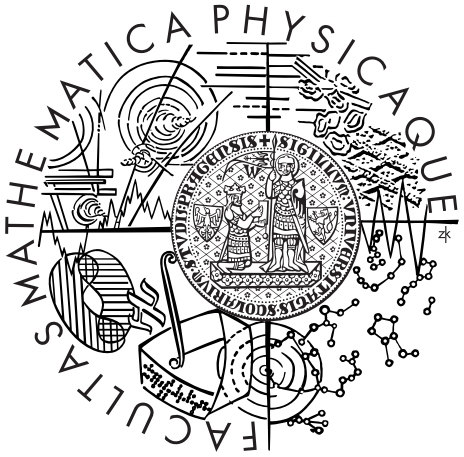
Charles University in Prague

Faculty of Mathematics and Physics

MASTER'S THESIS



Bc. Lukáš Kúdela

Multi-Agent Systems and Organizations

Department of Theoretical Computer Science and Mathematical Logic

Thesis supervisor: Prof. RNDr. Petr Štěpánek, DrSc.

Study programme: Computer Science (N1801)

Specialization: Theoretical Computer Science (1801T010)

Prague 2012

First I would like to thank my supervisor, prof. RNDr. Petr Štěpánek for sparking my interest in multi-agent systems, coming up with a interesting and challenging topic and for the time we have spent together discussing matters more or less related to the subject of this thesis. I would also like to say that I am thankful to my parents, Libuša and Ivan, who have always believed in the importance of quality education and made their best to ensure their children receive the best one possible. I am also very grateful to them for never giving me a reason to worry about issues that would prevent me from fully concentrating on my studies. Next, I would like express to my gratitude to my brother, Jakub, who was helping me with editing the images in this thesis. Finally, I would like to thank Monika, who has been bearing all the consequences of having a thesis-writing boyfriend and has managed not to lose her nerve. Once again, thank you!

Najprv by som chcel poďakovať môjmu vedúcemu, prof. RNDr. Petrovi Štěpánkovi, za vzbudenie môjho záujmu o multiagentové systémy, navrnutie zaujímavej témy, ktorá bola výzvou a za čas, ktorý sme spoločne strávili diskutovaním o veciach viac či menej súvisiacích s predmetom tejto práce. Chcel by som taktiež povedať, že som vďačný svojím rodičom, Ivanovi a Libuši, ktorí vždy verili, že kvalitné vzdelanie je dôležité a urobili všetko preto, aby ich deti dostali to najlepšie možné. Som im taktiež vďačný za to, že mi nikdy nedali dôvod trápiť sa záležitosťami, ktoré by mi zabránili plne sa sústrediť na štúdium. Ďalej by som chcel vyjadriť vďačnosť môjmu bratovi, Jakubovi, ktorý mi pomáhal s upravovaním obrázkov v tejto práci. Napokon by som chcel poďakovať Monike, ktorá znášala všetky následky toho, že jej priateľ písal túto prácu a podarilo sa jej pri tom nestratiť nervy. Ešte raz, ďakujem Vám!

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

Prague, April 10, 2012 Lukáš Kúdela

Title: Multi-Agent Systems and Organizations
Author: Bc. Lukáš Kúdela
Author's e-mail address: lukas.kudela@gmail.com
Department: Department of Theoretical Computer Science and Mathematical Logic
Thesis Supervisor: Prof. RNDr. Petr Štěpánek, DrSc.
Supervisor's e-mail address: petr.stepanek@mff.cuni.cz

Abstract: Multi-agent systems (MAS) are emerging as a promising paradigm for conceptualizing, designing and implementing large-scale heterogeneous software systems. The key advantage of looking at components in such systems as autonomous agents is that as agents they are capable of flexible self-organization, instead of being rigidly organized by the system's architect. However, self-organization is like evolution—it takes a lot of time and the results are not guaranteed. More often than not, the system's architect has an idea about how the agents should organize themselves—what types of organizations they should form. In our work, we tried to solve the problem of modelling organizations and their roles in a MAS, independent of the particular agent platform on which the MAS will eventually run. First and foremost, we have proposed a metamodel for expressing platform-independent organization models. Furthermore, we have implemented the proposed metamodel for the *Jade* agent platform as a module extending this framework. Finally, we have demonstrated the use of our module by modelling three specific organizations: remote function invocation, arithmetic expression evaluation and sealed-bid auction. Our work shows how to separate the behaviour acquired through a role from the behaviour intrinsic to an agent. This separation enables organizations to be developed independently of the agents that will participate in them, thus facilitating the development of the so-called open systems.

Keywords: multi-agent systems, organizations, roles, metamodel

Názov: Multiagentové systémy a organizácie
Autor: Bc. Lukáš Kúdela
E-mailová adresa autora: lukas.kudela@gmail.com
Katedra: Katedra teoretické informatiky a matematické logiky
Vedúcí práce: Prof. RNDr. Petr Štěpánek, DrSc.
E-mailová adresa vedúceho: petr.stepanek@mff.cuni.cz

Abstrakt: Multiagentové systémy (MAS) sa ukazujú ak sľubná paradigma pre konceptualizáciu, návrh a implementáciu rozsiahlych heterogénnych softvérových systémov. Hlavná výhoda nazerania na komponenty v takých systémoch ako na autonómne agenty spočíva v tom, že ako agenty sú schopné flexibilnej samoorganizácie, namiesto toho, aby boli rigidne zorganizované systémovým architektom. Avšak, samoorganizácia je ako evolúcia—vyžaduje veľa času a výsledky nie sú zaručené. Systémový architekt má často predstavu o tom, ako by sa mali agenti organizovať—aké typy organizácií by mali vytvárať. V našej práci sme sa pokúsili vyriešiť problém modelovania organizácií a ich rolí v MAS, nezávisle na konkrétnej agentovej platforme, na ktorej MAS napokon pobeží. V prvom rade sme navrhli metamodel na popis platformovo nezávislých organizačných modelov. Ďalej sme navrhnutý model implementovali pre agentovú platformu *Jade* ako modul rozširujúci tento framework. Napokon sme predviedli použitie nášho modulu namodelovaním troch konkrétnych organizácií: vzdialené volanie funkcie, vyhodnotenie aritemetického výrazu a aukcia obálkovou metódou. Naša práca ukazuje ako oddeliť rolou nadobudnuté chovanie od chovania, ktoré je neoddeliteľnou súčasťou agenta. Táto separácia umožňuje, aby boli organizácie vyvýjané nezávisle od agentov, ktoré v nich budú participovať a tým uľahčuje vývoj tzv. otvorených systémov.

Kľúčové slová: multiagentové systémy, organizácie, role, metamodel

Contents

Introduction **1**

1 Agents and Multi-Agent Systems **3**

1.1 Agents 3

1.2 Multi-Agent Systems 5

2 Organizations in Multi-Agent Systems **7**

2.1 Agent-Centric Multi-Agent Systems 8

2.2 Organization-Centric Multi-Agent Systems 9

2.3 Organizational Concepts 10

3 Modelling Organizations—Existing Approaches **11**

3.1 Models and Metamodels 11

3.1.1 Models and Modelling 11

3.1.2 Metamodels and Metamodelling 12

3.2 Aalaadin 13

3.2.1 Core Model 14

3.2.2 Methodological Model 16

3.2.3 MadKit 17

3.3 O&P 17

3.3.1 Integrated Metamodel 18
3.3.2 Agent Classifiers and Agent Model 18
3.3.3 Group Model 21
3.3.4 Agent Role Assignment 24
3.4 PIM4Agents 25
3.4.1 JadeOrgs 26
3.5 powerJade 27

4 Platform-Independent Metamodel—Thespian 31
4.1 Static Model 32
4.1.1 Organization Model 33
4.1.2 Player Model 34
4.1.3 Protocol Model 35
4.1.4 Design-Time Model 36
4.1.5 Run-Time Model 37
4.1.6 Integrated Static Model 37
4.2 Dynamic Model 38
4.2.1 Player and Organization Interaction 38
4.2.2 Player and Role Interaction 42

5 Metamodel Implementation—Thespian4Jade 47
5.1 Agent Platform 47
5.2 Jade 48
5.3 Thespian4Jade 49
5.3.1 States and Parties 49
5.3.2 Organization, Role and Competences 51

5.3.3 Player and Responsibilities 54

5.3.4 Protocols and Messages 56

5.3.5 Utilities 58

6 Examples **61**

6.1 Example 1: Remote Function Invocation 62

6.2 Example 2: Arithmetic Expression Evaluation 68

6.3 Example 3: Sealed-Bid Auction 77

Conclusions and Future Work **81**

Bibliography **83**

List of Figures **88**

A CD-ROM Contents **93**

Introduction

Autonomous agents were conceived by the artificial intelligence community as entities capable of autonomous behaviour in their environment, most preferably a behaviour that could be considered intelligent. Distributed AI community was especially interested in groups of such agents sharing a common environment and capable of interacting with one another—multi-agent systems. Multi-agent systems have also attracted a fair amount of attention from the software engineering community in the recent years because they represent a new way of harnessing the complexity involved in analysing, designing and ultimately implementing large-scale heterogeneous software systems.

Agent-oriented programming is the latest stage in the evolution of programming paradigms. Technologically, it may not seem like a big improvement over its predecessor—object-oriented programming. After all, contemporary large-scale multi-agent systems are implemented mostly in object-oriented languages, and agent-oriented languages are experiencing difficulties in finding their way from academia to industry. However, agent-oriented programming represents a huge leap conceptually. It is a lot easier for a human mind to think in terms of agents exchanging messages with one another than in terms of objects invoking methods on one another simply because the former resembles something people are very familiar with—a human society. Therefore, it makes perfect sense to turn our attention to human societies when looking for ways to expand our notion of multi-agent systems and deepen our understanding of them.

In almost any society, organizations of all kinds exist: be they explicit (e.g. a business company) or implicit (e.g. a group of friends playing football), permanent (e.g. a family) or temporary (e.g. a buyer-seller interaction). The members of an organization play roles that exist in this organization and interact according to protocols defined between these roles. Organizations have emerged as a natural way to institutionalize behavioural patterns and patterns of interaction, and be doing this they make human interaction more predictable and as such more efficient.

It is important to realize that by becoming a member of an organization and assuming a role in it, a person is not giving up their individuality or autonomy. They are still allowed (and often expected) to bring their personal approach to the role they play, and most importantly, they are free to leave the organization on terms known at the time of entering it.

Since there are multi-agent systems that would benefit from the predictability of agents' behaviour and interaction, there is a demand for approaches to model organizations in multi-agent systems.

The aim of this thesis is to provide theoretical foundation, together with design and implementation tools, to model organizations in multi-agent systems. This general aim can be broken down into four specific objectives:

1. Investigate some of the metamodel-based approaches to modelling organizations in multi-agent systems.
2. Propose a new organizational metamodel inspired by the existing ones.
3. Implement this metamodel in a free and open-source mainstream general-purpose agent platform.
4. Demonstrate its use with a number of examples.

The ultimate goal is to contribute to the effort of making multi-agent systems a more powerful paradigm for conceptualizing, designing, and implementing software systems.

The rest of this thesis is organized as follows. Chapter 1 is a brief overview of autonomous agents and multi-agent systems and can be skipped by a reader familiar with them. In chapter 2, we discuss the motivation for introducing organizations to multi-agent systems and talk about some key organizational concepts for the first time in an informal manner. In chapter 3, we present four existing metamodel-based approaches to modelling organizations in multi-agent systems from which we drew inspiration for our own work. Chapter 4 is where the presentation of our work begins. It introduces *Thespian*—a platform-independent metamodel for modelling organizations in multi-agent systems. It is the core chapter of this thesis. In chapter 5, we introduce *Thespian4Jade*—a platform-specific implementation of *Thespian*. This chapter describes all important packages and classes and provides guidelines on using them to develop organization-centric multi-agent systems in the *Jade* agent platform. In chapter 6, we demonstrate the use of *Thespian4Jade* with three examples of organization-centric multi-agent systems: function invocation, expression evaluation and auction. The last chapter concludes our discussion and suggests possible directions for future work.

Chapter 1

Agents and Multi-Agent Systems

The purpose of this chapter is to introduce *agents* and *multi-agent systems*. The introduction is kept as short as possible, sticking to the most fundamental characteristics. Nevertheless, it provides all necessary background to understand the ideas presented in this thesis.

A slightly more detailed overview of multi-agent systems can be found in [5] and [6]. A more in-depth (yet still general) introduction [1] has become the *de facto* standard textbook on multi-agent systems. [2] is another excellent introduction to multi-agent systems in the context of distributed artificial intelligence. Alternatively, in [3] the authors take an algorithmic, game-theoretic and logical[1] approach to multi-agent systems. To our knowledge, [4] is the most complete coverage of multi-agent systems in Czech, while [7] is an overview by the same author.

1.1 Agents

Unfortunately, no single and universally accepted definition of an agent exists in the agents community. However, this does not seem to be a problem at all; despite the lack of agreement on terminological details, many researchers are coming up with interesting agent theories and numerous practitioners are developing useful agent applications. Still, it is important that at least some definition of an agent exists—if for nothing else, then to protect it from being overused and thus stripped of any meaning.

Two usages of the term *agent* can be recognized. The first is weaker and does not appear to be disputed; the second is stronger and generates more discussion in the community. In this thesis, it is sufficient to use the weaker notion of agency.

An *agent* is a computational entity that is *situated in an environment* and that

[1] As in mathematical-logical, not rational.

is *capable of autonomous action* in this environment in order to meet its design objectives [6]. Being situated in an environment means that the agent can *sense* (or *perceive*) it through its *sensors* and *act* upon it through its *actuators* (or *effectors*). Note that the definition does not specify the *type* of environment an agent inhabits; agents can occupy many different types of environments, for example a physical, virtual or software environment. Being capable of autonomous action means that the "agents are able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behavior." [6]

An important type of agent—especially in artificial intelligence—is an *intelligent agent*. In [6], an *intelligent agent* is defined as an agent capable of *flexible* autonomous action in order to meet its design objectives, where flexibility means three things:

- *reactiveness*—an ability to respond in a timely fashion to changes that occur in the agent's environment,
- *proactiveness*—an ability to exhibit goal-directed behaviour by taking the initiative, and
- *social ability*—an ability to interact with other agents (and possibly humans).

Figure 1.1 shows an abstract view of agent interacting with its environment. The agent obtains *percepts* from the environment as input (using its sensors) and exerts *actions* upon the environment as output (using its actuators). This interaction is usually an ongoing, non-terminating one.

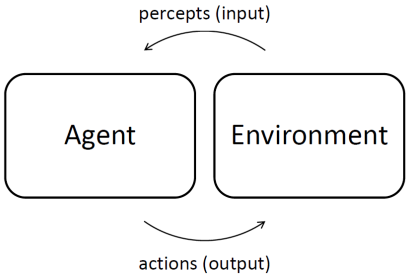


Figure 1.1: An agent interacting with its environment

An example of an agent is a *softbot*—an agent situated in a software environment interacting with it via commands. A softbot's sensors are commands meant to provide information about the environment (e.g. `ls` or `pwd` in Unix) and its effectors are commands meant to change the environment (e.g. `mv` or `compress` in Unix).

1.2 Multi-Agent Systems

There is a popular slogan in the multi-agent systems community: "There's no such thing as a single agent system" [1].

Multi-Agent system[2] (MAS) is a system composed of multiple interacting agents.

Like a single agent,a MAS is situated in an environment, which means that all its constituent agents are situated in the same environment. While the field of agents studies agents' interaction with their environment and their reasoning about it, the field of MASs deals with agents' interaction with other agents and their reasoning about them. The inter-agent interaction can be either *direct* (via messages) or *indirect* (via environment).

In this thesis, we do not consider agents' interaction with their environment and we do consider only direct inter-agent interaction—*communication*. Hence, we do not assume anything about the environment in which MASs are situated. In fact, as far as the discussion in this thesis goes, MASs do not have to be situated in any environment, i.e. situated in an *empty* (non-perceivable, non-affectable) environment.

Most importantly, we do not assume any social structure of MASs; a MAS is really just a group of agents with no special relationships among them. In other words, all agents are equal. It is the aim of this thesis to introduce social structure to MASs.

[2] Alternatively spelled multiagent system.

Chapter 2

Organizations in Multi-Agent Systems

"The achievements of an organization are the results of the combined effort of each individual."
— Vince Lombardi, American football coach

In this chapter, we discuss the motivation for introducing organizations to MASs, introduce two conceptions of a MAS and talk about some key organizational concepts for the first time in an informal manner.

Consider a problem with the following properties:

- It can easily be *decomposed* into well-defined subproblems. These subproblems can either be decomposable themselves or atomic. Note that the subproblems do not necessarily have to resemble the original problem or each other[1].
- The subproblems can be solved more or less *independently* and therefore, providing enough computational entities are available, concurrently. Note that this is a quantitative characteristic, not a categorical one.

Such a problem can be solved by first pondering how a society of humans (viewed as intelligent autonomous computational entities) would go about solving it, then modelling this society as a MAS and finally running the system. However there is an issue we must address—organizational structure of human societies.

In all human societies but the most primitive ones, various types of *organizations* emerge to facilitate cooperation among their members. In these organization types *roles* appear and *interaction protocols* governing their interaction crystallize

[1] A characteristic necessary for divide and conquer algorithms.

among them. We are facing the challenge of carrying these *organizational concepts* over to the realm of MASs.

Unfortunately, there is no standardized way to impose organizational structure upon MASs. It should come as no surprise to anybody who is familiar with MASs that no single, precise and universally accepted notions of organization, role or interaction protocol currently exist among researchers. In a plain vanilla MAS, every agent can (in principle) talk to any other agent, regardless of whether this is desirable or even allowed in the society being modelled by the MAS.

In the next two sections, we will introduce two ways of looking at MASs:

- *agent-centric*, focused on the structure of individual agents (the traditional viewpoint), and
- *organization-centric*, focused on the structure of agent societies (a novel perspective).

2.1 Agent-Centric Multi-Agent Systems

An *agent-centric multi-agent system* (ACMAS) is the classical conception of a MAS. It focuses on the architecture of individual agents, being oblivious to the structure of their society.

An ACMAS has the following characteristics [16]:

- Every agent has a public *agent identifier*[2] and it can be addressed with it.
- An agent can communicate with any other agent[3].
- An agent provides a set of services, which are available to every other agent in the system.
- It is the responsibility of each agent to constrain its accessibility and the accessibility of its services to other agents.
- It is the responsibility of each agent to define its relations, contracts, etc. with other agents.

Perhaps ironically, the absolute freedom of interaction in an ACMAS is the cause of many of its shortcomings [16]:

[2] *Agent identifier* (AID) is a name that identifies (that is, labels the identity of) a unique agent.

[3] Of course, the agent needs to know the other agent's AID, but since these are public, this is not an obstacle.

- Predicting the behaviour of the whole system from the behaviour of its constituent components is extremely difficult, if not downright impossible, due to high probability of emergent behaviour.

- Because there is no implicit security management, it is easy for a malicious agent to unknowingly misuse or even intentionally abuse the system.

- It is not possible to apply the principles of *modular design*. Agents cannot be grouped into modules with different visibilities to the outside world (public vs. private) at design-time, let alone at run-time.

- It is not possible to pursue the *framework approach*. There is only one framework—the agent platform itself—and it is impossible to define sub-frameworks with specific interactions.

2.2 Organization-Centric Multi-Agent Systems

An *organization-centric multi-agent system* (OCMAS) is the modern conception of a MAS proposed in [16]. It focuses on the structure of an agent society, paying no attention to the architecture of the individual agents.

Organizations provide a natural way of describing structure of a MAS and interactions among its constituent agents. This description is situated on the *organization level* of an OCMAS—the level above the *agent level*, which is the only level considered in an ACMAS. The organizational level contains abstract representations of the concrete organizations occurring on the agent level.

The following are the characteristics of an OCMAS [16]:

- The organizational level imposes social structure and patterns of interaction upon agents, but does not prescribe how agents should behave; it merely demarcates the space within which the agents can express their individuality.

- The organizational level does not place constraints on the architecture of the agents; deliberative as well as reactive agents can take part in an organization as longs as they behave in an expected way.

- The organizations provide a way to partition a MAS into bounded contexts of interaction. Whereas the structure of an organization is known to its members who are able to interact with one another, it is opaque to the non-members whose interaction with the organization (and its members) is limited.

