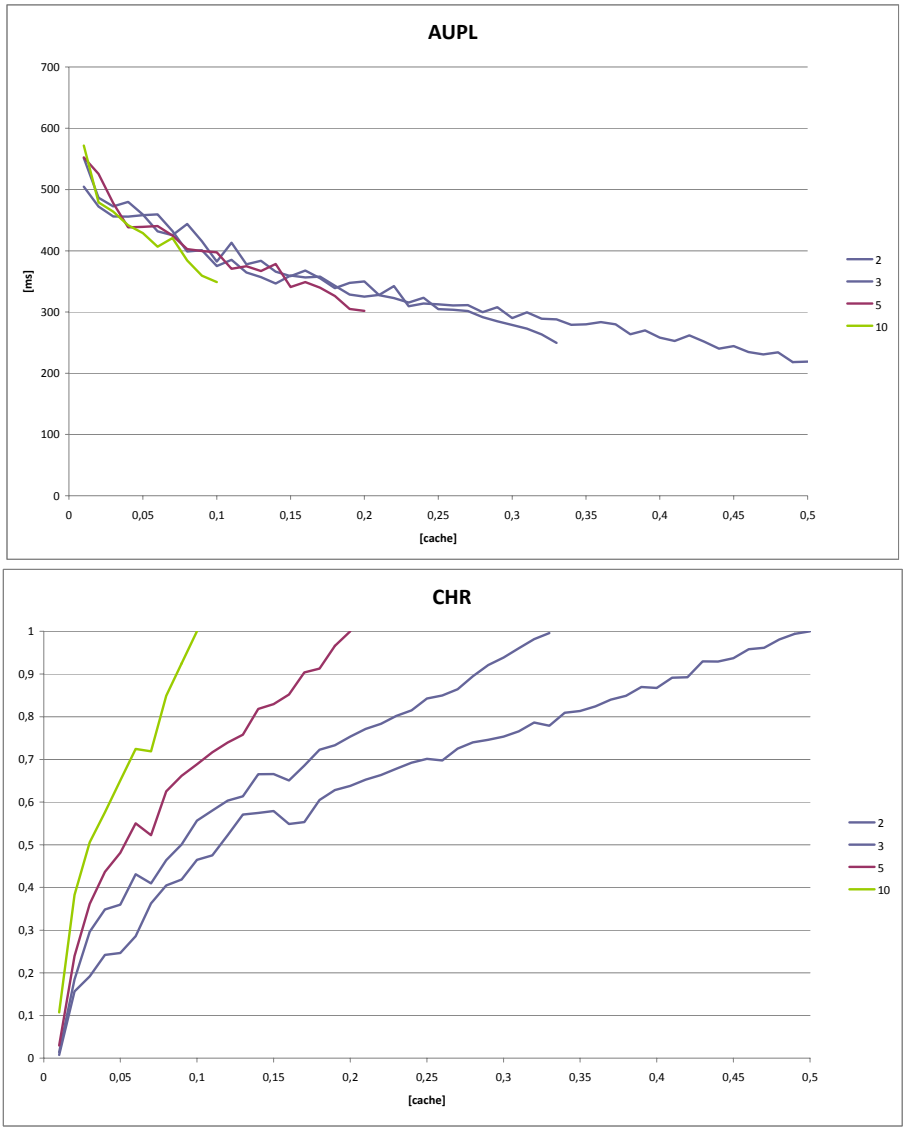



Figure 4.20: Performance indices for the M/S_2 architecture by varying the number of surrogates and with Zipf content request distribution and Rank eviction policy.

# 5 A Process for Agent Specification, Simulation and Implementation

PASSIM (Process for Agent Specification, Simulation and IMplementation) is an agent-oriented software development process that uses simulation for validating the requirements of the agent system under-development [24]. The creation of the PASSIM process was carried out through the composition of parts coming from two existing methodologies: PASSI (Process for Agent Societies Specification and Implementation) [23] and ELDAMeth. The composition of this new process can be regarded as an experiment of Situational Method Engineering (SME) [60] which is currently supported by several approaches in the literature [13, 25, 41, 61, 94]. In particular, PASSIM was created according to a process-driven approach [25, 41] which involves:

*(i)* **The choice or the definition of a software development life-cycle suitable for the specific problem and for the specific application domain.** An iterative-incremental life-cycle was chosen which is partly also derived from the Royce's final waterfall model [100] and specifically introduces the simulation phase to validate the system design before coding. In particular, the chosen life-cycle is articulated into five phases (see Figure 5.1): (1) Requirements Specification, (2) Design, (3) Prototyping, (4) Coding, and (5) Deployment. After the Prototyping phase, the designers can either proceed with the remaining part of the process, if they want to implement the software final release, or use the results of the Prototyping to feedback the *Design* phase and/or the *Requirements Specification* phase.

*(ii)* **The selection of suitable method fragments for carrying out each phase of the chosen software development life-cycle. Method fragments can be both derived from already existing methodologies or ad-hoc defined ones.** Table 5.1 reports the method fragments which were selected from both the PASSI methodology and the proposed methodology for carrying out each phase of the chosen software development life-cycle of Figure 1. For

each method fragment the Table shows the related activities and delivered work products. The selection of these fragments was easily performed since all the method fragments of the two exploited methodologies were available and ready-to-use. The so obtained software development process (PASSIM) consists of five phases carried out by six different method fragments.

*(iii)* **The adaptation of method fragments in order to allow their integration in the new methodology.** The *Prototyping* method fragment has been modified to take as input the work products produced by the *Agent Implementation* method fragment, selected from PASSI. In particular, the modified version of the method fragment translates the structural and dynamic diagrams produced by the *Agent Implementation* fragment into an agent system model based on ELDA MAS meta-model (see Section 3.1.2).

The phases of PASSIM carried out by the method fragments selected from PASSI are fully supported by the PASSI Toolkit (PTK), developed as a Rational Rose plug-in, whereas the *Prototyping phase* is supported by the ELDATool (see Section 3.4).

In the following subsections each phase of PASSIM is described with reference to the method fragments selected for carrying it out.

## *5.1 PASSIM*

### 5.1.1 Requirements Specification

The Requirements Specification phase is carried out by the System Requirements method fragment selected from PASSI that produces a model of system requirements in terms of agency and purpose. This method fragment is composed by four atomic fragments: Domain (Requirements) Description, Agents Identification, Roles Identification and Tasks Specification.

The Domain Description produces a use-case diagram that represents actors and use-cases (a functional description of the system) identified for the system using a hierarchical decomposition if its is required by the problem complexity. In the Agents Identification, agents are identified by assigning responsibility to each agent for a part of the functionalities of the whole system; this fragment produces a use-case diagram, called Agents Identification diagram (AId). In particular, the designer clusters some of the use cases within a package and gives it the name of the agent that will be responsible for accomplishing the specific functionalities of the clustered use cases. Once all the use cases have been assigned to the identified agents, the designer can define scenarios in which the agents will be involved (Roles Identification). Such scenarios are modeled through a set of UML sequence diagrams which show that each agent may be involved in several different activities and may appear more than once in each scenario playing different

roles. Finally, in the Tasks Specification, the tasks of each agent are specified through UML activity diagrams.

## 5.1.2 Design

The Design phase is carried out by two (composed) method fragments extracted from PASSI: the Agent Society and the Agent Implementation.

### 5.1.2.1 The Agent Society Fragment

The Agent Society composed method fragment includes four atomic method fragments: Domain Ontology Description, Communication Ontology Description, Roles Description, and Protocols Description.

In the Domain Ontology Description the design of the domain ontology is performed by means of a class diagram (DOD diagram) that describes the ontology in terms of concepts (categories, entities of the domain), predicates (assertions on properties of concepts) and actions (performed in the domain). This diagram can also be regarded as an XML schema that can be used to obtain a Resource Description Framework (RDF) [36, 96] which encodes the ontological structure.

The Communication Ontology Description produces a class diagram (COD diagram) that shows all the agents and all their communications (relationships among agents). This diagram is drawn on the basis of the AId (see Section 5.1.1). A class is introduced for each agent, and an association is introduced for each communication between two agents. Being communications a way to exchange knowledge, it is also important to introduce the proper data structure (coming from the entities described in the DOD diagram) in each agent. The association line that represents each communication is drawn from the initiator of the conversation to the other agent (participant) as can be deduced from the description of their interaction performed in the Roles Identification. Each communication is characterized by three attributes, Ontology, Agent Interaction Protocol and Content Language, which are grouped into an association class. The roles, initially identified in the Agents Identification, are completely defined in the Roles Description that produces a UML class diagram in which classes are used to represent roles. In particular, each role uses several elementary tasks to implement its complex behavior and, finally, roles are grouped in packages representing agents.

The Protocols Description is required only when the FIPA standard protocols are not sufficient to solve some communication problems and new protocols must be introduced.

Table 5.1: The method fragments of PASSIM.

| PHASE | COMPOSED METHOD FRAGMENT | ATOMIC METHOD FRAGMENTS | WORK PRODUCT (KIND) | SOURCE METHODOLOGY |
|---|---|---|---|---|
| Requirements Specification | System Requirements | - Domain Description<br>- Agents Identification<br>- Roles Identification<br>- Tasks Specification | - Domain Description diagram (use-case diagram)<br>- Agents Identification diagram (use-case diagram)<br>- Roles Identification diagrams (sequence diagram)<br>- Tasks Specification diagrams (activity diagram) | PASSI |
| Design | Agent Society | - Domain Ontology Description<br>- Communication Ontology Description<br>- Roles Description<br>- Protocols Description | - Domain Ontology Description diagram (class diagram)<br>- Communication Ontology Description diagram (class diagram)<br>- Roles Description diagram (class diagram)<br>- Protocols Description (sequence diagram) | PASSI |
| | Agent Implementation | - Agent Structure Definition<br>Agent Behavior Description | - Single-Agent Structure Definition diagrams (class diagram)<br>- Multi-Agent Structure Definition diagram (class diagram)<br>- Single.Agent Behavior Description diagrams (activity/state diagram)<br>- Multi-Agent Behavior Description diagram (activity/state diagram) | PASSI |
| Prototyping | Simulation-based Prototyping | - ELDA-based Multi-Agent-System Model Definition<br>- Multi-Agent System Code Generation<br>- Simulation Implementation<br>- Simulation Execution | - Multi-Agent System Distilled StateChart Simulation Model ($MAS_{DSC}$ diagram)<br>- MAS Code (C($MAS_{DSC}$) diagram)<br>- Simulator Program<br>- Simulation Results | Proposed Simulation-based process |
| Coding | Code | - Code Reuse<br>- Code Refinement | - Code for the target agent platform | PASSI |
| Deployment | Deployment | - Deployment Configuration | - Deployment Diagrams | PASSI |

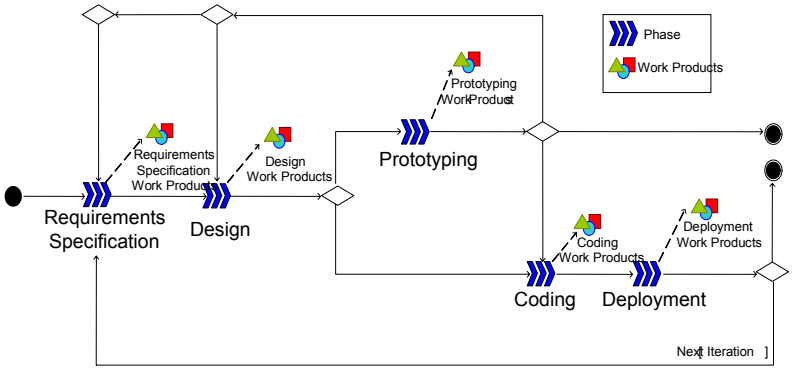



Figure 5.1: The software development life-cycle of PASSIM.

### 5.1.2.2 The Agent Implementation Fragment

The Agent Implementation method fragment is composed by two different atomic fragments, each of them carried out at both the multi- and single-agent level of abstraction. The multi-agent level models the overall structure of the system (MAS structure and behavior, inter-agent communications, etc.). The single-agent level of abstraction focuses on the implementation details of each agent.

In particular, the following two atomic method fragments are carried out at the multi- and single-agent levels:

- Agent Structure Definition (ASD), which uses conventional class diagrams to describe the structure of agents (represented by classes) and produces both the Single-Agent Structure Definition (SASD) diagrams and the Multi-Agent Structure Definition (MASD) diagram;
- Agent Behavior Description (ABD), which uses activity diagrams or statecharts to describe the behavior of agents and produces the Single-Agent Behavior Description (SABD) diagrams and the Multi-Agent Behavior Description (MABD) diagram.

The MASD diagram represents the multi-agent system from the structural point of view. Agents are represented as classes with their behaviors in the operation compartment and attributes specifying the agent knowledge.

The agent behavior at the multi-agent level is described by the MABD diagram. This is a UML activity diagram used to illustrate the dynamics of the system during the agents' lifecycle. In this diagram, the involved agents and their tasks are represented with swim-lanes, operations are displayed as activities, and transitions among activities represent events like method invocations (when relating activities in the same swim-lane), new behavior instantiations/invocations (when relating activities of the same agent but in different swim-lanes) or messages (when activities from two different agents are involved).

In the SASD diagram one class diagram is used for depicting the internal structure of each agent. This is a very detailed diagram, reporting attributes

and methods of both the agent class and the classes of the tasks. The details of the behavior of each agent are specified in the SABD diagram.

### 5.1.3 Prototyping

The Prototyping phase is carried out by a (composed) method fragment, Simulation-based prototyping, which is composed by four atomic method fragments: *ELDA-based Multi-Agent-System Model Definition*, *Multi-Agent-System Code Generation*, *Simulation Implementation, and Simulation Execution*.

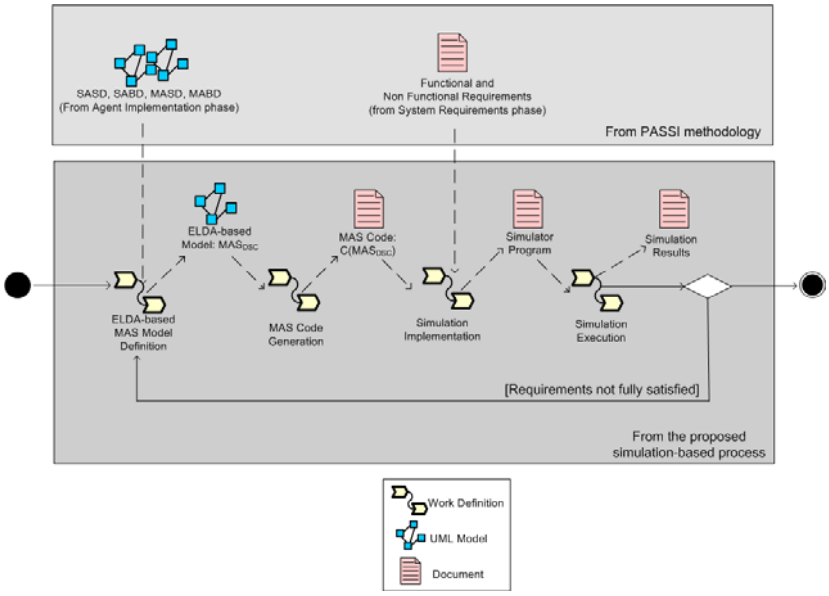



Figure 5.2: The ELDA-based Simulation method fragment.

The *ELDA-based Multi-Agent-System Model Definition* is enabled by ELDA MAS Meta-Model which supports the specification of the agents types and the interaction protocols among them. The ELDA-based specification of a MAS, denoted as $MAS_{DSC}$, is expressed as:

$$MAS_{DSC} = \{Beh(AT1), Beh(AT2), \ldots, Beh(ATn)\},$$

where Beh(ATi) is the specification of the dynamic behavior of the i-th agent type modeled according to the ELDA model (see section 3.1.1). The *ELDA-based Multi-Agent-System Model Definition* is an adapted fragment which takes as input the structural and dynamic diagrams (SASD, SABD, MASD, and MABD diagrams) produced by the Agent Implementation which are semi-automatically translated into a $MAS_{DSC}$ as described in section 5.2.

