ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

SECURE-AGENTS SIMULATOR

Elaborata nel corso di: Sistemi multiagente

Progetto di: ANDREA ZAGNOLI

Indice

Introduction

This project has been developed under the courses of Multi-agent Systems and Technologies and Systems for Security, with the aim of building a coherent simulator of a multi-agent system composed by different kind of agents willing to communicate with one another and using particular sensible-data-related services in a secure manner.
In the specific case the simulation is concerned with online purchases and cash withdrawal; a pool of *client*s wonder around the virtual world, stumbling upon ads of different products sold by different companies; from time to time, the single *client* gets convinced to buy something and acts accordingly, checking his finances, contacting the reseller of the particular product and buying the product. If the *client*, somehow, is low with cash first has to go to the ATM and withdraw some money.
Each *reseller*, in a dual way, continually ads its products and waits for tricked *costumer*s in order to conclude a sell.
The whole purchase process needs to be protected against unwanted access (given the exchange of sensible data, such credit card numbers and personal information), this is done following the de-facto standard architecture commonly adopted nowadays in situation like that: such as, first, via the use of a simple version of Public-Key-Infrastructure (PKI) - which provides services for public certificate (X509) creation, management and retrieval - for the purpose of establishing a secure channel and then through the use of a symmetric cryptography scheme for the reminder of the communication process.
Furthermore, even the action of withdrawing cash from the ATM has to be secured using an appropriate authentication schema.

The system provides a visual *gui* to directly set-up and manage

the entire simulation process (i.e. populate the world with clients and resellers) and to monitor/interact with the ongoing simulation, inspecting and modifying current values of relevant information (such as public certificates).

In the reminder of this paper will be given account of the different phases of analysis and design following (not so strictly) the *SODA* methodology, therefore adopting the A&A meta-model.

Capitolo 1

Why an agent-oriented model

As usual, the first issue to tackle when building a system, after the process of carefully framing the whole problem at hand, is to decide which meta-model is best suited to first design and then implement it. The matter is not so simple at it may seem, because, obviously, choosing a not proper level of abstraction could bring to a failure either for an explosion of complexity (when a low level meta-model is employed) or for having adopted a too much powerful level with which is difficult to properly map system's concepts.
Subsequently, once the perceived proper meta-model has been chosen, arise the matter of selecting an adequate methodology as a blueprint to follow in order to first formally analyze and then design the whole system.
In the remainder of this chapter will be given reasons, other than being the mere subject of the entire course, for choosing the abstraction of *agent* as basic building block for development of the system, then will be briefly described both the meta-model and the methodology adopted.

1.1 The agent abstraction

An agent is an autonomous computational entity, [2]; from this brief definition, according to the need, can stem both the notion of *weak* or

strong agency: the first being more concerned with software-engeneering requirements, such as encapsulation of control, pro-activity, reactivity and interactivity (with both other agents and the environment - whatever it might be), whilst the latter adding to the former the further requirement of cognitivity - encompassing reasoning and mental attitudes - being more focused on the pursue of a notion of intelligent entity [12].
Identifying the *weak* notion, between the two cited above, as the proper form of agency from a software engineering point of view (there's no need for intelligence here), an explanation of why this abstraction is the right one for the problem at hand is due. As possible to deduce by Jennings, the best abstraction is the one that most reduces the conceptual gap between the solution of the problem as one can think at first and the entity (or entities) really at hand for its actual design and implementation, meaning the one (or ones) that maps in the best way the system's concepts, [4]. In this particular case, it comes natural to think at the simulation-instance, in particular, and the whole system, in general, as composed by active entities (*client* and *reseller*) acting according to internal goals (buying and selling products respectively) which interact with one another and use or exploit passive entities as mere service providers (such as the *ATM* or part of the *PKI infrastructure*).
And here comes the problem: generally, when dealing with agent meta-models, there is a complete lack of a notion of service, meaning a natural counterpart of the autonomy of the agent such as a passive abstraction, enabling and supporting the agent's actions. In fact, in analogy with the object-oriented world where is common to say *everything is an object*, it is also common to say in an agent-oriented context *everything is an agent*.
Often, when dealing with complex system and especially in software engineering, the need of those kind of entities arises; in this case, in particular, comes natural to identify components, such as an ATM, as a passive and reactive component instead of an autonomous one; properties of pro-activness or sociality would be overabundant and would oblige the designer, or the developer, to a *stretch* of abstraction in order to capture such an entity into such a powerful abstraction.
In order to settle this matter, more and more meta-models are introdu-

cing concept of passive entities at an agent level, called with different names and with slightly different properties or formalization, but all underlying the same basic characteristics: i) reactiveness, ii) designed for a purpose and iii) exhibiting its functions via an interface.
At this point, for what stated above, a meta-model of this kind through which analyze, design and develop the whole simulator system seems a natural choice. The one here adopted is the *Agent and Artifact* (*A&A*) meta-model.

1.1.1 Agent & Artifact meta-model

A&A is a conceptual framework characterized in terms of three basic abstractions: i) *agent*, ii) *artifact* - being the needed computational entity aimed to be used by agents - and iii) *workspace* (as a way to topologically divide the system) [9].
In *A&A* the notion of *environment* is considered as a first-level abstraction and plays a central role in both the design and implementation phases [8]. In fact, the *environment*, being the set of *artifact*s developed into a certain *multi-agent-system*, act as both an enabler and constrainer for the agents: it expands agents ability, in the same time posing important limitation to what they can perceive or manipulate. Each artifact is explicitly engineered to offer a certain coherent set of functionalities (*operation*), defined for the particular application, and gets created, used, shared or manipulated by agents accordingly to its usage rules or protocol.
To briefly sum-up, a system formed by multiple agents, or what is commonly defined a *MAS*, modeled through *A&A* meta-model consists in a set of agents, each with its own personal goal(s), interacting with one another and using specifically-designed artifacts as service providers in order to fulfill the system common goal.

1.2 Methodology

Generally speaking, the increase of complexity in nowadays software systems impose not only a need to model and engineer the model of the

system itself, but also the whole development process which returns such a model as a result, adopting a systematic and organized approach appropriate for the given problem and context [11]. An indefinite number of methodologies have been developed since, each applicable to a specific context, but all prescribing a certain coherent approach aimed at solving the problem by putting in relation a number of preselected methods [3]; in this case an agent-oriented methodology is required, one that suitable characterizes agent-oriented abstractions - *agents* and *environment* - and lets easily frame and define the system's interactions. Having already committed to a particular meta-model, though is not mandatory, it would be better to stick with a methodology that fully embodies the concepts that it proposes, in order to avoid dealing with abstraction mismatch. For this reason, in the context of this system development process will be used the SODA methodology, briefly introduced in the remainder of this chapter.

1.2.1 SODA

SODA (Societies in Open and Distributed Agent spaces) is an agent-oriented methodology for the analysis and design of agent-based systems, which takes into account both the Agents and Artifacts (A&A) meta-model, and a mechanism to manage the complexity of system description.
The methodology is organised in two phases, each structured in two sub-phases: the *Analysis* phase, which is composed of the *Requirements Analysis* and the *Analysis* steps, and the *Design* phase, which is composed of the *Architectural Design* and the *Detailed Design* steps [10].

Capitolo 2

System analysis

The first phase of the methodology, as said, is further dived in two sub-categories: i) requirements analysis and ii) analysis itself. In the first sub-phase the system's requirements and the external environment are analysed and modelled, whilst in the latter the system's requirements are modelled in terms of tasks, functions, topologies and dependencies, [6].

2.1 Requirements analysis

The goal of *Requirements Analysis* is the characterisation of both the customers' requirements and the legacy components with which the system should interact, as well as to highlight the relationships among requirements and legacy systems. Several abstract entities are introduced for requirement modelling: in particular, requirement and actor are used for modelling the customers' requirements and the requirement sources, respectively, while the external-environment notion is used as a container of the legacy-systems that represent the legacy resources of the environment. The relationships between requirements and legacy systems are then modelled in terms of suitable relation entities.

The *secure-agents* system simulates a world in which different agents (*client*s and *reseller*s) cooperate with one another and with the surrounding environment to carry out purchases of goods. Agents

want to communicate in a secure manner when the communication itself concerns sensible data, such as credit-cards numbers, passwords or precious informations in general.

The system requirements can be split into three different categories each dealing with a particular aspect

- simulation requirements
- simulation-instance requirements
- security requirements

2.1.1 Simulation requirements

Simulation requirements are concerned with the entities needed to manage the entire simulation life-cycle (set-up, start, pause and stop) along with the tools aimed to its monitoring. Those kind of requirements are listed below:

- set-up simulation environment and participants
- launch, pause and stop a simulation-instance
- inspecting relevant simulation's data/parameters (public certificates)
- check minimum requirements for launching the simulation (at least one reseller and one costumer)

2.1.2 Simulation-instance requirements

Simulation-instance requirements are concerned with the entities needed to carry out an entire simulation, such as resellers, costumers, ATMs and so on and so forth.
Each simulation instance has to be composed by at least one client and one reseller, otherwise it has not much sense at all.