2.3 Organizational Concepts

An *organization* is a a structured group of agents, which imposes rules on the behaviour and mutual interaction of its members. These rules are imposed by roles and interaction protocols defined in the organization.

A *role* is an interface between an organization and its member—the organization interacts with its members through their roles. Is is also an interface between the organization members themselves—the members interact with each other via the roles they play. A role always exists and operates within the context of its defining organization.

When playing a role, a player is entitled to exercise the role's competences but also obliged to fulfil its responsibilities. A *competence* is an operation the role's player *can* perform as a result of playing that role. A *responsibility* is an operation the role's player *has to* perform as a consequence of playing that role.

An *interaction protocol* is a institutionalized pattern of interaction[4] between two or more roles in an organization. It defines by intension a set of possible communication scenarios between the players of these roles. In the context of a protocol, the participating roles (or their players) are called *parties*.

[4]In this thesis, the only kind of interaction we consider is communication. Therefore, we will use the terms *interaction* and *communication* interchangeably.

Chapter 3

Modelling Organizations—Existing Approaches

In this chapter, we will introduce existing metamodel-based approaches to modelling organizations in MASs. The initiative to model organizations in MASs using platform-independent metamodels began with the publication of a seminal paper [13] by Jacques Ferber and Oliver Gutknecht and continues to this day. We will introduce four metamodels: *Aalaadin*, *O&P*, *PIM4Agents* and *power-Jade*. All of them have influenced the design of our metamodel to a greater or lesser degree.

In software engineering, a *platform-independent model* (PIM) is a model of a software system, that is independent of the specific technological platform used to implement it [51]. The main motivation to use a PIM is to build the model once and then automatically transform it to any number of platform-specific models for different deployment platforms.

The *platform-specific model* (PSM) is a model of a software system, that is bound to a specific technological platform, for example, a hardware environment (processor), operating system or software environment (virtual machine)). PSMs are indispensable for the actual implementation of a software system [52]

3.1 Models and Metamodels

3.1.1 Models and Modelling

Before talking about metamodels and metamodelling, its absolutely necessary to have a clear understanding of models, modelling and two basic relationships:

representation and conformance.

A *model* is a simplified *representation* of a certain reality, for example, a system [10]. A system can be represented by a set of different models. Each model captures a specific aspect (or view) of the system, depending on the purpose of that particular model. A model must not represent the system with absolute preciseness; it is useful only because it is a simplified representation [10].

A model also has to be expressed in some modelling language. Therefore, the full definition of a model is the following: a *model* is a simplified *representation* of a certain reality *conforming* to the rules of a certain modelling language [10]. In short, a model represents a system and conforms to a metamodel. Figure 3.1 illustrates both relationships.

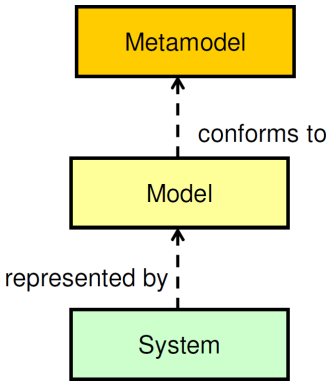


Figure 3.1: The representation and conformance relationships [10]

Modelling, in the most general sense, is the use of a model to represent a certain reality for some cognitive purpose.

A model is characterized by *contextual substitutability*—it should be able to answer a given set of questions in the same way the system would answer them [10].

3.1.2 Metamodels and Metamodelling

A *metamodel* is a special kind of model that specifies the abstract syntax of a *modelling language* [11].

A metamodel represents an abstract syntax of a modelling language; it does *not* represent a model or a set of models[1]. A metamodel conforms to a meta-metamodel.

Consider the metalayers shown figure 3.2:

[1]The popular expression "model of a model" is particularly confusing.

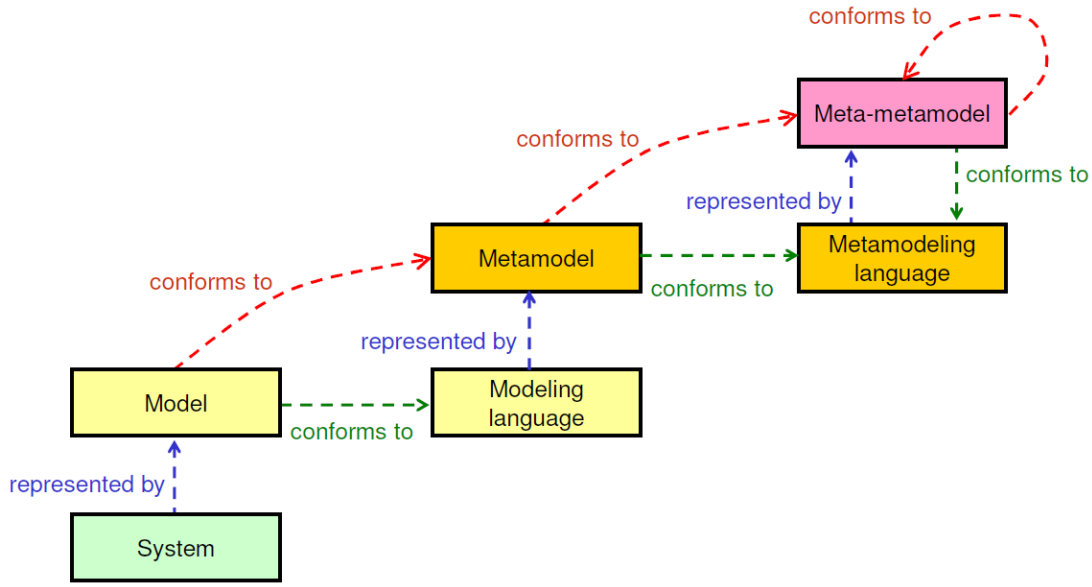


Figure 3.2: Metalayers [10]

On the bottom level, a *model* conforms to a modelling language whose abstract syntax is represented by a *metamodel*. Transitively, we can say that a *model* conforms to a *metamodel*.

On the middle level, a *metamodel* conforms to a metamodelling language whose abstract syntax is represented by a *meta-metamodel*. Transitively, we can say that a *metamodel* conforms to a *meta-metamodel*.

On the top level, A *reflexive meta-metamodel* conforms to a language whose abstract syntax is represented by *itself*. Transitively, we can say that a *meta-metamodel* conforms to *itself*.

The represented-by and conforms-to relationships are essentially different; arranging them in the same direction might be confusing.

Since metamodels are models themselves, we will use the less cumbersome term—*model*—to refer to a metamodel where it is obvious from the context that we are talking about the metamodel and not one of the models it specifies.

3.2 Aalaadin

This section introduces the *Aalaadin* metamodel[2] [12], [13], [14] and [16], proposed in 1997 by Jacques Ferber, Oliver Gutknecht and their colleagues from Montpellier 2 University in Montpellier, France. The overview presented here is distilled from the seminal paper on *Aalaadin* [12].

[2] *Aalaadin* is the old name; the metamodel is now known as *AGR* (for "Agent, Group and Role"). We will use the fancier old name.

To understand the following text, it is essential to make a distinction between an *abstract organization* and a *concrete organization*. An *abstract organization* is the organization specification that exists in a MAS at design-time, whereas a *concrete organization* is the actual organization that exists in a MAS at run-time. Put differently, an abstract organization is a (possibly infinite) set of all imaginable organizations conforming to a common specification (sharing the same role structure) and a concrete organization is a member of this set.

Later we will use the terms *organization type* and *organization token* to refer to an abstract organization and a concrete organization respectively. These terms try to capture the essence of the relationship between an abstract and concrete organization (namely, a concrete organization being an instance of an abstract organization and conversely, an abstract organization being a class of a concrete organization)

The *Aalaadin* metamodel comprises two models: *Core model* and *Methodological model*.

3.2.1 Core Model

The *Core model* contains concepts for modelling concrete organizations, the so-called *core* concepts: *Agent*, *Group* and *Role*. Figure 3.3 illustrates the *Core model*.

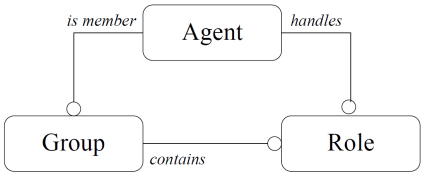


Figure 3.3: The *Core model* [12]

Agent

An *agent* is defined in [12] as an active communicating entity which plays roles within groups.

Aalaadin does not prescribe any particular agent architecture. Indeed, any MAS metamodel striving for generality should impose as few constraints upon the resulting MAS models as possible. After all, the decision of which agent architecture to employ is best made by the the MAS designer and relates to the MAS as such, not just its organizational structure. As we will see, none of the metamodels introduced in this thesis force the MAS designer to adapt a concrete definition of agenthood.

Group

In [12], a *group* is defined as atomic set of agent aggregation. In its most basic form, a group is just a way to tag a set of agents, i.e. it has no structure.

Groups have the following characteristics:

- An agent can be a member of a number of groups simultaneously. This means that groups can overlap, which is major point of *Aalaadin*.
- A new group can be founded by any agent; an agent must request its admission to an existing group.
- A group may be local or distributed across multiple machines.

The real advantage of grouping agents becomes apparent when we use roles to impose some structure to these groups.

Role

A *role* is an abstract representation of an agent function, service or identification within a group [12]. An agent can play multiple roles, each of which is local to a particular group. Similarly to group admission, playing a role in a group must be requested by the candidate agent (already a member of the group) and awarded by the group founder agent.

In *Aalaadin*, the communication is related to roles. Since an agent can play multiple roles, it can be engage in several independent dialogues simultaneously.

The following characteristics are part of a role definition:

- a *uniqueness characteristic*—an indication whether the role is *single* or *multiple*,
- a list of *competences*—conditions the candidate agent must satisfy to be eligible to play the role, and
- a list of *capacities*—abilities attributed to an agent while it is playing the role.

A *single role* can be played by at most one agent in a group, whereas a *multiple role* can be played by any number of agents within a group. By default a role is multiple, does not require any competences and does not provide any capacities.

A special role in a group is the *group manager* role, which is automatically granted to the group founder. It has a competence to handle group membership and

role playing requests. It also has a capacity to revoke roles and cancel group membership.

3.2.2 Methodological Model

The *Methodological model* contains concepts for modelling abstract organizations, the so-called *methodological* concepts: *Organization structure*, *Group structure*, *Interaction* and *Agent class*. These concepts are not present directly in concrete organizations, but only serve during the analysis and design phases. Their purpose is to describe abstract organizations from which concrete organizations, described using the core concepts, will ultimately be derived. Figure 3.4 shows the integrated *Aalaadin* metamodel. The dotted ellipsis is the demarcation line between the *Core model* and *Methodological model*.

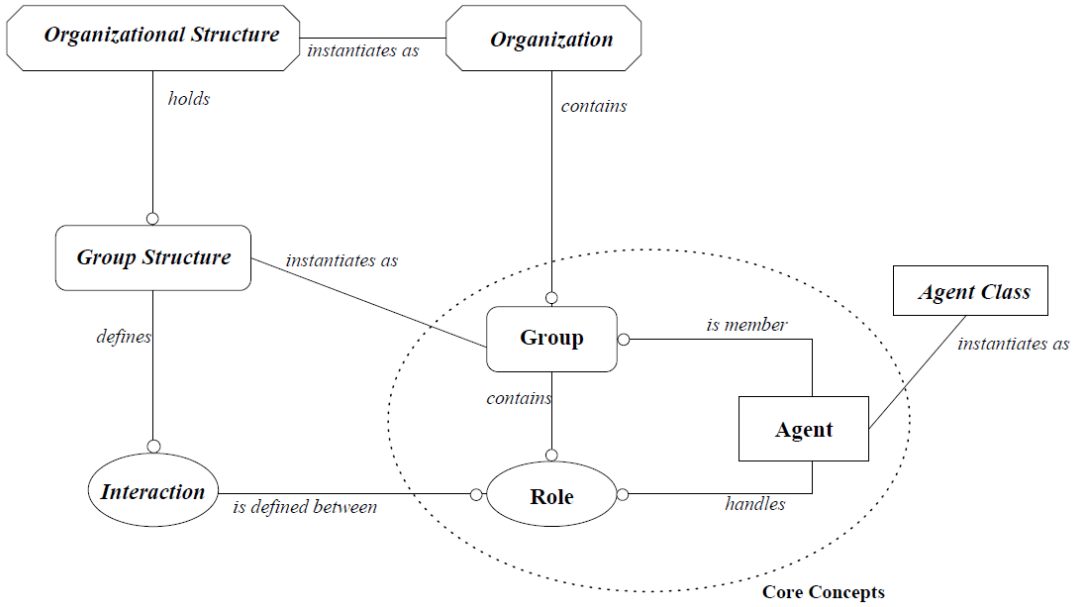


Figure 3.4: The *Aalaadin* metamodel [12]

Group Structure

A *group structure* is an abstract description of a group [12]. It identifies all roles comprising the group and defines interactions among them.

A group structure is defined by

- a set of available roles that can be played by agents in the group, and
- a set of valid interaction schemes between the roles.

Note that an actual group might be a partial instantiation of its defining group structure. This means that during the MAS run, there might be a moment when some roles defined in the group structure are not played in an actual group. This dynamic nature of groups allows for a great deal of run-time flexibility.

Organizational Structure

The *organizational structure*, as defined in [12], is a set of group structures expressing the design of a multi-agent organization scheme.

The organizational structure can be seen as the specification of the problem to be solved (organization to be modelled) using a MAS. Any sort of heterogeneity within a single system (e.g. agent architecture heterogeneity or language heterogeneity) can be managed by different group structures involved in the organizational structure.

Similarly to groups, an asctual organization can be a incomplete manifestation of its defining organizational structure. This means that while a MAS is running, there may be a point when some groups defined in the organizational structure are not present in an organization. This also contributes to the overall run-time flexibility.

3.2.3 MadKit

The authors of *Aalaadin* also developed an agent platform implementing their metamodel called *MadKit*[3] [12], [13] and [15].

The basic philosophy of the *Aalaadin/MadKit* architecture is to use the platform itself for its own management wherever possible. *MadKit's* main design principles are *micro-kernel architecture* and *agentification of services*—all services except for the most fundamental ones provided by the micro-kernel are implemented as agents, organized in groups and identified by roles [13].

3.3 O&P

This section introduces the *O&P* metamodel[4] [17], [18], [19], [20] and [21], put forward in 2001 by James J. Odell, H. van Dyke Parunak and their colleagues. The overview presented here is extracted from the most complete paper on *O&P* [21].

[3] Multi-Agent Development Kit —`http://www.madkit.org/`

[4] The metamodel was not given a name by its authors. In this thesis, we will call it *O&P*.

3.3.1 Integrated Metamodel

Figure 3.5 shows the integrated *O&P* metamodel proposed in [21]. The following subsections will focus on parts of the integrated metamodel that can be studied in isolation. We present the integrated metamodel before discussing its parts so that the reader can follow the discussion knowing how each part fits into the big picture.

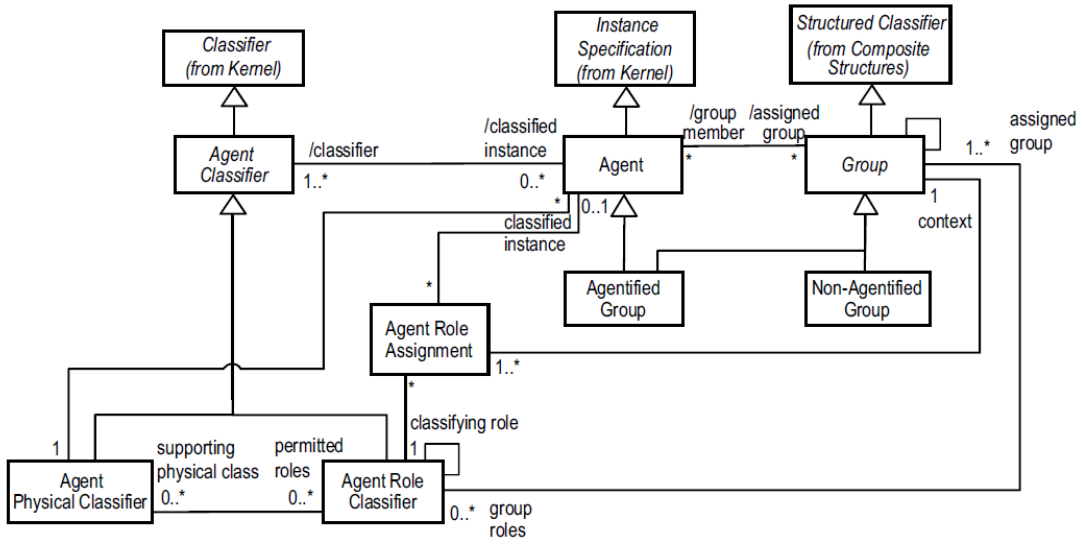


Figure 3.5: The *O&P* metamodel [21]

To understand the *O&P* metamodel, it is essential to differentiate between the *Classifier* and *Class* UML classes. In short, *Classifier* does not have the features associated with a OOP class (e.g. the extended class, the list of implemented interfaces or attributes), while *Class*, obviously, has them. *Class* is in fact a specialization of *Classifier*. It is important to make this distinction, because the agent classification is based on an extension of *Classifier*, not *Class*. The reason for this is that the authors did not want to impose object-orientation upon their metamodel. After all, it is not at all expected of an agent to exhibit behaviour intrinsic to an object, such as polymorphism.

3.3.2 Agent Classifiers and Agent Model

Agent Classifier

Agent Classifier is a UML *Classifier* that specifically provides a way to classify agent instances by a set of features that they have in common [21]. Classification is important because it enables a common definition of a set of entities that are in some sense similar, i.e. share some features and/or capabilities.

Figure 3.6 shows *Agent Classifier* and its two specializations: *Agent Physical*

Classifier and *Agent Role Classifer*.

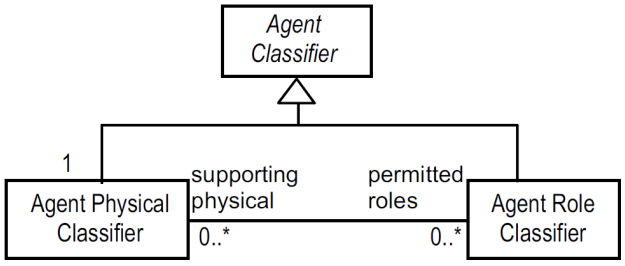


Figure 3.6: *Agent Classifier* and its two specializations: *Agent Physical Classifier* and *Agent Role Classifier* [21]

Agent Physical Classifier

The purpose of *Agent Physical Classifier* is to define a set of features that an agent classified with it has independent of roles it plays [21]. Every agent must be classified with exactly one physical classifier[5] and is never reclassified during its lifetime.

In contract to role classifiers, physical classifiers attribute primary and permanent features to agents. Examples of physical classifiers from the real world are *Human*, *Male* or *Female*.

Figure 3.7 shows some examples of physical classifiers forming a small class hierarchy. Notice the «agent physical classification» stereotype.

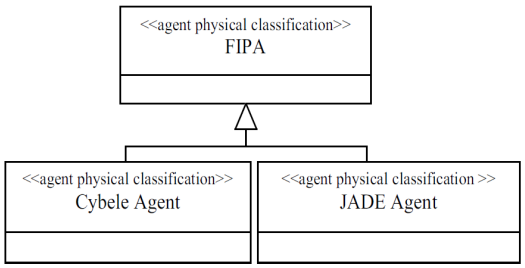


Figure 3.7: Examples of physical classifiers forming a class hierarchy [21]

Agent Role Classifier

Agent Role Classifier is a classifier that defines a set of features that an agent classified with it acquires. An agent can be classified with more than one role classifier at once (*multiple classification*) and can be reclassified over time (*dynamic classification*).

[5]Compare this with OOP, where every object must be an instance of exactly one class.

In comparison to physical classifiers, role classifiers ascribe secondary and transient features to agents. An example of a role classifier from the real world would be *Chess player*.

Figure 3.8 depicts a small class hierarchy of role classifiers. Notice the «agent role» stereotype.