The *Multi-Agent-System Code Generation* is made according to the ELDAFramework (see Section 3.2) and fully supported by ELDATool (see Section 3.4): given a $MAS_{DSC}$, it produces $C(MAS_{DSC})$ representing the code of $MAS_{DSC}$.
The *Simulation Implementation* and *Simulation Execution* are supported by ELDASim (see Section 3.3.1): on the basis of functional and non-functional requirements and the MAS code, a simulator program can be implemented by using ELDASim in the *Simulation Implementation*; in the *Simulation Execution* the simulator program is executed to obtain the simulation results containing validation traces and performance parameter values. Moreover, the simulation results can be used to feed back the *ELDA-based Multi-Agent-System Model Definition*.

### 5.1.4 Coding

The Coding phase is carried out by the Code (composed) method fragment selected from PASSI which produces the code of the MAS under-development. The Code is composed by two atomic method fragments:

1. Code Reuse, in which code generation is directly supported by the PTK. In particular, it is possible to generate not only the skeletons but also largely reusable parts of the methods implementation based on a repository of reused patterns and associated design descriptions. Currently, the pattern repository includes a set of reusable portions of JADE and FIPA-OS agents and corresponding behaviors; a more detailed description of the pattern repository can be found in [20, 26];
2. Code Refinement, where code is manually completed by the programmer.

### 5.1.5 Deployment

The Deployment phase is carried out by the Deployment (composed) method fragment selected from PASSI which specifies the distribution of the parts of the system (agents) across the available agent platforms. The Deployment is composed by only the Deployment Configuration atomic method fragment which produces the deployment diagrams describing the allocation of agents to the available agent platforms and any constraints on agent migration. In particular, these diagrams also specify the libraries or hardware devices (sensors or actuators) that should be available on the agent platforms in order to ensure the proper system functionalities.

## *5.2 Adapting the design for the prototyping*

As previously introduced, in order to prototype the MAS under-development, the work products of the Agent Implementation, carried out in the Design phase, must be translated into a Multi-Agent System Distilled StateChart Model ($MAS_{DSC}$) which represents the work product of the Multiagent-System ELDA-based Model Definition (see Table 5.1). The input to the translation

process consists the SASD, SABD, MASD, and MABD diagrams whereas the output of the translation process is represented by a $MAS_{DSC}$.
The translation process is semi-automatic which means that these diagrams are automatically translated into a $MAS_{DSC}$ skeleton and, then, the $MAS_{DSC}$ skeleton is manually refined through programming. In particular, the following steps are carried out:

1. The agent types of the $MAS_{DSC}$ are directly derived from the agent types of the MASD diagram through a one-to-one mapping.
2. The interactions in terms of events exchanged between the agent types of the $MAS_{DSC}$ are directly derived from the MABD diagram.
3. The ADSC of an agent type is based on the SASD and the SABD diagrams of the agent type. As a SASD is a platform-dependent diagram (e.g. FIPA-OS-based or JADE-based) the SASDs are designed to be ELDA-model oriented. In particular, attributes and methods of the agent type are inserted into the ADSC as state variables and supporting functions, respectively. These state variables and supporting functions need to be manually finalized, i.e. the specific type of all the state variables is defined and the methods are implemented. The activities reported in the SABD diagram become states of the ADSC and the transitions among activities become transitions among the states corresponding to these activities. Finally, the ADSC has to be refined through manual programming which is needed for model consistency and optimization purposes. This refinement step involves the introduction/deletion of states, transitions, transition labels (event[guard]/action), state variables and supporting functions.

In the following we use a simple example to show how the semi-automatic translation from the work products of the Agent Implementation to a $MAS_{DSC}$ can be obtained.
The example MAS we considered is composed of two agent types: (i) an information retrieval agent (IRA) whose task is to visit a given number of locations to retrieve information through a query; (ii) an information provider agent (IPA) whose task is to process the query received from the IRA and to provide it with the query result.
The work products produced by the Agent Implementation activities are shown in Figure 5.3, 5.4, 5.5, 5.6.

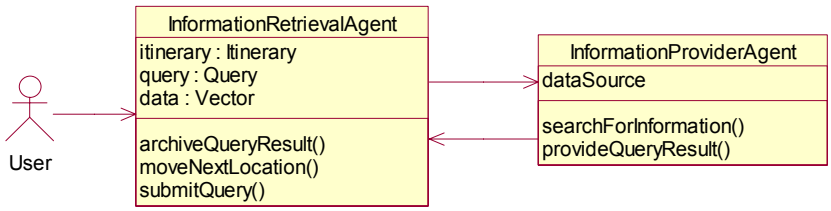



Figure 5.3: the MASD of the example MAS.

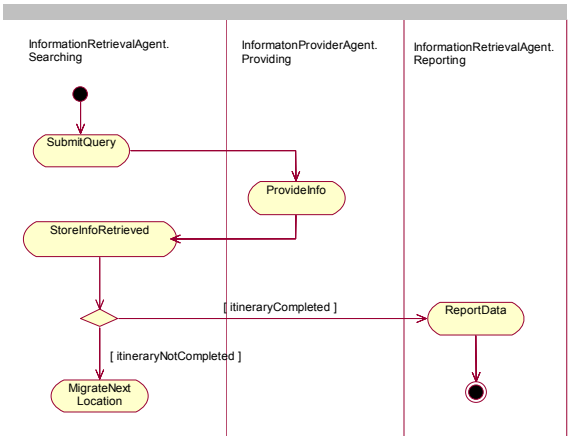
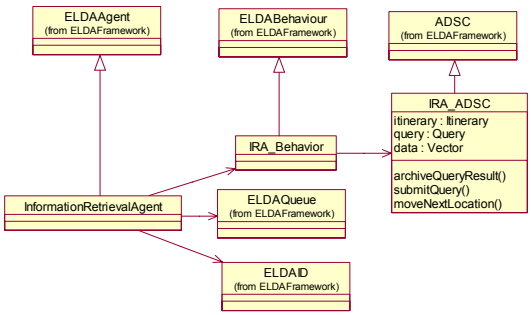
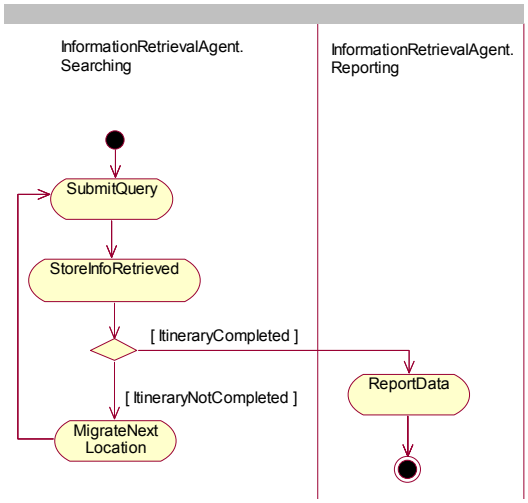



Figure 5.4: the MABD of the example MAS.




(a)




(b)

Figure 5.5: the (a) SASD and (b) SABD of the IRA.

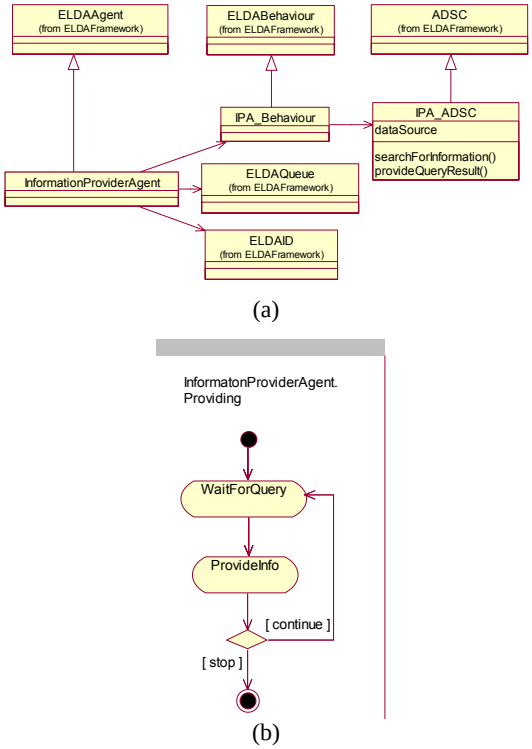



Figure 5.6: the (a) SASD and (b) SABD of the IPA.

Given the MASD of the example MAS (Figure 5.3), the agent types of the $MAS_{DSC}$ are: InformationRetrievalAgent and InformationProviderAgent.

***InformationRetrievalAgent***

Given the MABD of the example MAS (Figure 5.4), the events exchanged between the two agent types are: QUERYREQUEST (QUERY) and QUERYINFORM (QUERYRESULT), which correspond to the two main messages of the FIPA Query Protocol which was selected for the communication between the two agents.

Given the SASD and SABD diagrams of the InformationRetrievalAgent (see Figure 5.5), the ADSC skeleton of the InformationRetrievalAgent of the $MAS_{DSC}$ reported in Figure 5.7 was obtained. The names of the states of the ADSC have as suffix the names of the activities of the SABD diagram and as postfix “Done” which means that the activity corresponding to the state has been carried out. The event labeling the transition from SubmitQueryDone to StoreInfoRetrieveDone corresponds to the message QUERYINFORM sent from the IPA agent. The events labeling the transitions from StoreInfoRetrieveDone are derived from the guards of the selection block of the IRA SABD diagram (see Figure 5.5b). The event labeling the transition

from MigrateNextLocationDone to SubmitQueryDone assumes the name of the activity corresponding to the target state as each transition of a DSC must be labeled by an event.

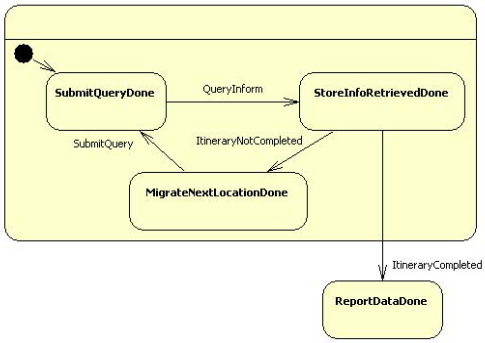
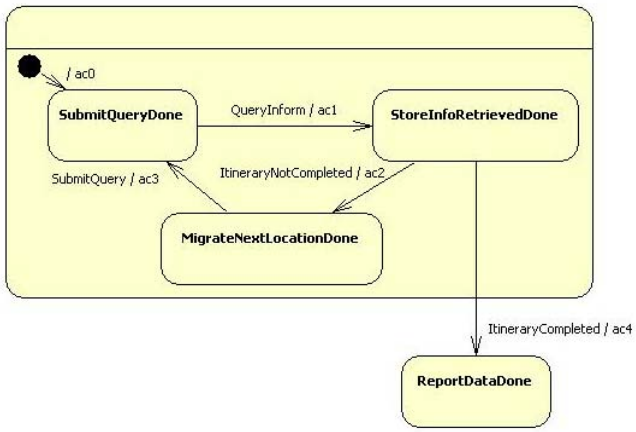



Figure 5.7: the ADSC skeleton of the IRA.

The ADSC of the InformationRetrievalAgent which was obtained after refinement is shown in Figure 5.8. The actions have been purposely defined "by programming" on the basis of the state variables and supporting functions derived from the SASD diagram (see Figure 5.5a).




```
ac0: generate(new QueryRequest(self(), IPA, query));
ac1: QueryInform qi = (QueryInform)evt;
archiveQueryResult(qi.getInfo());
if (itinerary.hasNextLocation())
 generate(new ItineraryNotCompleted(self()));
else
 generate(new ItineraryCompleted(self()));
ac2: Location nextLoc = itinerary.getNextLocation();
generate(new Move(self(), nextLoc, new SubmitQuery(self())));
ac3: reportData();
ac4: ac0;
```

Figure 5.8: the refined ADSC of the IRA.

***InformationProviderAgent***

Given the SASD and SABD diagrams of the InformationProviderAgent (see Figure 5.6), the ADSC skeleton of the InformationProviderAgent of the $MAS_{DSC}$ was obtained (see Figure 5.9). Two states are derived: WaitForQueryDone, referring to the end of the WaitForQuery activity, and ProvideInfoDone, referring to the end of the ProvideInfo activity. The event labeling the transition from WaitForQueryDone to ProvideInfoDone corresponds to the message QueryRequest sent from the IRA agent. The Continue event labeling the transition from ProvideInfoDone to WaitForQueryDone is derived from the guard of the selection block of the IPA SABD diagram (see Figure 5.6b).

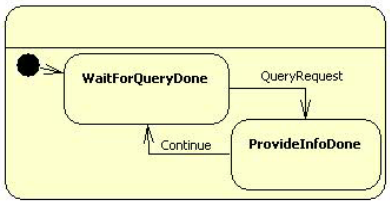
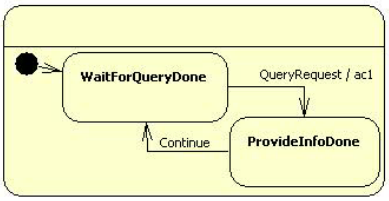



Figure 5.9: the ADSC skeleton of the IPA.

The ADSC of the InformationProviderAgent which was obtained after refinement is shown in Figure 5.10. The actions have been purposely defined "by programming" on the basis of the state variables and supporting functions derived from the SASD diagram (see Figure 5.6a).




```
ac1: QueryRequest qr=(QueryRequest)evt;
Result r = searchForInformation(qr.getQuery());
generate(new QueryInform(self(), qr.getSource(), r);
generate(new Continue(self(), self()));
```

Figure 5.10: the refined ADSC of the IPA.

## 5.3 A case study: from the analysis to the validation of an Agent-based E-Marketplace

An electronic marketplace (e-Marketplace) is a platform for buyers and sellers exchanging products and services [33, 98]: (i) buyers specify the items they want to buy, along with their desired price ranges; (ii) the e-Marketplace then matches trading partners for the buyers and provide the Request for Quotation (RFQ); (iii) on the basis of the specification and price

range, sellers return the quotation to the buyers and wait for the confirmation; (iv) after receiving all quotations, buyers can select the best offer and issue a purchase order to the selected sellers. Nowadays, many e-Marketplaces are based on software agents which are capable of fully supporting and automating the stages of the consumer-buying behavior (CBB) model [57, 74]. The CBB model defines the decision process which consumers undergo when purchasing a product. Such a process is articulated in six stages:

1. ***Need Identification***: This stage characterizes the buyer that becomes aware of some unmet/desired need. Within this stage, the buyer can be stimulated through product information.
2. ***Product Brokering***: This stage comprises the retrieval of information to help determine what to buy. This encompasses the evaluation of product alternatives based on buyer-provided criteria. The result of this stage is the "consideration set" of products.
3. ***Merchant Brokering***: This stage combines the "consideration set" from the previous stage with merchant-specific information to help determine who to buy from. This includes the evaluation of merchant alternatives based on buyer-provided criteria (e.g., price, warranty, availability, delivery time, reputation, etc.).
4. ***Negotiation***: This stage is about how to settle on the terms of the transaction. The negotiation varies in duration and complexity depending on the market.
5. ***Purchase and Delivery***: The purchase and delivery of a product can either signal the termination of the negotiation stage or occur sometime afterwards.
6. ***Product Service and Evaluation***: This post-purchase stage involves product service, customer service, and an evaluation of the satisfaction of the overall buying experience and decision.