Those requirements are:

- resellers advertise their products
- costumers wonder around the virtual world, perceiving ads of different products
- costumers, sometimes, buy products from a particular reseller
- resellers, when a certain costumer requests it, sell their advertised products
- costumers, before buying a product, check if they need any money and, if so, they withdraw some from the nearest ATM

2.1.3 Security requirements

Security requirements involve and encompass both types of requirements cited above to provide a general security layer upon which and through which the system can be secured. From an external point of view it has to provide mechanisms to safely store and manage certificates, handle robust authentication processes and define a clear communication protocol through which different parts of the system can exchange messages with confidence. In a more precise way those requirements are the following:

- in order for the agents to communicate in a secure way, a *PKI* (*Public Key Infrastructure*) has to be established; from this general assertion stems different requirements, in terms of resources needed and constraints both within the interaction space and the behaviour of single agents
  - a *certificate authority* is needed with the aim to evaluate, grant and deny certificate requests
  - a *registration authority* as a 24/7 on-line service to which submit certificate requests
  - a *directory service* as a on-line database representing the current set of approved certificates, accessible by everyone at necessity
  - each *reseller* has to obtain a certificate before selling products

  - each *costumer* needs the chosen *reseller*'s certificate before buying the wanted item
  - certificate, encryption keys and passwords have to be saved in a safe way in order to prevent external and unauthorized access
  - communication between a willing-to-buy *costumer* and a more-than-incline-to-sell *reseller* has to occur according to the following interaction protocol: i) first, the *costumer* sets-up a session with the *reseller*, using the public key obtained through the certificate of the latter, ii) then, *reseller* acknowledges the request using its private key, iii) the *costumer* buys the product using the session-key to protect the messages and iv) finally, the reseller communicate its acceptance always using the session-key (note that for the sake of simplicity, each purchase always terminates successfully)

- the *ATM* has to employ a mechanism to safely ensure authentication of *costumer* wanting to withdraw some money: no unregistered agent has to have the possibility of completing the operation, whilst no registered agent has to be denied of doing that

- lastly, a general requirement involving not only a single simulation-instance, but the entire system as a whole states the necessity of adopting measures to prevent unauthorized access to the resources of the system, first through the definement of a clear access policy and then through its application throughout the entire context of the application

2.2 Analysis

The Analysis step expresses the abstract requirement representation in terms of more concrete entities such as tasks, functions and topologies. Tasks are activities requiring one or more competences, while functions are reactive activities aimed at supporting tasks. The relations highlighted in the previous step are now the starting point for the

definition of dependencies (interactions, constraints, etc.) among the abstract entities. The structure of the environment is also modeled in terms of topologies, i.e. topological constraints over the environment. Topologies are often derived from legacy-systems and requirements, but also functions and tasks could induct some topological constraints.
The *secure-agents* system since, as said, is built from scratch has no legacy-systems.

Moving from requirements phase to the analysis phase the abstractions identified in the first step are mapped onto the abstractions adopted in this stage to generate the corresponding model.

2.2.1 Tasks

Each task, as said, is defined as an activity that requires one or more competences and the use of functions [from AO's slides]. Starting from the different requirements cited before, the active parts are extrapolated and better defined.

A costumer has to carry out the following tasks:

- *wondering-around* - each costumer walks throughout the world perceiving ads of products they might want to buy; every now and then arises the necessity of purchasing an item, when that's the case the costumer checks its finances, engaging in the tasks of withdrawing some money, if needed, and then buying the product from the specific reseller

- *withdrawing-cash* - this task defines the operation engaged by each costumer whenever needs cash; he goes to the ATM, authenticates himself and draws moneys.

- *buying-product* - when the costumer decides to get something from a reseller, he contacts and establishes with him a transaction in which the good is purchased; in doing that, the costumer, has to follow the PKI protocol in order to prevent external breaches which might cause loosing sensible-information; for this rea-

son he has to obtain the reseller's certificate from the *directory-service* and use the retrieved public-key to agree on a session-key with the reseller himself, which will be consequently used to proceed with the real purchase part of the communication.

A reseller, on the other hand, has to perform dual tasks:

- *obtaining-certificate* - the first thing a reseller has to do, before it can safely sell products within this simulated world, is obtaining a certificate from the *certificate-authority* which will allow costumers to communicate with him in a discrete way
- *advertising-products* - once the reseller is ready for business it has to attract willing costumers advertising his products using ad-hoc billboards disseminated throughout the whole world
- *selling-products* - in the meantime of advertising products, a reseller has to wait for incoming customer's requests for products: whenever an incoming request is received the required item gets always sold to the costumer (for sake of simplicity, as already mentioned); analogously to the costumers, the reseller is aware of the PKI protocol and interprets the first message, once decrypted using its private-key, as the session-key used in the reminder of the transaction.

Further tasks have to cope with *certificate-authority* activities:

- *creating-root-certificate* - for first, the *certificate-authority* has to create a self-signed certificate with which signing all resellers' certificates
- *handling-request* - this task defines the cyclic behavior of the authority in which it handles incoming certificate-requests from the *registration-authority* and, if granted, generates the corresponding certificates, publishing them into the *directory-service*. The generation of a given certificate is subordinated to the acceptance of the incoming request, in which the identity of the applicant should be verified; in this case, the verification process is delegated to the user,which decides whether or not granting a particular certificate, through the *gui*.

The tasks considered so far are all concerned with simulation-instance requirements, but since the system is developed as a simulator, tasks to control simulations' life-cycle have also to be considered

- *shutdown* - is a common activity which deals with the need for each agent inside a simulation-instance (be it a reseller, a costumer or even the certificate-authority) to be aware of the current state of the simulation; whenever the simulation is paused, stopped or resumed this task cause the agent to behave accordingly (i.e. pausing all its activities whenever the pause-button is pressed). The control of a simulation-instance is user-side, so those commands are perceived directly through the *gui*, which signals the simulation's state whenever it's changed.

- *managing* - this activity handles the creation and destruction parts of both active and passive components of a simulation-instance, when a simulation is first started, or ultimately stopped; in the first case it generates all the passive resources needed and then spawns the agents according to the configuration chosen by the user. In the second case, instead, the activity is responsible for disposing previously used resources and clearing the environment before launching another instance of simulation. Both simulation-instance's commands (start or stop) and configuration settings are drawn from the *gui*.

2.2.2 Functions

Each function is a reactive activity that aimed at supporting tasks; as seen above, each task performing some activity relies on some already provided functionalities. Here again, as already made with tasks, *functions* are derived from the requirements and then explicitly defined, describing their aim within the system.

Each costumer has to do with an *atm* when he needs money; the functionalities provided by such a component are similar with the ones provided by ATM in the real world, but what makes the virtual one different is the mechanism used to authenticate customers; normally a badge and a password are needed, in this case, in order to prevent

the direct insertion of the user's password, which might be peeked by a near stranger and has serious problem in case of badge-cloning, the *atm* employs a *challenge/response* protocol, which poses to a given user a question that only him, relaying on its password, could answer - the costumer uses his password to compute the answer but never shares it with anyone, not even with the *atm*.

- *insert-badge(customer-name)* - this operation returns a random value encrypted using the customer's password
- *insert-response(customer-name, answer)* - this operation check if the answer provided by the costumer is correct (if it's the random number first encrypted), in that case the authentication process is complete, otherwise it fails.

For reason of simplicity and of keeping the system complexity at low levels, a withdrawn operation is here considered completed when the authentication process has succeeded, no further functionalities dealing with the actual withdrawn of money are provided.

All resellers rely on billboards to advertise their products; here a simple version of *billboard* is considered offering only one functionality

- *advertise(reseller-name)* - this operation causes the billboard to display a particular reseller ad, which is perceived by all passing costumers

The use of a PK-infrastructure rises the need of certain services provided by ad-hoc resources: a so called *registry-authority* and a *directory-service*; the first being a 24/7 service through which resellers can post their certificate requests and the *certificate-authority* can retrieve them. The second one acting as an on-line database of certificates, used by costumers, to retrieve resellers' certificates, and by the *certificate-authority*, which publish them. So, the security-related functionalities are the following:

- *submit-CSR(certificate-request)* - operation used by a reseller, when requesting a certificate
- *get-new-request()* - operation used by the *certificate-authority* in order to get and then evaluate a request

- *add-certificate(certificate)* - operation used by the *certificate-authority* to publish a brand-new certificate

- *delete-certificate(customer-name)* - operation used when the need of deleting a certificate arises (i.e. for suspension reasons)

- *lookup(reseller-name)* - function used by a costumer to obtain the certificate of a particular reseller

The *gui* services as a mediator between the end-user and both the simulator, providing suitable commands to manage simulation-instances life-cycle and signaling back changes in the latter's state, and the simulation-instance currently running itself, if any, providing means through which the user can interact with it (inspecting or monitoring its state in terms of certificates and granting or denying certificate requests).

- *submit-new-request(certificate-request)* - this operation causes the *gui* to pop-up an ad-hoc dialog in which the applicant's information are described and two buttons, for the user to *grant* or *deny* the certificate, are provided; when one of the two button is pressed the decision is passed back to the simulation-instance's *certificate-authority*, which behaves accordingly.

- the *gui* also reflects back to the system the state of the simulation every time it changes, due to user decisions, signaling its current value in order for the currently active tasks to behave accordingly

2.2.3 Dependencies

A dependency is any relationship (interactions, constraints,..) among other (tasks and/or functions) abstract entities. When describing both tasks and functions some dependencies have already been cited, though not properly highlighted; in this sections some of the most important dependencies of the system will be listed.

A first set of dependencies can be those temporally constraining tasks execution with one another.

- each costumer engages in the *wondering-around* task, whenever he wants to buy a product he engages in the *buying-product* task, but only after the completion of the *withdrawing-money* task, if necessary; when the item has been purchased, hence terminating the related task, the *wondering-around* task is resumed. Along side this sequential execution, lies the *shutdown* task, which supervise the execution and properly stops the other tasks when the simulation requires it.

- each reseller, as the first action, has to obtain a certificate through the one-time-executed *obtaining-certificate* task; once successfully completed, both *advertising* and *selling-product* tasks are cyclically carried-out; as for the customer, the *shutdown* task runs along side those task controlling their termination when demanded

- the *certificate-authority*'s behavior is constrained by the temporal precedence of the one-time-executed *creating-root-certificate* task, with respect to the cyclical *handle-new-request* task; those are both controlled by the usual *shutdown* task

- all those activities cited above, being simulation-instance dependent, will be subordinated to the task controlling the simulation life-cycle, named *managing*, which will spawns the agents, hence their activities.