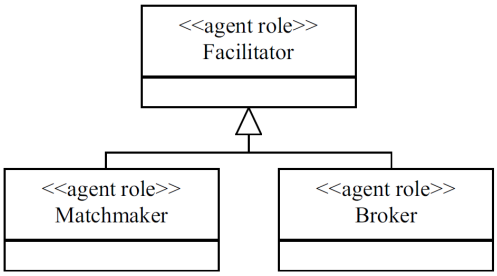


Figure 3.8: Examples of role classifiers forming a class hierarchy [21]

Agent

In *O&P*, the basic concepts are *Agent Classifier* and *Agent*. These modelling constructs are considered fundamental, because they enable a MAS designer to model agents classes and agent instances respectively. Agent classes are the design-time constructs providing the classification of the run-time constructs—agent instances.

Association between Agent Physical Classifier and Agent Role Classifier

The association between *Agent Physical Classifier* and *Agent Role Classifier* specifies which role classifiers are permitted for each physical classifier, independent of the capabilities of the individual agents classified with that particular physical classifier [21].

Figure 3.9 illustrates this association. It can be interpreted as follows. *Jade* agents can play the `Broker` and `Manager` roles, and *Cybele* agents can take on the role of `Broker`, `Trust Manager` and `Buyer`.

Association between Agent and Agent Classifier

The association between *Agent* and *Agent Classifier* defines agents' features. Each agent classifier classifies an agent as a member of a set of agents sharing some physical or role-related features.

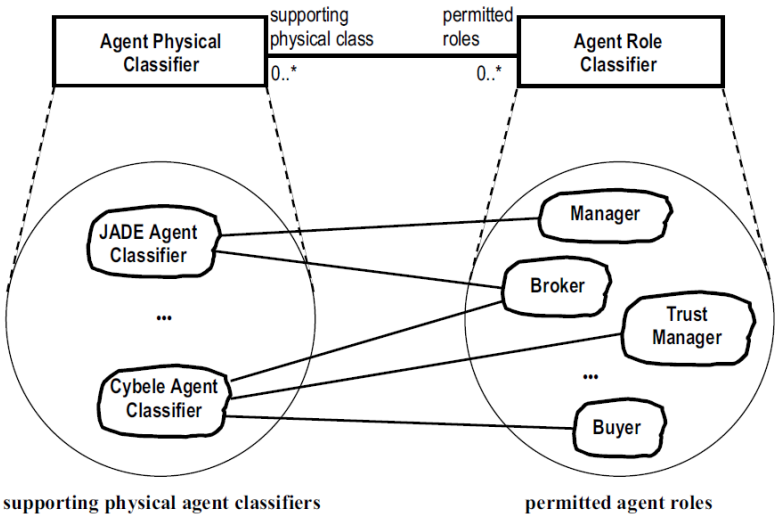


Figure 3.9: The association between *Agent Physical Classifier* and *Agent Role Classifier* [21]

There are two main differences between the physical and role classification. First, the role classification is *multiple* whereas the physical classification is *single*. While an agent can be classified with more than one (or even none) role classifiers at the same time, it must be classified with exactly one physical classifiers. Second, the role classification is *dynamic* in contrast to physical classification, which is *static*. Dynamic classification means that an agent can be declassified or reclassified with another role after the initial classification; static classification is invariant in time.

Figure 3.10 illuminates this association. It can be read as follows. `Agent1`, a *Jade* agent, is a `Manager`; `Agent2`, a *Cybele* agent, is a `Manager` and `Buyer`; `Agent3`, another *Cybele* agent, is a `Trust Manager`; and `Agent4`, also a *Cybele* agent, is a `Broker`.

3.3.3 Group Model

Group

A *group* is a set of agents that are related via their roles, where these links must form a connected graph within the group [21]. This is the agent-centric way of lookig at a group. Another way of to look at it is the role-centric way: a group is a composite structure consisting of interrelated roles, where each of the group's roles has a number of agent instances [21] playing that role. A group can be formed to exploit the synergy of its members, resulting in an entity capable of performing operations that none of its constituents alone is capable of performing on its own.

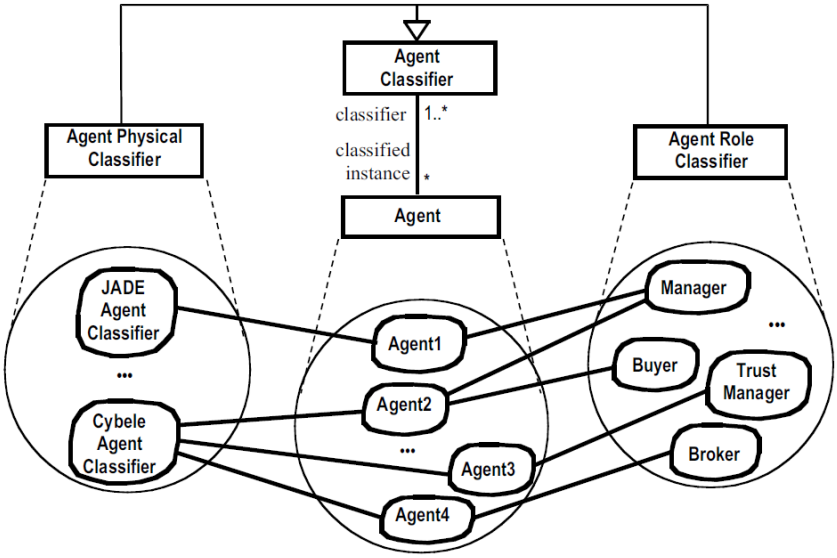


Figure 3.10: The association between *Agent* and *Agent Classifiers* [21]

Figure 3.11 shows the *Group* class and its associations with *Agent* and *Role*. The abstract *Group* class extends the UML *Structured Classifier*, which means that *Group* is defined as composite structure[6].

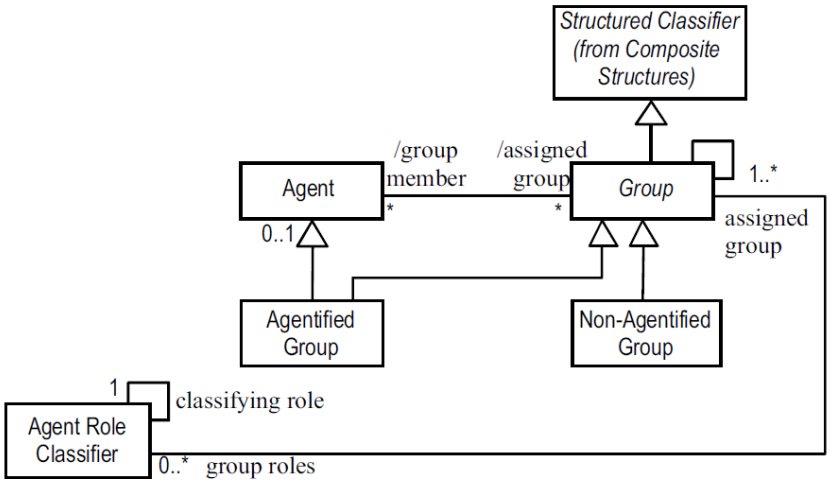


Figure 3.11: The *Group* class and its associations [21]

Association between Group and Agent

Conceptually, a group is constituted by a set of agents playing roles within that group. The roles that the agents can play are represented by one or more agent role classifiers associated with this group. Therefore, the set of agents forming a group can be derived from the group via the agent role classifiers [21].

[6]In UML, *Structured Classifier* can be thought of as a structured set of classifiers. From this perspective, *Group* is a structured set of *Agent Role Classifiers*.

Association between Group and Role

Figure 3.12 illustrates the association between *Group* and *Role*. Note that groups containing no roles are not allowed; each group must contain at least one role. Also observe that each role has to be defined in at least one group, since roles only make sense within the context of a group.

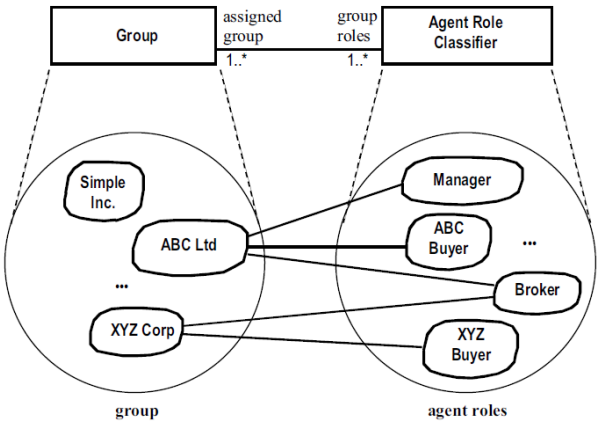


Figure 3.12: The association between *Group* and *Role* [21]

Agentified and Non-Agentified Groups

O&P differentiates between two types of groups: *agentified* and *non-agentified*.

An *agentified group* is a group that is also an agent in its own right, which means it has its own capability to interact [21]. An agentified group can communicate with other agents (or agentified groups) directly, i.e. without a representative agent. It can also be a member of other groups (agentified or not) and play roles like any other agent. To achieve this in *O&P*, *Agentified Group* is a subclass of both the *Group* and *Agent* classes. Figure 3.13 shows an example of an agentified group. Notice the «agent» stereotype used to mark the group as agentified.

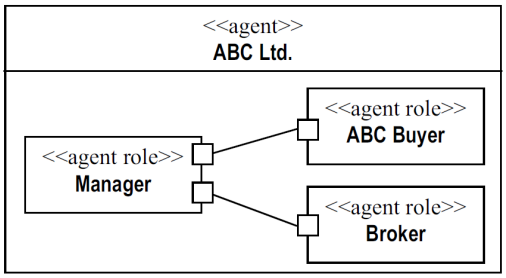


Figure 3.13: An example of an agentified group [21]

A *non-agentified group*, while still being a first-class citizen, is not an agent in and of itself, meaning it has no capability to interact of its own. A non-agentified group

always communicates with other agents (including agentified groups) through one of its members acting as an intermediary. This is achieved in *O&P* by *Non-Agentified Group* subclassing only the *Group* class and not the *Agent* class. An example of a non-agentified group is shown in figure 3.14. Notice the absence of the «agent» stereotype.

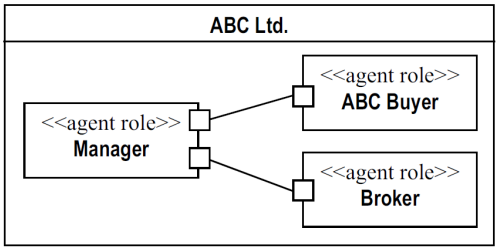


Figure 3.14: An example of a non-agentified group [21]

3.3.4 Agent Role Assignment

The assignment of roles to agents is dynamic, i.e. it changes in time, and is modelled by *Agent Role Assignment*. Figure 3.15 shows the *Agent Role Assignment* class and its associations.

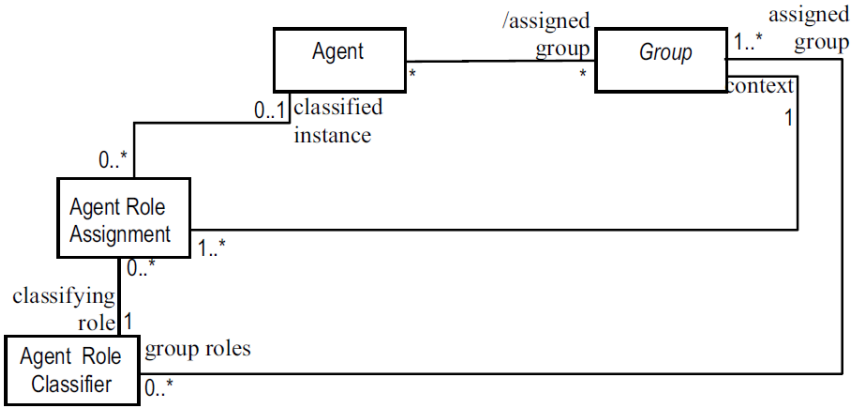


Figure 3.15: The *Agent Role Assignment* class and its associations [21]

Agent Role Assignment as a Ternary Association

A model with a direct association between *Agent* and *Agent Role Classifier* could represent agents playing roles. However, such model would not be able to represent a situation where an agent plays a role in one group but does not play it in another. To model this kind of situation, it is necessary to augment the agent-to-role association with a group context. This yields a ternary association, reified[7]

[7] *Reification* is the process of turning an implicit abstract idea about some concept into an explicit concrete model of that concept.

in *O&P* as the *Agent Role Assignment* class, whose instances link an agent to a role in a group.

Position

It is possible to associate a group with a role leaving an agent unspecified. Such association is called a *position* and represents a situation where a concrete agent playing a role within a particular group is yet to be determined. This turns out to be an extremely useful modelling concept, since more often than not, the organization modeller does not know (or simply does not care) which agent will actually take a particular position when the MAS is run. Unfortunately, this association is not reified in *O&P*. If it was reified as the *Position* class, the *Agent Role Assignment* class could be viewed as a reified association between the *Position* and *Agent* classes.

3.4 PIM4Agents

This section introduces the *PIM4Agents* metamodel[8] [22], [23] and [24], proposed in 2007 by Christian Hahn, Cristián Madrigal-Mora and Klaus Fischer from German Research Centre for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI). We will present only a brief overview (distilled from [23]) since our work does not draw much inspiration from *PIM4Agents*.

PIM4Agents has been specifically designed to be employed in the Model-driven engineering (MDE) software development methodology, more precisely in Model-driven architecture (MDA) by Object Management Group (OMG). Apart from the platform-independent metamodel itself, the authors have proposed two platform-specific metamodels: *JackMM* and *JadeMM* for the *JACK* and *Jade* agent platforms respectively. They have also described two sets of model transformations to convert PIMs to PSMs: *PIM4Agents-to-JackMM* and *PIM4Agents-to-JadeMM*.

Core Model

To support adaptability, *PIM4Agents* is structured around a small core that can be augmented with extensions to model specific aspects of MASs, for example, security. Figure 3.16 shows the core model.

The metamodel, like previously introduced metamodels, is built around the concept of *agent*, an autonomous entity capable of sensing its environment and acting upon it. Each agent has access to a set of *resources* from its surrounding *environment* [23].

[8] For "Platform-Independent Model for Agents".

A *behaviour* can be atomic or composed of sub-behaviours. This way, a whole hierarchy of specific behaviours can be created. A behaviour may also send or receive *messages* according to a *protocol*. A *capability* allows to group conceptually related behaviours [23].

A *role* is an abstraction of the social behaviour of an agent in a given social context, usually a *cooperation*; it specifies the responsibilities of an agent in that social context. A *cooperation* represents the interaction between agents playing the required set of roles. The detailed realisation of this interaction is described by a *protocol* that specifies the *messages* exchanged between the involved roles and at which point in time they are to be expected. A protocol is executed by a set of behaviours sending and receiving messages in accordance to their roles.

Agents can take part in an *organization*, a special kind of cooperation that also has the same characteristics as an agent. Being a cooperation, an organization can have its own internal protocol that specifies how it coordinates its members. Being also an agent, an organization can play roles in other organizations (*super-organization*) and has capabilities which can be performed by its members, be they agents or other organizations (*sub-organizations*).

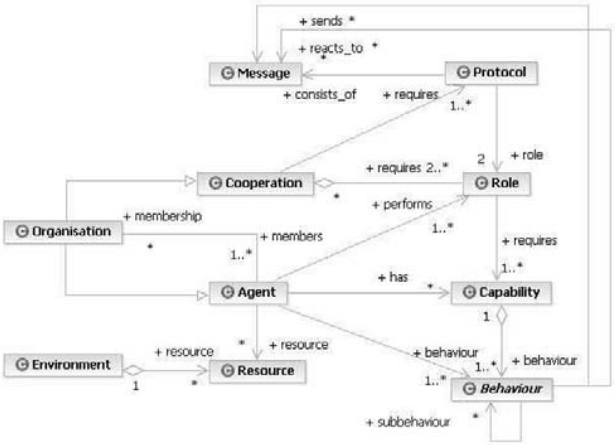


Figure 3.16: The *PIM4Agents* core model [23]

3.4.1 JadeOrgs

JadeOrgs, [25] and [26], is an extension of the *Jade* framework that implements the *JadeMM* platform-specific metamodel.

JadeMM is defined using Eclipse Modeling Framework (EMF) to exploit EMF's code generation facility. Given a platform-independent model of a MAS (conforming to *PIM4Agents*), the corresponding *Jade/JadeOrgs* platform-specific model (conforming to *JadeMM*) can be derived automatically.

3.5 powerJade

This section introduces the *powerJade* metamodel[9] [35], [36], [37] and [38], put forward in 2008 by Matteo Baldoni, Guido Boella and their colleagues from University of Turin in Turin, Italy. The overview presented here is extracted from the most complete paper on *powerJade* [38].

powerJade is inspired by the authors' previous work on *powerJava*—an extension of the Java programming language with an explicit role construct, based on an ontological analysis of roles described in [27], [28], [29] and [30].

An organization belongs to the social reality and it can only be interacted with via the roles it defines [33]. Specifically, it is not an object that could be manipulated from the outside like objects in OOP. The concept of organization is useful not only when modelling problem domains including organizations per se. Indeed, we can view every object as an organization offering different ways of interacting with it, each represented by a different role.

powerJade is a unification of two orthogonal models:

- the *organizational structure model*, that models the static aspects of organizations, and
- the *role dynamics model*, that models their dynamic aspects.

Organizational Structure Model

The model in [31] is focused on organizational structure. An ontological analysis of roles in yields the following properties of roles [31]:

- A role instance is always associated with an instance of the organization class to which it belongs and with a player instance.
- The definition of a role depends on the organization it belongs to.
- Role operations (called *powers*) have access to the state of the organization and other roles in the organization.
- To be granted a role, the player must be able to perform operations (called *requirements*) which can be requested while it plays the role.

The ontological status of organizations and roles does not differ completely from that of agents or even objects [31]. On one hand, organizations and roles, unlike agents, are not autonomous and act via their members and players. Additionally,

[9] The name is intentionally uncapitalized.

roles, unlike objects, do not exist as independent entities, since they are necessarily linked to organizations. On the other hand, organizations and roles, like agents, are descriptions of complex behaviour. In the real world, organizations are considered legal entities; they can even act like agents, albeit via a representative role. Since they share some properties with agents, they can be modelled using similar primitives.

Role Dynamics Model

The model in [32] is focused on role dynamics. Four operations pertaining to the role dynamics are defined [32]:

- When an agent *enacts* a role, it acquires the role and the role is said to be *enacted*.
- When an agent *deacts* a role, it relinquishes the role.
- When an agent *activates* a role, it starts playing the role, and the role is said to be *active*.
- When an agent *deactivates* a role, it stops playing the role, and the role is said to be *inactive*.

Even though it is possible (and very common) for an agent to be enacting multiple roles at the same time, at any given moment, only one of these can be active. Naturally, it is possible that at some moment none is active. In particular, when an agent is invoking a power, the role whose power it invokes must be active.

Unified Metamodel

The unified *powerJade* metamodel was created by merging the organizational structure model found in [31] and the role dynamics model described in [32]. Organizations and roles are not just design-time abstractions and players are not just isolated agents; they are all agents interacting with one another. A logical specification of this unified metamodel can be found in [34].

Powers and Requirements

On a final note, roles in *powerJade* can be compared to interfaces from OOP. Just like an interface is a contract between a calling class and called class, a role is a contract between an organization and a player.

In OOP, the relationship between a class and the interfaces it implements is a rigid one—the interfaces a class implements are part of its design-time definition and cannot be implemented or un-implemented at run-time. In contrast, the relationship between a player and roles it enacts in *powerJade* is a flexible one—the roles the a player enacts are not part of its design-time definition and can be enacted and deacted at run-time.

Interfaces in OOP declare methods and events. When implementing an interface, a class has to implement its methods and it can raise its events. Similarly, roles in *powerJade* define *requirements* and *powers*. When eancting a role, a player has to execute its requirements and it can invoke its powers. Thus, role requirements correspond to interface methods (both are responsibilities of player/called class), while role powers are analogous to interface events (both are competences of the player/called class).

Chapter 4

Platform-Independent Metamodel—Thespian

In this chapter, we will present our platform-independent metamodel for modelling organizations in MASs—*Thespian*. *Thespian* is one of our two contributions to the field, the other one being *Thespian4Jade*.