The objective of our case study is to apply PASSIM to the design and validation of an agent-based e-Marketplace (AeM) which supports stages 3, 4, and 5 through the following specific consumer-buying process [41]:

i. *Request Input*. When users wish to buy a product, they identify a set of product parameters (product description, maximum price $P_{MAX}$), log into the e-Marketplace and submit a request containing the product parameters. The e-Marketplace checks if users are trustworthy (i.e. from a commercial and security viewpoint) and decides if requests can be accepted. If so, the Consumer Assistant System (CAS) of the e-Marketplace starts satisfying the user request.
ii. *Searching*. The CAS obtains a list of locations of vendors by using the Yellow Pages Service (YPS) of the e-Marketplace. The YPS is a federation of autonomous components at which vendors register to advertise their products. In particular the following YPS organizations were established:
    - Centralized: each YPS component stores a complete list of vendors;

- One Neighbor Federated: each YPS component stores a list of vendors and keeps a reference to only one other YPS component;
- M-Neighbors Federated: each YPS component stores a list of vendors and keeps a list of at most M YPS components.

iii. *Contracting & Evaluation*. The CAS interacts with the found vendors to request an offer ($P_{OFFER}$) for the desired product, evaluates those received, and selects an offer, if any, for which the price is acceptable (i.e., $P_{OFFER} \leq P_{MAX}$).

iv. *Payment*. The CAS purchases the desired product from the selected vendor using a given amount of e-cash (or bills). The following steps are performed to execute the money transaction between the CAS and the vendor:
   - the CAS gives the bills to the vendor;
   - the vendor sends the bills to its bank;
   - the bank validates the authenticity of the bills, exempts them from reuse, and, finally, issues an amount of bills equal to that previously received to the vendor;
   - the vendor notifies the CAS.

v. *Reporting*. The CAS reports the buying result to the User.

This description can be considered as an initial requirements document on the basis of which the Requirements Specification phase is carried out. In the following subsections selected work products of the first four phases of PASSIM (see Section 5.1) are shown and described. In particular, Section 5.3.1 presents the Requirements Specification work products, Section 5.3.2 shows the Design work products, and, finally, Section 5.3.3 shows the establishment and the results of the Prototyping phase which allows for both functional validation and performance evaluation of the MAS under-development.

### 5.3.1 The Requirements Specification phase

From the previously reported description of the system to be designed, the AId (see Section 5.1.1) was drawn which reports three actors (User, Vendor and Bank) and the use cases, coming from the Domain Description, which were packaged into the following six agents:

- User Assistant Agent (UAA) is associated with a user and assists her/him in looking for a specific product that meets her/his needs and buying the product according to a specific buying policy.
- Yellow Pages Agent (YPA) represents an entry point of the federated yellow pages service (or “Yellow Pages”) which provides the location of agents selling a given product.
- Vendor Agent (VA) represents the vendor of specific goods.
- Mobile Consumer Agent (MCA) is an autonomous mobile agent dealing with searching, contracting, evaluation, and payment of goods.

- Access Point Agent (APA) represents the entry point for the e-marketplace, accepts requests for buying a product from a registered UAA and fulfils them by generating a specific MCA.
- Bank Agent (BA) represents a reference bank of MCA and VA.

It is worth noting that the <<communicate>> relationship shown in Figure 5.11 represents agents interaction.

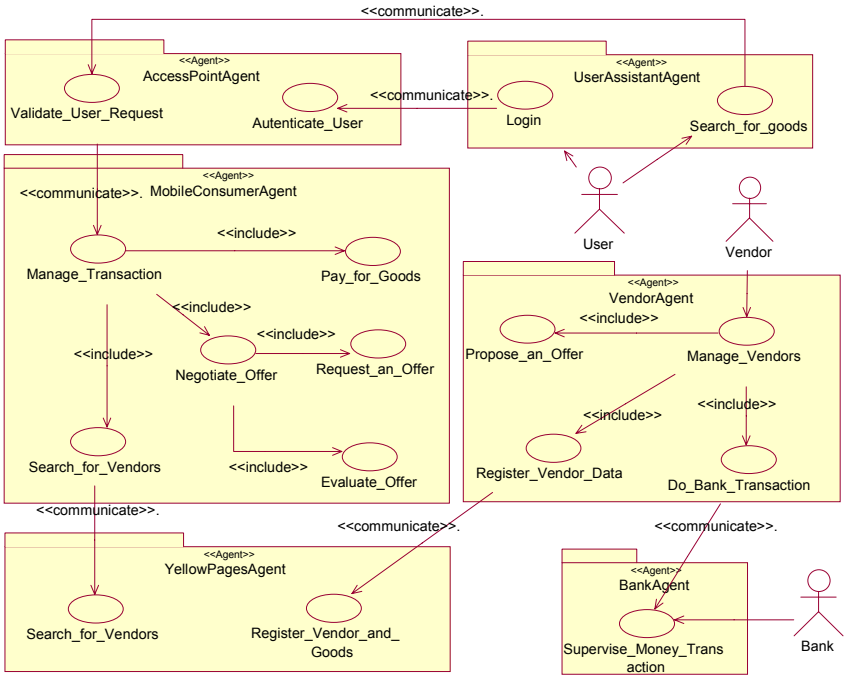



Figure 5.11: The AId for the proposed case study.

On the basis of the AId, the Roles Identification diagram (RId) was designed. A portion of the obtained RId is shown in Figure 5.12 where the APA (UserRequestValidatorAndForwarder role) after validating the order, forwards it to the MCA (Searcher role); afterwards the MCA asks for the vendors list to the YPA (VendorListProvider role). After getting the list, the MCA (Contr&Eval role) contacts all the VAs (OfferProposer role) and asks them for their offers. Finally, the MCA selects the best offer and pay for the product (Payer role).

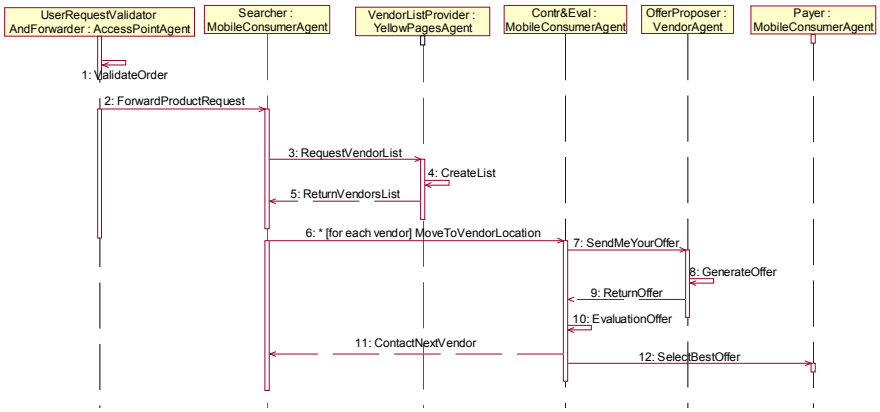



Figure 5.12: A portion of the RId regarding a specific-product vendors search scenario.

An initial definition of the dynamic behavior of each agent is the work product produced by the last atomic method fragment of this phase (Tasks Specification). The Tasks Specification produces a set of Tasks Specification diagrams (one for each identified agent) which are UML activity diagram representing the agent tasks. Each diagram is composed of two swim-lanes (see Figure 5.13): the right-hand highlights the roles of the agent which the diagram refers to and the activities the agent performs in playing these roles, whereas the left-hand reports the roles played by other agents interacting with the agent of right-hand.

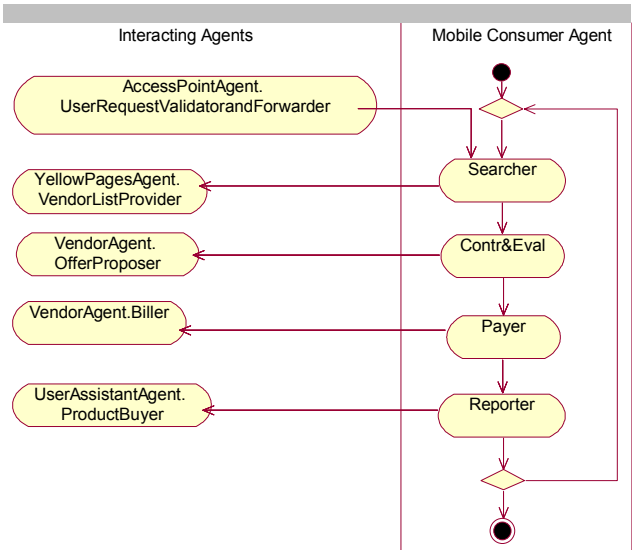



Figure 5.13: The Tasks Specification diagram for the MCA.

In figure 5.13, the Tasks Specification diagram of the MCA is shown. In particular, the MCA is involved in: (i) searching the list of vendors through a query to the YPA (Searcher role), (ii) contracting with VAs and evaluating their offers (Contr&Eval role), (iii) buying the product from the VA proposing the best offer (Payer role), (iv) reporting the transaction results to the UAA (Reporter role). Afterwards the MCA can either play again the Searcher role or be terminated.

### 5.3.2 The Design phase

The Agent Society method fragment (see Section 5.1.2) produces diagrams which represent social interactions and dependencies among the identified agents (see Section 5.3.1). A portion of the DOD diagram is reported in Figure 5.14 in which some concepts, predicates and actions used to define the problem domain are shown. For instance the Vendor concept (representing the vendor of the real-world scenario) is related with the Product(s) it sells. A vendor registers its products in the agent-based yellow pages service by executing the RegisterProduct action which is performed by the VA (action Actor) and its outcome received by the YPA (action Receiver).
A portion of the COD diagram is reported in Figure 5.15. It shows three identified agents (APA, VA, MCA) and two communications among them (Forward_Product_Request, Offer_Request). In particular, the Offer_Request communication happens when the MCA asks the VA for the best offer (see the scenario reported in Figure 5.12). This communication refers to the OfferPrice predicate from the ontology of Figure 5.14 and adopts the FIPAQuery agent interaction protocol and the RDF content language. Roles played by agents during the interaction (as described in the RIds) are reported at the beginning and the end of the association line.
The Agent Implementation method fragment (see Section 5.1.2.2) produces work products representing the MAS architecture. In particular, a portion of the Multi-Agent Structure Description (MASD) diagram, which describes the structure of the VA, MCA and APA agents, is shown in Figure 5.16. It is worth noting that the VA is in relationship with an (human) actor; this is an extension of UML that is useful to represent all the possible agent relationships (communications and GUI-based interactions with the user) in a unique diagram.
A portion of the obtained Multi-Agent Behavior Description (MABD) diagram is reported in Figure 5.17, which illustrates the activities occurring during the Vendor_Request communication between MCA and YPA and the Offer_Request communication between MCA and VA. In particular, this portion of the MABD diagram describes the request for the VA list from the MCA to the YPA, the migration of the MCA to the retrieved VA location and the contracting phase carried out by the MCA with the VA.

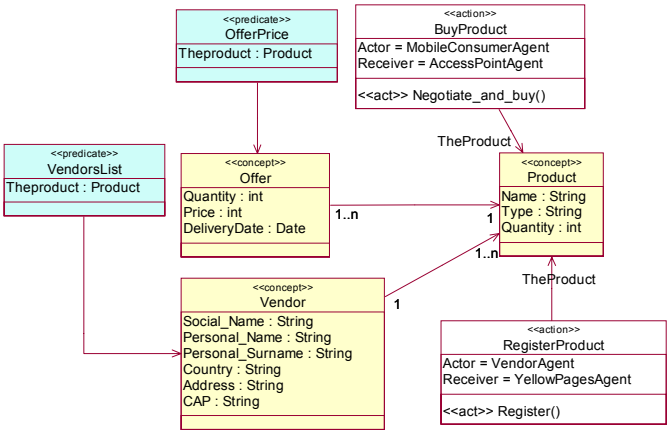



Figure 5.14: A portion of the DOD diagram.

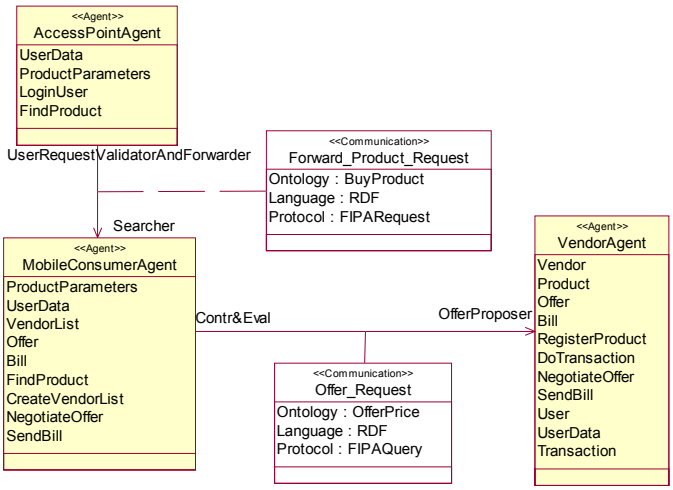



Figure 5.15: A portion of the COD diagram.

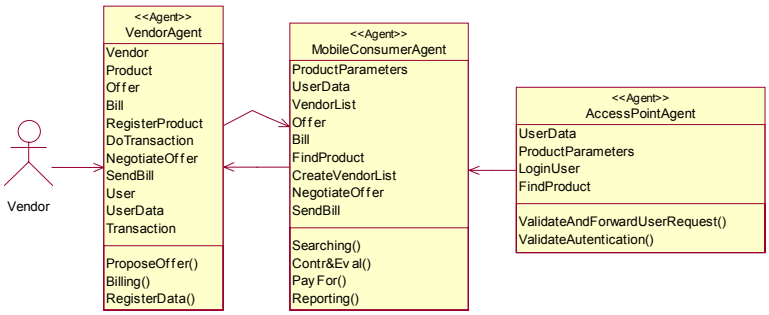



Figure 5.16: A portion of the MASD diagram.

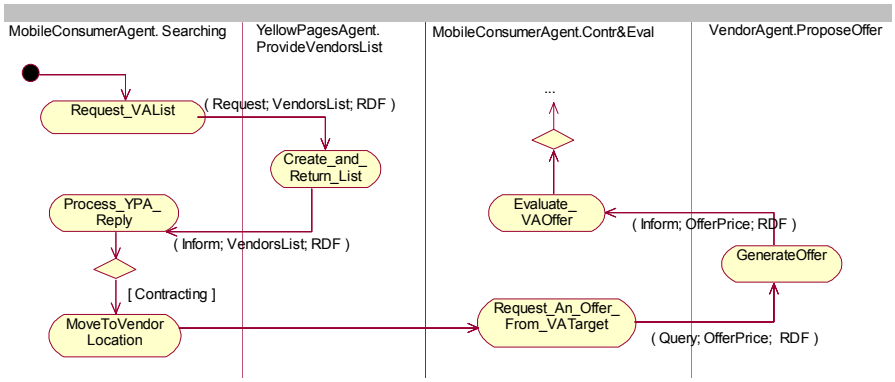



Figure 5.17: A portion of the MABD diagram with some interactions among MCA, YPA and VA.

Figure 5.18 shows the Single-Agent Behavior Description (SABD) diagram for the MCA, which provides a high-level specification of the behavior of the MCA. In particular, the MCA plays 4 different roles in the following sequence: Searching, Contr&Eval, PayFor and Reporting. They also correspond to the phases of the MCA lifecycle. In particular, in the Searching phase, the MCA moves to the location of the next YPA (YPATarget), requests the list of vendors (VAList), and processes the reply (YPA_Reply). If the Searching phase is not completed ([Searching] is evaluated to true), the MCA continues searching. If the guard [Contracting] holds (i.e. the VAList is not empty) the MCA passes into the Contr&Eval phase. If the guard [Reporting] holds (i.e. the VAList is empty) the MCA directly goes into the Reporting phase. In the Contr&Eval phase, the MCA moves to the location of a vendor in the VAList (VATarget), requests an offer (VAOffer) and evaluates it. If the MCA decides to accept the received VAOffer (i.e. the guard [BuyingSoon] holds) or another received VAOffer (i.e. the guard [MovingAndBuying] holds), it passes into the PayFor phase. If the MCA desires a new offer, it keeps contracting (i.e. guard [Contracting] holds true). If no offer is selected the MCA goes into the Reporting phase (i.e. guard [Reporting] holds true). Finally, in the Reporting phase, the MCA moves to the APA location and reports to its UAA.
Figure 5.19 shows the SASD diagram for the MCA and its derived agents. In particular, two specific MCAs are derived:

- the ItineraryConsumerAgents (or ICA), which performs the Searching and Contr&Eval phases (see Figure 5.18) by sequentially moving from one location to another within the e-Marketplace;
- the ParallelConsumerAgent (or PCA), which performs the Searching and Contr&Eval phases (see Figure 5.18) by means of the support of a set of mobile agents:
  - the ItinerarySearcherMobileAgent or the SpawningSearcherMobileAgent for carrying out sequential or parallel searching of vendors during the Searching phase;

- the ContractorMobileAgent for carrying out parallel negotiation during the Contr&Eval phase.