A second cluster of dependencies are the ones relating different tasks in terms of direct interactions (in this case communication).

- the most important dependency of this kind is the one putting in relation with one another the tasks *selling-product* and *buying-product*, which lead the costumer to purchase and, dually, the reseller to sell a particular product

Another kind of dependencies is found in the activities reliance on some specific functionalities in order to carry-out the tasks at hand and, stemming from that and from particular resources usage, the consequent order in which certain of those functionalities have to be used in order to provide consistent results. Given the high cardinality

of this set of dependencies, in order to keep brief the report, only the main or the ones demanding a thorough explanation will be described.

- costumers and billboards are related, the first ones perceiving ads, in the *wondering-around* task, displayed by the seconds; furthermore customers interact with *directory-service*'s *lookup* functionality, when in need of obtaining a particular certificate within the *buying-product* task

- each costumer, when needed, uses the *atm* functionalities, which in order to provide a meaningful outcome have to be employed in a certain order: as common sense would suggest, first the *insert-badge* must be used and only consequently can and must be used the *insert-response* operation. Attempt of using the resource in a different manner will result in authentication-failures.

- each reseller's task *obtain-certificate* uses the *registration-authority submit-CSR* operation when requesting his own certificate and perceiving its publication from the *directory-service* in order to follow up with his activities. Furthermore he uses the billboard functionality *advertise* within the *advertising-products* activity

- the *certificate-authority*'s *creating-root-certificate* uses *directory-service*'s *addCertificate* operation in order to add its certificate to the repository; furthermore its *handling-request* task uses both *registration-authority*'s *get-new-request* and *directory-service*'s *addCertificate* operations when dealing with certificate requests - in order to perform the applicant identity verification it relies, as said, on the *gui*, using the latter's *submit-new-request* operation.

- customers, resellers and the *certificate-authority* alike perceive, within the *shutdown* task, simulation-related commands from the *gui*, which signals whenever the currently running simulation-instance changes its state

- as for task-to-task dependencies for the agents, the *managing* task is related to all simulation-instance's resources being the activity which creates and disposes them at demand, furthermore it relays on perceptions coming from the *gui*, in order

to initially start and populate or definitely stop and dispose a simulation-instance

The last type of dependency here highlighted, but not used within the system here considered, is the one among different functions, in which one uses one or more functionalities provided by other resources.

2.2.4 Topologies

The topology of a system can be generally defined as the logical configuration in which are arranged its composing entities. In the case of the *secure-agent* system, its topological divisions stem directly from the main division between requirements: *simulator* requirements and *simulation-instance* requirements; in first approximation then, is here very straightforward to lay-out the main logical compartments in which the system can be separated

- a *simulator* topology, to which will belong all components related to the management of the simulations, their supervision and management
- and a *simulation-instance* topology composed by all those elements having to do with a single simulation development (like resellers, costumers, ATMs and so on)

Capitolo 3

System design

3.1 Architectural Design

The main goal of this stage is to assign responsibilities of achieving tasks to roles, and responsibilities of providing functions to resources. To this end, roles should be able to perform actions, and resources should be able to execute operations providing one or more functions. The dependencies identified in the previous phase become here interactions and rules. Interactions represents the act of the interaction among roles, among resources and between roles and resources, while rules enable and bound the entities' behaviour. Finally, the topology constraints lead to the definition of space, i.e. conceptual places structuring the environment.

3.1.1 Roles and actions

Roles are defined as the abstractions responsible for the achievement of one or more coherent tasks (previously identified), which in turn are capable of engaging in a set of ordered actions in order to carry-out their individual job.
Within the *secure-agent* system can be framed different kind of roles, each with a different and well-defined goal to fulfill: two main roles, *customer* and *reseller*, the *certificate-authority* role, the *simulation-manager* and *life-cycle-controller* roles; they all will be briefly intro-

duced in the reminder of this section.

The *customer* role has the obvious final aim of living a good and fulfilling life, wondering around all day long wasting his time; unfortunately the filthy virtual world is filled with ads of products of evil resellers eager to make more and more money. Every now and often every poor *customer* thinks he needs some product he saw in an ad, so, in order to settle that necessity and to finally return doing nothing, he has to buy that product; money don't come easy, either, so when he is low with cash, a withdrawal at the local ATM is due. Tasks and related actions of the *costumer* role are listed below:

- the *wondering-around* task is able to engage in the actions of *perceive-ad* and of critically *decide-weather-to-buy* or not given the amount of ads of the same product he has already seen before

- the *withdrawing-cash* task has to use the atm in compliance with its specifications, firstly *inserting-the-badge*, then *computing-the-answer* and ultimately to give the correct response to successfully authenticate himself and draw some money out of the machine; given the already discussed mechanism of authentication used by the atm, the *costumer*, in order to correctly compute the correct answer to the posed question, is also capable to *decrypt-message* according to a certain key - in this particular case its badge's password.

- *buying-product* is achieved through the execution of the following actions: i) first, trying to *obtain-reseller-certificate* - if unable the entire task cannot succeed, therefore the purchase is abandoned - subsequently ii) the customer has to *generate-session-key* through which communicate with the reseller during the second phase of the transaction - the one in which the purchase of the item is carried-out - then iii) communicate with the reseller through the actions *send-message* and *receive-message*, in order to exchange informations, and the actions *encrypt-message* and *decrypt-message*, with which the messages are (de)cyphered before (or after) be sent (or received).

The *reseller* role has the obvious final aim of getting richer and richer at the expanses of poor costumers, selling their products all day long. In order to make him and his items famous around the virtual world, he disseminates it with convincing and compelling ads. The entire life of the reseller is therefore spent waiting for willing costumers; when a tricked one contacts him, the game is on and the reseller is able to conclude a sell.
Unfortunately every world has its rules, even the virtual ones, and the *secure-agent* system makes no exception: since security is required when exchanging sensible informations, the reseller has to provide himself, and therefore his costumers, with the minimum level of safety required by the PK Infrastructure - has to first obtain an his-own certificate through the *certificate-authority* and then to comply with the imposed communication protocol.
Tasks and related actions of the *reseller* role are listed below:

- the *obtaining-certificate* task is the first executed by the reseller and has the aim of providing him with a proper certificate; in doing that the agent should be able to *create-certificate-request*s and to *submit-request* using the *registration-authority*

- the *advertising-product* task has to carry out the job of publishing ads, using the *billboard*; in here the reseller engages in the action of *publishing-ad*s

- *selling-product* is the main activity of a reseller in which he engages in the actions of i) *waiting-for-costumer* and, when one arrives, ii) communicating with him through *send-message* and *receive-message* actions, always after (or before) the actions of *encrypt-message* or *decrypt-message* according to the protocol

Another important simulation-instance role is the *certificate-authority*, which has the goal of acting as the authority, within the virtual world, with the powers of granting or denying well-formed certificate to the applicants and the duty of publishing the granted ones to the *directory-service*.
Tasks and related actions of the *certificate-authority* role are listed below:

- the *creating-root-certificate* task has to comply with the job of creating a self-signed root certificate with which the *c-a* will be then able to sign future certificates; in doing that the agent engages in the actions of *create-certificate* and *publish-certificate* via the *d-s*

- the *handling-request* task has to do with the managing of incoming certificate requests: the *certificate-authority* first *wait-for-incoming-request*s through the *r-a*, then *ask-user-for-validation*, then, if granted, *create-certificate* for the reseller and finally *publish-certificate* using the *d-s*, as required by the PKI

All previous identified roles where concerned with simulation-instance activities, the *simulation-manager* role, instead, is concerned with simulator activities and has the job of supervising the whole simulation and handling, at user request, its creation and its disposal. This role engages only in one main activity, denoted as *managing*, in which it continuously *waits-for-simulation-instance-state-change*s and when one occurs, according to the new state and the configuration of the simulation environment decided by the user, engages in the actions of *spawning-agents* and *creating-resources* or of *stopping-agents* and *disposing-resources* to clear the workspace.
The last important role, both simulation-instance and simulator related, is the *life-cycle-controller*: this role, with its only *shutdown* task, has the aim of making *simulation-instance* activities aware of external simulation state changes and reflect them back to the agents themselves, acting accordingly; for example when a simulation-instance is stopped by the user, the *shutdown* task stops all agent's current activities. In order to fulfill the job, the role *waits-for-simulation-instance-state-change* and then acts accordingly on the agent, interacting with his activities.

3.1.2 Resources and operations

A resource is defined as the abstraction that provides some functions used by the active parts of the system to carry-out their job; each resource is a set of coherent *functions* highlighted in the analysis phase and, in turn, each function is composed by an ordered set of operation

- representing what the resource itself is able to provide. In this section will be presented the main resources which build-up the *secure-agent* system and the related operations.

The first resource here described is the *atm*. The *atm* is a tool that has the aim of providing money-related services; in this particular case the machine at hand is only concerned with the authorization phase, instead of the entire operation-cycle, hence will provide *functions*, and related operations, to serve such purpose

- the function *insert-badge*, which has seen in the previous phase of the methodology, has to provide a proper question for authenticate the costumer at hand, identifying whether or not he is actually who he claims to be; the operations undertaken in order to provide such a service are i) first, *generate-random-number*, then ii) *encrypt-number* using the customer's own password and finally iii) *display-question* to the user

- the function *insert-response* , on the contrary, has the aim of checking whether the answer provided by the costumer is the right one, therefore authenticating him, or not - in that case the authentication will fail and the access denied. The operation needed for such a service are simply the *comparison* of the provided answer with the random-generated number and the *communication-of-authentication-result* to the user

Another resource identified within the system is the *billboard*, which has to provide services that let resellers publishing their ads for costumers to see. This resource is a straightforward one, being composed by only one operation, namely *advertise*, which simply operates *displaying* the ads in the virtual world

The *registration-authority* resource is the 24/7-active service through which certificate-applicants submit their requests and the *certificate-authority* retrieve them, adding an intermediate level of indirection between the two active parts, allowing temporal decoupling. *Functions* and related operations are below listed

- the *submit-CSR* stores the current certificate-request in a local set

- on the contrary the emphget-new-request function retrieve from the local set one of the requests and returns it

The other resource having to do with the adopted public-key infrastructure is the *directory-service*, acting as a certificate-repository in which storing newly created certificates and through which retrieving needed ones. Its functions, *add-certificate* and *lookup*, respectively store and return a particular certificate.