Thespian is a metamodel to which platform-independent models of organizations in MASs must conform; more precisely, it is a model[1] representing a modelling language [2] to which platform-independent models of organizations in MASs must conform.

The metamodel is named after *Thespis of Icaria* (present-day Dionysos, Greece), who lived in the 6th century BC and, according to certain Ancient Greece sources and especially Aristotle, was the first person ever to appear on stage as an actor playing a character in a play (instead of speaking as themselves) [53].

The design of *Thespian* is inspired, to a greater or lesser degree, by all four metamodels introduced in the previous chapter. Similarly to *Aalaadin*, *Thespian's* static model can be partitioned into a *Run-time model* and *Design-time model* (called *Core model* and *Methodological model* respectively in *Aalaadin*.) Like *O&P*, *Thespian* can be used to model *holonic* MASs[3]. Similarly to *PIM4Agents*, *Thespian's* static model contains concepts for modelling protocols and messages exchanged in these protocols. Overall, *PIM4Agents* has had the least influence on *Thespian* from all four metamodels. And like *powerJade*, *Thespian* can be used to represent role *competences* and *responsibilities* (called *powers* and *requirements* respectively in *powerJade*). Furthermore, *Thespian's* dynamic model also makes a distinction between *enacting* a role and actually *activating* it. All in all, *pow-*

[1] Recall that a metamodel is a kind of model.

[2] When can call this language the *Thespian modelling language*.

[3] A *holonic* MAS is a MAS where agents are *holons*—simultaneously a *whole* and a *part*. This means that the agents in a holonic MAS can not only form aggregations, but these aggregations are full-fledged agents; in particular, they can form other aggregations and so on. An *object* from OOP is another example of a holon.

erJade has had the greatest influence on *Thespian* from all four metamodels.

The metamodel encompasses

- a *static model* for modelling static (structural) aspects of organizations, and
- a *dynamic model* for modelling their dynamic (behavioural) aspect.

In the following two sections, we will introduce both models in detail.

4.1 Static Model

The *static model* is used to model static (structural) aspects of organizations: such as:

- an organization's role structure and protocols,
- a role's competences and responsibilities, or
- a player's capabilities.

The metamodel model can be partitioned in two orthogonal ways: *functional* and *technical*. First, both partitions will be described, and then the integrated static model will be presented.

The *functional partition* divides the concepts according to the area they represent: organization, player or protocol. This is a more natural partition of the two, since fewer dependencies among concepts from different parts exists than in the other partition. The organization/protocol part of the MAS can be designed (and implemented) independently of the player part; indeed agent developed by one team can play roles in organizations developed by another team.

The *technical partition* separates the concepts based on whether they represent design-time or run-time entities. The design-time entities are created already at design-time and are usually implemented as *agent classes* in the target agent platform; they are analogous to object classes in OOP. The run-time entities are created only at run-time and are usually implemented as *agent instances* in the target agent platform; they are analogous to object instances in OOP.

Before continuing with the presentation of *Thespian*, it is important to explain the *type-token distinction*. In disciplines such as philosophy and knowledge representation, the type-token distinction is a distinction that separates a *concept* from objects that are particular *instances* of that concept [54]. *Thespian* has been designed to support the type-token distinction and the correct differentiation between types and their tokens is a recurring theme in this chapter.

As an example of the type-token distinction, consider a MAS. The *specification* of the MAS (its source code) is a *MAS type* and its *manifestation* (a running MAS) is a *MAS token*. Just like a type can (and usually does) have many tokens, a MAS specification can have multiple manifestations—the same source code can be run many times, each time yielding a different run.

4.1.1 Organization Model

The *Organization model* (figure 4.1) contains concepts whose instances model organizations and roles with their competences and responsibilities.

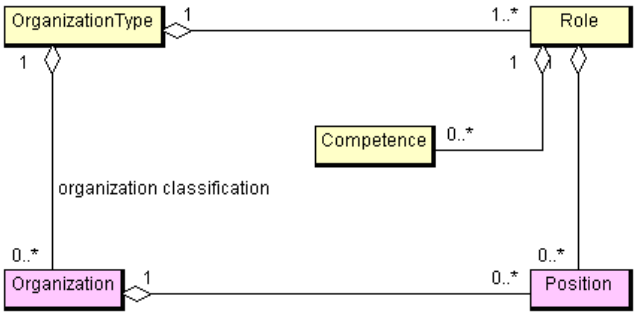


Figure 4.1: The Organization model

Organization and Organization Type

To enable the type-token distinction for organizations, *Thespian* contains concepts for modelling both an *organization type* and an *organization token*: *Organization type* and *Organization*.

Organization type (also called *Organization class*) is a class of organizations sharing the same role structure; it is a design-time entity. It contains a set of *roles* defining its role structure. It can be instantiated to yield an *organization*.

Organization (also called *Organization instance*) is an actual organization in a running MAS; it is a run-time entity. It is classified by an *organization type* which specifies its role structure. Note that despite begin a run-time entity, an organization has to be declared at design-time, because in thesis we do not consider creating and destroying organization during at run-time. When declared at design-time, the organization is created at MAS start-up.

Role and Position

Thespian contains concepts for modelling both an *role type* and *role token* to enable the type-token distinction for roles: *Role* and *Position*.

Role is a role specification within an organization; it is a design-time entity. It has a set of *competences* and a set of *responsibilities* defining its function in its containing organization type. Furthermore, it has a *multiplicity* differentiating between a *single role*—one that can be played by at most one player in one organization—and a *multiple role*—one that can be played by more than one player in one organization. Currently, neither *Thespian* not *Thespian4Jade* support differentiating between a *mandatory role*—one that has to be played at all times—and an *optional role*—one that does not have to be played at all times.

A *Position* (sometimes called *Role instance*) is a role manifestation within an actual organization; it is a run-time entity. It is a realization of a *role* which specifies its competences. It belongs to an organization and is played by a player. Note that a position is usually not declared at design-time; it is created when an player starts playing a role in an organization and destroyed when that player stops doing so.

Competence

Let us consider a player playing a role. As a result of playing the role, the player gains competences a responsibilities associated with that role.

Competence is an operation a player playing a role *can* invoke as a result of playing that role; it is a design-time entity. A *Competence* can require an argument from a player after its invocation (but before its execution), in which case the argument type (a Java type) has to be specified. Also it can provide a return value to the player after its execution, in which case the return value type (a Java type) has to be specified.

4.1.2 Player Model

The *Player model* (figure 4.2) contains constructs for modelling players and their responsibilities.

Player and Player Type

To facilitate the type-token distinction for players, *Thespian* contains concepts for modelling both a *player type* and a *player token*: *Player type* and *Player*.

Player type (also called *Player class*) is a class of players sharing the same capabilities; it is a design-time entity. It has a set of *responsibilities* defining its capabilities when playing a role. It can be instantiated to yield a *player*.

Player is an actual player in a running MAS; it is a run-time entity. It is classified

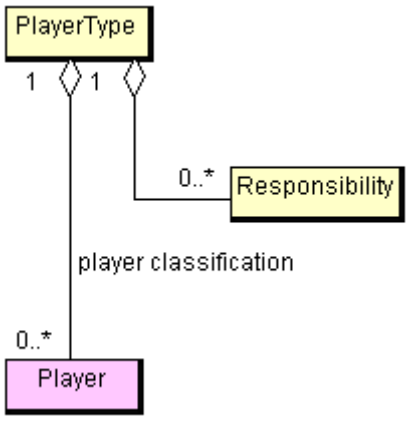


Figure 4.2: The Player model

by a *player type* which specifies its . Note that despite begin a run-time entity, a player has to be declared at design-time, because in this thesis, we do not consider creating and destroying players at run-time. When declared at design-time, the player is created at MAS start-up.

Responsibility

Responsibility is an operation a player playing a role *must* execute as a result of playing that role; it is a design-time entity. A *Responsibility* can provide an argument to a player after its invocation (but before its execution), in which case the argument type (a Java type) has to be specified. Also it can request a return value from the player after its execution, in which case the return value type (a Java type) has to be specified.

4.1.3 Protocol Model

The *Protocol model* (figure 4.3) contains abstractions whose instances represent interaction protocols between a player and an organization or role, and among roles themselves.

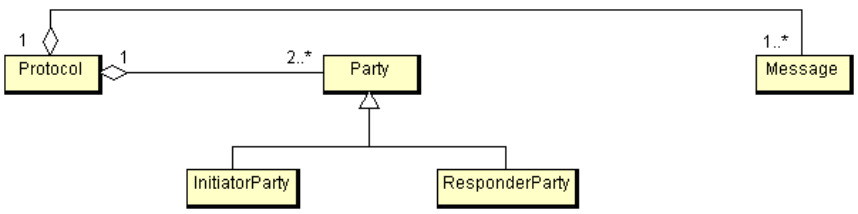


Figure 4.3: The Protocol model

Protocol

An *Interaction protocol* (or simply *Protocol*) is an institutionalized pattern of interaction (communication) between two or more roles within an organization; it is a design-time entity. It defines parties involved in the interaction and messages exchanged in the communication.

A realization of an interaction protocol is an *interaction scenario*—a sequence of actions performed (messages exchanged) by two or more positions within an organization; a scenario is a purely run-time entity. In other words, a protocol is a framework and a scenario (is) one of its possible instantiations. Since scenarios are usually not explicitly modelled[4], *Thespian* does not contain the concept of an *Interaction scenario*.

The theme of type-token distinction is at play here: instances of *Protocol* model *protocol types* and instances of *Scenario* represent *protocol tokens*.

Party

A *Party* is a role involved in a protocol; it is a design-time entity. A relationship between roles and protocols is a many-to-many one—a role can participate in multiple protocols and at least two different roles have to take part in a protocol. A party is a reification (embodiment) of this relationship. A *Party* is either an *Initiator party*—one that initiates the protocol—or a *Responder party*—one that responds to the initiated protocol.

Message

A *Message* is a piece of information exchanged between two parties in a protocol; it is a design-time entity.

4.1.4 Design-Time Model

The *Design-time model* (figure 4.4) contains concepts whose instances model the design-time MAS entities. These entities, as their name suggests, are created and/or modified at design-time by the MAS designer, and they constitute the *MAS specification*.

[4]Scenarios can be explicitly modelled in snapshots.

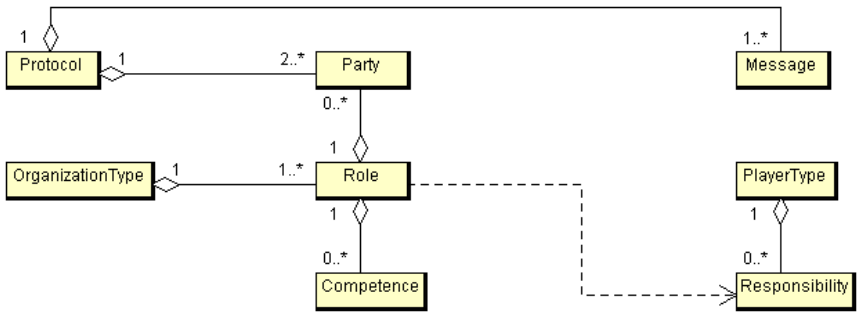


Figure 4.4: The Design-time model

4.1.5 Run-Time Model

The *Run-time model* (figure 4.5) contains constructs that model the run-time MAS entities. These entities, as their name implies, are created and/or modified at run-time by the MAS itself, and they make up the *MAS manifestation*. They models of MAS manifestations are also referred to as *snapshots*.

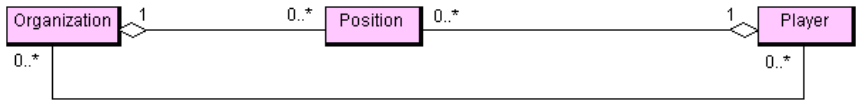


Figure 4.5: The Run-time model

4.1.6 Integrated Static Model

Figure 4.6 shows the integrated *Thespian* static model.

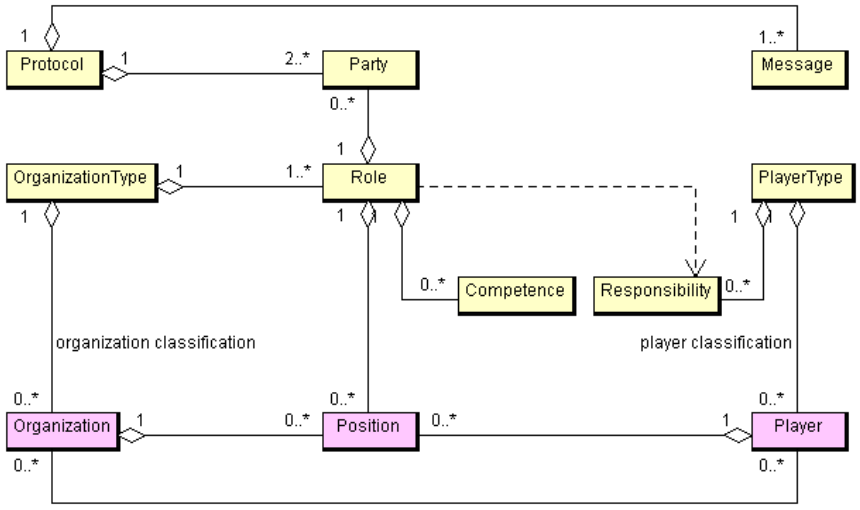


Figure 4.6: The *Thespian* static model

We would like to emphasize what we perceive as key property of *Thespian*: there is no association between *Role* and *Player type*, no link between a specific *role* and

a particular *player type*. This means that there is no *design-time* dependency between roles and player types; all connections arise at *run-time* and happen between *positions* and *players*.

4.2 Dynamic Model

The *dynamic model* is used to model dynamic (behavioural) aspects of organizations, such as:

- players playing roles within organizations,
- players exercising their roles' competences and fulfilling their roles' responsibilities, or
- players subscribing to organization events, or
- organizations publishing events.

4.2.1 Player and Organization Interaction

A player and an organization interact using four protocols: *Enact role*, *Deact role*, *Subscribe to event* and *Publish event*.

Enacting a Role

To *enact a role in an organization* means to assume a role in an organization. Naturally, only a non-enacted single role or a multiple role can be enacted. More precisely, a single role can only be enacted in a concrete organization in which it is not already enacted by any player, and a multiple role can be enacted even if it is already enacted. A player who wants to enact a role in an organization, initiates the *Enact role* protocol with that organization.

The *Enact role* protocol consists of the following steps:

1. The player sends an *Enact role request* message to the organization, containing the name of the role it wants to enact.
2. The organization receives the request message and sends back either
   - a *Required Responsibilities* message listing the role's responsibilities, if such role exists and can be enacted, or
   - a *Failure* message otherwise.

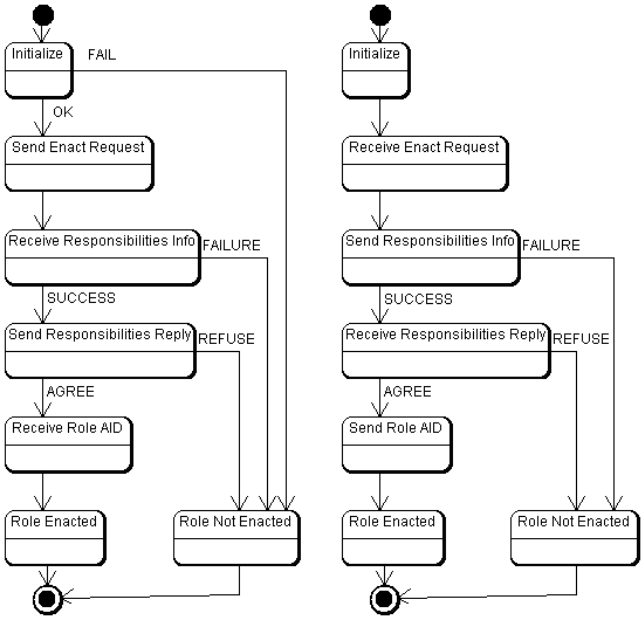


Figure 4.7: The *Enact role* protocol

3. Upon receiving a *Required Responsibilities* message, the player determines if it has the required capabilities to fulfil the role's responsibilities and replies either with

   - an *Agree* message, if it has the capabilities, or
   - a *Refuse* message otherwise.

4. Upon receiving an *Agree* message, the organization creates a position and sends a *Role AID* message, containing the position's AID to the player. The organization then ends its part in the protocol.

5. The player receives the *Role AID* message and ends its part in the protocol.

Deacting a Role

To *deact*[5] *a role in an organization* means to leave a role in an organization. Naturally, only an enacted and inactive (see section 4.2.2 for details) role can be deacted. More precisely, a role can only be deacted in a concrete organization in which it is enacted by the player and inactive. A player who wants to deact a role in an organization, initiates the *Deact role* protocol with that organization.

The following steps comprise the *Deact role* protocol:

1. The player sends a *Deact role request* message to the organization, containing the name of the role it wants to deact.

[5]The word *deact* does not exist in English, it is a made-up word.

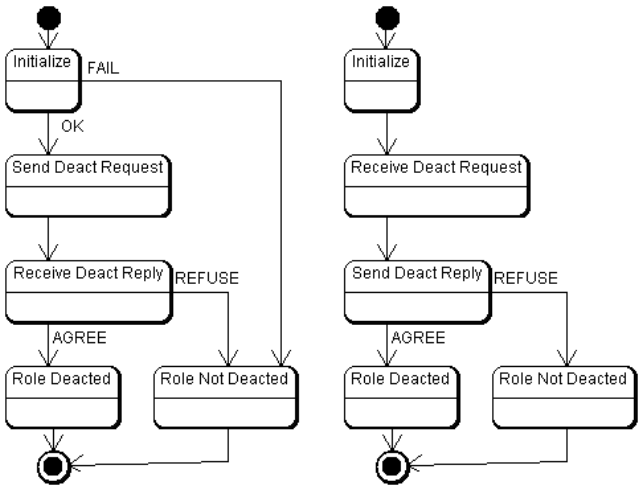


Figure 4.8: The *Deact role* protocol

2. The organization receives the request message and sends back either

   - an *Agree* message, if the role exists and can be deacted, or
   - a *Refuse* message otherwise.

   The organization then ends its part in the protocol.

3. The player receives the reply message and ends its part in the protocol.

Subscribing to an Event

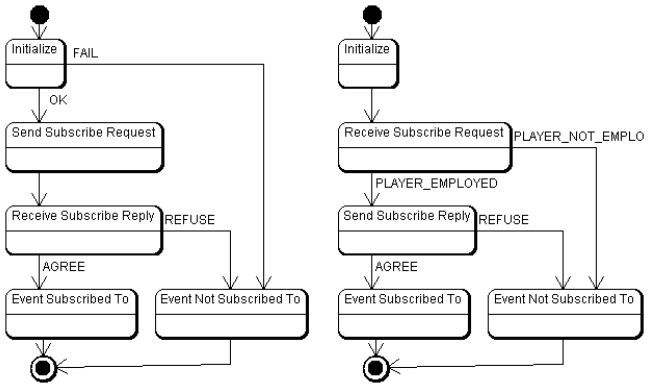


Figure 4.9: The *Subscribe to event* protocol

To *subscribe to an event in an organization* means to register for notification of an event occurring in an organization. A player can subscribe to the *role enacted*, *role deacted*, *role activated* and *role deactivated* events. Naturally, only a member player can subscribe to an event. More precisely, an event can only be subscribed to in a concrete organization of which the player is a member, i.e. enacts a role in it. A player who wants subscribe to an event in an organization, initiates the *Subscribe to event* protocol with that organization.

The *Subscribe to event* protocol consists of the following steps:

1. The player sends a *Subscribe to event request* message to the organization, containing the name of event it wants to subscribe to.

2. The organization receives the request message and immediately terminates if the message comes from a player that is not a member of that organization. If the message comes from a member player, it sends back either

   - an *Agree* message, if the event exists, or
   - a *Refuse* message otherwise.

   The organization then ends its part in the protocol.