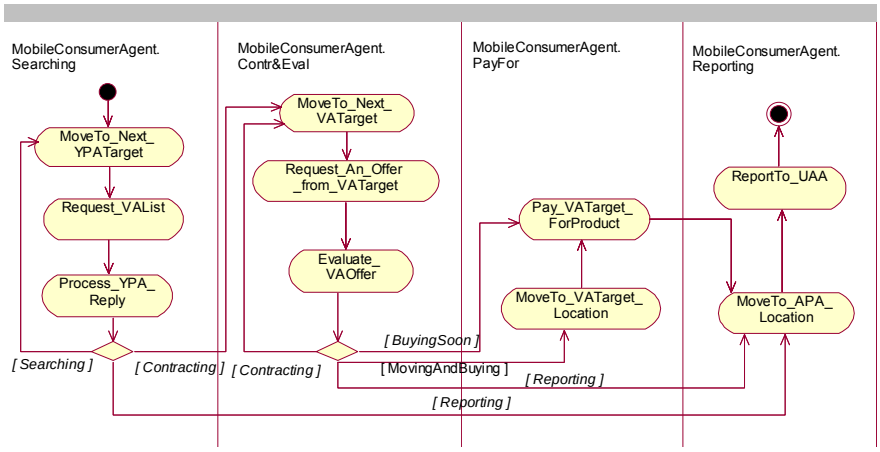



Figure 5.18: The SABD diagram for the MCA.

## 5.3.3 The Prototyping phase

The aim of the Prototyping phase is the functional validation of the designed AeM and the performance evaluation of different types of MCAs for optimization purposes. In particular, the functional validation is carried out on the basis of simple simulation scenarios aiming at validating the behavior of the agent types, the agent interaction protocols, and the global behavior of the AeM. The performance evaluation is carried out to evaluate the completion time of the buying task of different types of MCAs.

In the following subsections, first the refined ELDA-based MCAs derived from the Multiagent-System ELDA-based Model Definition are described (Section 5.3.3.1) and, then, the functional validation (Section 5.3.3.2) and the performance evaluation (Section 5.3.3.3) are presented.

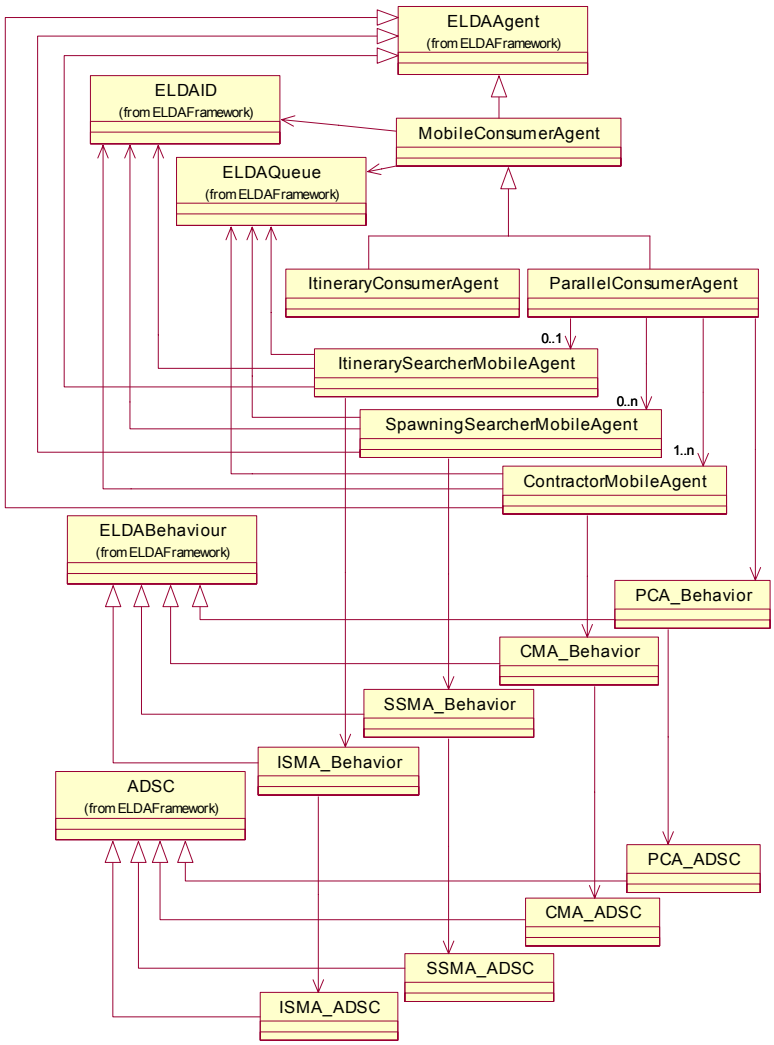



Figure 5.19: The SASD diagram for the MCA and derived agents.

### 5.3.3.1 ELDA-based MCAs

Two types of ELDA-based MCAs were obtained according to the adapting Multiagent-System ELDA-based Model Definition (see Section 5.2): ICA and PCA. Both ICA and PCA are equipped with policies for searching and buying (see Table 5.2) during the Searching and the Contr&Eval phases, respectively.

Table 5.2: Searching and Buying Policies of MCA.

| SEARCHING POLICY (SP) | DESCRIPTION |
|---|---|
| ALL | All YPA agents are contacted |
| PA-PARTIAL | A subset of YPA agents are contacted |
| OS-ONE-SHOT | Only one YPA agent is contacted |
| **BUYING POLICY (BP)** | **DESCRIPTION** |
| MP-Minimum Price | The MCA first interacts with all the VA agents; then, it buys the product from the VA offering the best acceptable price |
| FS-First Shot | The MCA interacts with the VA agents until it obtains an offer for the product at an acceptable price, then it buys the product |
| FT-Fixed Trias | The MCA interacts with a given number of VA agents and buys the product from the VA which offers the best acceptable price |
| RT-Random Trias | The MCA interacts with a random number of VA agents and buys the product from the VA which offers the best acceptable price |

In the following, we focus on the PCA as the ICA possesses a more simple behavior, which is encompassed by the PCA. Figure 5.20 shows the refined ADSC (see Section 3.1.1.2) of the PCA which was derived from the MASD, MABD, SASD and SABD diagrams (see Figures 5.16-5.19) of the MCA and from the SABD diagram specific to the PCA, not reported here for the sake of brevity, which is a specialization of the SABD diagram of the MCA.

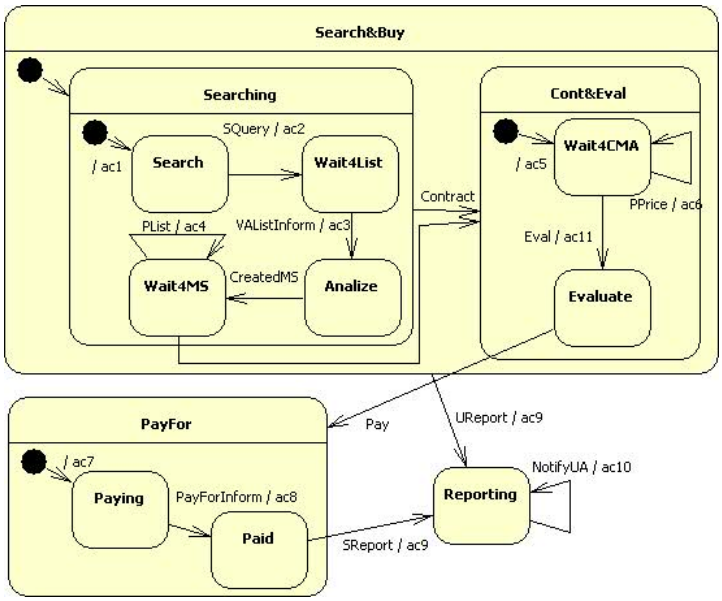



Figure 5.20: The ADSC of the PCA.

Table 5.3: Association between the activities of the SABD diagram of the MCA and the ADSC action of the PCA.

| SABD ACTIVITY | ADSC ACTION |
|---|---|
| MoveTo_Next_YPATarget | ac1, ac2 |
| Request_VAList | ac3 |
| Process_YPA_Reply | ac4 |
| MoveTo_Next_VATarget | sa1 |
| Request_An_Offer_From_VA_Target | ac5 |
| Evacuate_VAOffer | ac6 |
| MoveTo_VATarget_Location | ac11 |
| Pay_VATarget_ForProduct | ac7, ac8 |
| MoveTo_APA_Location | ac9 |
| ReportTo_UAA | ac10 |

The messages that the MCA exchanges with the YPA, VA, and UAA agents during its lifecycle, reported in the MABD diagram, are implemented through events in the ADSC; the association between messages and events is reported in Table 5.4 for the interactions with YPA and VA.

Table 5.4: Association between the messages of the MABD diagram of the MCA and the ADSC events of the PCA.

| MABD MESSAGE | SENDER→RECEIVER | ADSC EVENT |
|---|---|---|
| (Request, VendorsList, RDF) | MCA → YPA | VAListRequest |
| (Inform, VendorsList, RDF) | VA → YPA | VAListInform |
| (Query, OfferPrice, RDF) | MCA → VA | OfferPriceQuery |
| (Inform, OfferPrice, RDF) | VA → MCA | OfferPriceInform |
| (Request, Payment, RDF) | MCA → VA | PayForRequest |
| (Inform, Payment, RDF) | VA → MCA | PayForInform |

The names of the composite states of the ADSC corresponds to the names of the phases of the MCA shown in the related SABD diagram (see Figure 5.18). For the sake of modularity the Searching and Contr&Eval states are embodied into the Search&Buy state.
The activities reported in the SABD diagram are implemented by the actions of the ADSC; the association between activities of the SABD diagram and actions is reported in Table 5.3.
The PCA agent fulfils the searching phase in the Searching state. In particular, as soon as the PCA agent is created, it moves (ac1) to the first Yellow Page Agent (YPA) location and locally interacts (ac2) with the YPATarget by sending it the VAListRequest event. The YPATarget replies to the PCA agent with the VAListInform event which can contain a list of VA

agents with the linked YPA agents. After processing the reply (ac3), the PCA agent can do one of the following:

- create an Itinerary Searcher Mobile Agent (ISMA), which sequentially moves from one YPA location to another, if the YPS organization is of the One-Neighbor Federated type, and pass (ac4) into the contracting phase as soon as a PList event sent by the ISMA agent is received;
- create M Spawning Searcher Mobile Agents (SSMAs), if the YPS organization is of the M-Neighbors Federated type, and pass (ac4) into the contracting phase when all the PList events sent by the directly created SSMA agents are processed. In particular, an SSMA agent moves to the assigned YPA agent and, in turn, creates a child SSMA agent for each reachable YPA agent. This parallel searching technique generates a spawning tree, with SSMA agents as nodes, which is rooted at the PCA agent. If an SSMA agent interacts with a YPA agent which has already been visited by an SSMA agent belonging to the same spawning tree, the YPA agent notifies the SSMA agent which then returns to its parent;
- directly pass into the contracting phase if the YPS organization is of the Centralized type;
- report an unsuccessful search to the UAA agent.

The contracting phase accomplished in the Contr&Eval state involves the creation (ac5) of Contractor Mobile Agent (CMA) according to the modes reported below. Each CMA agent is able to move to the assigned VA location, contract with the VA agent, and report the obtained offer. The VA offers (PPrice events) reported by the CMA agents are evaluated and a decision about when and from which VA agent to purchase is therefore taken (ac6). In the PayFor state the PCA agent pays (ac7) the VA agent using the PayForRequest event which contains the bills. After receiving the PayForInform event, the PCA agent passes (ac8) to the Reporting state from where it moves back (ac9) to the original APA location and finally reports (ac10) to its UAA agent.

When using agent techniques in e-Marketplaces, a large number of agents are generated in the e-Marketplace network which could lead to many problems such as server loading, network congestion and, more generally, scalability of the whole system [69]. So to optimize the performance of the PCA during the Contr&Eval phase with respect to time and resources, two types of CMAs (see Figure 5.21) have been defined:

- Full Parallel CMA (CMA_FP): the PCA spawns an instance of the CMA_FP for each VA location so that the CMA_FP contracts with the assigned VA and returns the obtained offer to the PCA. The advantage of this solution is that CMAs, once created by the PCA, can soon move to the assigned VA location and contract with the VA so minimizing waiting times. However, the creation of a large number of CMAs on a single agent server can increase the agent server load as well as the network congestion in the proximity of the agent server. Moreover, if

the buying policy is of the MP type, such solution is effective; otherwise, such solution would create more CMAs than those needed.

- Binary CMA (CMA_BIN): after organizing the list of the VAs retrieved in the Search phase as a binary tree, the PCA spawns a CMA_BIN to the VA location, root of the tree. A CMA_BIN, in turn, spawns at most two other CMA_BIN agents if the left and/or right branches/leaves exist. In this operational mode, at most two agents are created on a single agent server so reducing the server load due to agent creation and the network congestion due to agent migration [110]. According to the way the CMA_BIN returns the results of the negotiation with the VA to the PCA, the following types of CMA_BIN have been modeled:
  - CMA_BIN_FW_R2PCA: the agent directly reports to the PCA through an external event. The advantage of this solution rests on its simplicity whereas, if the number of CMA created is high, there would be a high number of external events targeting the PCA which would become a bottleneck.
  - CMA_BIN_FW_R2O: the agent reports to its owner (i.e. the CMA agent that has spawn it) through an external event. In this way, only the root CMA agent reports to the PCA. In this mode, the disadvantage of the previous solution is avoided.
  - CMA_BIN_BW_R2O: the agent reports to its owner (i.e. the agent that has spawn it) by moving to its site. Also in this case, only the root CMA agent reports to the PCA. This operational mode preserves the same advantage as the previous one and, in addition, can be effectively exploited in the case the agents can only communicate through local interactions (e.g. based on tuple-based systems).

Figure 5.22 shows the ADSC of the CMA_BIN_FW_R2PCA; the ADSCs of the other CMA_BINs are variants of the ADSC of the CMA_BIN_FW_ R2PCA. Migration and child spawning are carried out in the Migrate_And_Create state, whereas negotiation is carried out in the Contract state. In particular, after its creation the CMA moves to the location of the assigned VA (ac0), where it tries to spawn two other CMAs, and goes into the Contract state (ac1). In this state, the CMA sends the OfferPriceRequest event to the VA (ac2), process (ac3) the offer contained in the OfferPriceInform event and, finally, reports to the PCA (ac4).

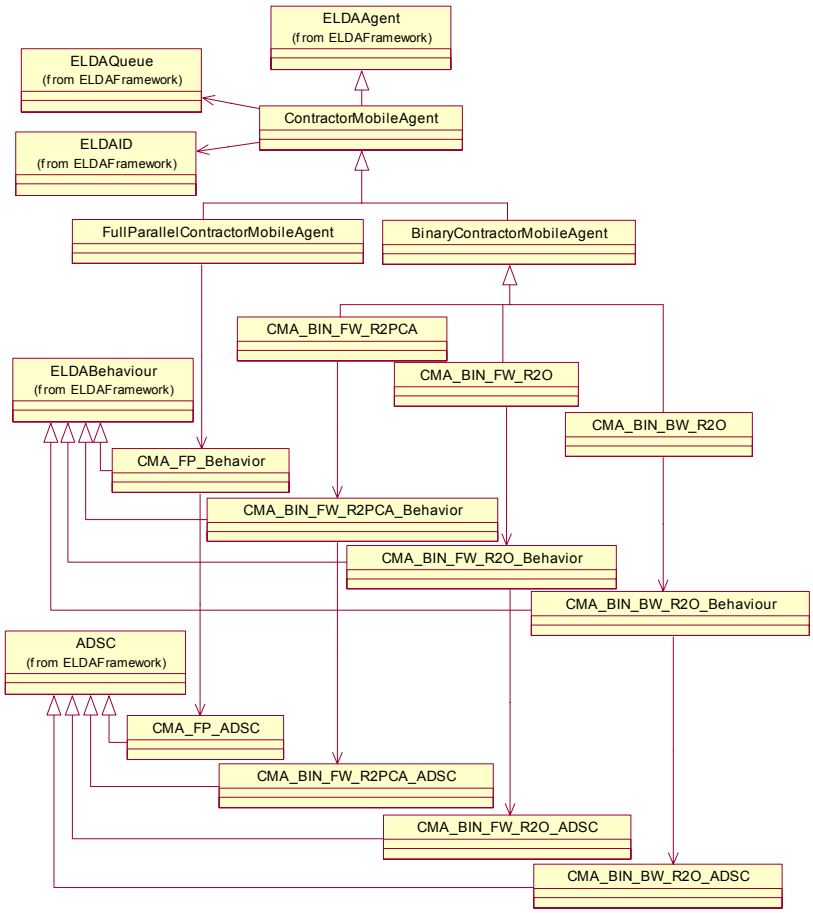



Figure 5.21: the SASD of the CMA.