The last resource identified within the system is the *gui*; since its nature, it is a slightly different resources than the other ones: this has to do with the fact that it services as a bridge between the *simulation-instance* and the *simulator* active parts of the system (resellers, costumers and the *certificate-autority* on one side and the *simulation-manager* and the user on the other). From a simulation-instance point of view, the *gui* offers the *submit-new-request* functions, which lets the *certificate-authority* entrust the identity verification process to the final user, and notifies simulation state changes; from the simulator point of view, instead, it lets interact with the simulation.

3.1.3 Interactions and rules

The previously identified dependencies and constraints among tasks, among functions and between tasks and functions will be here mapped onto abstractions like *interactions*, namely representing the acts of the 'data-flow' among roles, among resources and between roles and resources, and *rules*, as enablers of boundaries of the entities behavior. Most of these considerations have already been made in the previous phase of analysis, therefore in this section, only the most significant ones will be highlighted.

Interactions considered, and used, within the *secure-agent* system are of the kind role-to-role and role-to-resource; the main dependency of the first type is the one concerning the communication between the role of *costumer* and the role of *reseller*, from which the exchange of goods for money stems. Interactions of the second kind arise when a role exploits a given resource in order to carry-out a certain action; examples of this dependency are given from the use of the *customer* role of the services provided by the *atm*, the *directory-service* and the

billboard, or the role of *reseller* employment of the functionalities exposed by the *registration-authority* and the *billboard*, only to cite a few.

On the other hand, rules arise whenever certain constraints have to be respected by an entity in order to behave consistently within the system - e.g. following an established communication or usage protocol. In the system at hand those constraints, has already presented before, are both inter-role (as temporal dependencies among different tasks within a single role), intra-roles (as communication protocol which has to followed when a costumer and a reseller engage in a transaction) and among roles and resources in terms of the latter's usage (as the correct order in which a costumer has to use atm's operations).
An important set of rules, not presented yet, is the one constraining the boundaries of roles action within a particular space: explicitly

3.1.4 Spaces

Starting from the topology defined in the analysis phase, here spaces, as conceptual loci in the environment, will be defined along with the entities operating within each one of them.
The *simulator* space is the one through which is possible to control the simulations and is concerned with both user-side aspects, such as the *gui*, its interface and its managing, and internal aspects, meaning the logic needed to handle the simulations. In this space are located both the *gui* resource and the *simulation-manager* role.
The other space is the *simulation-instance* one; this logical compartment, managed by entities in the first one, is composed by all those components, both active and passive, having to do with the simulation it-self: *costumers*, *resellers*, the *certificate-authority* are the active roles, whilst *atm*, *billboard*s, the *directory-service* and the *registry-authority* are the passive resources. The *gui* too is placed within that space. It's important to notice that spaces (or topology, or workspaces for that matter) are not physical separator of entities, but merely logical separators which limit roles actions and resources visibility; so not surprisingly the same resource (or role) could be placed within different compartments for logical purposes. In this case the *gui* is placed both within the *simulator* space and the *simulation-instance* space,

acting as a bridge between the two worlds: signaling simulation state changes from the outside-in or reflecting back simulation advancement from the inside-out.

3.2 Detailed Design

This is the final step of the methodology leading to a detailed model of the system which, at last, represents entities and interactions highlighted so far in terms of A&A components. In this phase the system is designed in terms of agents and societies, representing its active parts, artifacts, compositions and aggregates, representing passive resources providing functionalities upon which active ones rely, workspaces, as the topological boundary logically separating the different components, and, lastly, in terms of uses, links to, manifests and speaks to, as the interaction glue of the system itself which ties together all the components. In the reminder of this chapter a final model will be provided in terms of those abstractions adopted by the A&A meta-model.

3.2.1 Agents

Starting from different requirements, passing through tasks which are then both grouped in terms of roles and described as an ordered set of actions, the process is here completed defining the agents - as autonomous entities able to play several roles - which populate the system.

Simulation-instance agents

As already mentioned at the beginning of the paper and as perceived throughout the whole development of the methodology, which, given the simplicity of the system at hand, was more a designed outcome rather than the process leading to this very point (exception made for some peculiar aspects), should not be of any surprise that the main agents composing the virtual world of a simulation of the *secure-agents* system are the *costumer*s, the *reseller*s and the *certificate-authority*.

The *costumer-agent* represents the active part of the system which buys not-so-needed products in order to fulfill its material needs which, in turn, stand in the way of its final goal: the pursue of happiness; furthermore the agent oughts to be aware of its status within the simulation lifecycle, being controlled by external forces, with the goal of behaving according to the simulation's current state (that is, acting when the simulation is running, whilst suspending all its tasks when the simulation is paused or stopped). As a result, each *costumer-agent* engages simultaneously in both the *costumer-role*, concerned with the first of its goals, and the *life-cycle-controller-role*, concerned instead with its latter aim.

In a dual way, the *reseller-agent* is the villain, representing the capitalistic and evil part of the virtual world with the main objective of stockpiling enormous amount of money selling products; furthermore, since not even the powerful forces of the free-market can compete with the almighty controller of the simulation, the *reseller-agent*, like the *costumer-agent*, has to be aware of external life control and behaving accordingly. Therefore the two roles played by every *reseller-agent* are the *reseller-role*, fulfilling the first of its two goals, and the *life-cycle-controller-role*, which acts toward a simulation-state awareness.

In the middle of this selling/buying process stands the *certificate-authority-agent* which has the main goal of managing in a consistent way the public-key infrastructure that provides the system with an underlying level of security. As already mentioned before, the certificate authority here described and represented is only a mere simplification of a real one, being only responsible of granting, denying and publishing certificates. For this end the agent engages in the *certificate-authority-role*, previously described. Furthermore, like all the other simulation-instance agents, the *certificate-authority-agent* plays simultaneously the *life-cycle-controller-role*, providing the required responsiveness to external life-cycle commands.

It is worth noting how the goal proper to the single agent (such as selling products for a reseller or wondering around the world and buying what's needed for the customer) is subordinated to the multi-

agent system goal, which is providing a working simulator. For this reason each and everyone of the agents composing the simulation-instance space play the *life-cycle-controller-role* which controls and governs the other one in response to simulator commands. Those individual goals are sometimes in conflict and in competition with the system main goal, hence, when those situations occur, the formers must bend themselves and adapt to the latter.

Simulator agents

The *simulation-manager-agent* plays the *simulation-manager-role* and represents the active part of the system responsible for the management of a simulation-instance life-cycle, creating a band-new one when the user requires it, whilst cleaning the space and disposing all resources when the simulation is stopped.

3.2.2 Artifacts

An artifact is a computational entity aimed at the use by agents; artifacts as reactive but not proactive entities which offers functionalities exploited by the agents within the MAS to fulfill their individual and social goal. Here a distinction among different kind of artifacts can be made according to their particular functions: i) *individual artifacts*, ii) *social artifacts* and iii) *environmental artifacts*.
In this section, using as a starting point *resources* and *operations* highlighted in the previous phase of architectural design, the artifacts building up the environment in which the agents operate will be described along their *usage-interface*.

Social artifacts

Social artifact are exploited by more than one agent, mediate between two or more agents in a MAS. In general, social artifacts typically provide MAS with a service which is in the first place meant to achieve a social goal of the MAS, rather than an individual agent goal

The *atm-artifact* stems directly from the *atm* resource identified in the previous phase and serves as the tool though which costumers can

withdraw cash, after a proper authentication; as already said before, for sake of simplicity, a withdrawal process is here considered concluded whenever the customer identity is validated and authenticated within the *atm*. The *atm-artifact* usage interface is as follows:

- the *insert-badge(customer-name)* operation signals back to the customer a particular answer related to its claimed identity
- the *insert-response(response)* operation checks whether the given response is correct and then signals back to the costumer the result of the verification process (*authentication-succeeded* or *authentication-failed*)

Another social artifact is the *registration-authority-artifact*, stemming from the *registration-authority* resource previously described, which acts as a mediator between resellers willing to obtain a certificate needed to sell product and the *certificate-authority*, the agent who's responsible for the verification of applicant's identity. The usage interface offered by this artifact is described below

- the *submit-CSR(certificate-request)* operation is used by the resellers to file a certificate-request
- the *get-new-request* operation is used by the *certificate-authority* to retrieve a new certificate-request to evaluate; the operation signals back to the agent the particular certificate-request

The *directory-service-artifact*, derived from the *directory-service* resources, services as the repository of certificate through which is possible register or retrieve certificates. It's usage interface follows:

- the *add-certificate(certiifcate)* operation adds to the repository a new certificate and signals back to the interested reseller the certificate it-self; the operation is used by the *certificate-authority* whenever grants a new certificate in order to make it public and accessible.
- the *lookup(reseller-name)* operation is used by a customer, willing to by a product from a certain reseller, in order to obtain

the reseller's certificate; the operation signals back to the customer the required certificate, if contained within the repository, a *look-up-failed* otherwise.

The last social artifact composing the system is the *billboard* used by resellers to ad their products and perceived by the wondering costumers. Its usage interface is simple since it is composed by only one operation

- the *advertise(reseller-name)* operation, exploited by the resellers, signals the particular ad to the unwitting customers.