3. The player receives the reply message and ends its part in the protocol.

Publishing an Event

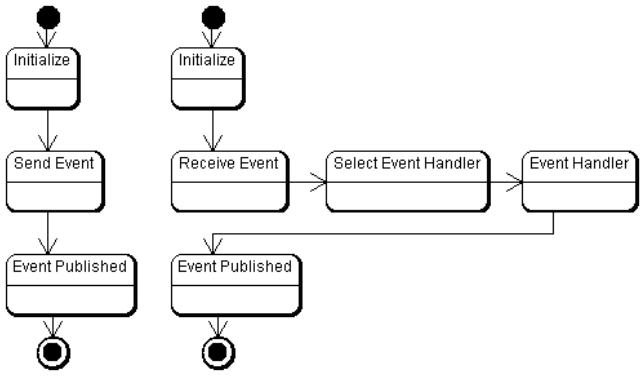


Figure 4.10: The *Publish event* protocol

To *publish an event to subscribed players* means to notify the players subscribed to an event (event subscribers) of that event occurring. An organization that wants publish an event to subscribers, initiates the *Publish event* protocol with the event subscribers.

The following steps comprise the *Publish event* protocol:

1. The organization sends an *Event* message to the event subscribers, containing the name of the event and its argument. The organization then ends its part in the protocol.

2. A player receives the message and handles the event. The player then ends its part in the protocol.

4.2.2 Player and Role Interaction

A player and a role interact using four protocols: *Activate role*, *Deactivate role*, *Invoke competence* and *Invoke responsibility*.

In concrete terms, a player does not really interact with a *role* because a role is an abstract entity in a running MAS. It interacts with a *position*—a concrete realization of that role in a specific organization. However, in the following text, we refer to a *role* when we mean a *position* because in an abstract way of speaking, a player *does* interact with a role.

Activating a Role

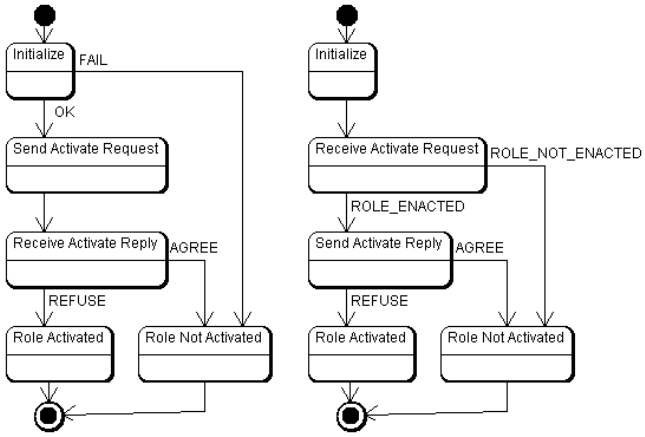


Figure 4.11: The *Activate role* protocol

To *activate a role* means to begin playing a role, i.e. start exercising its competences and fulfilling its responsibilities. Naturally, only an enacted and inactive role can be activated. More precisely, a role can only be activated in a concrete organization in which it is already enacted by the player and not yet activated. Furthermore, a player may activate only one role at a time. A player who wants to activate a role, initiates the *Activate role* protocol with that role.

The *Activate role* protocol consists of the following steps:

1. The player sends a *Activate role* request message to the role.
2. The role receives the request message and immediately terminates if the message comes from anyone else than its enacting player. If the message comes from its enacting player, it sends back either
   - an *Agree* message, if the role can be activated, or
   - a *Refuse* message otherwise.

   The role then ends its part in the protocol.

3. The player receives the reply message and ends its part in the protocol.

Deactivating a Role

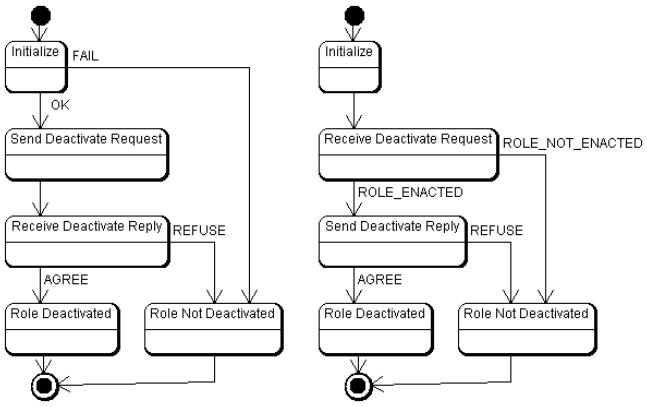


Figure 4.12: The *Deactivate role* protocol

To *deactivate a role* means to stop playing a role, i.e. stop exercising its competences and fulfilling its responsibilities. Naturally, only an enacted and active role can be deactivated. More precisely, a role can only be deactivated in a concrete organization in which it is enacted by the player and already activated. A player who wants to deactivate a role, initiates the *Deactivate role* protocol with that role.

The following steps comprise the *Deactivate role* protocol:

1. The player sends a *Deactivate role* request message to the role.
2. The role receives the request message and immediately terminates if the message comes from anyone else than its enacting player. If the message comes its enacting player, it sends back either
   - an *Agree* message, if the role can be deactivated, or
   - a *Refuse* message otherwise.

   The role then ends its part in the protocol.
3. The player receives the reply message and ends its part in the protocol.

Invoking a Competence

Invoking a competence on a role happens when a player calls upon its active role to exercise a competence. Note that a competence can only be invoked on an active role. A player who wants to invoke a competence on its role, initiates the *Invoke competence* protocol with its role.

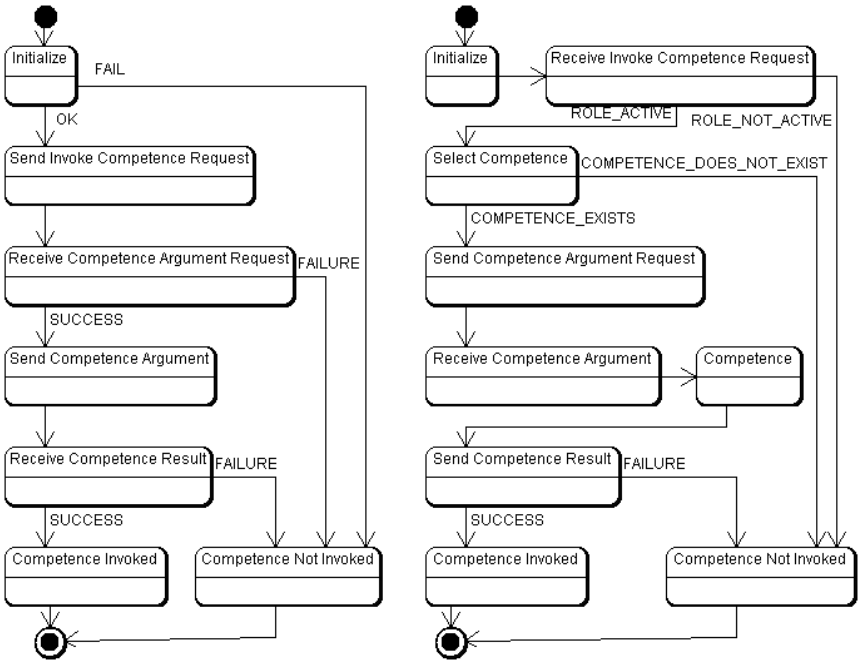


Figure 4.13: The *Invoke competence* protocol

The *Invoke competence* protocol consists of the following steps:

1. The player sends an *Invoke competence request* request to the role, containing the name of the competence to invoke.

2. The role receives the request message and immediately terminates if the message comes from anyone else than its enacting player or if it is not active. If, on the other hand, the message comes from its enacting player and it is active, it sends back either

   - a *Competence argument request* message asking the player to provide the competence argument, if the competence exists, or
   - a *Failure* message otherwise.

3. Upon receiving the *Competence argument request* message, the player sends the *Competence argument* message to the role, carrying the competence argument.

4. The role receives the competence argument, executes the competence and sends back either

   - a *Competence result* message carrying the competence result, in case the competence executed successfully, or
   - a *Failure* message otherwise.

   The role then ends its part in the protocol.

5. The player receives the reply message and ends its part in the protocol.

Invoking a Responsibility

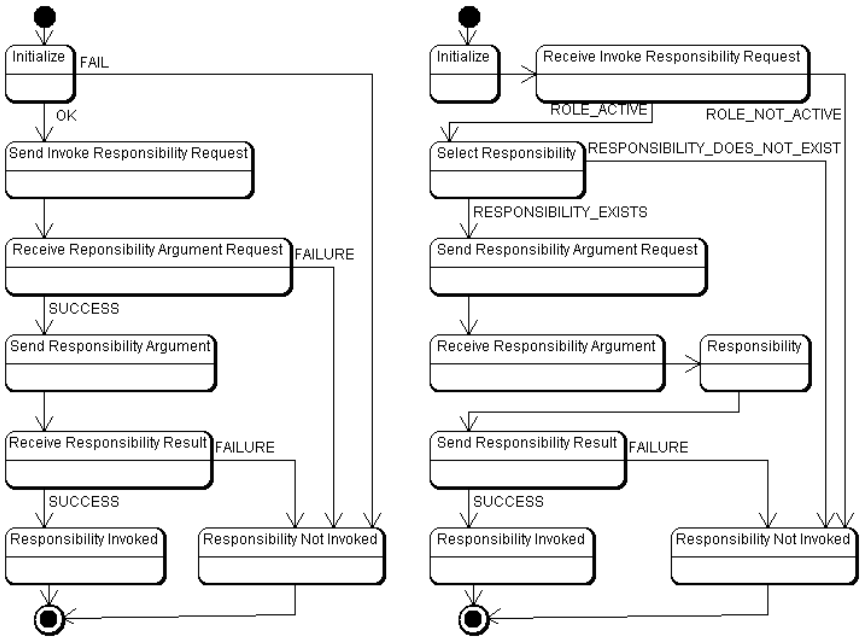


Figure 4.14: The *Invoke responsibility* protocol

Invoking a responsibility on a player happens when a role calls upon its player to fulfil a responsibility. Note that a responsibility can only be invoked by an active role. A role who wants to invoke a responsibility on its player, initiates the *Invoke responsibility* protocol with its player.

The following steps comprise the *Invoke responsibility* protocol:

1. The role sends an *Invoke responsibility request* request to the role, containing the name of the responsibility to invoke.
2. The player receives the request message and immediately terminates if the message comes from anyone else than its active role. If, on the other hand, the message comes from its active role, it sends back either
   - a *Responsibility argument request* message asking the role to provide the responsibility argument, if the responsibility exists, or
   - a *Failure* message otherwise.
3. Upon receiving the *Responsibility argument request* message, the role sends the *Responsibility argument* message carrying the responsibility argument.
4. The player receives the responsibility argument, executes the responsibility and sends back either
   - a *Responsibility result* message carrying the responsibility result in case the responsibility executed successfully, or
   - a *Failure* message otherwise.

The player then ends its part in the protocol.

5. The role receives the reply message and ends its part in the protocol.

Chapter 5

Metamodel Implementation—Thespian4Jade

In this chapter, we will present our implementation of the *Thespian* metamodel for the *Jade* agent platform—*Thespian4Jade*. However, first we talk briefly about agent platforms in general and about *Jade* in particular.

5.1 Agent Platform

An agent platform is a software platform on which multi-agent systems—targeting this platform—can be run. Being a software platform, it abstracts the underlying hardware platform and operating system. Every agent platform comprises

- a *run-time environment*, in which a MAS can be run and
- an *framework*, using which a MAS can be built[1].

There are many platforms available, both proprietary and free/open source ones.

The primary distinguishing factor among agent platforms is the degree of generality. The most general among them do not impose any particular agent architecture and usually pick a general-purpose programming language (typically an object-oriented language, like Java or C#) as their agent programming language. On the other hand, the most specific agent platforms do prescribe a concrete agent architecture and generally introduce some (declarative) domain-specific language to program agents in.

As our aim was to introduce organizational concepts as first-class citizens to the MAS landscape, we had to steer towards the agent platform offering the richest

[1] A framework is actually an optional component of an agent platform; there are agent platforms that do not come with a framework, e.g. 3APL.

extension possibilities. Therefore, we considered only the most general free and open source agent platforms: *Jade*, *Janus*[2] and *Jason*[3]. Eventually, we chose *Jade* which, at the time of writing this thesis, appears to be the de facto standard implementation platform in the MAS research community.

5.2 Jade

Jade[4] is a general-purpose agent platform. It provides a run-time environment in which the agents live and interact with one another. Jade also manages the lifetimes of the agents running on the platform, delivers messages (asynchronous message passing) and provides other services (e.g. the yellow pages service). It also comes with an extensive base class libraries (e.g. the `Agent` class), and the user-defined classes are expected to inherit from these base classes. A multi-agent system running on *Jade* (viewed as a platform) is basically an instantiation of *Jade* (viewed as a framework).

Jade simplifies the implementation of multi-agent systems by acting as a middleware that complies with the FIPA[5] specifications[6] and by providing a set of graphical tools that support the debugging and deployment phases. The platform can be distributed across multiple machines (possibly running different operating systems) and it can be configured via a remote graphical tool. The configuration can be even changed at run-time by moving agents from one machine to another, as and when required. Jade is written in the Java programming language.

Jade has been specifically designed with extensibility in mind, and since our goal is to extend the multi-agent systems with organizational concepts, this is the platform of our choice. The fact that it places great emphasis on extensibility also manifests in its choice of the agent programming language—the Java programming language.

A feature of *Jade* we find particularly noteworthy is its minimalistic architecture. Only the most fundamental services (e.g. message delivery) are hard-wired. Whenever possible, a service is implemented as a full-fledged agent operating on the platform (e.g. white and yellow pages service). Even the graphical tools are only GUI front-ends to these service agents. This minimalistic architecture is further evidence that *Jade* takes extensibility seriously.

Jade is a free and open source software, distributed under GNU Lesser General Public License, version 2[7]. The copyright holder is Telecom Italia.

[2] http://www.janus-project.org/Home
[3] `http://jason.sourceforge.net/Jason/Jason.html`
[4] Java Agent Development Framework — `http://jade.tilab.com/`
[5] The Foundation for Intelligent Physical Agents — `http://www.fipa.org/`
[6] `http://www.fipa.org/specifications/index.html`
[7] `http://www.gnu.org/licenses/lgpl-2.1.html`

The version of *Jade* used in this thesis is 4.1.1 released on November 18th, 2011. To run *Jade* 4.1.1, the Java Platform, Standard Edition (Java SE) 1.4 or higher is required.

5.3 Thespian4Jade

Thespian4Jade is our implementation of the *Thespian* metamodel for the *Jade* platform. It is a module extending the *Jade* framework with classes that can be subclassed to define platform-specific models of organizations in a MAS running on the *Jade* platform. *Thespian4Jade* is one of our two contributions to the field, the other one being *Thespian*.

5.3.1 States and Parties

The `thespian4jade.behaviours` package contains abstractions of *Jade's* behaviours—interaction protocol parties and their states. An *interaction protocol party* (or simply *party*) is a behaviour of a participant in an interaction protocol and a *state* is one of its sub-behaviours. In order for parties to be composable, their design follows the *Composite* design pattern:

- `IState` plays the role of *Component*,
- concrete states play the role of *Leaf*, and
- `Party` and its subclasses play the role of *Composite*.

The `Responder` abstract class implements a responder behaviour that receives protocol initiation messages and invokes the corresponding protocol responder parties.

To implement a concrete responder

1. extend the `Responder` class, and
2. implement the constructor to add the protocols using the `addProtocol(protocol: Protocol)` method.

States

The `thespian4jade.behaviurs.states` package contains classes that implement states in parties.

The `IState` interface specifies a state of a party. It declares convenience methods that streamline the process of composing states into parties.

The `ISenderState` interface, an extension of `IState`, specifies a state in which a message is sent to all receivers using the `send(message: Message, receivers: AID[])` method. It has two implementations:

- `OneShotBehavioursSenderState` implementing a sender state that is also a *Jade's* one-shot[8] behaviour, and
- `FSMBehaviourSenderState` implementing a sender state that is also a *Jade's* FSM behaviour[9].

The `IReceiverState` interface, an extension of `IState`, specifies a state in which a message is received from any sender using the `receive(message: Message, senders: AID[])` method. It has two implementations:

- `OneShotBehavioursReceiverState` implementing a receiver state that is also a *Jade's* one-shot behaviour, and
- `FSMBehaviourReceiverState` implementing a sender state that is also a *Jade's* FSM behaviour.

Sender States

The `thespian4jade.behaviours.states.sender` package contains, above all, the `OuterSenderState` abstract class implementing a state in which the party sends a message to the other party. It is further specialized as

- `SingleSenderState<TMessage>` implementing a state in which only one kind of message is sent,
- `SendSuccessOrFailure<TMessage>` implementing a state in which two kinds of messages can be sent—one in case of success and the other one in case of failure—and
- `SendAgreeOrRefuse` implementing a state in which either an *AGREE* or *REFUSE* message can be sent.

Receiver States

The `thespian4jade.behaviours.states.receiver` package contains the `OuterReceiverState` abstract class implementing a state in which the party receives

[8] An simple behaviour in *Jade* that executes only once.

[9] A composite behaviour in *Jade* with sub-behaviour arranged into a finite-state machine.

a message from the other party. It it further specialized by classes analogous to the sender state classes, except they receive messages instead of sending them.

Special States

The `thespian4jade.behaviours.states.special` package contains classes implementing some special types of states. The `StateWrapperState<TState>` abstract class implements a state that enables to wrap another state (or a party for that matter) that needs to be provided an argument just before its execution[10] and provides a result just after its execution. It has two specializations:

- `InvokeCompetenceState<TArgument, TResult>` that wraps the *Invoke competence* initiator party and
- `InvokeResponsibilityState<TArgument, TResult>` that wraps the *Invoke responsibility* initiator party.

The `EventHandler<TArgument>` abstract class is an implementation of an event-handling state that is invoked (synchronously) from the *Publish event* protocol responder party.

Parties

The `tehspian4jade.behaviours.parties` contains the `Party<TAgent>` abstract class modelling a party and its two specializations:

- `InitiatorParty<TAgent>` representing a party that initiates an interaction, and
- `ResponderParty<TAgent>` representing a party that responds to the initiation of others.

The `IResultParty<TResult>` interface specifies a party that produces a result, e.g. initiator parties in the *Invoke competence* or *Invoke responsibility* protocols.

5.3.2 Organization, Role and Competences

The `thespian4jade.core.organization` package contains the classes modelling organizations and their roles with competences.

[10] Usually because it is not yet available at the time of its creation.

The `Organization` abstract class—one of *Thespian4Jade's* three core classes—models an organization [11]. Since organizations in *Thespian4Jade* are agentified[12], the class is an extension of *Jade's* `Agent` class. The `Organization_Responder` class implements an organization's responder behaviour configured to respond to the *Enact role*, *Deact role* and *Subscribe to event* protocols.

To implement a concrete organization

1. extend the `Organization` class, and
2. override the `action()` method to add the organization's roles using the `addRole(roleClass: Class)` method.

The following classes implement an organization's behaviour in all four role-related infrastructure protocols:

- `Organization_EnactRole_ResponderParty` implements a responder party in the *Enact role* protocol,
- `Organization_DeactRole_ResponderParty` implements a responder party in the *Deact role* protocol,
- `Organization_SubscribeToEvent_ResponderParty` implements a responder party in the *Subscribe to event* protocol, and
- `Organization_PublishEvent_InitiatorParty` implements an initiator party in the *Publish event* protocol.

The `Role` abstract class—another one of *Thespian4Jade's* three core classes—models a role and its instances model the role's positions. Since, like organizations, roles are agentified, the class extends *Jade's* `Agent` class. The `Role_Responder` class implements a role's default responder behaviour configured to respond to the *Activate role*, *Deactiavte role* and *Invoke competence* protocols.

To implement a concrete role

1. extend the `Role` class, and
2. override the `action()` method to schedule the role's custom responder behaviour (if the role acts as a responder party in any application-logic protocol) and to add the role's competences using the `addCompetence(competenceClass: Class)` method.

[11] Actually, the `Organization` class models an organization *type*, and its instances model organization *tokens*.

[12] An agentified organization is an agent in its own right—it it autonomous and can interact (communicate) with other agents.

The following classes implement a role's behaviour in all four role-related infrastructure protocols:

- `Role_ActivateRole_ResponderParty` implements a responder party in the *Activate role* protocol,
- `Role_DeactivateRole_ResponderParty` implements a responder party in the *Deactivate role* protocol,
- `Role_InvokeCompetence_ResponderParty` implements a responder party in the *Invoke competence* protocol, and
- `Role_InvokeResponsibility_InitiatorParty` implements an initiator party in the *Invoke responsibility* protocol.