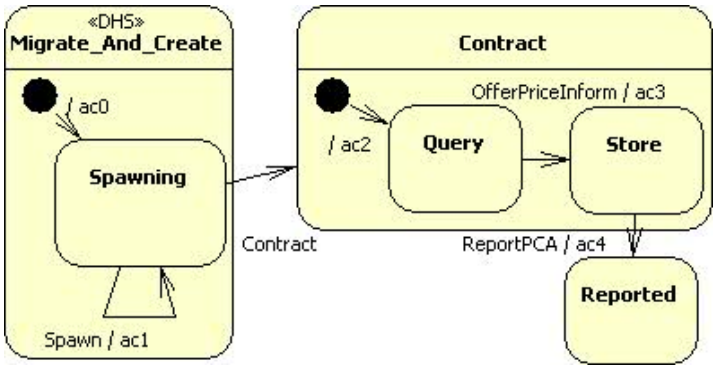



Figure 5.22: The ADSC of the CMA_BIN_FW_R2PCA.

### 5.3.3.2 Functional Validation

Functional validation is supported by ELDASim (see Section 3.3.1) through the generation of event traces which can be analyzed off-line to validate agent behaviors, agent interaction protocols and the behavior of the whole MAS.

Validation of a single agent type behavior relies on a simple simulation scenario which allows for the generation of the response of the agent behavior to all its admissible events. Validation of an agent interaction protocols is based on simple simulation scenario which allows for the generation of the flow of events exchanged between the involved agents. Validation of the whole system is carried out by setting more complex simulation scenarios. In particular, the simulation scenario for the validation of the global behavior of the AeM, also used during the performance evaluation phase, was set up as follows:

- Each stationary agent (UAA, APA, YPA, VA, BA) executes in a different agent server.
- Agent servers are mapped onto different network nodes which are completely connected through links having the same characteristics and modeling the communication delay (δ) as a lognormally distributed random variable.
- Each VA is reachable from any YPA and sells the same set of products.
- Each product is always offered by a VA at a fixed price, which is an integer number uniformly distributed between a minimum ($PP_{MIN}$) and a maximum ($PP_{MAX}$).
- The user is willing to pay, for a desired product, a maximum price $P_{MAX}$, which is an integer value between $PP_{MIN}$ and $PP_{MAX}$.

An indirect functional validation of the AeM was carried out by defining the following index, calculating such index both through analytical methods and simulation, and comparing the outcoming results:

- the Probability of Successful Buy (PSB), which is defined as the probability of successfully buying a desired product within the e-Marketplace.

On the basis of the assumptions made for the simulated e-marketplace, PSB can be easily calculated as follows:

$$PSB = 1-[(PP_{MAX}-P_{MAX})/(PP_{MAX} -PP_{MIN}+1)]V,$$

where: V is the number of VA agents contacted by the MCA for buying the product, $PP_{MAX}-P_{MAX}$ represents the number of prices that exceed $P_{MAX}$ (i.e. that are not acceptable for the user), whereas $PP_{MAX}-PP_{MIN}+1$ represents the number of all the possible prices for the product. V depends on the BP adopted by the MCA; in particular: if BP is of the MP type or of the FS type V=NVA; if BP is of the FT type V is VFT=NVA/2+1 as in the simulations the

MCA always performs NVA/2+1 trials; if BP is of the RT type V belongs to the range [1..NVA].
The values of PSB calculated both analytically and through simulation for each defined BP and with $PP_{MAX}$=200, $PP_{MIN}$=100, PMAX=$PP_{MIN}$, and NVA=100, are reported in Figure 5.23. It is worth noting that the analytical value for BP=RT is calculated by using the mean value of the uniform distribution defined in the range [1..NVA].

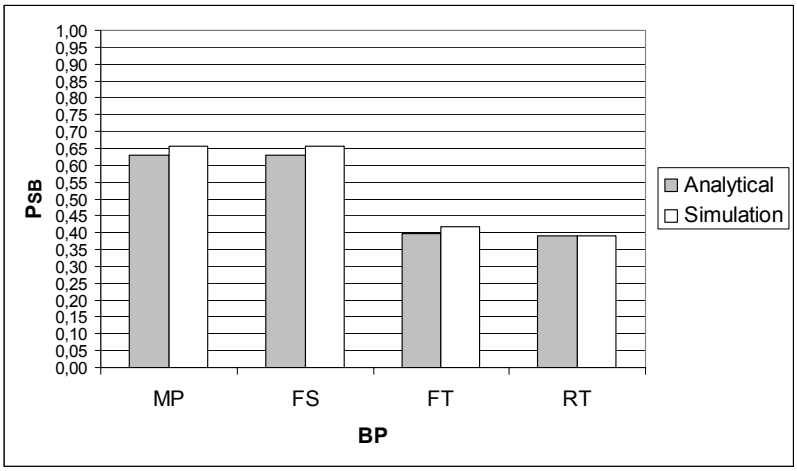



Figure 5.23: Evaluation of PSB for the defined BPs with $PP_{MAX}$=200, $PP_{MIN}$=100, $P_{MAX}$=$PP_{MIN}$, and NVA=100.

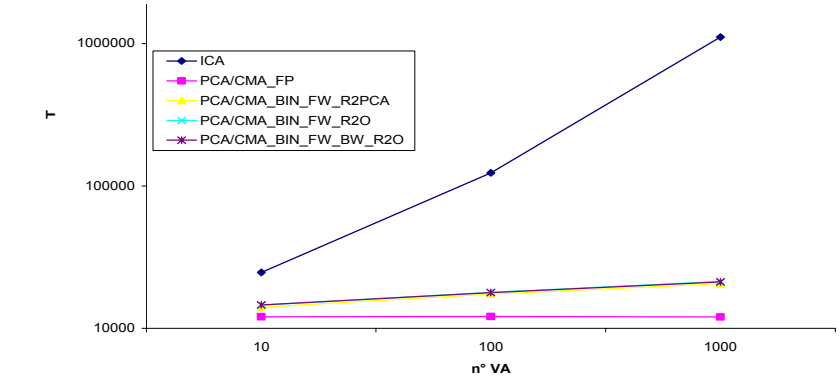
Such results confirm that the global behavior of the AeM is correct and this confirmation also provides an indirect functional validation of the AeM.

## 5.3.3.3 Performance Evaluation

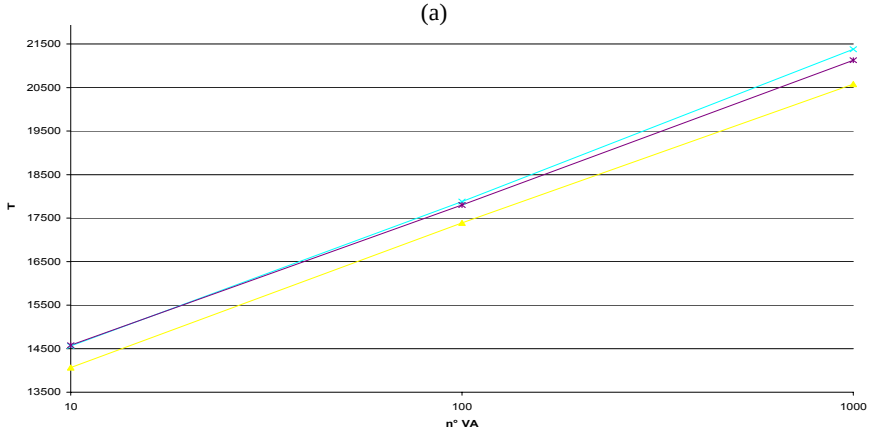
The aim of the performance evaluation phase is to evaluate and compare the efficiency of the 5 types of MCA (ICA, PCA/CMA_FP, PCA/CMA_BIN_FW_R2PCA, PCA/CMA_BIN_FW_R2O, PCA/CMA_BIN_BW_R2O) in terms of the following performance index:

$$\text{Buy Task Completion Time (TBTC)} = T_{CREATION} - T_{REPORT}$$

where, $T_{CREATION}$ is the creation time of the MCA and $T_{REPORT}$ is the reception time of the MCA report.
Given the scenario described in section 5.3.3.2, the evaluation of the TBTC performance index is focused on an MCA adopting a searching policy (SP) of the ALL type and a buying policy (BP) of the MP type (see Table 5.3), moreover it is supposed that $P_{MAX}$=$PP_{MAX}$ so always guaranteeing a successful purchase at the best price.




(a)




(b)

Figure 5.24: (a) Evaluation of TBTC for the five types of MCA with SP=ALL, BP=MP, NYPA=10 and variable NVA; (b) Zoom in of the TBTC curves of the PCA/CMA_BIN agents.

The results, obtained adopting a YPA organization in which the YPAs are logically connected as a binary tree, are reported in Figure 5.24(a-b) with NYPA=10 and varying NVA, where NYPA is the number of the YPA agents and NVA is the number of the VA agents.

The results show that the PCA outperforms the ICA and that the PCA/CMA_FP is the better solution from the point of view of time efficiency even though it suffers the resource consumption issues highlighted in Section 5.3.3.1. It is worth saying that the simulated PCA/CMA_FP is only an ideal implementation and that the obtained curve is a lower bound for a real implementation of the PCA/CMA_FP. Among the PCA/CMA_BINs, the PCA/CMA_BIN_FW_R2PCA exhibits better performance even though it could cause bottlenecking issues at the PCA site.

# 6 A Multi-Coordination based process for the design of mobile agent interactions

Code mobility paradigms have been introduced to support the design and the implementation of flexible, dynamic and reconfigurable distributed applications in terms of software components which are not confined in a single run-time context for their entire lifecycle but can migrate autonomously or on-demand across different contexts [52]. Among them, the most fascinating paradigm is represented by the mobile agents, executing software components capable of autonomous migration by retaining code, data and execution state. Although it is advocated that the exploitation of mobile agents can provide many benefits [68], they have introduced specific and not yet fully addressed issues that actually limit their advertised wide-spread use [109]. Among them, an interesting issue concerns with the design of mobile agent interactions. To deal with this issue several approaches have been to date defined which are mainly based on mobile agent interaction/coordination design patterns [1, 29] and coordination models [4, 15, 16, 17, 21, 78, 85, 86, 93] (refer to chapter 2 for a description of models and patterns). Recent interesting proposals on how to obtain and integrate agent-oriented coordination models are the programmable coordination spaces [97] and the multi-coordination approach [40]. The former proposal is based on the concept of Linda-like reactive tuple space in which the reactions can be programmed through a logic-based language (ReSpecT) so that already existing coordination models as well as new ones can be easily programmed. Conversely, the multi-coordination approach enables an integrated and simultaneous exploitation of multiple coordination models (both existing and to-be-defined) so that agents can choose among a variety of different coordination models which best fit their interaction needs. This can actually enhance design effectiveness, improve efficiency, and enable adaptability in dynamic and heterogeneous computing environments. All the aforementioned coordination models can be effectively used for supporting mobile agent interactions.

Although interaction patterns and coordination models can be jointly exploited for the design of mobile agent interactions, an automated design process which includes these techniques and produces effective and efficient design solutions (or coordination solutions) is still not available. To address this lack, this chapter proposes the Multi-Coordination based Process (MCP)

for the design of mobile agent interactions which, starting from a set of application-specific agent coordination requirements, produces an effective coordination solution by using two subsequent phases (Modeling and Evaluation).
To show a concrete application of the proposed process, a significant case study related to mobile agent-based distributed information retrieval is presented. In particular, alternative coordination solutions, which use different coordination models (asynchronous message passing, Linda-like tuple space, publish/subscribe), are produced on the basis of specific agent coordination requirements and evaluated against significant time- and resource-related performance indices.

## 6.1 The Multi-Coordination based Process (MCP)

The proposed Multi-Coordination based Process (MCP) [42] is iterative and consists of the two phases (Modeling and Evaluation) shown in Figure 6.1. The Modeling phase, on the basis of a coordination statement (CS) which derives from a preliminary analysis and includes a description of the agents along with their interactions (coordination requirements - CRs), and a set of coordination properties (CPs), provides alternative coordination solutions which fulfill the CS. In the Evaluation phase, a specific solution is chosen among such alternative coordination solutions which are evaluated through simulation and then compared on the basis of ad-hoc defined performance indices (e.g. time and resource consumption).

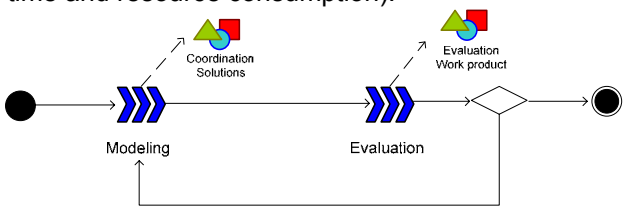



Figure 6.1: The MCP design process.

Each phase of the process is described in details and exemplified in the following two sections with reference to a simple yet effective case study concerning with a distributed information retrieval task in a distributed computing system. A possible solution consists in carrying out this task through a coordinated set (or task force) of mobile agents. In particular, a user can search for specific information over a network of federated information locations by creating and launching a task force of mobile agents (called searcher agents) onto different locations. As soon as the task force finds the desired information, the user (represented by the owner agent) is notified with the found information.

## 6.2 The Modeling Phase

The Modeling phase is detailed in Figure 6.2 and is composed by three subsequent activities (IP Selection and Setting, IP-CM Matching, Selection and Design of Coordination Solutions) described and exemplified with reference to the case study in the following correlated sub-sections.

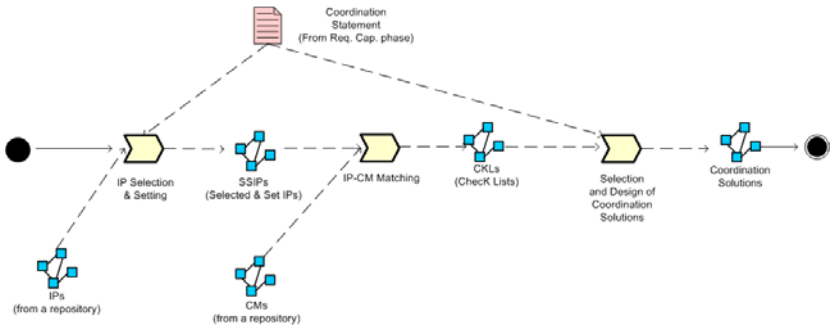



Figure 6.2: The Modeling phase.

### 6.2.1 IP Selection and Setting

Starting from the CS this activity identifies the rules governing the agent interactions by exploiting interaction design patterns (IPs). In particular, for each coordination requirement (CR), this activity (i) selects the IPs, from a given IP repository, which best fit the requirement, and (ii) sets the characteristics of each selected IP on the basis of the CPs so obtaining the set of selected and set IPs (SSIP) related to the requirement. In particular, CPs characterize the interactions among the agents as identified in the CRs on the basis of the following characteristics:

- Number of participants (PN), which can assume values in the range [2..N].
- Participant identity (PI), which concerns with the mutual knowledge among interacting agents. PI can therefore assume the values known or unknown.
- Locus (L), which indicates remote or local interactions among agents. L can assume the values local or remote.
- Temporality (T), which refers to the type of temporal coupling among interacting agents. T can assume two values: async for time decoupling and sync for time coupling.