Environmental artifacts

Environmental artifacts mediate between a MAS and an external resource. In principle, environmental artifacts can be conceived as a means to raise external MAS resources up to the agent cognitive level.

In the case of the *secure-agent* system the only external resource which the MAS needs to be aware of is the interaction of the simulator-user who builds up and the controls the simulations. The resource previously described and here defined as an environmental artifact is the *gui* which acts as a mediator between the user and both the simulator and simulation-instance workspaces, manifesting from the outside-in changes in the simulation state, whenever they occur, caused by the user and displaying from the inside-out certificate-request for user's consideration, every time a reseller request a certificate. Its usage interface has only one operation.

- *submit-new-request* used by the *certificate-authority* to demand identity verification to the user: the operation displays a form presenting the request to the user signaling back to the *certificate-authority* the user's decision (certificate granted or not)

3.2.3 Workspaces

A workspace is a conceptual locus in the environment containing both agents and artifact and logically separating the entire MAS environment. In the *secure-agents* system were identified two different spaces,

in the phase of architectural design, which are now described as workspaces, indicating which agents and artifacts build them up.

The *simulation-instance* workspace logically divides each single simulation from those parts of the system that controls them; it contains all the agents and artifacts which form a *secure-agent* simulation as specified in the initialization phase by the controller user, namely: *customer-agents*, *reseller-agents*, a *certificate-authority*, a *billboard*, an *atm*, a *registry-authority*, a *directory-service* and the *gui*.

The *simulator* workspace, on the other hand, contains all those entities controlling each simulation: a *simulation-manager-agent* and the *gui*.

3.2.4 The interactions dimension and its constraints

Starting from the system's requirements from which its constraints are derived and passing through the metaphors of *interaction* and *rule*s singled out in the architectural design, here the process will be completed, identifying the interactions among metaphors used in this phase along with the rules constraining the interaction space. The different kind of interactions, along with the rules constraining them can be divided up in: i) communication agent-to-agent, in terms of *speaks-to*, and related rules as *protocol* of exchanged messages, ii) agent exploitation of an artifact, namely *use*, with the related *usage*, that is the instructions of the particular artifact in order to consistently use it according to the specific need of the agent and iii) the artifact intention of signaling something to the external world, be it interesting or not, in terms of *manifest* interaction - even here specifying the *protocol* of signals emitted.

Agent-to-agent communication

This kind of interaction involves generally two (or more, depending on the semantic of the interaction itself) agents which directly communicate with one another exchanging message according to a previously

agreed protocol.

Within the *secure-agents* system there is only one interaction of this kind which is the communication that occurs between a customer and a reseller. Generally, when the need of defining a communication protocol arises, especially in complex contexts, such as open systems or where the communication itself is not as trivial as the one adopted in this case, in order to properly define syntax and semantic of the protocol itself formal tools are used; in this case only four messages are exchanged within the system, therefore a simple description of both the semantic and the syntax is given in a non formal manner.

- the first message of the communication is always the one sent by the customer to the reseller to communicate him his willingness to purchase a product. Since no actual purchase is carried out during the process the only information required here in this message is the *session-key*, generated by the customer and used in the reminder of the communication; the message, given the security requirements, is encrypted with the reseller's public key. Formally: $\{sessionkey\}_{PbK_{reseller}}$

- the message in response sent by the reseller to the customer has the aim of acknowledging the correct establishment of the session, therefore the message will contain the string *ack?* followed by the id of the session just established and will be encrypted with the reseller's private key. Formally: $\{ack?sessionID\}_{PvK_{reseller}}$

- at this point of the communication the session is established, and the customer can signal to the reseller his intention to buy the product: in this simple case the customer simply sends to the reseller a *buy-product* string, encrypted with the session key. Formally: $\{buyproduct\}_{SessionKey}$

- the last message of the communication is sent by the reseller communicating to the customer that the purchase is succeeded: the message simply contains a *ack* string. Formally: $\{ack\}_{SessionKey}$

Artifact exploitation by agents

These kind of interactions involve an agent which in order to achieve one (or more) of his goals uses functionalities offered by a specific artifact. There are different kind of *use*, differing from one another for the intended agent-side semantic: i) an agent uses an artifact without expecting any particular result back, therefore continuing its activities, ii) an agent uses an artifact and needs the result in order to carry on with further activities, so he has to wait for the artifact to complete whatever operation it's engaged in and signal back the result and iii) the agent using the artifact needs the result, but later in the time, and can carry on with his own activities until the result is indispensable. In the *secure-agents* system the following agent-to-artifact interaction are identified:

- usages of the first kind are: the resellers using the *advertise* operation on the *billboard* or the *submit-new-request* on the *registry-authority*, or the *certificate-authority* using the *add-certificate* operation on the *directory-service*. All those usages has no particular usage-protocol attached
- usages of the second kind, on the other hand, are: the *certificate-authority* getting new requests from the *registry-authority*, through the *get-new-request* operation or submitting new requests to the *gui*, through the *submit-new-request*; furthermore, the customers exploiting the *lookup* operation of the *directory-service* in order to obtain a particular certificate or using the *atm* functionalities in order to withdrawn some money; the last one needs to follow a particular protocol in order to correctly carry out a withdrawn operation: as already described, the costumer first needs to invoke the *insert-badge* operation and then the *insert-response*.

Artifact manifestation

In this kind of interaction an artifact intends to manifest a certain information at the surrounding environment and interested agents. In this scenario there are an artifact acting as a *source* of signals and a set - even empty - of agents (willing to receive those kind of information);

furthermore a syntax and a particular semantic need to be defined for each of these signals.
Interactions of such a kind for the system at hand are listed below

- *simulation-state-changes*: the source is the *gui* signaling when the user change the state of the simulation; interested in this kind of signals are all the agents living within the virtual world of the simulation-instance workspace, namely *customers*, *resellers* and the *certificate-authority*. The syntax is as follow `start | pause | stop` and the semantic is self-explanatory

- *ads*: the source of this kind of signals is the *billboard* whilst the interested agents are, this time, only the customers. Their syntax is `ad <reseller-name>`, indicating that the ad perceived comes from the reseller identified with the *reseller-name* attribute

3.2.5 A comprehensive model

The following figures are an informal way of representing the system structure and main interactions: the first one, 3.1, shows an overhaul view of the system with its two workspaces and describe in detail the simple structure of the *simulator-workspace* and how it interacts with entities of the other workspace. In the second figure, 3.2, instead, is depicted the slightly more complex model of the *simulation-instance-workspace* in terms of the entities composing it (agents and artifacts) and of their relations.