Organization Knowledge Base

The `thespian4jade.core.organization.kb` package hold the implementation of an organization's knowledge base. In *Thespian4Jade*, organizations and roles are agentified, and therefore can have knowledge about other agents (players in particular) the system.

The `OrganizationKnowledgeBase` represents an organization's knowledge base. It stores knowledge about the players enacting roles in the organization (e.g. their responsibilities). It provides two views:

- *Query*, accessed via the `query()` method, exposing API to query the knowledge base, and
- *Update*, accessed via the `update()` method, exposing API to update the knowledge base.

Competence

The `thespianjade.core.organization.competence` package includes classes representing role competences. The `ICompetence<TArgument, TResult>` interface models a competence with a typed argument and result. It declares two methods common to all competences:

- `setArgument(argument: TArgument)` that sets the argument passed to the competence, and
- `getResult(): TResult` that gets the result returned from the competence.

A competence can be either *synchronous* or *asynchronous*.

Behaviours in agent-oriented programming correspond to methods in OOP, except that while the asynchronous method invocation has to be programmed explicitly in OOP, in agent-oriented programming, the asynchronous invocation of behaviours is implicit. For example, in *Jade* a behaviour is invoked[13] by calling the `addBehaviour(behaviour: Behaviour)` method of the `Agent` class. The method immediately returns and the invoked behaviour is executed asynchronously and concurrently with other behaviours. If a behaviour is to be invoked synchronously from another behaviour in *Jade*, the calling behaviour has to *include* the called behaviour as its sub-behaviour[14].

A synchronous competence is modelled by the `SynchronousCompetence<TArgument, TResult>` abstract class, a kind FSM behaviour. To define a concrete synchronous competence, extend this class and implement the competence logic as a beahviour included as a state (or states) of this FSM behaviour. An asynchronous competence is represented by the `AsynchronousCompetence<TArgument, TResult>` abstract class, a kind of one-shot behaviour. To define a concrete asynchronous competence, extends this class and implement the competence logic as a behaviour invoked from the overriden `action()` method.

5.3.3 Player and Responsibilities

The `thespian4jade.core.player` package contains the classes modelling players with responsibilities.

The `Player` abstract class—the last one of *Thespian4Jade's* three core classes—models a player[15]. Since the player is basically an agent able to play roles in organizations, the class extends *Jade's* `Agent` class. The `Player_Responder` class implements a player's responder behaviour configured to respond to the *Publish event* and *Invoke responsibility* protocols.

To implement a concrete player

1. extend the `Player` class, and

2. override the `action()` method to add the player's responsibilities using the `addResponsibility(responsibilityClass: Class)` method.

The following classes implement a player's behaviour in all eight infrastructure protocols:

[13] More precisely, scheduled for execution.
[14] For example, the calling FSM behaviour includes the called behaviour as one of its states.
[15] Actually, the `Player` class models a player *type*, and its instances model player *tokens*.

- `Player_EnactRole_InitiatorParty` implements an initiator party in the *Enact role* protocol,
- `Player_DeactRole_InitiatorParty` implements an initiator party in the *Deact role* protocol,
- `Player_SubscribeToEvent_InitiatorParty` implements an initiator party in the *Subscribe to event* protocol,
- `Player_PublishEvent_ReponderParty` implements a responder party in the *Publish event* protocol,
- `Player_ActivateRole_InitiatorParty` implements an initiator party in the *Activate role* protocol,
- `Player_DeactivateRole_InitiatorParty` implements an initiator party in the *Deactivate role* protocol,
- `Player_InvokeCompetence_InitiatorParty` implements an initiator party in the *Invoke competence* protocol, and
- `Player_InvokeResponsibility_RespodnerParty` implements a responder party in the *Invoke responsibility* protocol.

Player Knowledge Base

The `thespian4jade.core.player.kb` package holds the implementation of a player's knowledge base. In *Thespian4Jade*, players, being agents, can have knowledge about other agents (organizations and roles in particular) the system.

The `PlayerKnowledgeBase` represents a players's knowledge base. It stores knowledge about organizations in which the player enacts roles and the enacted roles (e.g. their competences). Like an organization's knowledge base, it also provides the *Query* and *Update* views.

Responsibilities

The `thespian4jade.core.player.responsibility` package includes classes representing player responsibilities. The `IResponsibility<TArgument, TResult>` interface models a responsibility with typed argument and result. It prescribes two methods common to all responsibilities:

- `setArgument(argument: TArgument)` that sets the argument passed to the responsibility, and
- `getResult(): TResult` that gets the result returned from the responsibility.

Similarly to a competence, a responsibility can be either synchronous or asynchronous, represented by the `SynchronousResponsibility<TArgument, TResult>` and `AsynchronousResponsibility<TArgument, TResult>` abstract classes respectively. The same that has been said about competences applies to responsibilities as well.

5.3.4 Protocols and Messages

The `thespian4jade.protocols` package contains the classes modelling protocols. By virtue of good design, the same base classes that are used to model the application-agnostic protocols in the framework itself (between a player and an organization or role) can be used to represent the application-specific protocols in the applications using the framework (between roles within an organization).

The `Protocol` abstract class models an interaction (or communication) protocol. Although *Thespian* has been designed to support protocols between more than two interacting parties, *Thespian4Jade* currently supports only protocols between two parties: the initiator and responder parties. A concrete protocol must specify the protocol-initiating performative and override two abstract methods (*Abstract factory* design pattern):

- `createInitiatorParty()` that creates a new `InitiatorParty`, and
- `createResponderParty()` that creates a new `ResponderParty`.

Concrete protocols are singletons—they are never instantiated directly but are retrieved from the protocol registry.

The `ProtocolRegistry` class implements the protocol registry. When the framework or an application needs a specific protocol, it does not instantiate the protocol class, but rather uses it as a key to retrieve that protocol's singleton from the protocol registry.

The `Protocols` static class contains the keys to retrieve the infrastructure protocols from the protocol registry. There will be an identically named static class of similar purpose in each of our examples, holding the keys to retrieve the application-logic protocols from the protocol registry.

Organization Protocols

The `thespian4jade.protocols.organization` package and its sub-packages contain classes modelling the protocols that control the interaction between a player and an organization:

- `EnactRoleProtocol` models the *Enact role* protocol,
- `DeactRoleProtocol` represents the *Deact role* protocol,
- `SubscribeToEventProtocol` models the *Subscribe to event* protocol, and
- `PublishEventProtocol` represents the *Publish event* protocol.

Role Protocols

The `thespian4jade.protocols.role` package and its sub-packages include classes representing the protocols that govern the communication between a player and a role:

- `ActivateRoleProtocol` models the *Activate role* protocol,
- `DeactivateRoleProtocol` represents the *Deactivate role* protocol,
- `InvokeCompetenceProtocol` models the *Invoke competence* protocol, and
- `InvokeResponsibilityProtocol` represents the *Invoke responsibility* protocol.

Messages

The `thespian4jade.language`package holds classes modelling messages exchanged in the protocols. The `Message` abstract class models a message with structured content; it is an abstraction over *Jade's* `ACLMessage` representing a message with unstructured content. A concrete message must specify the performative and override two abstract methods:

- `generateACLMessage(): ACLMessage` that converts this T4J message to an ACL message to be sent, and
- `parseACLMessage(aclMessage: ACLMessage)` that converts a received ACL message to this T4J message.

A message is either a *text message* or a *binary message* depending whether its payload is interpreted as a piece of text or a serializable Java object.

A text message is modelled by the `TextMessage` abstract class. To define a concrete text message, subclass `TextMessage`, and override its two abstract methods:

- `generateContent(): String` that generates the content of an ACL message from this text message, and

- `parseContent(content: String)` that parses the content of a received ACL message initializing this text message.

A binary message is represented by the `BinaryMessage` abstract class. To define a concrete binary message, extend `BinaryMessage` and override its two abstract methods:

- `getContentObject(): Serializable` that gets a serializable object from this binary message to become the ACL message's *content object*, and
- `setContentObject(contentObject: Serializable)` that sets the content object of a received ACL message initializing this binary message.

In situations where the content of a message is an unstructured piece of text, and therefore can be inserted to the message at once (as opposed to being built step by step using various setters), a *simple message* can be used. A simple message—a fallback to the unstructured-content approach—is modelled by the `SimpleMessage` class—a thin wrapper over *Jade's* `ACLMessage` class.

The `IMessageFactory<TMessage>` interface specifies a message factory used in various receiver states to create a new message. It follows the *Abstract factory* design pattern.

5.3.5 Utilities

The `thespian4jade.utilities` package contains various utility classes for general use across the whole framework. The `ClassHelper` static class defines (static) helper methods that aid class-related reflection, most notably dynamic class instantiation. The `StringUtils` static class defines (static) utility methods assisting with string manipulation performed when formulating messages.

This `thespian4jade.asynchrony` package includes types that support asynchrony in the framework. Although equivalents of these types exist in Java SE standard class libraries, we have chosen to define our own lightweight versions that provide only the necessary functionality.

The `IObserver` and `IObervable` interfaces specify the *Observer* design pattern as employed in *Thespian4Jade*. A developer can implement `IObservable` either from scratch or they can delegate the implementation to an instance of the `Observable` class, which already implements the interface. Caution has to be exercised when delegating the implementation, as the original observable object has to be passed as an argument to the `notifyObservers()` method.

The `Future<T>` class models a *future*[16]—an object that acts as a proxy for a

[16] Also called a *promise* or *IOU* (for "I Owe You").

initially unknown result of a yet-to-be-completed computation. Note that a `Future<T>` implements both the `IObserver` and `IObservable` interfaces.

The `thespian4jade.example` package holds classes that extend the base classes (e.g. `Player`) with functionality that is used in all three examples, yet is not belong to the extended class. The `RoleEnacterPlayer` class models a player whose intention is to enact a role in an organization, and both the role and the organization are predetermined. The `CompetenceInvoker` class, represents a player who intends to invoke a predetermined competence. Such player is also a role-enacter since the competence has to be invoked on some predetermined role.

Chapter 6

Examples

In this chapter, we will present three examples that demonstrate the use of the *Thespian4Jade* module to model organizations in MASs. The examples are ordered by the complexity of the MAS social structure—the first can be considered a toy problem, while the last one is a real-world problem.

The MAS in each example is presented in two parts: specification and manifestation. The specification part presents the model of the MAS and the manifestation part presents a potential run of the MAS.

In all examples, the MAS is modelled using three sub-models: a) Organization and role model, b) Protocol model and c) Player model.

In all examples, the MAS runs in five stages: 1) role enactment, 2) role activation, 3) competence and responsibility invocation, 4) role deactivation and 5) role deactment. The third stage is the problem-solving stage—a period of time during which the agents solve the problem itself by exercising their competences and fulfilling their responsibilities. The other stages are infrastructure stages—time intervals during which agents organize themselves into organizations, enact/deact roles and activate/deactivate them. For the first example, we will provide a detailed description of all five stages; for the second example, a brief description for all five stages will suffice; and for the third example, a brief description of the problem-solving stage will be sufficient.

In all three examples, we assume

- the concrete organization in which the agents want to participate already exists, and
- that the agents already know its AID[1].

Situations where the second assumption or even both assumptions do not hold

[1] *Agent ID*—an agent's address, in the format <agent-name>@<platform-name>.

lie outside the scope of this thesis (see Conclusion and Future Work).

The complete interaction diagrams[2] are too large to be reproduced here; they can be found on the companion CD-ROM.

6.1 Example 1: Remote Function Invocation

This example demonstrates a simple organization—*remote function invocation*. The purpose of this organization is to facilitate remote function invocation by grouping two agents: one agent *invokes* a function and the other one *executes* it. The reason the organization is formed in the first place is because the agent invoking the function is not capable of executing it itself. In this example, an agent invokes the factorial function, but it should be obvious that any function could be invoked this way.

Specification

Organization Part

The *Function invocation* organization type (modelled by the `FunctionInvocation_Organization` agent class) contains two roles—*Invoker* and *Executor*—and one protocol—*Invoke function*. *Function invocation* has one instance in the running MAS: the *function-invocation* organization (modelled by the `functionInvocation_Organization` agent instance).

The *Invoker* role (modelled by the `Invoker_Role` class) can invoke a function to be executed. The *Invoker* role is a *single* role. It has one competence—*Invoke function*—and no responsibilities.

The *Invoke function* competence (modelled by the `InvokeFunction_Competence` class) is a competence to invoke a function. It has one argument—the function argument—and one result—the function value.

The *Executor* role (modelled by the `Executor_Role` agent class) can execute a function upon its invocation. The *Executor* role is a *single* role. It has no competences and one responsibility—*Execute function*.

The *Execute function* responsibility (modelled by the `ExecuteFunction_Responsibility` class) is a responsibility to execute a function for some argument. It has one argument—the function argument—and one result—the function value.

[2]Diagrams showing interaction between agents in a MAS.

Protocol Part

The *Invoke function* protocol (modelled by the `InvokeFunctionProtocol` class) is a protocol by which an *Invoker* (the initiator party, modelled by the `Invoke-Function_InitiatorParty`) requests an *Executor* (the responder party, modelled by the `InvokeFunction_RespodnerParty`) to execute a function (the factorial function in this example).

The *Invoke function request* message (modelled by the `InvokeFunctionRequestMes-sage` class) is a message sent by an *Invoker* to an *Executor* requesting the latter to execute a function for a particular argument.

The *Invoke function reply* message (modelled by the `InvokeFunctionReplyMes-sage` class) is a message sent by an *Executor* to an *Invoker* informing the latter about the value of the executed function.

Player Part

The *Blank* player type (modelled by the `Blank_Player` agent class) is a player with no capabilities. It has one instance in the running MAS: *player1*. *player1* (modelled by the `player1` agent instance) intends to enact the *Invoker* role in the *function-invocation* organization and to exercise the role's *Invoke function* competence—to invoke a function to be executed by the player of the *Executor* role.

The *Factorial computer* player type (modelled by the `FactorialComputer_-Player` agent class) is a player capable of computing the factorial function. It has one instance in the running MAS: *player2*. The intention of *player2* (modelled by the `player2` agent instance) is to enact the *Executor* role in the *invoke-organization* organization and fulfil the role's *Execute function* responsibility—to execute the function invoked by the player of the *Invoker* role.

Manifestation

Stage 1: Role Enactment

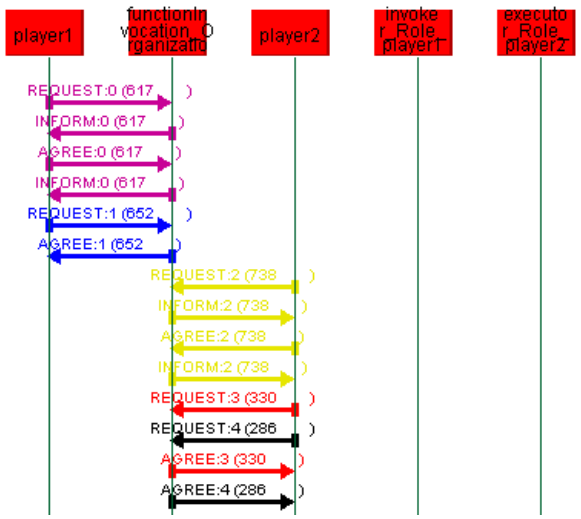


Figure 6.1: Stage 1: Role enactment

The **purple** interaction scenario between *player1* and the *function-invocation* organization follows the *Enact role* protocol. *player1* requests *function-invocation* to enact the *Invoker* role (1st message) and is informed about the role's responsibilities (2nd message) . Since *Invoker* has no responsibilities, *player1* agrees that it can indeed enact the role (3rd message). *invoke-organization* then creates the *invoker-player1* position (modelled by the `invoker_Role_player1` agent instance) and informs *player1* of its AID.

The **blue** interaction scenario between *player1* and the *function-invocation* organization follows the *Subscribe to event* protocol. *player1* requests *function-invocation* to subscribe to the *Role activated* event (1st message) and *function-invocation* agrees (2nd message).

The **yellow** interaction scenario between *player2* and the *function-invocation* organization follows the *Enact role* protocol. *player2* requests *function-invocation* to enact the *Executor* role (1st) and is informed about the role's responsibilities (2nd message). Since *player2* can perform one *Executor*'s responsibility—the *Execute function* responsibility—it agrees that it can indeed enact the role (3rd message). *function-invocationn* then creates the *executor-player2* position (modelled by the `executor_Role_player2` agent instance) and informs *player2* of its AID (4th message).

The **red** interaction scenario between t*player2* and the *function-invocation* organization follows the *Subscribe to event* protocol. *player2* requests *function-invocation* to subscribe to the *Role activated* event (1st message) and *function-invocation* agrees (2nd message).

The **black** interaction scenario between *player2* and the *function-invocation* organization follows the *Subscribe to event* protocol. *player2* requests *function-invocation* to subscribe to the *Role deactivated* event (1st message) and *function-invocation* agrees (2nd message).

Stage 2: Role Activation

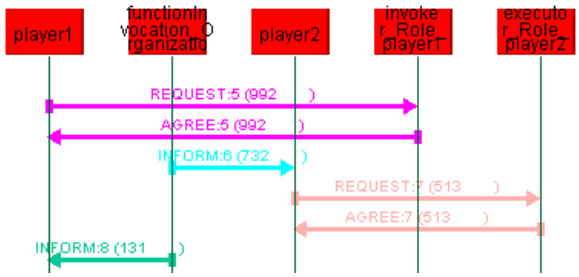


Figure 6.2: Stage 2: Role activation

The **magenta** interaction scenario between *player1* and the *invoker-player1* position follows the *Activate role* protocol. *player1* requests *invoker-player1* to activate the role (1st message) and the position promptly agrees (2nd message).

The **cyan** interaction scenario between the *function-invocation* organization and *player2* follows the *Publish event* protocol. *invoke-organization* raises a *Role activated* event (for the *Invoker* role) and *player2* handles it by activating its *Executor* role (the **pink** interaction scenario).

The **pink** interaction scenario between *player2* and the *executor-player2* position follows the *Activate role* protocol. *player2* requests *executor-player2* to activate the role (1st message) and the position immediately agrees (2nd message).

The **teal** interaction scenario between the *function-invocation* organization and *player1* follows the *Publish event* protocol. *invoke-organization* raises a *Role activated* event (for the *Executor* role) and *player1* handles it by invoking the *Invoke function* competence (the **green** interaction scenario).

Stage 3: Competence and Responsibility Invocation

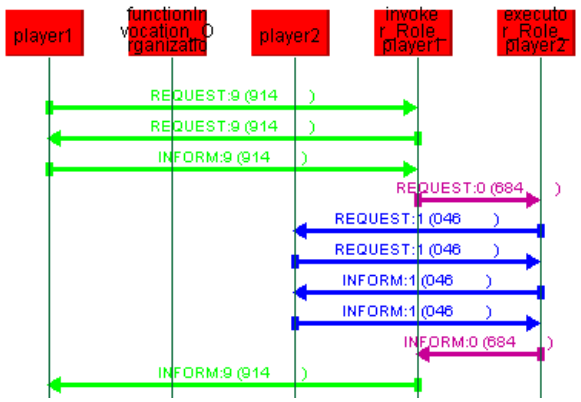


Figure 6.3: Stage 3: Competence and responsibility invocation

In the following discussion, is is important to keep in mind that the *Invoke competence* protocol is being used to invoke the *Invoke function* competence. The distinction between these two usages of the word *invoke* is a prerequisite to understanding the discussion of Stage 3.

The **green** interaction scenario between *player1* and the *invoker-player1* position follows the *Invoke competence* protocol. *player1* requests *invoker-player1* to invoke the *Invoke function* competence (1st message) and is in turn requested to provide the competence argument (factorial argument—10, 2nd message). *player1* then promptly informs *invoker-player1* about the argument (3rd message). After *invoker-player1* executes the competence (the **purple** interaction scenario), it informs *player1* about its result (factorial value—3628800, 4th message).

The **purple** interaction scenario between the *initiator-player1* and *executor-player2* positions follows the *Invoke function* protocol. *initiator-player1* requests *executor-player2* to execute a particular function (factorial) for a particular argument (10, 1st message). *executor-player2*, after invoking the *Execute function* responsibility on its player (the **blue** interaction scenario), informs *invoker-player1* about the function return value (3628800, 2nd message).