The set of SSIPs, one SSIP for each coordination requirement, constitutes the result of this activity.

With reference to the case study, the proposed solution for the coordination of the task force during its information retrieval task is based on the following CRs:

- CR1: every time a searcher agent visits a location not yet searched by other agents of the same task force, it notifies the other agents that such

location has already been searched so avoiding unnecessary and resource-consuming duplicate searches;
- CR2: as soon as a searcher agent finds the desired information on a given location, it reports the found information to the owner agent;
- CR3: when a searcher agent finds the desired information on a given location, it signals such event to all the other searcher agents to stop them;

and on the following CPs:
- CPa: the task force is constituted by at least two searcher agents;
- CPb: the agents of the task force may or may not know each other whereas they know the identity of the owner agent and vice-versa;
- CPc: the interactions among all the agents (searcher and owner) are always asynchronous.
- CPd: the interactions required by CR1 may be local or remote, that required by CR2 and CR3 are remote.

Figure 6.3 shows, with reference to each CR, the selected IPs and the setting of their characteristics carried out by also taking into account the aforementioned CPs. In particular, the IPs selected from the repository for modeling the interactions as derived from the CRs are the following:
- Location-based notification (LBN), which involves agents passing through a given location to be notified about events occurring/occurred in such location.
- Report to owner (R2O), which involves a child agent reporting to its owner agent when its task is completed.
- Group-based notification (GBN), which involves an agent notifying all its peer agents when a given event occurs.

The star indicates that the value of the PI characteristic of the LBN and GBN IPs was not fixed according to the CPb.

| IP | IP CHARACTERISTICS | | | |
|---|---|---|---|---|
| | *PN* | *PI* | *L* | *T* |
| LBN | 2..N | * | LOCAL | ASYNC |
| GBN | 2..N | * | REMOTE | ASYNC |

(a) The SSIP for $CR_1$

| IP | IP CHARACTERISTICS/PROPERTIES | | | |
|---|---|---|---|---|
| | *PN* | *PI* | *L* | *T* |
| R2O | 2 | KNOWN | REMOTE | ASYNC |

(b) The SSIP for $CR_2$

| IP | IP CHARACTERISTICS/PROPERTIES | | | |
|---|---|---|---|---|
| | *PN* | *PI* | *L* | *T* |
| GBN | 2..N | * | REMOTE | ASYNC |

(c) The SSIP for $CR_3$

Figure 6.3: The result of the IP Selection and Setting activity.

## 6.2.2 IP-CM Matching

Starting from both a set of coordination models (CMs) and the set of the SSIPs, as it results from the previous activity (see Figure 6.2), this activity,

for each CR and characterized IP in the associated SSIP, and for each CM constructs a checklist (CKL) which specifies the characteristics of the IP supported by the considered CM. In presence of a not completely characterized IP (i.e. an IP with one or more not fixed characteristics), different complete characterizations of the IP are to be considered (one for each possible combination of the values of the not fixed characteristics).
The set of the so obtained CKLs constitutes the result of this activity.
With reference to the case study and to the result of the previously described activity (see Figure 6.3), the obtained CKLs, one for each CR, are reported in Figure 6.4. In particular, the considered CMs are the following:

- Local Linda-like tuple space (LTS), which supports a high number of participants, allows temporal decoupling but only local interaction is supported [40].
- Topic-based publish/subscribe (TPS), which supports a high number of participants, allows for distributed interactions and does not require temporal coupling between participants [71].
- Queue–based unicast asynchronous message passing (QAMP), which supports a variable number of participants, allows for both local and remote interactions, does not require temporal coupling, but requires spatial coupling among participants [115].

With reference to the PI characteristic, which was not fixed for the LBN and GBN IPs, in Figure 6.4.a-c all the different complete characterizations of the IPs are considered.

### 6.2.3 Selection and Design of Coordination Solutions

This activity partitions the set of the obtained CKLs in subsets (candidate solutions) which are obtained by selecting for each CR one and only one related CKL, and selects the subsets which satisfy specific admissibility and optimality criteria. In particular, the admissibility criteria are based on the CPs whereas the optimality criteria are based on the support of the IP characteristics by the considered CM.
Finally, from each selected subset a coordination solution is provided by implementing the related coordination models.
The so obtained coordination solutions constitute the result of this activity and of the whole modeling phase (see Figure 6.5).
With reference to the case study, the selected criteria are:

- Admissibility. Only the solutions which have the same value of the PI characteristics for the IPs associated to CR1 and CR3 are admissible.
- Optimality. The selected solutions are those in which the considered CM fully supports the IP (the related CKL is completely set with ‘x’).

| CHARACTERIZED IP | CM | SUPPORTED IP CHARACTERISTICS | | | |
|---|---|---|---|---|---|
| | | *PN* | *PI* | *L* | *T* |
| LBN [2..N, *known*, local, async] | LTS | x | | x | x |
| | TPS | x | | | x |
| | QAMP | x | x | x | x |
| LBN [2..N, *unknown*, local, async] | LTS | x | x | x | x |
| | TPS | x | x | | x |
| | QAMP | x | | x | x |
| GBN [2..N, *known*, remote, async] | LTS | x | | | x |
| | TPS | x | | x | x |
| | QAMP | x | x | x | x |
| GBN [2..N, *unknown*, remote, async] | LTS | x | x | | x |
| | TPS | x | x | x | x |
| | QAMP | x | | x | x |

(a) The CKLs for $CR_1$

| CHARACTERIZED IP | CM | SUPPORTED IP CHARACTERISTICS | | | |
|---|---|---|---|---|---|
| | | *PN* | *PI* | *L* | *T* |
| R2O [2, known, remote, async] | LTS | x | | | x |
| | TPS | x | | x | x |
| | QAMP | x | x | x | x |

(b) The CKLs for $CR_2$

| CHARACTERIZED IP | CM | SUPPORTED IP CHARACTERISTICS | | | |
|---|---|---|---|---|---|
| | | *PN* | *PI* | *L* | *T* |
| GBN [2..N, *known*, remote, async] | LTS | x | | | x |
| | TPS | x | | x | x |
| | QAMP | x | x | x | x |
| GBN [2..N, *unknown*, remote, async] | LTS | x | x | | x |
| | TPS | x | x | x | x |
| | QAMP | x | | x | x |

(c) The CKLs for $CR_3$

Figure 6.4: The result of the *IP-CM Matching* activity.

On the basis of the obtained results, it is possible to refine and/or modify the choices made in the activities of the modeling phase; in particular: (i) other IPs and CMs can be chosen and/or defined to be subsequently used for the IP Selection and Setting and IP-CM Matching activities; (ii) new criteria of admissibility and optimality can be defined in the Selection and Design of Coordination Solutions activity for generating a new set of alternative coordination solutions; (iii) different implementation descriptions for the indentified coordination solutions can be given.

| CR | CHARACTERIZED IP | CM | IMPLEMENTATION DESCRIPTION |
|---|---|---|---|
| $CR_1$ | LBN [2..N, *known*, local, async] | QAMP | When a searcher agent searches in a location which has not been already searched by another agent of its task force, it clones itself so that its clone, stationing in this location, can prevent the other agents of its task force from searching. As soon as an agent visits a location looks for a clone of an agent of its task force to avoid searching. |
| $CR_2$ | R2O [2, *known*, remote, async] | QAMP | When a searcher agent finds the desired information, it sends a message containing the found information to its owner. |
| $CR_3$ | GBN [2..N, *known*, remote, async] | QAMP | A searcher agent which has found the desired information sends a notification message to all the other searcher agents of the task force to stop them. |

(a)

| CR | CHARACTERIZED IP | CM | IMPLEMENTATION DESCRIPTION |
|---|---|---|---|
| $CR_1$ | GBN [2..N, *known*, remore, async] | QAMP | A searcher agent to notify that it has searched a given location sends a message containing the location identifier to all the other searcher agents of the task force. |
| $CR_2$ | R2O [2, *known*, remote, async] | QAMP | When a searcher agent finds the desired information, it sends a message containing the found information to its owner. |
| $CR_3$ | GBN [2..N, *known*, remote, async] | QAMP | A searcher agent which has found the desired information sends a notification message to all the other searcher agents of the task force to stop them. |

(b)

| CR | CHARACTERIZED IP | CM | IMPLEMENTATION DESCRIPTION |
|---|---|---|---|
| $CR_1$ | LBN [2..N, un*known*, local, async] | LTS | When a searcher agent searches in a location which has not been already searched by another agent of its task force, it inserts a signaling tuple into the LTS to signal that this location has been searched. As soon as an agent visits a location and reads the signaling tuple, it avoids searching. |
| $CR_2$ | R2O [2, *known*, remote, async] | QAMP | When a searcher agent finds the desired information, it sends a message containing the found information to its owner. |
| $CR_3$ | GBN [2..N, un*known*, remote, async] | TPS | When a searcher agent finds the desired information, it publishes an event of a specific topic related to its task force which signals the stop of the retrieval task. All the other agents of the task force will be thus asynchronously notified since they subscribed to the specific topic at creation time. |

(c)

| CR | CHARACTERIZED IP | CM | IMPLEMENTATION DESCRIPTION |
|---|---|---|---|
| $CR_1$ | GBN [2..N, un*known*, remote, async] | TPS | A searcher agent to notify that it has searched a given location publishes an event of a specific topic related to its task force and containing the location identifier. All the other agents of the task force will be thus asynchronously notified since they subscribed to the specific topic at creation time. |
| $CR_2$ | R2O [2, *known*, remote, async] | QAMP | When a searcher agent finds the desired information, it sends a message containing the found information to its owner. |
| $CR_3$ | GBN [2..N, un*known*, remote, async] | TPS | When a searcher agent finds the desired information, it publishes an event of a specific topic related to its task force which signals the stop of the retrieval task. All the other agents of the task force will be thus asynchronously notified since they subscribed to the specific topic at creation time. |

(d)

Figure 6.5: The result of the *Selection and Design of Coordination Solutions* activity

## 6.3 The Evaluation phase

In this phase (see Figure 6.6) the alternative coordination solutions identified in the modeling phase are evaluated through simulation and then compared on the basis of ad-hoc defined performance indices.

The phase is currently supported by the proposed methodology and related tools which provide rapid prototyping of the coordination solutions and their simulation. In particular, the ELDA model directly supports the concept of multi-coordination by providing multiple coordination models through which the interactions among the agents can be easily modeled. Moreover, its related tools support the visual programming of the modeled coordination solution and the automatic generation of code which can be directly executed by a ELDASim (see Section 3.2, Section 3.3 and Section 3.4).

The Evaluation phase starts with two parallel activities:

- Simulation Model Definition, which provides an ELDA-based simulation model for each considered alternative coordination solution.
- Performance Indices Definition, which defines specific performance indices for evaluation and comparison purposes also taking into account the CRs and CPs.

In particular, the defined ELDA-based simulation models are subsequently implemented (Code Generation and Simulation Implementation activities) and simulated (Simulation Execution activity) for being evaluated and compared with reference to the identified performance indices. The simulation results therefore allow to identify the best coordination solution.

On the basis of the obtained results, it is possible to refine and/or modify the choices made in the activities of the modeling phase; in particular: (i) performance indices can be modified and/or new performance indices can be defined in the Performance indices Definition activity; (ii) agent-based simulation models can be refined in the Simulation Model definition activity; (iii) new simulator programs can be defined and executed.

In the following subsection an example of evaluation with reference to the case study is presented.

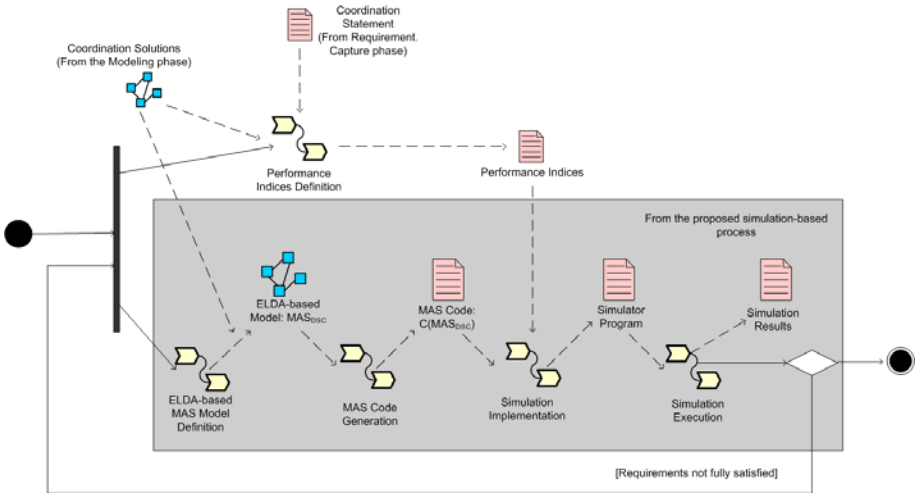



Figure 6.6: The Evaluation phase.

### 6.3.1 Performance Evaluation of Coordination Solutions: an example

With reference to the case study, in the Performance Indices Definition activity the following performance indices have been defined:

- Task completion time ($T_{TC}$): the temporal gap between the spawning of the first created searcher agent and the first report message received from the owner agent.
- Number of coordination messages (NM): the number of coordination messages transmitted through the network.
- Notification time ($T_N$): the temporal gap between the information finding and the notification to the last searcher agent.
- Number of visits ($N_V$): after finding the information: the total number of locations visited by the searcher agents after the information finding.
- Number of searches ($N_S$): after finding the information: the total number of the locations searched by the searcher agents after the information finding.

To exemplify the Evaluation phase, all the alternative coordination solutions reported in Figure 6.5 have been implemented and integrated into a simulator program along with the calculation of the defined performance indices; in the following, solutions reported in Figure 6.5a, 6.5b, 6.5c and 6.5d will be named A, B, C, D solutions, respectively.

The Simulation Execution activity relies on two simulation parameters (the number of locations and the number of searcher agents) and on the following settings of the simulation topology at network and information level:

- Locations are connected through a fully connected logical network composed of FIFO channels. In particular, channels are characterized by

the same delay and bandwidth parameters modeled as uniform random variables.

- The information to be found is contained exactly at one location and the locations keep references (randomly generated) to other locations at information level to be all reachable.

In particular, simulation runs are carried out with the number of locations equals to 100 and the number of searcher agents in the range [2-20]. Moreover, for each simulation run, all the alternative solutions are executed on the same network and information topologies. In Figures 6.7-6.11 the simulation results are reported; the obtained values of the performance indices are averaged over 50 simulation runs.

The $T_{TC}$ performance index, which measures the speed with which the information search task is carried out, decreases as the number of searcher agents increases (see Figure 6.7).

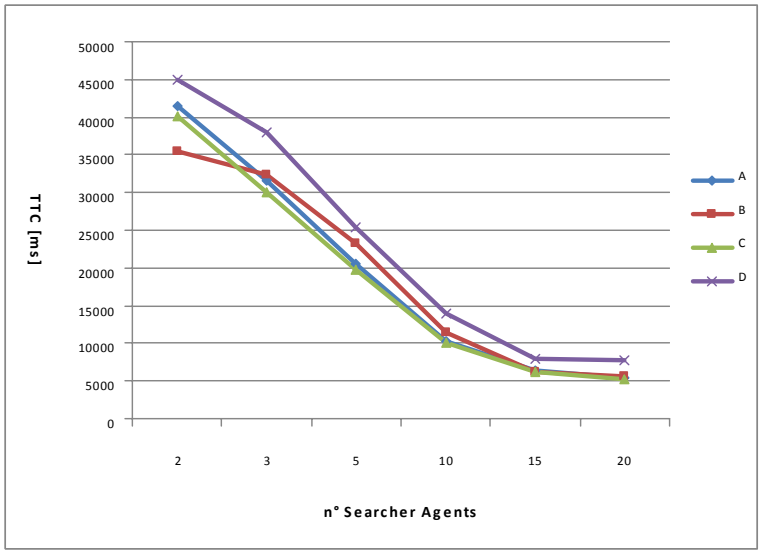



Figure 6.7: The Task Completion Time.

In fact, the use of more searcher agents augments the degree of parallelism which, consequently, increases the probability to find the searched information with a smaller number of migrations which are time-consuming. The performances of the all the solutions are almost the same.