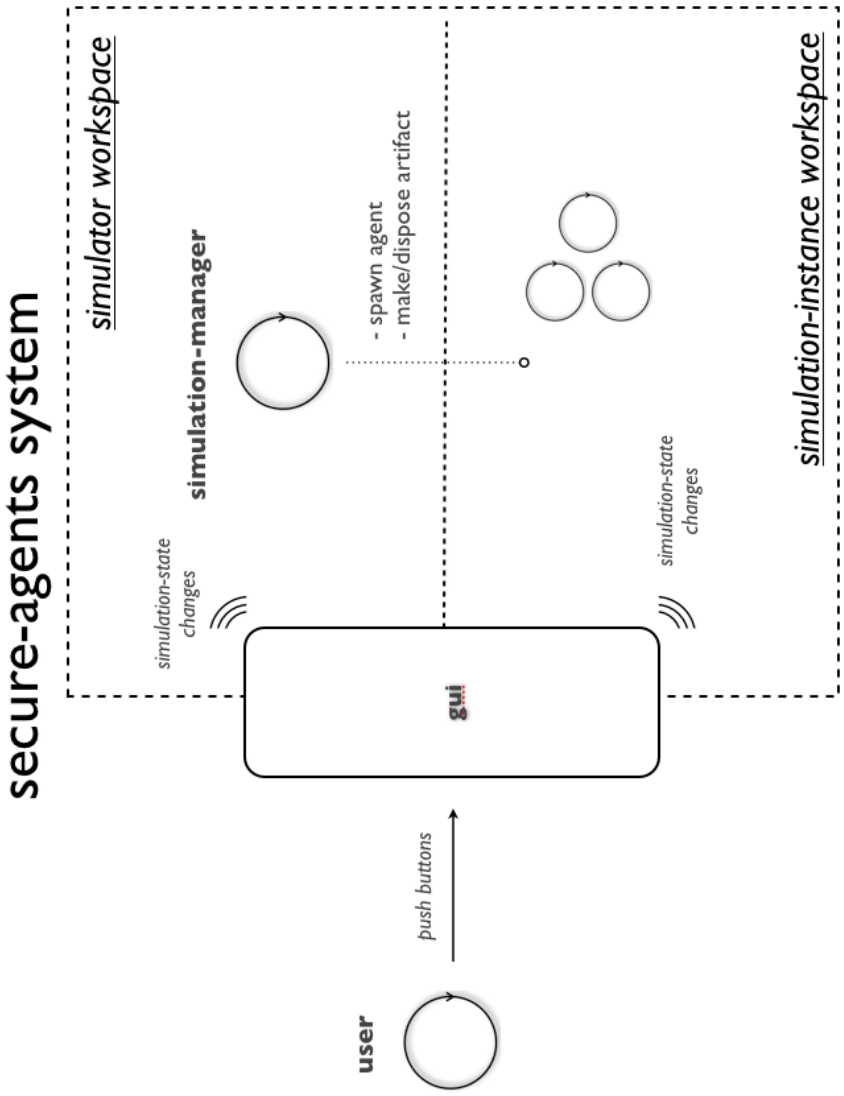


Figura 3.1: Application Model

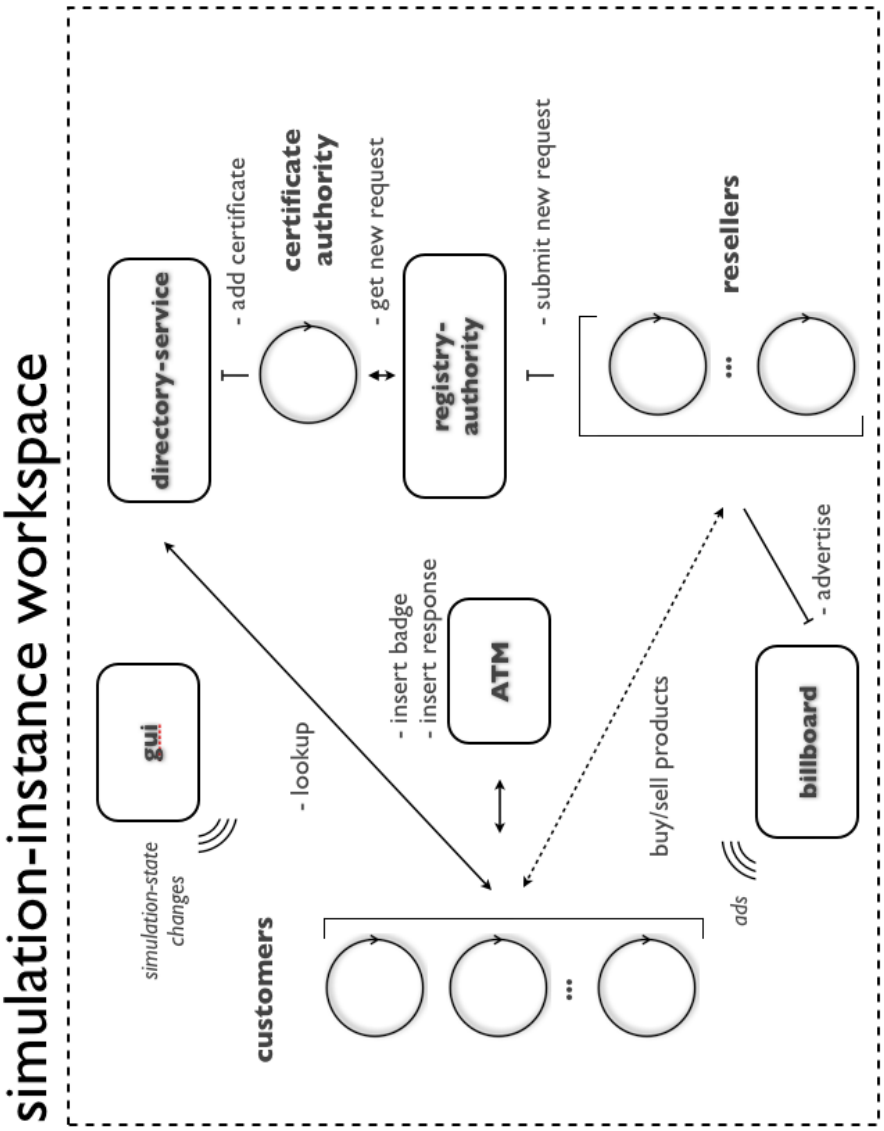


Figura 3.2: Simulation-instance model

Capitolo 4

Security in depth

Security is an hot topic in general for all applications requiring confidence, integrity, authenticity and availability (or a subset) for the information it treats. Security can be seen as a dimension orthogonal to the application domain, as a set of desired characteristics which, in order to be met, require the employment of different principles, mechanisms and tools which, in turn, add constraints on all the dimensions of the space of the systems, namely structure, interaction and behavior; for this reason, as well as these dimensions, security needs to be taken into account through the whole development process of a system (from requirements to implementation). Security engineering deals in a methodological manner with security-related problems and the conventional view is that while software engineering is about ensuring that certain things happen, security is about ensuring that they dont [1].
In this section, though already explicitly considered from the beginning of the report, a more in depth description of the system's security-related characteristics will be given, along with an explicit attempt to highlight the process through which those requirements are, first, singled out - as if only a single general requirement of security (such as 'all application's aspects have to be secured') was given instead of a much more detailed set, as it is indeed (?)- and then translated into architectural constraints and implementation mechanisms. This process goes through the phases of risk analysis, security requirements identification, design and implementation/testing.

4.1 Risk analysis and security requirements

Risk analysis is the first step of the process aimed to identify and tackle security issues arising within a certain system, placed in a given context; the goal of this first phase is to explicitly determine the system's prone-to-be-attacked goods and for each one of them assess both the cost of a possible loss and the cost of suitable countermeasures. The result of this phase are a set of high -level security requirements.
As said, in this section, the system is considered as given without detailed security requirements, with the only constraint that the environment in which multiple agents (resellers and customers) operate has to be reliable. From here a number of subsequent reasonings can be made: for first, the costumers communicating with the resellers, when purchasing an item, have to exchange sensible information (such has credit card numbers and so on), hence mechanisms to make the transactions unintelligible are needed to prevent external agents from seizing personal data. In such a context, messages can be identified as goods which have to be secured, with an high probability of being attacked and with a very high value both for the customer and for the application as a whole (the thief could steal money from the costumer without his knowledge and the system could be considered unreliable and no longer used).
A second line of thought stemming from the initial proposition leads to the identification of the *atm* as another good which needs to be protected against external intrusion: in fact, the *atm* gives costumers the possibility of withdrawing money from their account, provided that they have one, in order to purchase items from the resellers; now, let alone physical requirements (here not considered since examining a virtual *atm*), normally a correctly functioning *atm* requires its clients to pass some sort of authentication process before withdrawing money. The aim of this procedure is to safely identify clients without denying permission at eligible ones and granting it at false clients, therefore it has to be carefully designed. Another aspect of the *atm* security relies on the impossibility of external agents to come to know, in some way, one client's secret and exploit it to authenticate themselves; this is a twofold aspect: on one side there's a direct responsibility of each client

which has to be carefully trained to behave according to some rules of thumb in order to prevent social engineering. On the other side, the *atm* itself has to employ mechanisms to safely store its clients' secret in a way which prevent malevolent users to access them. Whilst the first aspect is concerned about agents behavior (treated later), the second can be directly addressed within the system carefully designing, along with the authentication process, the process of storing and accessing client's sensible data used by the *atm*.
The *atm*, as the messages exchanged between agents, are considered to be high valued both for the customers and for the system as a whole (again, if a thief can freely withdraw money from a customer's account it's a damage for both the customer itself, who looses money, and for the system as a whole, which looses credibility).
Both the above reasonings lead to a point in which agent's behavior is crucial for the safety of the entire system: each agent has to treat security-related information (such as passwords or encryption keys) in a precise way which ensures their secretness. Whilst in real-life scenario this aspect can be managed only by training people to adopt certain precautions when dealing or managing 'secrets', in a virtual world agents can be programmed to follow a particular behavioral protocol, which may employ certain tools of support, carefully designed to prevent social engineering.
Another argument that can be made about system's security is concerned with the access control, namely the policy which manage the usage of the system resources; in the case of a multi-agent system it can be translated into the way of discerning which agent can use which resources. This aspect is useful not only for system's internal security, as a means through which enable or constraint a certain agent to use (or not) a particular artifact, but also for the design of system's interaction dimension, constraining the agents' interaction space.

In the table that follows are summed up the critical aspects of the system identified and the high-level requirements stemming from them.

Critical aspect	Value	Cost	Requirements
Messages	High	Practicable	Make each exchanged message unintelligible to external observers
ATM	High	Practicable	i) Reliable authentication process. ii) Reliable mechanisms to safely store and access sensible data
Agent behavior	High	Practicable	Design a behavioral protocol to ensure secrets reserve
Access control	Medium	Practicable	Design a set of policy and mechanisms to define and implement resources access control

4.2 Security design

Once defined the security requirements demanded by the system itself and by the users of the system, the next step in the process is the one aimed at designing the model of the entire system toward the fulfillment, other than system functional requirements, of these security requirements. In doing that, other than starting from scratch, a first look has to be given to the infinite number of design patterns about security existing in literature defining architectural guidelines on how model different systems, underlined by the same security problems. In this case literature provides us with a full-fledged set of patterns suitable for the problems at hand; in the reminder of this section each of the previously highlighted problem will be taken into account and addressed following a related pattern.

4.2.1 Exchanging secure messages

Setting the stage; a first problem arises within a certain system: two different parties want to communicate with one another, exchanging

sensible information in a secure manner; a first solution leading to subsequent problems relies on the application of an encryption procedure to all outgoing messages using a secret-key shared only between the two interested parties. Such a solution does not scale well when dealing with systems in which multiple, initially unknown, customers desire to communicate with a-priori known resellers: how's possible for two strangers having a secret in common? Another solution, though still partial, can be found through the use of an asymmetric encryption schema: each agent (be it a reseller or a costumer) has its own pair of private and public key, the first one kept secret and the second one made available for every-one to know; now when two agents want to communicate they simply use the other's public key to encrypt every outgoing message and use their own private key to decrypt any ingoing message. In a perfect world, where performances are not considered, this solution would be enough, but in reality asymmetric encryption, in order to ensure an equivalent level of security, employs much more computational time than the symmetric one, making it not suitable for securing conversation concerning the exchange of many long messages. Here's the geniality: an hybrid approach using an asymmetric schema only in the first part of the communication with the purpose of exchanging a secret key and than using this one, with a symmetric schema, for the reminder of the communication. Such a schema ensures security with respect of performances, but still lacks something. How can one be sure that a particular public key belongs to a certain agent and not to another one claiming to be the first? If a system cannot provide this safety the whole security approach put in place so far is useless. What's the point of encrypting messages to make them unintelligible to others if they are sent to a wrong, and maybe malicious, receiver? At this point certificates come along: certificates (or chain of certificates), associated with each public key and released from trustworthy authorities, are the tools through which a client can be assured (within a certain degree of confidence) about the paternality of a public key.
This solution, named as a whole as PKI *Public Key Infrastructure*, is fully described, along with its components and protocols, in a set of patterns referred as *Cryptographic Key Management*; its main aspects will be here presented in a brief way.