The **blue** interaction scenario between the *executor-player2* position *player2* follows the *Invoke responsibility* protocol. *executor-player2* requests *player2* to invoke the *Execute function* responsibility (1st message) and is in turn requested to provide the responsibility argument (factorial argument—10, 2nd message). *executor-player2* then immediately informs *player2* about the argument (3rd message). After *player2* executes the responsibility, it informs *executor-player2* about its result (factorial value—3628800, 4th message).

Stage 4: Role Deactivation

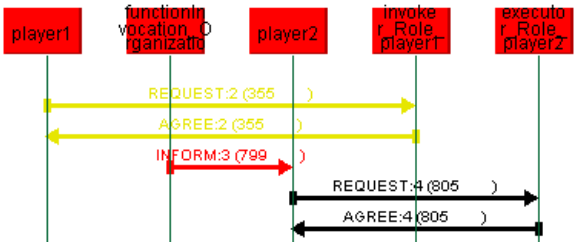


Figure 6.4: Stage 4: Role deactivation

The **yellow** interaction scenario between *player1* and the *invoker-player1* position. *player1* requests *invoker-player1* to deactivate the role (1st message) and the position promptly agrees (2nd message).

The **red** interaction scenario between the *function-invocation* organization and *player2* follows the *Publish event* protocol. *invoke-organization* raises a *Role deactivated* event (for the *Invoker* role) and *player2* handles it by deactivating its *Executor* role (the **black** interaction scenario).

The **black** interaction scenario between *player2* and the *executor-player2* position. *player2* requests *executor-player2* to deactivate the role (1st message) and the position immediately agrees (2nd message).

Stage 5: Role Deactment

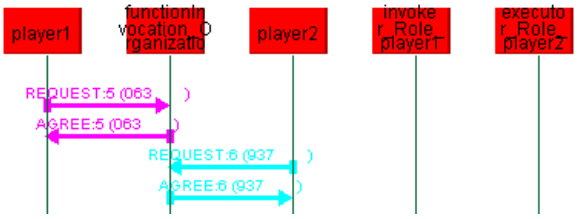


Figure 6.5: Stage 5: Role deactment

The **magenta** interaction scenario between *player1* and the *function-invocation* organization follows the *Deact role* protocol. *player1* requests *function-invocation* to deact the *Invoker* role (1st message). *function-invocation* abolishes the *invoker-position1* position and confirms this for the player (2nd message).

The **cyan** interaction scenario between *player2* and the *function-invocation* organizations follows the *Deact role* protocol. *player2* requests *function-invocation* to deact the *Executor* role (1st message). *function-invocation* abolishes the *executor-player2* position and confirms this for the player (2nd message).

6.2 Example 2: Arithmetic Expression Evaluation

This example demonstrates a not-so-simple organization—*arithmetic expression evaluation*. The purpose of this organization is to facilitate divide-and-conquer evaluation of arithmetic expressions by grouping five agents: one agent *breaks the expression down* (the *divide* part) and the other four compute *addition*, *subtraction*, *multiplication* and *integral division* (the *conquer* part), each agent computing one arithmetic operation. The reason the organization is formed in the first place is because the agent breaking the expression down is not capable of computing any arithmetic operation itself. In this example, agents evaluate simple arithmetic expressions—consisting of natural numbers, four basic arithmetic operations (addition, subtraction, multiplication and integral division) and parentheses. However, it should be apparent that any arithmetic expressions could be evaluated this way.

Specification

Organization Part

The *Expression evaluation* organization type (modelled by the `ExpressionEvaluation_Organization` agent class) contains five roles—*Evaluator*, *Adder*, *Subtractor*, *Multiplier* and *Divider*—and two protocols—*Evaluate expression* and *Evaluate binary operation*. *Expression evaluation* has one instance in the running MAS: the *expression-evaluation* organization (modelled by the `expressionEvaluation_Organization` agent instance).

The *Evaluator* role (modelled by the `Evaluator_Role` agent class) can evaluate a simple arithmetic expression. The *Evaluator* role is a *multiple* role[3]. It has one competence—*Evaluate*—and no responsibilities.

The *Evaluate* competence (modelled by the `Evaluate_Competence` class) is a competence to evaluate a simple arithmetic expression. It has one argument—an expression (a string)—and one result—the value of this expression (an integer).

The *Adder* role (modelled by the `Adder_Role` agent class) can perform addition of two simple arithmetic expressions. The *Adder* role is a *multiple* role[4]. It has no competences and one responsibility—*Add*.

The *Add* responsibility (modelled by the `Add_Responsibility` class) is a responsibility to perform addition of two integers. It has two arguments—a pair of addends—and one result—their sum.

[3]However, only one agent plays the *Evaluator* role in our example.
[4]However, only one agent plays the *Adder* role in our example.

The *Subtractor* role (modelled by the `Subtractor_Role` agent class) can perform subtraction of two simple arithmetic expressions. The *Subtractor* role is a *multiple* role[5]. It has no competences and one responsibility—*Subtract*.

The *Subtract* responsibility (modelled by the `Subtract_Responsibility` class) is a responsibility to perform subtraction of two integers. It has two arguments—a minuend and a subtrahend—and one result—their difference.

The *Multiplier* role (modelled by the `Multiplier_Role` agent class) can perform multiplication of two simple arithmetic expressions. The *Multiplier* role is a *multiple* role[6]. It has no competences and one responsibility—*Multiply*.

The *Multiply* responsibility (modelled by the `Multiply_Responsibility` class) is a responsibility to perform multiplication of two integers. It has two arguments—a pair of factors—and one result—their product.

The *Divider* role (modelled by the `Divider_Role` agent class) can perform division of two simple arithmetic expressions. The *Divider* role is a *multiple* role[7]. It has no competences and one responsibility—*Divide*.

The *Divide* responsibility (modelled by the `Divide_Responsibility` class) is a responsibility to perform integral division of two integers. It has two arguments—a dividend and a divisor—and one result—their quotient.

In the following, we will use the *Binary operator* abstract role to refer to refer to the *Adder*, *Subtractor*, *Multiplier* or *Divisor* role where it is not necessary to distinguish between them.

Protocol Part

The *Evaluate expression* protocol (modelled by the `EvaluateExpressionProtocol` class) is a protocol by which a *Binary operator* (the initiator party, modelled by the `EvaluateExpression_InitiatorParty`) requests an *Evaluator* (the responder party, modelled by the `EvaluateExpression_ResponderParty`) to evaluate an expression.

The *Evaluate expression request* message (modelled by the `EvaluateExpressionRequestMessage` class) is a message sent by a *Binary operator* to an *Evaluator* requesting the latter to evaluate an expression.

The *Evaluate expression reply* message (modelled by the `EvaluateExpressionReplyMessage` class) is a message sent by an *Evaluator* to a *Binary operator* informing the latter about the value of the evaluated expression.

[5]However, only one agent plays the *Subtractor* role in our example.
[6]However, only one agent plays the *Multiplier* role in our example.
[7]However, only one agent plays the *Divider* role in our example.

The *Evaluate binary operation* protocol (modelled by the `EvaluateBinaryOperationProtocol` class) is a protocol by which an *Evaluator* (the initiator party, modelled by the `EvaluateBinaryOperation_InitiatorParty`) requests a *Binary operator* (the responder party, modelled by the `EvaluateBinaryOperation_ResponderParty`) to evaluate a binary operation.

The *Evaluate binary operation request* message (modelled by the `EvaluateBinaryOperationReqestMessage` class) is a message sent by a *Evaluator* to a *Binary Operator* requesting the latter to evaluate a binary operation between two operand expressions.

The *Evaluate binary operation reply* message (modelled by the `EvaluateBinaryOperationReplyMessage` class) is a message sent by a *Binary operator* to an *Evaluator* informing the latter about the value of the evaluated binary operation.

Player Part

The *Blank* player type (modelled by the `Blank_Player` agent class) is a player with no capabilities. It has one instance in the running MAS: *player1*. *player1* (modelled by the `player1` agent instance) intends to enact the *Evaluator* role in the *expression-evaluation* organization and to exercise the role's *Evaluate* competence—to have an expression evaluated by the players of the *Binary operator* roles.

The *Addition computer* player type (modelled by the `AdditionComputer_Player` agent class) is a player capable of computing the addition operation. It has one instance in the running MAS: *player2*. *player2* (modelled by the `player2` agent instance) intends to enact the *Adder* role in the *expression-evaluation* organization and fulfil the role's *Add* responsibility—to compute addition during the evaluation of the expression from the player of the *Evaluator* role.

The *Subtraction computer* player type (modelled by the `SubtractionComputer_Player` agent class) is a player capable of computing the subtraction operation. It has one instance in the running MAS: *player3*. The intention of *player3* (modelled by the `player3` agent instance) is to enact the *Subtractor* role in the **expression-evaluation** organization and perform the role's *Subtract* responsibility—to compute subtraction during the evaluation of the expression from the player of the *Evaluator* role.

The *Multiplication computer* player type (modelled by the `MultiplicationComputer_Player` agent class) is a player capable of computing the multiplication operation. It has one instance in the running MAS: *player4*. *player4* (modelled by the `player4` agent instance) intends to enact the *Multiplier* role in the *expression-evaluation* organization and fulfil the role's *Multiply* responsibility—to compute multiplication during the evaluation of the expression from the player of the *Evaluator* role.

The *Division computer* player type (modelled by the `DivisionComputer_Player`

agent class) is a player capable of computing the division operation. It has one instance in the running MAS: *player5*. The intention of *player5* (modelled by the `player5` agent instance) is to enact the *Divisor* role in the *expression-evaluation* organization and fulfil the role's *Divide* responsibility—to compute division during the evaluation of the expression from the player of the *Evaluator* role.

Manifestation

Stage 1: Role Enactment

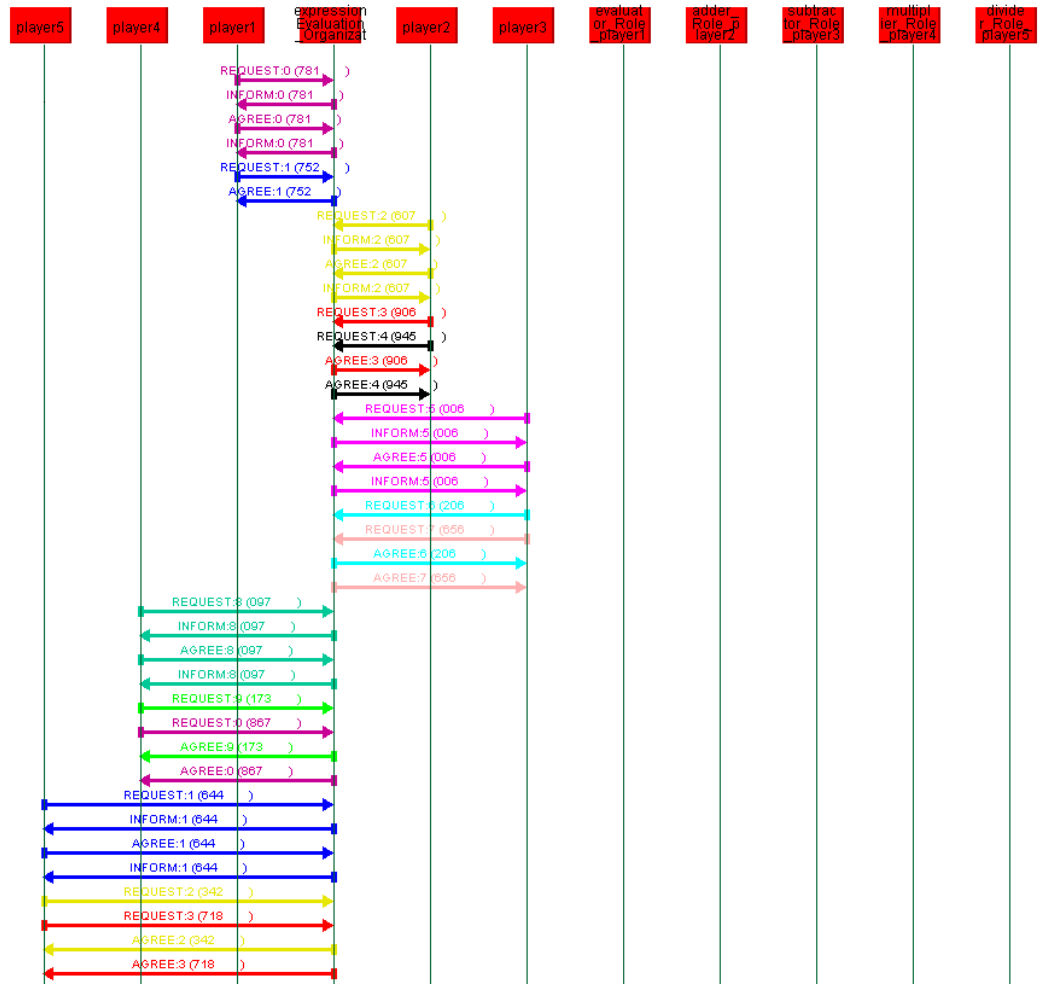


Figure 6.6: Stage 1: Role enactment

In the **first purple** interaction scenario, *player1* enacts the *Evaluator* role, resulting in the creation of the *evaluator-player1* position. In the **first blue** interaction scenario, *player1* subscribes to the *Activate role* event.

In the **first yellow** interaction scenario, *player2* enacts the *Adder* role, resulting in the creation of the *adder-player2* position. In the **first red** and **black** interac-

tion scenarios, *player2* subscribes to the *Activate role* and *Deactivate role* events respectively.

In the **magenta** interaction scenario, *player3* enacts the *Subtractor* role, resulting in the creation of the *subtractor-player3* position. In the **cyan** and **pink** interaction scenarios, *player3* subscribes to the *Activate role* and *Deactivate role* events respectively.

In the **teal** interaction scenario, *player4* enacts the *Multiplier* role, resulting in the creation of the *multiplier-player4* position. In the **green** and **second purple** interaction scenarios, *player4* subscribes to the *Activate role* and *Deactivate role* events respectively.

In the **second blue** interaction scenario, *player5* enacts the *Divider* role, resulting in the creation of the *divider-player5* position. In the **second yellow** and **second red** interaction scenarios, *player5* subscribes to the *Activate role* and *Deactivate role* events respectively.

Stage 2: Role Activation

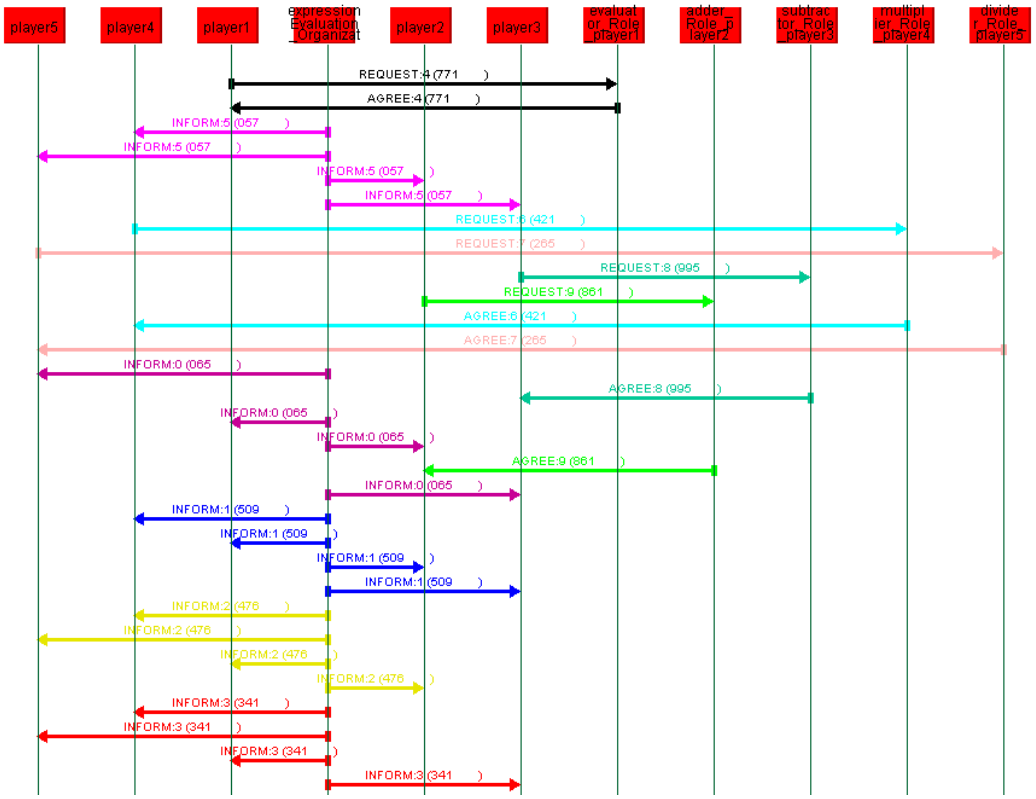


Figure 6.7: Stage 2: Role activation

In the **black** interaction scenario, *player1* activates its *Evaluator* role. In the **magenta** interaction scenario, the *expression-evaluation* organization publishes

the *Role activated* event (for the *Eveluator* role). *player2* reacts by activating its *Adder* role (the **green** interaction scenario), *player3* by activating its *Subtractor* role (the **teal** interaction scenario), *player4* by activating its *Multiplier* role (the **cyan** interaction scenario) and *player5* by activating its *Divider* role (the **pink** interaction scenario).

In the **green** interaction scenario, *player2* activates its *Adder* role. In the **red** interaction scenario, the *expression-evaluation* organization publishes the *Role activated* event (for the *Adder* role).

In the **teal** interaction scenario, *player3* activates its *Subtractor* role. In the **yellow** interaction scenario, the *expression-evaluation* organization publishes the *Role activated* event (for the *Subtractor* role).

In the **cyan** interaction scenario, *player4* activates its *Multiplier* role. In the **purple** interaction scenario, the *expression-evaluation* organization publishes the *Role activated* event (for the *Multiplier* role).

In the **pink** interaction scenario, *player5* activates its *Divider* role. In the **blue** interaction scenario, the *expression-evaluation* organization publishes the *Role activated* event (for the *Divider* role).

Note that that in all role activation scenarios, the player activating a role is not notified about the resulting *Role activated* event, although it is subscribed to it; there is no need to notify the player causing the event in the first place.

Stage 3: Competence and Responsibility Invocation

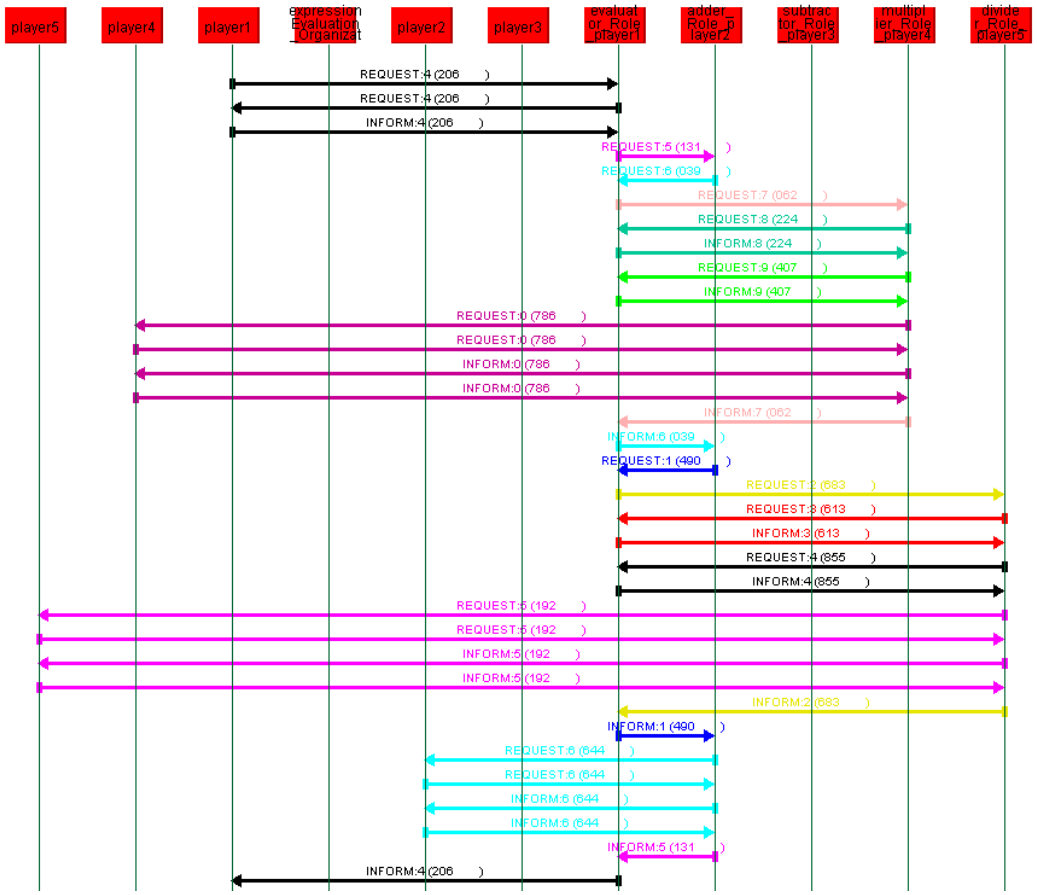


Figure 6.8: Stage 3: Competence and responsibility invocation

The *Evaluate expression* competence is invoked by *player1* on its *Evaluator* role and is carried out in a divide-and-conquer fashion by an *Evaluator*, an *Adder*, a *Multiplier* and a *Divider* by collaborating with one another and invoking responsibilities on their respective players. In this example the arithmetic expression to be evaluated is $(1 \cdot 2) + (4/2)$.