The $N_M$ parameter (see Figure 6.8), which measures the network load, is significantly better in the A and C solutions thus saving network resources with respect to the other solutions.

The $T_N$ performance index measures how fast all the searcher agents are notified after finding the information: the shorter $T_N$, the fewer are the resources consumed throughout the networked agent platform.

The A and C solutions outperform the other solutions (see Figure 6.9) as the network load is less heavy than the ones of the B and D solutions.

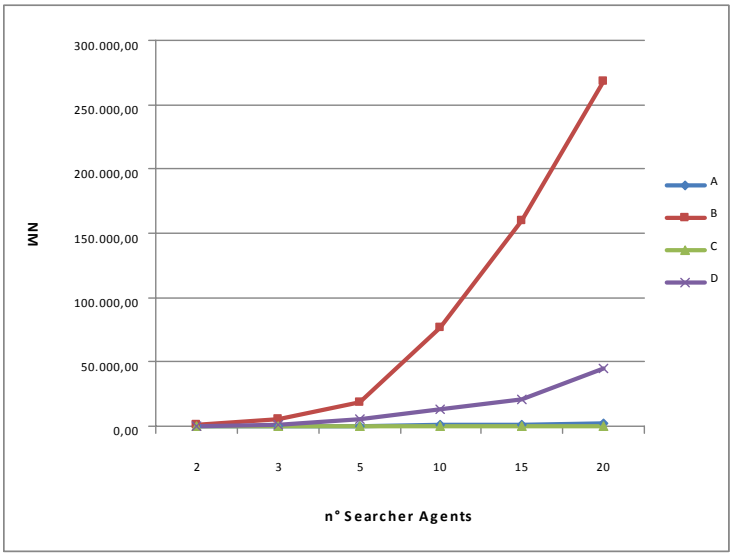



Figure 6.8: The Number of coordination messages.

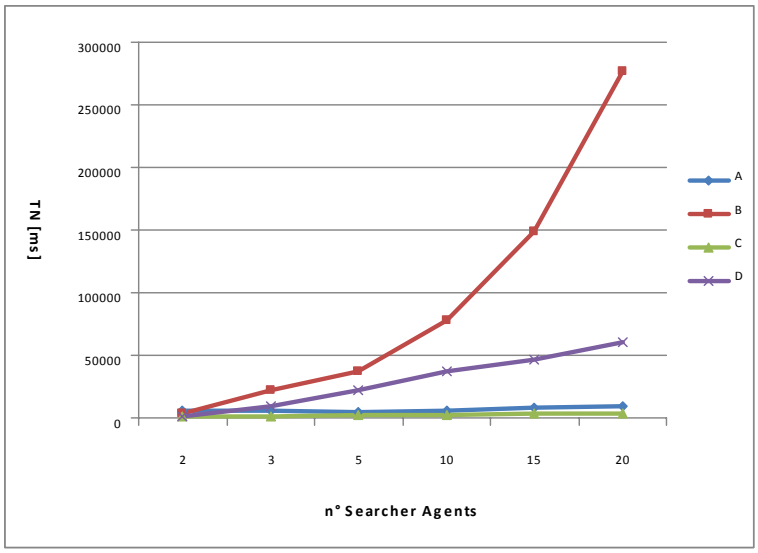



Figure 6.9: The Notification Time.

The $N_V$ and $N_S$ parameters are measures of the consumption of resources after the information is found. The values of such parameters should be kept as low as possible. As shown in Figures 6.10 and 6.11, the A and C solutions outperform the other solutions.

On the basis of the obtained simulation results (see Figures 6.7-6.11), the

solutions A and C are more effective than the other solutions. In fact, although the TTC value is similar to the solutions B and D, the other performance indices values are significantly better. Moreover, it's worth noting that solution A requires the cloning of a signalling agent for each visited location but such operation may not be allowed according to security policies of the locations.

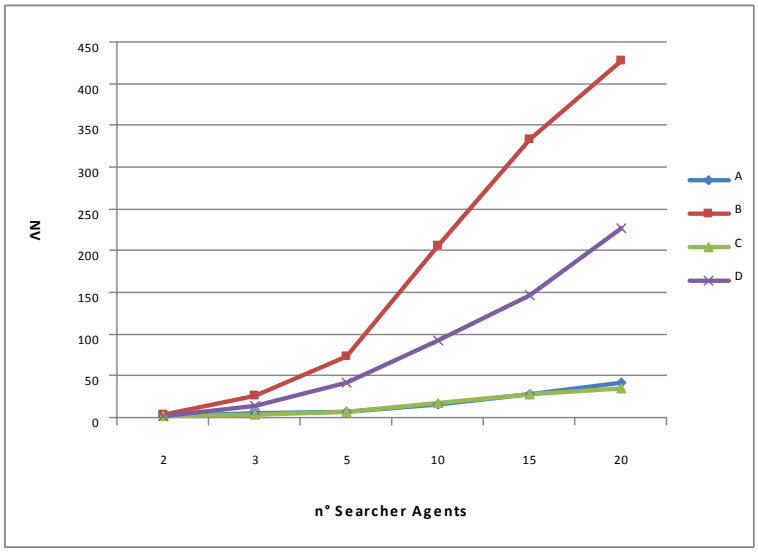



Figure 6.10: The Number of visits after finding information.

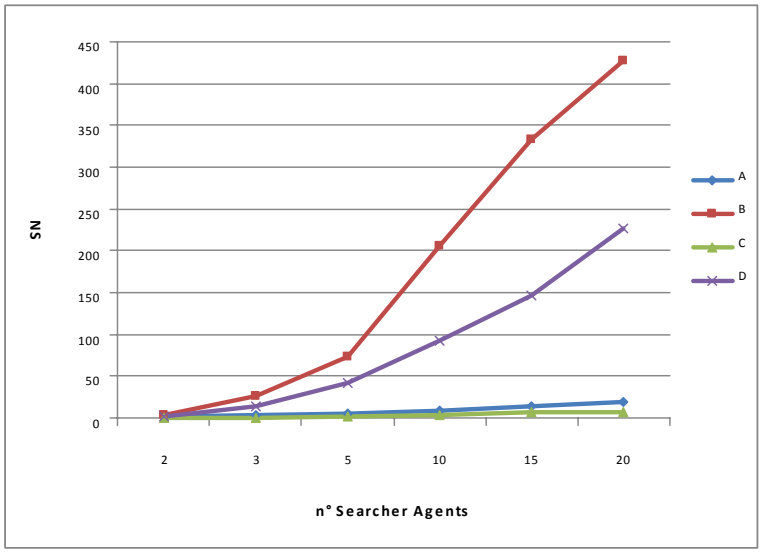



Figure 6.11: The Number of searches after finding information.

# 7 Conclusion and Future Work

## 7.1 Summary

Internet-based distributed applications need development methodologies able to capture their key characteristics through powerful abstractions at modelling, validation and implementation levels. The agent oriented software engineering (AOSE) has promoted effective agent-oriented models, frameworks and methodologies to fulfil the requirements posed by new kinds of distributed applications emerging in a variety of challenging application domains, e.g. e-Commerce, content delivery, information retrieval, pervasive computing.

This thesis has proposed ELDAMeth, a novel methodology supported by a CASE tool for the simulation-based prototyping of Internet-oriented distributed agents systems (DAS). Although a significant number of general-purpose and domain-specific agent-oriented methodologies have been already proposed, ELDAMeth incorporates important and distinctive key features for an effective prototyping of Internet-oriented DAS. Such features refer to the reference agent model, the defined methodology and the supporting CASE tool.

The defined ELDA agent model incorporates the three main enabling features for distributed agent systems: lightweight reactive/proactive behavior, multi-coordination and mobility. In particular: (i) reactive/proactive agent behavior based on lightweight architectures has been demonstrated to be particularly suitable to model reactive and/or proactive components in large-scale, dynamic and distributed environments; (ii) multi-coordination has been experimentally recognized as a new key feature enabling effective design and efficient execution of complex interactions among distributed agents; (iii) mobility has emerged as an enabling feature for defining new distributed algorithms for code and computation dissemination.

The proposed ELDAMeth is characterized by two important features: effective dynamic validation based on simulation and high-degree of integrability. The first feature, which is indeed the main distinctive feature of ELDAMeth, supports the validation of designed distributed agent systems in a simulated controlled environment to analyze both functional and performance-oriented requirements. The second feature, which partly relates to the first feature, allows using ELDAMeth to empower already existing

agent-oriented methodologies or to create new ad-hoc ones with a very effective validation phase before implementation and deployment.
Finally, the developed ELDATool fully supports ELDAMeth during all the prototyping process, from modelling to simulation. In particular, ELDATool provides highly effective visual support to the modelling of agent behaviors and subsequently automatic translation into code so minimizing programming errors and speeding up the prototyping process. Moreover, it supports the simulation configuration phase through general-purpose and case-specific graphical windows, the simulation execution which can be controlled (started, paused, and stopped) by a control panel, and the storing of the execution traces in an RDBMS.
ELDAMeth has been applied according to different perspectives: methodology-oriented and application-oriented.
In the former case ELDAMeth has been integrated with PASSI to obtain a full-fledged agent-oriented methodology, namely PASSIM, and has been used to create a new methodology for designing mobile agent interactions, namely MCP.
In the latter case ELDAMeth has been directly used to prototype distributed agent systems such as e-Marketplaces, architectures of surrogates for content delivery, and information retrieval systems. These different applications have demonstrated the suitability and great effectiveness of ELDAMeth for the rapid prototyping of Internet-based DAS.

## *7.2 Future Work*

A number of future research directions in relation to this thesis can be devised. In particular, the following three research directions will be investigated:
- *Enhancement of the ELDAMeth with an ad-hoc implementation/deployment phase after simulation*. Two approaches can be envisaged: (i) ELDAPlatform-oriented and (ii) Model-driven development. According to the former approach, the ELDAPlatform should be developed as a new agent platform able to execute ELDA agents programmed through the ELDAFramework. According to the second approach, the ELDA-based models, or PIMs (Platform-Independent Models) in the MDD (Model-Driven Development) language,should be converted into agent models specific to a target platform or PDMs (Platform Dependent Models). In particular, JADE will be considered as executing target platform due to its wide diffusion in academic and industrial contexts.
- *Extension of the ELDA agent model with the new agent-oriented concepts of Organization and Environment*. The ELDA model currently does not exploit interesting agent oriented concepts such as Organization and Environment. Nevertheless, these concepts could be seamlessly introduced as new types of spaces or meta-spaces apart from the already existing system and coordination spaces.

- *Formalization of the ELDA agent model to validate agent-based systems through formal methods*. The objective is to provide a formalization of the ELDA model through term rewriting and use rewriting logic and related tools (e.g. Maude) to validate ELDA-based models of agent systems.

# References


[1] Aridor, Y., and Lange, D., Agent Design Patterns: Elements of Agent Application Design, Second Intl. Conference on Autonomous Agents, IEEE, 1998.

[2] Astley, M., and Agha, G. A., Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management, ACM SIGSOFT 6th International Symposium on Foundations of Software Engineering (FSE), 1998.

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

[4] Baumann, J., Hohl, F., Radouniklis, N., Rothermel, K. and Strasser, M., Communication concepts for Mobile Agent Systems, 1st International Workshop on Mobile Agents (MA'97), Berlin, Germany, LNCS 1219, pp. 123-135, April 1997.

[5] Baumann, J., Hohl, F., Rothermel, K., Straßer, M., Mole - Concepts of a Mobile Agent System", The World Wide Web Journal, Vol. 1, No. 3, pp. 123-137, 1998

[6] Bellifemine, F. L., Claire, G., and Greenwood, D., Developing Multi-Agent Systems with JADE, Wesley, 2005.

[7] Bellifemine, F., Poggi, A., and Rimassa, G., Developing multi agent systems with a FIPAcompliant agent framework. Software Practice And Experience, 31, pp 103–128, 2001.

[8] Bernon, C., Cossentino, M., and Pavon, J. An Overview of Current Trends in European AOSE Research. Informatica 29, pp 379—390, 2005.

[9] Bernon, C., Cossentino, M., and Pavón, J., Agent Oriented Software Engineering, Knowledge Engineering Review, Vol.20, Issue 02, pp. 99-116, 2005.

[10] Boloni, L., and Marinescu, D. C., A multi-plane state machine agent model, Fourth International Conference on Autonomous Agents, Barcelona, Spain, pp. 80-81, ACM Press, 2000.

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

[12] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A., TROPOS: An Agent-Oriented Software Development Methodology, Journal of Autonomous Agents and Multi-Agent Systems, 8, 3, pp 203–236, 2004.

[13] Brinkkemper, S., Lyytinen, K. and Welke, R., Method engineering: Principles of method construction and tool support, International Federation for Information Processing, 1996.

[14] Buyya, R., Pathan, M., and Vakali, A., Content Delivery Networks: Principles and Paradigms, Lecture Notes Electrical Engineering, Vol. 9, Ch. 12, Springer, Aug, 2008.

[15] Cabri, G., Leonardi, L., and Zambonelli, F., Engineering Mobile Agent Applications via Context-dependent Coordination, IEEE Transactions on Software Engineering, 28, 11, pp 1040-1056, Nov. 2002.

[16] Cabri, G., Leonardi, L.., and Zambonelli, F., Mobile-agent coordination models for internet applications, IEEE Computer, 33, 2, pp 82-89, 2000.

[17] Cao, J., Feng, X. and Das, S.K., Mailbox-Based Scheme for Mobile Agent Communications, Computer 35, 9, pp. 54–60, 2002.

[18] Cardelli, L., and Gordon, D., Mobile Ambients, Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science, No. 1378, Springer-Verlag (D), pp. 140-155, 1998.
[19] Carzaniga, A., Rosenblum, D.S., and Wolf, A., Design and evaluation of a wide-area event notification service, ACM Transactions on Computer Systems, 19, 3, pp 332-383, 2001.
[20] Chella, A., Cossentino, M. and Sabatucci, L., Designing Jade systems with the support of case tools and patterns, Exp Journal, 3, 3, pp 86-95, 2003.
[21] Choi S., Kim, H., Byun, E., Hwang, C., and Baik, M., Reliable Asynchronous Message Delivery for Mobile Agents, IEEE Internet Computing, vol. 10, no. 6, pp. 16-25, 2006.
[22] Ciancarini, P., Coordination models and languages as software integrators, ACM Computing Surveys, 28, 2, pp 300-302, Jun 1996.
[23] Cossentino, M., From Requirements to Code with the PASSI Methodology, Agent-Oriented Methodologies, B. Henderson-Sellers and P. Giorgini (eds). Idea Group Inc., Hershey, PA, USA, 2005.
[24] Cossentino, M., Fortino, G., Garro, A., Mascillaro, S., and Russo W., PASSIM: A Simulation-based Process for the Development of Multi-Agent Systems, International Journal of Agent Oriented Software Engineering, Vol. 2, n. 2, pp. 132-170, 2008.
[25] Cossentino, M., Garro, A., Gaglio, S. and Seidita, V., Method fragments for agent design methodologies: from standardization to research, International Journal of Agent Oriented Software Engineering, 1, 1, pp 91-121, 2007.
[26] Cossentino, M., Sabatucci, L. and Chella, A., A Possible Approach to the Development of Robotic Multi-Agent Systems, IEEE/WIC Conf. on Intelligent Agent Technology (IAT'03), Halifax (Canada), 2003.
[27] Cugola G., Di Nitto, E., and Fuggetta, A., The Jedi event-based infrastructure and its application to the development of the OPSS WFMS, IEEE Transactions on Software Engineering, 27, 9, pp 827-850, 2001.
[28] da Silva, J. L. T. and Demazeau, Y., Vowels co-ordination model, First international joint conference on Autonomous Agents and Multi-Agent systems (AAMAS '02), pages 1129-1136, New York, USA, ACM Press, 2002.
[29] Deugo, D., Weiss, M., and Kendall, E., Reusable Patterns for Agent Coordination, Coordination of Internet Agents: Models, Technologies, and Applications, A. Omicini, F. Zambonelli, M. Klusch and R. Tolksdorf, Eds. Springer, Ch. 14, 2001.
[30] Eclipse - an open development platform, documentation and software, available at the World Wide Web: http://www.eclipse.org.
[31] Eclipse Modeling Framework Project (EMF), documentation and software, available at the World Wide Web: http://www.eclipse.org/modeling/emf/.
[32] ELDATool, documentation and software, http://lisdip.deis.unical.it/software/eldatool.
[33] Feldman, S., Electronic marketplaces, IEEE Computing, 4, Jul-Aug, pp 93-95, 2000.
[34] FIPA ACL Message Structure Specification, A description of the structure of FIPA ACL, http://www.fipa.org/specs/fipa00061/SC00061G.html.
[35] FIPA Agent Management Specification, Management for agents on FIPA agent platforms, http://www.fipa.org/specs/fipa00023/SC00023K.html.
[36] FIPA RDF Content Language Specification, A description of a FIPA content language based on the Resource Description FrameworkDocument FIPA XC00011B (2001/08/10). http://www.fipa.org/specs/ fipa00011/XC00011B.html.
[37] Fortino G., Garro, A., Mascillaro, S., and Russo, W., ELDATool: A Statecharts-based Tool for Prototyping Multi-Agent Systems, Workshop on Objects and Agents (WOA'07), Genoa, 24-25 September, 2007.
[38] Fortino, G, Garro, A. and Russo, W., A Discrete-Event Simulation Framework for the Validation of Agent-based and Multi-Agent Systems, Workshop on Objects and Agents (WOA'05), Camerino, Italy, Nov 14-16, 2005.
[39] Fortino, G. and Russo, W., Using P2P, GRID and Agent Technologies for the Development of Content Distribution Networks, Future Generation Computer Systems, doi:10.1016/j.future.2007.06.007, 2008.
[40] Fortino, G., and Russo, W., Multi-coordination of Mobile Agents: a Model and a Component-based Architecture, 20th Annual ACM Symposium on Applied Computing

(SAC'05), Special Track on Coordination Models, Languages and Applications, Santa Fe, NM, USA, Mar. 13-17, 2005.