The actors, active parts, in the solution are

- *users* - as the active parts interested in obtaining a certificate from an authority for a particular purpose, such as selling products on the net as in this case
- *clients* - as the active parts interacting with the *users* and using certificates to identify the *user*
- *certificate authority* - as the active part which handles the creation, suspension and revocation of certificates and manages certificate requests, granting or denying them after a well defined process of user's identity verification

The resources used and managed by the actors are:

- *certificates* - linking a public key to a given subject
- *directory service* - as a public available repository, in which certificates are stored for future reuse and consultation
- *registration authority* - as the mediator between *users*' certificate requests and *certificate authority*'s examination phase

Practically, *users* request for certificates via the *registration authority* whilst dually the *certificate authority* retrieves and verify them; if the verification process succeeds, a certificate related to the public key is generated by the *certificate authority* and then submitted to the *directory service* for persistent and publicly available storage. When a certain *client* needs to communicate with a particular *user*, he first has to obtain the latter's certificate via the *directory service*, then can contact him following the hybrid schema described above, using the public key contained in the just received certificate.

The *secure-agent* system employs such a solution to address communication secretiveness problems, defining at design time and implementing at run-time those components along with the interaction protocols here briefly defined and already described in the previous section.

4.2.2 ATM authentication problem

The authentication problem is another classic problem of security engineering, which arises whenever, within a system, there's the need of authorizing only certain users in doing something (i.e. accessing a resources or, as in the case at hand, withdrawing some money), basing the decision on the identity of such a user; the main problem here is to devise a process, an authentication procedure, which has to be simple to tackle by any authorized user, impossible for all the others. One of the most widely employed authentication schema relies on the use of a password, associated with each user, which as to be inserted, along with the user's name, during the process. This simple approach, if not properly adapted, can be very trivial to overcome even by non authorized users simply by the means of social engineering along with some elaborated mechanisms of brute-force attack (dictionary attack only to cite one) or with a man-in-the-middle attack. Two of the major weaknesses of the simple-password schema are i) that it is a so called passive authentication approach, in which the user unwittingly transmits his credentials to someone - allegedly a trustworthy receiver - and ii) that the credentials themselves are directly transmitted to the receiver and never changed during the time - the password is the same unless directly changed by the user. More sophisticated procedure have been proposed in order to address those problems, mainly adopting a *request/response* approach: those kind of processes relies on the use of a secret password not directly for authentication purposes, but for calculating the answer to the question posed by the authenticator. In this way, the password is never sent to the receiver - avoiding password leakage or phishing attacks - and dually the authenticator needs to know in advance the user's password in order to properly pose a correct question - assuring an user of the authenticator's identity.

In the system at hand, the atm authentication procedure employs a particular kind of the request/response approach: in particular, whenever a user wants to access the atm stating is identity, the atm poses him a question generating a random number which gets encrypted with the user's own password and then sent to the user as the request part of the request/response mechanism; if the user is indeed who he

says he is, he'll be able of decrypting the message, given the fact that he knows his own secret password, and answering with the previously random generated number to the machine as the response part of the procedure. At this point the authentication process succeeds if the number returned by the user to the atm is the same generated before, otherwise it fails. Both the atm and the user are afterward sure of the other part's identity.
The interaction protocol between the agent (`A`) and the machine (`M`) can be formalized as follows:

1. $A->M : ID\_A$
2. $M->A : \{RNG\}_{Psswd(A)}$
3. $A->M : RNG$

being `ID_A` the agent's name, `Psswd(A)` the agent's secret password shared with the atm and `RNG` the random generated number created by the atm for the particular question.

4.2.3 How to safely store and manage sensible data

Throughout this entire section the use of encryption keys and passwords has been evoked: one problem arising from their use is how they can be safely stored in a way which prevents external leakage; an example drawn from the system at hand could be an hypothetic atm which, though employing the most reliable authentication process, stores its users's passwords in a plain old text file - the strongest authentication process is useless if users's passwords can be easily accessed by anyone. In general, precautions have to be taken when storing sensible data such as passwords or encryption keys, in practice the data, before being saved are encrypted.
In the case of the *secure-agent* system both the artifacts and the agents employing security-related secrets are carefully designed following a strict usage protocol when managing such data; the protocol defines the behavior in terms of actions of the agents and in terms of operations of the artifacts when they both have to deal with those secrets, in particular:

- keys and passwords are stored in the secondary memory using the *keystore* tool, employing encryption mechanisms to mask data

- whenever keys or passwords are needed, they get retrieved from the *keystore*, kept in the primary memory for the minimum time required, in order to be used, and then removed

- though trivial, the agents cannot communicate with others their secret keys or passwords

4.2.4 Resources access control

The last security-related issue here taken into account is concerned with the problem known as resources *access control*: generally speaking, access control is aimed at allowing authorized users to access the system resources they need, while preventing unauthorized users to do the same, [5]. Many different approaches to address the problem have been proposed in literature (DAC, MAC, etc.), the one here adopted is the one which considered to be the fittest approach in agent-oriented context: *role based access control* (*RBAC*). Within this model access decisions dont depend on users names (or personal identity) but on the functions they are currently performing within the organization, namely, in more agent-oriented terms, their current role within the society. As already discussed in previous section, a *role* is a job function within the context of an organization with some associated semantics regarding the authority and responsibilities conferred to the agent that plays the role at a given time. A *permission* is an approval to perform an operation on some protected resources. Practically, access control policies in RBAC are a set of rules (permissions and denials) specifying which roles can access which operations on each resource composing the system environment. Hence, an agent, when playing a certain role, has to be forced to obey, by the mechanisms implementing the policy, the access rules constraining its role.
In the *secure-agents* system, for simplicity, it is adopted a closed-policy assumption, which states that what is not explicitly granted is to be considered forbidden; starting from here are defined then solely which

roles (*reseller*, *costumer*, etc.) exploit which operations on the artifacts, being all other interactions implicitly not allowed. The rules used within the system at hand can be summarized as depicted in the table below.

Role	Artifact [allowed operations]
Customer	ATM [`insert-badge`, `insert-response`] Directory-Service [`look-up`]
Reseller	Registration-Authority [`submit-CSR`] Billboard [`advertise`]
Certificate-authority	Registration-Authority [`get-new-request`] Directory-Service [`addCertificate`, `look-up`] GUI [`submit-new-request`]

In order to conclude this brief part on RBAC, it is worth pointing out two interesting things: the first is how, in this little example, the access rules resembles very much the interactions depicted in the previous section, that is roles can use only what they need in order to achieve their goals; this is partly true in the example at hand only because the simplicity of the entities employed, but in general access rules should be given according to what a role could, in line of principle, be allowed to do within an environment and not only to what it actually needs to do to bring about a particular state of affair, that is access rules should not be tailored on a particular business logic only. The second point interesting to be made is that, whilst in the *secure-agent* system roles are fixed at design time, in general agents during their lifecycle within an environment play several roles, even more than one concurrently; in those more complex situations, role access rules will also contain *static* and *dynamic separation of duty* constraints (SSD and DSD), which impose limits on the simultaneous roles an agent can play in order to avoid conflict of interests (such as the controlled cannot act as a controller and vice-versa) or granting more privileges than the ones strictly needed (violating the principle of minimum privilege).

Capitolo 5

Implementation, deployment and usage

After a lot of thought about a certain problem, eventually one has to get practical or the product, in this case an application, remains an abstract object and doesn't get built.
The final steps of the implicit process used to develop the system are *implementation*, *deployment* and *execution*. The reminder of this section comprehends a few notes about some critical implementation choices which have to be made, a brief explanation about how to configure and run the system and, finally, some snippets of the app's gui will be given along with a very short description of its usage.

5.1 Implementation

As already discussed in the first chapter of this report, the meta-model and subsequent methodology which drove the entire analysis and design steps were already selected, namely the *A&A* meta-model and the *SODA* methodology. Now in the implementation phase the entire system could be built using, in line of principle, whatever language, or framework or a set of them, comes to mind. For example it would be certainly possible to implement it using a today mainstream object-oriented language, such as Java, but that will lead to a wide semantic gap between the domain model chosen for the problem and the abstraction at hand to really build the system - lot of code not pertinent

with the business logic of the application would have to be written only to fill that gap.
It would be also possible to pick a general and pure agent-oriented programming language in which hold the statement *everything is an agent*, such as pure Jason, JADE or alike, but in those languages totally lacks a notion of environment as a first class abstraction. Here is considered as environment only what is external to the MAS and not those entities internal to the MAS itself that the agents can exploit for internal reasons (such as providing functions for or coordination among agents) or for bringing the real external environment (here namely what is not part of the system) to the cognitive level of the agents in a way which is consistent to the system logic; here again a semantic gap would exist, though less wide than the previous one.
Discarded pure agent-oriented frameworks or languages, there are still plenty of languages and frameworks which could be used, the most of which are slightly different implementations of the BDI abstract architecture (as most of the main pure agent-oriented languages cited above) in which agents are described has having mental attitudes such as belief intentions and desires and the goals they have to achieve are explicitly represented as a particular state of the agent's bb; those kind of languages having to do with a strong notion of agency, in which agents exhibit a some sort of intelligence, introduce a great deal of complexity , here not needed, and furthermore deal with abstractions, though mappable, different from the one used in the analysis and design phases.
The framework of choice in this case is simp-A (along with CartAgO), an annotation-based extension of the Java language, which adopts the *A&A* meta-model and provides the latter's concepts (*agent*, *activity*, *artifact*, *operation*, etc.) as first class abstractions. This is the most suitable choice due to the one-to-one relation between the results of the *detailed design* phase and the conceptual tools the framework suggests.
Implementation details are here not described, since they are outside the scope of this report.

5.2 Deployment

In this section will be briefly given some advices on how to run whole application.
The *secure-agent* system comes packeted into a compressed file, which once opened presents itself as a folder named *secure-agents* with the following internal structure:

+doc +img +lib +src +secure-agents.jar

Though the application could be executed simply double-clicking the *.jar* file, the most verbose method to do it is using the command-line, with which is possible to follow in a much deeper and detailed way what is really going on within the simulation system; in a terminal window type the following:

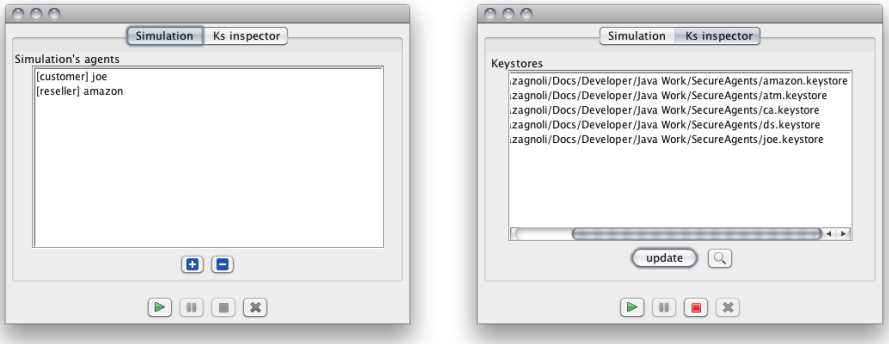
```
$ cd <path-of-the-directory>/secure-agents
$ java -jar secure-agents.jar
```

5.3 Usage

This section contains a brief explanation of the main commands appearing in the application's gui.
Once started the application shows its main window, with which is possible interacting with the simulator in several ways:

- Control the simulation life-cycle, through the bottom-bar buttons (*start*, *pause*, *stop* or *cancel* a simulation-instance)
- Only from the *Simulation tab*
  - control the simulation environment, adding or removing acting agents (both costumers and resellers) through the *plus* and *minus* blue-and-white buttons
  - inspecting the current simulation-instance's population
- Only from the *Ks-inspector tab*
  - inspecting the key-stores currently used within a simulation-instance

– internally inspecting one of the key-stores



(a) Main gui - simulation environment set-up (b) Main gui - key-store inspector

Figura 5.1: The main window of the application

In order to properly run a simulation instance, the environment must be populated with at least one agent of each kind, namely a *customer* and a *reseller*; when the system starts-up, the environment already contains a *customer*, *joe*, and a *reseller*, *amazon*, in this way it would be already possible starting a first simulation, otherwise it is possible to add, or remove, agents through the ad-hoc buttons. Once the environment is properly initialized the related simulation-instance can be started, with the apposite button; the simulation can be paused and then restarted for as long as wanted, but once it has been stopped can no longer be resumed - any subsequent pressure on the start button will create and run a brand-new simulation-instance with the same environment configuration as the one before.

Capitolo 6

Conclusions

In this brief report has been presented the process of developing a simple system employing agent-oriented meta-models, methodology and framework, posing particular focus on the phases of analysis and design.

It is worth saying, for the sake of correctness, that, in general, a problem is first analyzed in absolute terms, without any bias from a chosen meta-model, programming paradigm or underlying platform, hence trying to think at the problem itself and the concepts it suggests rather than thinking in terms of its solution. This procedure would allow the analysis to endure even after a change in the underlying stack of support (programming language, frameworks, mete-models, etc). Once the problem has been properly examined and the concepts it proposes have been singled out, the suitable tools to design and then implement the system can be chosen.
That being said, the approach used here is slightly different: first the agent abstraction has been highlighted as the one suitable to model the system, then the A&A meta-model, and the SODA methodology, have been chosen as the set of concepts and guidelines to follow and only at this point the system gets thoroughly analyzed and designed with a bottom-up mind-set: looking at it from a solution point-of-view, influenced by the meta-model's abstractions.
This inversion of order was mainly due to the need to use and put in practice agent-oriented theories, test them against a simple, but com-

prehensive, case study and draw some conclusions about the process as a whole, the models and the tools.

The main question to ask in conclusion, since the mandatory choice made at the beginning, is whether or not the agent-oriented paradigm and abstraction, in general, and the A&A meta-model, along with the SODA notions, have been adequate to analyzed and model the system at hand.
The agent abstraction can be seen, from a software engineering prospective, the natural extension of the concepts of modularity and encapsulation [7] and is suitable to characterize problems which can be seen as a set of autonomous but interacting activities: in this case, as emerged from the analysis, the *secure-agents* system is made-up by different actors, playing different roles, and communicating one another in order to carry out the simulator's tasks. But activities are not the whole system, resources too can be identified as taking part, though not an active one, as functionalities suppliers, mediators among agents and between agents and the external environment - examples of such resources can be the ATM or the billboards. Mapping those entities into agents would force the designer to model them as activities, which clearly are not, incurring in a stretch of abstraction. The A&A meta-model introduces the concept of artifact to explicitly address this need, directly conceiving purely reactive resources as first class abstractions through which tailor the agent's environment in order to provide suitable services.

A methodology can be seen as different set of notions, each at an always decreasing level of abstractions, which in a coherent and systematic way guide the construction of a system starting from its requirements down to its design model; the central part of the report has been focused on the analysis and design of the secure-agent system following the SODA methodology, which, following the A&A meta-model, proposes concepts like tasks, activities, actions and agents to lead the modeling of the active parts of the system and notions like functionalities, resources, operations and artifacts to suitably engineer the environment mediating and supporting those activities. Such a layered set of abstractions, on one hand, has been proven useful to de-

scribe the solution at a proper level, managing the complexity mainly derived from the presence of multiple active parts interacting with one another. One the other side, the methodology as a whole has provided a guideline to systematically transform a set of system requirements into a detailed model of its solution. One thing worth noting is that in this simple case study, the system was quite straightforward whilst the artifacts (in this case tables) produced by the different phases of the methodology were quite populated - containing many records; as the system complexity scales up, there is the need for tools to help the designer coping, in an automatic, way with those artifacts.

At the implementation phase, the `simpA` framework has been used; this has led to two main consequences, the first one being an almost one-to-one map between the concepts used to model the system, outcome of the detailed design phase, and the abstractions that the frameworks makes available - facilitating the phase itself. The second consequence stems from an inherent difference between the used framework and other agent-oriented frameworks: `simpA` and its runtime environment deal with agent-oriented abstractions from a more practical software-engineering view-point, concentrating its attentions to the agent pro-activity part and their relations (putting in place concepts as *activity* and *agendas*), demanding to the developer the burden to directly manage the reactive part - in practice ad-hoc activities which continually sense the environment have to be created. Other agent-oriented programming languages or framework (such as Jade or Jason) adopt a different approach, interleaving actions and perceptions, adapting agent's individual goals accordingly to the situation.
Each one of these two different characterizations of the agent internal architecture has its own pros and cons. `SimpA` has been chosen for its pragmatism and because it's essentially annotated Java, a well known programming language; on the other side it completely lacks that trade-off between pro-activity and reactivity distinguishing the agent notion (and which makes other frameworks more coherent with the more diffuse idea of agent), leading to less responsive systems, unless carefully designed.
However the need for an ad-hoc programming language properly de-

signed and tailored upon the agent abstraction, though adopting a software-engineer view, is needed and, at this point in time, almost due.

The report is concluded highlighting the main parts of the system that, for the sake of simplicity, have been oversimplified or not properly addressed. The first one is the communication between the agents: the *secure-agent* involves the communication between customers and resellers, which, in this case, employs simple strings and is not based on any common notion of ontology using any implementation of ACL. The second simplification here made is concerned with RBAC specification here concerned and implemented only considering static rules, without explicitly dealing with dynamic one.

Bibliografia

[1] R. J. Anderson and R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 1 edition, January 2001.

[2] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III. Agent Theories, Architectures, and Languages*, volume 1193 of *LNCS*, pages 21–35. Springer, 1996. ECAI'96 Workshop (ATAL'96) Budapest, Hungary, 12–13 Aug. 1996. Proceedings.

[3] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[4] N. Jennings. An agent-based approach for building complex software systems. In *Communications of the ACM*, Apr. 2001. Vol. 44, No. 4.

[5] A. Molesini, E. Denti, and A. Omicini. Rbac-mas and soda: Experimenting rbac in aose. In *ESAW*, pages 69–84, 2008.

[6] A. Molesini, E. Denti, and A. Omicini. Agent-based conference management: a case study in soda. *Int. J. Agent-Oriented Softw. Eng.*, 4(1):1–31, 2010.

[7] A. Natali and A. Molesini. *Costruire sistemi software: dai modelli al codice*. Number 9788874883349 in Progetto Leonardo. Esculapio, Bologna, Italy, second edition edition, November 2009.

[8] A. Omicini, A. Ricci, and M. Viroli. *Agens Faber*: Toward a theory of artefacts for MAS. *Electronic Notes in Theoretical Computer Science*, 150(3):21–36, 29 May 2006. 1st International Workshop "Coordination and Organization" (CoOrg 2005), COORDINATION 2005, Namur, Belgium, 22 Apr. 2005. Proceedings.

[9] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, Dec. 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.

[10] The soda methodology. `http://apice.unibo.it/xwiki/bin/view/SODA/`. ApiCe working group.

[11] I. Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010.

[12] M. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2009.

Elenco delle figure