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

[42] Fortino, G., Garro, A., Mascillaro, S., and Russo, W., A Multi-Coordination based Process for the Design of Mobile Agent Interactions, IEEE Symposium on Intelligent Agents, Nashville (TN), USA, March 30-April 2, 2009.

[43] Fortino, G., Garro, A., Mascillaro, S., and Russo, W., Agent-based Modeling and Simulation of Cooperative Content Distribution Networks, 2nd Int'l Workshop on Multi-Agent Systems and Simulation (MAS&S'07), St. Julian, Malta, 22-24 October, 2007, as part of the EUROSIS European Simulation and Modeling Conference.

[44] Fortino, G., Garro, A., Mascillaro, S., and Russo, W., Modeling Multi-Agent Systems through Event-driven Lightweight DSC-based Agents, 6th Int'l Workshop "From Agent Theory to Agent Implementation" (AT2AI'06), Estoril, Portugal, 13 May, 2008, held at the 7th Autonomous Agents and Multi-agent Systems (AAMAS).

[45] Fortino, G., Garro, A., Mascillaro, S., and Russo, W., Specifying WSN Applications through Agents Based on Events and States, IARIA/IEEE Int'l Conference SensorComm'07, Valencia, Spain, 14-20 October, 2007.

[46] Fortino, G., Garro, A., Mascillaro, S., and Russo, W., Using Event-driven Lightweight DSC-based Agents for MAS Modeling, International Journal of Agent Oriented Software Engineering, Vol. 4, n. 1, pp. 1-30, 2010.

[47] Fortino, G., Garro, A., Mascillaro, S., and Russo, W., Using multi-coordination for the design of mobile agent interactions, Workshop on Objects and Agents (WOA'08), Palermo, Italy, 17-18 November, 2008.

[48] Fortino, G., Garro, A., Mascillaro, S., Russo, W., and Vaccaro, M., Distributed architectures for surrogate clustering in CDNs: a simulation-based analysis, 4th ACM/HPDC Int'l Workshop on the Use of P2P, GRID and Agents for the Development of Content Networks (UPGRADE-CN'09), , ACM Press, Munich (Germany), 9 June, 2009.

[49] Fortino, G., Mastroianni, C., and Russo, W., A Hierarchical Control Protocol for Group-Oriented Playbacks Supported by Content Distribution Networks, Journal of Network and Computer Applications, Elsevier, 32, 1, pp. 135-157, 2009.

[50] Fortino, G., Russo, W. and Zimeo, E., A Statecharts-based Software Development Process for Mobile Agents, Information and Software Technology, 46, 13, pp.907-921, Elsevier, Amsterdam, The Netherlands, 2004.

[51] Fox A., Gribble, S.D. Chawathe, Y., Brewer, E.A., and Gauthier, P., Cluster-Based Scalable Network Services, 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, Oct. 5-8, 1997.

[52] Fuggetta, A., Picco, G.P., and Vigna, G., Understanding Code Mobility, IEEE Transaction. on Software Engineering, 24, 5, pp 342–361, 1998.

[53] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[54] Gardelli, L., Viroli, M. and Omicini, A., On the Role of Simulation in the Engineering of Self-Organising Systems: Detecting Abnormal Behaviour in MAS, Workshop on Objects and Agents (WOA'05), Camerino (Italy), pp. 85-90, 2005.

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

[56] Griss, M. L, Fonseca, S., Cowan, D., and Kessler, R., SmartAgent: Extending the JADE Agent Behavior Model, ACM Press, 2003.

[57] Guttman, R. H., Moukas, A. G. and Maes, P., Agent-mediated electronic commerce: a survey, The Knowledge Engineering Review, 13, pp 147-159, 1998.

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

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

[60] Harmsen, F. and Brinkkemper, S., Design and Implementation of a Method Base Management System for a Situational CASE Environment, 2nd Asia-Pacific Software Engineering Conference (APSEC'95), IEEE CS Press, Brisbane, pp. 430-438, 1995.

[61] Henderson-Sellers, B., Method Engineering for OO Systems Development, Communications of the ACM, 46, 10, pp.73-78, 2003.
[62] Jennings, N. R., On Agent-Based Software Engineering, Artificial Intelligence Journal, 117, 2, pp 277-296, 2000.
[63] Jensen, A. J., Kasper, H., and Demazeau, Y., Reactive agent mechanisms for scheduling manufacturing processes, 6th Int. Workshop From Agent Theory to Agent Implementation (AT2AI-6), eds. Jung, Michel, Ricci & Petta, Estoril, Portugal. May 13, 2008.
[64] Kendall, E., Murali Krishna, P., Pathak, C. and Suresh, C.B., Patterns of Intelligent and Mobile Agents, Second Intl. Conference on Autonomous Agents, IEEE, 1998
[65] Kendall, E., Role Models: Patterns of Agent System Analysis and Design, Agent Systems and Applications/Mobile Agents (ASA/MA-99), ACM, 1999
[66] Kessler, R., Griss, M., Remick, B., and Delucchi, R., A hierarchical State machine using JADE behaviours with animation visualization, Int. Conf. on Autonomus Agents and Multi Agents Systems, New York City, 19-23 July, 2004.
[67] Kolp, M., Giorgini, P., and Mylopoulos, J., A Goal-Based Organizational Perspective on Multi-Agent Architectures, Eighth Intl. Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), 2001.
[68] Lange, D.B., and Oshima, M., Seven good reasons for Mobile Agents, Communications of the ACM, 42, 3, pp 88-89, 1999.
[69] Leung, C.S., Sum, J., Shen, H., Wu, J. and Young, G., Analysis and Design of an Agent Searching Algorithm for e-Marketplaces, Cluster Computing, 7, pp 85-90, 2004.
[70] Lilius, J. and Paltor, I. P., The semantics of UML State Machines. Technical Report N. 273. Turku Centre of Computer Science (TUCS), 1999.
[71] Loke, S. W., Padovitz, A., Zaslavsky, A., and Tosic, M., Agent Communication Using Publish-Subscribe Genre: Architecture, Mobility, Scalability and Applications, Annals of Mathematics, Computing & Teleinformatics, 1, 2, pp 35-50, 2004.
[72] Luck, M., McBurney, P., and Preist, C., A Manifesto for Agent Technology: Towards Next Generation Computing. Autonomous Agents and Multi-Agent Systems, 9, 3, pp 203-252, 2004.
[73] Luck, M., McBurney, P., Shehory, O., Willmott, S., Agent technology roadmap (A Roadmap for agent based computing). AgentLink, 2005.
[74] Maes, P., Guttman, R. and Moukas, A., Agents that Buy and Sell: Transforming Commerce as we Know It, ACM Communications of ACM, 42, 3, pp 81-91, 1999.
[75] Marinescu, D.C., Internet-based Workflow Management, New York: John Wiley & Sons, Inc, 2002.
[76] Martelli M., Mascardi, V. and Zini, F., Specification and Simulation of Multi-Agent Systems in CaseLP, Appia-Gulp-Prode Joint Conf. on Declarative Programming, L'Aquila, Italy. M.C. Meo and M. Vilares-Ferro (eds), pp. 13-28, 1999.
[77] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A., Handbook of Applied Criptography, CRC Press, 1997.
[78] Murphy, A., Picco, G. P., and Roman, G., LIME: A Middleware for Logical and Physical Mobility, International Conference on Distributed Computing Systems, IEEE CS, 2001.
[79] Ni, J., and Tsang, D.H.K., Large-Scale Cooperative Caching and Application-Level Multicast in Multimedia Content Delivery Networks, IEEE Communications, 43, 5, pp.98-105, May, 2005.
[80] North, M. J., Collier, N.T., and Vos, J.R., Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit, ACM Transactions on Modeling and Computer Simulation, 16, 1, January, pp 1–25, 2006.
[81] Nwana, H.S., Software Agents: an overview, Knowledge Engineering Review, 11, 3, pp 205–244, 1996.
[82] Object Management Group, Unified Modelling Language Specification v. 2.0, 2005 (N. formal/2005-07-05).
[83] Omicini, A. and Zambonelli, F., A Coordination of Mobile Agents for Information Systems: the TuCSoN Model, AI*IA'98 Workshop on Knowledge Integration, Padova, Italy, 1998.

[84] Omicini, A., and Zambonelli, F., Challenges and Research Directions in Agent Oriented Software Engineering, Autonomous Agents and Multi-Agent Systems, 9(3), pp. 253-284, 2004.
[85] Omicini, A., and Zambonelli, F., Coordination for Internet application development, Autonomous Agents and Multi-Agent Systems, 2, 3, pp 251–269, Sept. 1999.
[86] Omicini, A., Ossowsky, S., and Ricci, A., Coordination infrastructures in the engineering of multi-agent systems, Methodologies and Software Engineering for Agent Systems, New York, 2004, Kluwer.
[87] Padovitz, A., Agent communication using Publish-Subscribe genre: Architecture, Mobility, Scalability and Applications, Annals of Mathematics, Computing and Teleinformatics, 1, 3, pp 35-50, 2004.
[88] Papadoupolos, G.A., and Arbab, F., Coordination models and languages, Advances in Computers 46, Academic Press, 1998.
[89] Pavon, J., and Gómez-Sanz, J., Agent Oriented Software Engineering with INGENIAS, Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03) (Lecture Notes in Computer Science, 2691). Berlin: Springer, pp. 394-403, 2003
[90] Pavon, J., Sansores, C., and Gomez-Sanz, J., Modeling of Social Systems with Ingenias, 1st Workshop on Multi-Agent Systems and Simulation (MAS&S'06), Palermo, Italy, 2006.
[91] Peine, H., Application and Programming Experience with the Ara Mobile Agent System, Software Practice and Experience, 32, 6, pp 515-541, 2002.
[92] Peng, G., CDN: Content Distribution Network, Technical Report TR-125, Experimental Computer Systems Lab, Stony Brook University, 2003.
[93] Picco, G. P., Murphy, A. L., and Roman, G. C., LIME: Linda meets mobility, ACM Press, 1999.
[94] Ralyté, J. and Rolland, C., An assembly process model for method engineering, 13th Conference on Advanced Information Systems Engineering (CAISE01), Interlaken, (Switzerland), pp.267-283, 2001.
[95] Raman, S., and McCanne, S., A model, analysis, and protocol framework for soft state-based communication, ACM SIGCOMM Computer Communication Review, 29(4), pp. 15-25, 1999.
[96] RDF (Resource Description Framework), Model and Syntax Specification, W3C Recommendation, 22-02-1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
[97] Ricci, A., Viroli, M., and Omicini, A., Programming MAS with artifacts, Workshop on Programming Languages for Multi-Agent Systems (PROMAS), 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'05), Utrecht, The Netherlands, 2005.
[98] Ripper, P.S., Fontoura, M.F., Maia Neto, A. and de Lucena, C.J.P., V-Market: A framework for agent e-commerce systems, World Wide Web, 3, pp 43-52, 2000.
[99] Rohl, M. and Uhrmacher, A.M., Controlled Experimentation with Agents - Models and Implementations, 5th Int'l Workshop Engineering Societies in the Agents World, Toulouse (France), Oct 20-22, 2004.
[100] Royce, W., Managing the Development of Large Software Systems, IEEE WESCON, Aug 26, pp. 1-9, 1970.
[101] Sarjoughian, H.S., Zeigler, B.P. and Hall. S.B., A Layered Modeling and Simulation Architecture for Agent-based System Development, IEEE, 89, 2, pp. 201-213, 2001.
[102] Shi, L., Gu, Z-M, Tao, Y-C, Wei, L., and Shi, Y., Modeling Web objects' popularity, International Conference on Machine Learning and Cybernetics, 18-21 Aug., 4, pp.2320-2324, 2005.
[103] Sierra, C., Rodríguez-Aguilar, J. A., Noriega, P., Esteva, M. and Arcos, J.L., Engineering Multi-agent Systems as Electronic Institutions, Novática, 170, 2004.
[104] Silva, A.R., Romao, A., Deugo, D., and Mira da Silva, M., Towards a reference model for surveying mobile agent systems, Autonomous Agent and Multi-Agent Systems, 4, 3, pp 187-231, 2001.
[105] Sivasubramanian, S., van Halderen, B., and Pierre, G., Globule: a User-Centric Content Delivery Network., 4th International System Administration and Network Engineering Conference, Sept. 2004.

[106] Tolksdorf, R., Coordination patterns of mobile information agents, Cooperative Information Agents II, Springer-Verlag, vol.1435 of LNAI, pp 246-261, 1998.
[107] Tveit, A., A survey of Agent-Oriented Software Engineering, First NTNU CSGS Conference, May, 2001.
[108] Uhrmacher, A.M. and Scattenberg, B., Agents in Discrete Event Simulation, 10th European Simulation Symposium ``Simulation in Industry -- Simulation Technology: Science and Art'' (ESS'98), SCS Publications, pp 129-136, 1998.
[109] Vigna, G., Mobile Agents: Ten Reasons For Failure, IEEE International Conference on Mobile Data Management (MDM'04), Berkeley, CA, USA, 19-22 January 2004.
[110] Wang, Y., Tan, K-L., and Ren, J., A Study of Building Internet Marketplaces on the Basis of Mobile Agents for Parallel Processing, World Wide Web: Internet and Web Information Systems, 5, 1, pp. 41-66, 2002.
[111] Weiss, M., Pattern-Driven Design of Agent Systems: Approach and Case, Conference on Advanced Information Systems Engineering (CAiSE), Springer, LNCS 2681 ,2003.
[112] Wooldridge, M., An Introduction to MultiAgent Systems, John Wiley & Sons Ltd, 2002.
[113] Wooldridge, M., Jennings, N. R., and Kinny, D., The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems, 3, 3, pp 285–312, 2000.
[114] Xu, D., Yin, J., Deng, Y., and Ding, J., A Formal Architectural Model for Logical Agent Mobility, IEEE Trans. Software Eng. 29, 1, pp 31-45, 2003.
[115] Zhou, X.Y., Arnason, N., and Ehikioya, S.A., A proxy-based communication protocol for mobile agents: protocols and performance, IEEE Conference on Cybernetics and Intelligent Systems, vol. 1, pp 53-58, 1-3, Dec. 2004.