In the **first black** interaction scenario, the *evaluator-player1* position is requested by *player1* to invoke the *Evaluate* competence for the expression $(1 \cdot 2) + (4/2)$. First, it parses the expression, finds the operation to be applied last—addition—and splits the expression into two sub-expressions to be added. Next, it requests *adder-player2* to evaluate their sum (the **first magenta** interaction scenario). Finally, it reports the value (4) to *player1*.

In the **first magenta** interaction scenario, the *adder-player2* position is requested to evaluate the sum of the expressions $(1 \cdot 2)$ and $(4/2)$. First, it requests *evaluator-player1* to evaluate both expressions (the **first cyan** and **blue** interaction scenarios). Next, it invokes the *Add* responsibility on *player2* to calculate the sum of their values (the **second cyan** interaction scenario). Finally, it reports the sum (4) to *evaluator-player1* (the original **first magenta** interaction scenario).

In the **first cyan** interaction scenario, the *evaluator-player1* position is requested to evaluate the expression $(1 \cdot 2)$. First, it parses the expression, finds the last-to-be-applied operation—multiplication—and splits the expression into two sub-expressions to be multiplied. Next, it requests *multiplier-player4* to evaluate their product (the pink interaction scenario). Finally, it reports the value (2) to *adder-player2*.

In the pink interaction scenario, the *multiplier-player4* position is requested to evaluate the product of the expressions 1 and 2. First, it requests *evaluator-player1* to evaluate both expressions (the **teal** and **green** interaction scenarios). Next, it invokes the *Multiply* responsibility on *player4* to calculate the product of their values (the **purple** interaction scenario). Finally, it reports the product (2) to *evaluator-player1* (the original pink interaction scenario).

In **teal** interaction scenario, the *evaluator-player1* is requested to evaluate the expression 1. It parses the expression, finds out it is a number (bottom case) and reports the value (1) to *multiplier-player4*.

In **green** interaction scenario, the *evaluator-player1* is requested to evaluate the expression 2. It parses the expression, finds out it is a number (bottom case) and reports the value (2) to *multiplier-player4*.

In the **blue** interaction scenario, the *evaluator-player1* is requested to evaluate the expression $(4/2)$. First, it parses the expression, finds the last-to-be-applied operation—division—and splits the expression into two sub-expressions to be divided. Next, it requests *divider-player5* to evaluate their product (the yellow interaction scenario). Finally, it reports the value (2) to *adder-player2*.

In the yellow interaction scenario, the *divider-player5* position is requested to evaluate the quotient of the expressions 4 and 2. First, it requests *evaluator-player1* to evaluate both expressions (the **red** and **black** interaction scenarios). Next, it invokes the *Divide* responsibility on *player5* to calculate the quotient of their values (the **second magenta** interaction scenario). Finally, it reports the quotient (2) to *evaluator-player1* (the original yellow interaction scenario).

In **red** interaction scenario, the *evaluator-player1* is requested to evaluate the expression 4. It parses the expression, finds out it is a number (bottom case) and reports the value (4) to *divider-player5*.

In **black** interaction scenario, the *evaluator-player1* is requested to evaluate the expression 2. It parses the expression, finds out it is a number (bottom case) and reports the value (2) to *divider-player5*.

Stage 4: Role Deactivation

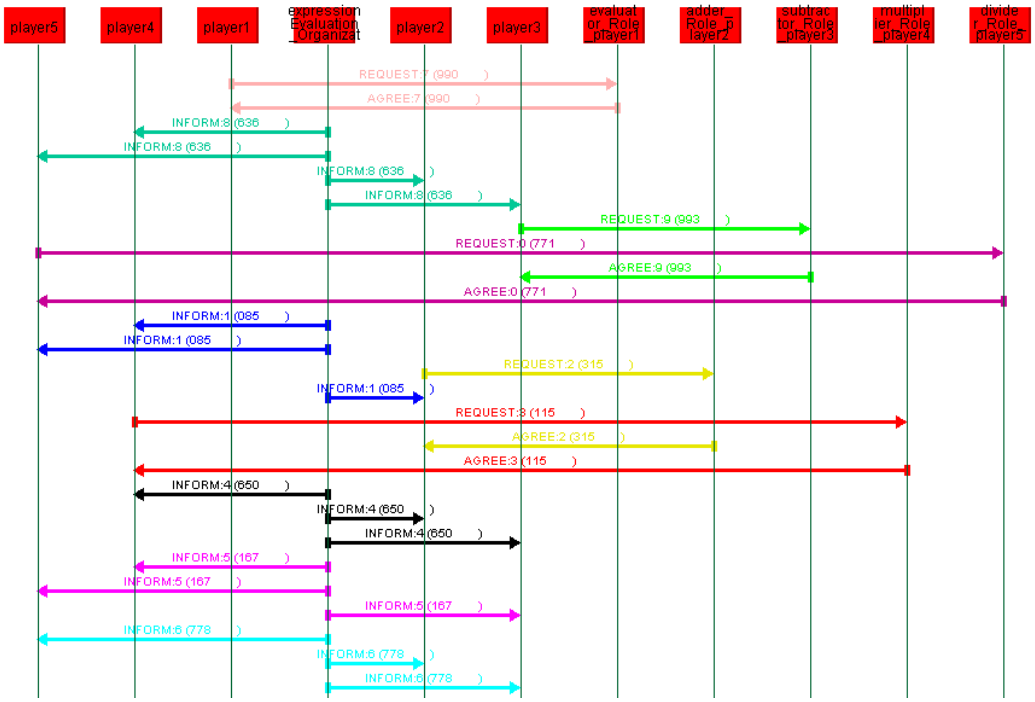


Figure 6.9: Stage 4: Role deactivation

In the pink interaction scenario, *player1* deactivates its *Evaluator* role. In the **teal** interaction scenario, the *expression-evaluation* organization publishes the *Role deactivated* event (for the *Evaluator* role). *player2* reacts by deactivating its *Adder* role (the **yellow** interaction scenario), *player3* by deactivating its *Subtractor* role (the **green** interaction scenario), *player4* by deactivating its *Multiplier* role (the **red** interaction scenario) and *player5* by deactivating its *Divider* role (the **purple** interaction scenario).

In the **yellow** interaction scenario, *player2* deactivates its *Adder* role. In the **magenta** interaction scenario, the *expression-evaluation* organization publishes the *Role deactivated* event (for the *Adder* role).

In the **green** interaction scenario, *player3* deactivates its *Subtractor* role. In the **blue** interaction scenario, the *expression-evaluation* organization publishes the *Role deactivated* event (for the *Subtractor* role).

In the **red** interaction scenario, *player4* deactivates its *Multiplier* role. In the **cyan** interaction scenario, the *expression-evaluation* organization publishes the *Role deactivated* event (for the *Multiplier* role).

In the **purple** interaction scenario, *player5* deactivates its *Divider* role. In the **black** interaction scenario, the *expression-evaluation* organization publishes the *Role deactivated* event (for the *Divider* role).

Stage 5: Role Deactment

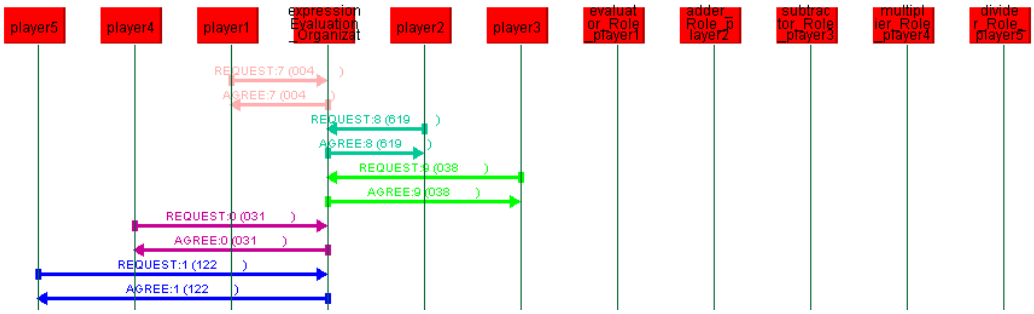


Figure 6.10: Stage 5: Role deactment

In the pink interaction scenario, *player1* deacts its *Evaluator* role and the *evaluator-player1* position is abolished.

In the teal interaction scenario, *player2* deacts its *Adder* role and the *adder-player2* position is abolished.

In the green interaction scenario, *player3* deacts its *Subtractor* role and the *subtractor-player3* position is abolished.

In the purple interaction scenario, *player4* deacts its *Multiplier* role and the *multiplier-player4* position is abolished.

In the blue interaction scenario, *player5* deacts its *Divider* role and the *divider-player5* position is abolished.

6.3 Example 3: Sealed-Bid Auction

This example demonstrates a relatively complex organization—*sealed-bid auction*. The purpose of this organization is to facilitate sealed-bid auction by bringing together several agents: one agent *auctions* an item and the other agents *bid* for it.

A *sealed-bid auction* is a type of auction in which bidders simultaneously submit bids to the auctioneer without knowledge of the amount bid by other participants. Two sub-types of a sealed-bid auction are supported, depending on whether the winner pays the amount bid—a *first-price auction*, also called an *envelope auction*—or an amount equal to the next highest bid—*a second-price auction*, also known as a *Vickrey auction*.

In this example, agents participate in an envelope auction selling and buying famous paintings, but it should be obvious that any of the two auction types

could be used to sell any items.

Specification

Organization Part

The *Auction* organization type (modelled by the `Auction_Organization` agent class) contains two roles—*Auctioneer* and *Bider*—and two protocols—*Envelope auction* and *Vickrey auction*. *Auction* has one instance in the running MAS: the *auction* organization (modelled by the `auction_Organization` agent instance).

The *Auctioneer* role (modelled by the `Auctioneer_Role` agent class) can auction an item in an auction. The *Auctioneer* role is a *single* role. It has one competence—*Auction*—and no responsibilities.

The *Auction* competence (modelled by the `Auction_Competence` class) is a competence to auction an item. It has several arguments–mainly the name of the item and the *reserve price*[8]—and several results—particularly the winner's AID and *hammer price*[9].

The *Bidder* role (modelled by the `Bidder_Role` class) can bid for an item in an auction. The *Bidder* role is a *multiple* role. It has no competences and one responsibility—*Bid*.

The *Bid* responsibility (modelled by the `Bid_Responsibility` class) is a responsibility to bid for an item. It has several arguments—mainly the name of the item—and several results—the bid amount in particular.

Protocol Part

The *Envelope auction* protocol (modelled by the `EnvelopeAuctionProtocol` class) is a protocol defining an envelope auction by which an *Auctioneer* (the initiator party, modelled by the `EnvelopeAuction_InitiatorParty` class) can determine the best buyer from among the *Bidders* (responder party, modelled by the `EnvelopeAuction_RespoderParty` class).

The *Vickrey auction* protocol (modelled by the `VickreyAuctionProtocol` class) is a protocol defining a Vickrey auction by which an *Auctioneer* (the initiator party, modelled by the `VickreyAuction_InitiatorParty` class) can choose the best buyer from among the *Bidders* (responder party, modelled by the `VickreyAuction_RespoderParty` class).

[8]The smallest price at which a seller is willing to sell an item.

[9]The price at which an item is eventually sold.

The *Auction call-for-proposals (CFP)* message (modelled by the `AuctionCFPMessage` class) is a message sent by an *Auctioneer* to all *Bidders* calling for proposals for the item's price (bids).

The *Bid propose* message (modelled by the `BidProposeMessage` class) is a message sent by a *Bidder* to an *Auctioneer* proposing a price for the item (bid).

Player Part

The *Participant* player type (modelled by the `Participant_Player` agent class) is a player capable of biding for an item. *Participant* has three instances in the running MAS: *player1*, *player2* and *player3*. All of them intend to enact both the *Auctioneer* and *Bidder* roles in the *auction* organization and exercise the *Auctioneer* role's *Auction* competence—to auction an item to the players of the **Bidder** role. They also intend to fulfil the *Bidder* role's *Bid* responsibility—to bid for an item auctioned by the player of the *Auctioneer* role. *player1* (modelled by the `player1` agent instance) intends to play the role of **Auctioneer** in the *first* round and be a **Bidder** in the second and third rounds. *player2* (modelled by the `player2` agent instance) intends to act as *Auctioneer* in the *second* round and be a *Bidder* in the first and third rounds. *player3* (modelled by the `player3` agent instance) intends to play the role of *Auctioneer* in the *third* round and be the *Bidder* in the first and second rounds.

Manifestation

Stage 3: Competence and Responsibility Invocation

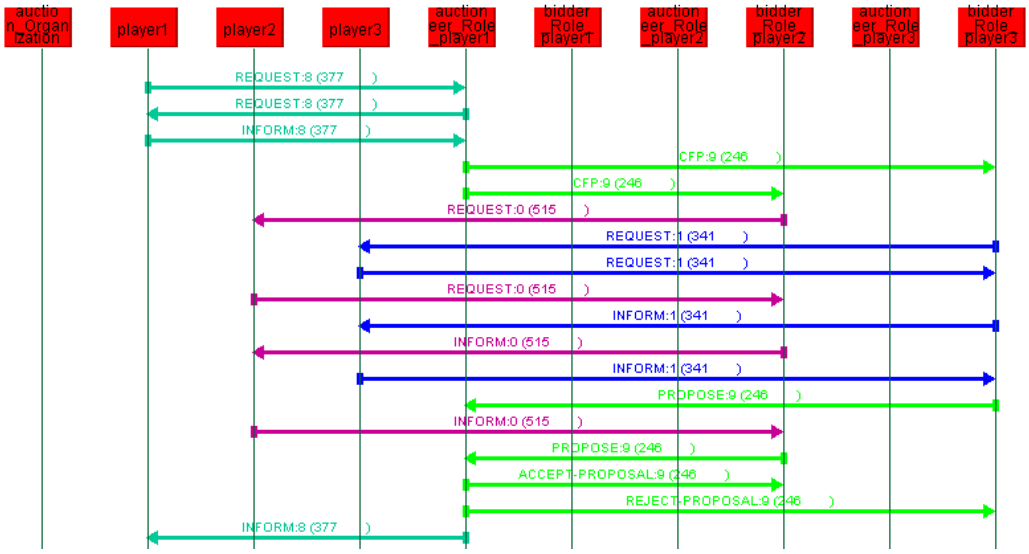


Figure 6.11: Stage 3: Competence and responsibility invocation

First, *player1* invokes the *Auction* competence for Jason Pollock's *No. 5, 1948* for the reservation price of \$140M[10] on its *auctioneer-player1* position (the **teal** interaction scenario).

Next, *auctioneer-player1* calls for proposals for the painting's price from all *Bidder* positions (the **green** interaction scenario).

To obtain their proposals, the *bidder-player2* and *bidder-player3* positions invoke the *Bid* responsibility on their *player2* and *player3* respectively (the **purple** and **blue** interaction scenarios respectively). *player2* bids \$156.8M[11] and *player3* makes a bid of \$155.8M.

bidder-player2 and *bidder-player3* propose their bids to *auctioneer-player1* (the **green** interaction scenario again), who determines the auction winner and the hammer price and also determines whether the auction is ultimately successful, i.e. whether the hammer price is at least the reservation price. The winner is the highest-bidding bidder—*bidder-player2*. Because this is an envelope auction, the hammer price is the highest amount bid— \$156.8M. Since the hammer price is way above the reservation price, the auction is successful.

Next, *auctioneer-player1* informs the winner—*bidder-player2*—that his proposal has been accepted and all other bidders—*bidder-player3*—that their proposal has been rejected (still the **green** interaction scenario).

Finally, *auctioneer-player1* informs *player1* about the outcome of the auction—whether it was successful or not and in case it was, who is the winner and what is the hammer price (the **teal** interaction scenario again).

[10]The actual original price at the private sale of this painting via Sotheby's in 2006.
[11]The actual adjusted price at the private sale of this painting.

Conclusions and Future Work

The objectives of this thesis were to investigate some of the metamodel-based approaches to modelling organizations in multi-agent systems, to propose a new organizational metamodel inspired the existing ones, to implement this metamodel in a free and open-source mainstream general-purpose agent platform, and to demonstrate its use with a number of examples.

We have investigated a total of four organizational metamodels and have drawn inspiration from all of them. We have observed that they all more or less agree on the core concepts (organization, role), but offer different views of the auxiliary concepts.

Inspired by the existing ones, we have proposed a new organizational metamodel—*Thespian*. *Thespian* is a fusion of investigated metamodels and some original ideas, for example, the mechanism of subscribing to organization events, their publishing and subsequent handling. *Thespian* was designed to strike a balance between expresiveness and simplicity; it provides enough concepts for modelling real world organizations, but at the same avoids introducing modelling constructs peripheral to this domain. In other words, in *Thespian* we have made an effort to provide a tool that does just one job—modelling organizations—and does it well.

To verify the applicability of *Thespian*, we have implemented it as an extension for *Jade* (the most popular free and open-source general-purpose agent platform) called *Thespian4Jade*. To our knowledge, *Thespian4Jade*—being an actually implemented[12] organizational extension of an existing general-purpose agent platform—is an unprecedented[13] effort.

In order to demonstrate the use of *Thespian4Jade*, we have modelled three example organizations ranging from the simplest possible (remote function invocation) through a sightly more involved (arithmetic expression evaluation) to a real-world organization (sealed-bid auction). With these examples we have highlighted the separation of behaviour acquired through a role from the behaviour inherent to a player. This decoupling of innate and acquired behaviour has one particularly appealing consequence—an organization can be designed and developed independently from the players who will participate in it.

[12] As opposed to just specified.

[13] And at the time of writing this thesis also unique.

Although we are confident *Thespian* is a sufficiently expressive metamodel and *Thespian4Jade* is a useful *Jade* extension, there are a few places where we made certain assumptions and imposed some restrictions. This way, we could do without some features that would otherwise have been indispensable. These features are outlined here as the strongest candidates for any future work.

It is assumed that a concrete organization in which a player wants to enact a role already exists. It has to be specified at compile-time, and it is created at MAS start-up. Undoubtedly, a more flexible approach, where a player could request that an organization of certain type be created, would be a step in the right direction. For this to work, a special agent—*Organization manager*—would have to exist capable of creating organizations on demand.

Currently, only a player can initiate role enactment. This is not an accurate reflection of reality. In the real world, not only a job-seeker can initiate the recruitment process; a company can take the initiative by selecting candidates for a position from a pool of all job-seekers and making them an offer. Two features would have to be added to *Thespian* in order to support this process: *yellow pages for players* that would enable an organization to search for existing players with certain capabilities and *organization-initiated role enactment*.

At present, role deactment can only by initiated by a player. Again, this is not true in the real world. In reality, an employee quitting a job is not the only way for them to stop being employed; a company can fire an employee if they fail to fulfil their position's responsibilities. In order for *Thespian* to support this procedure, two features would have to be added: *responsibilities fulfilling monitoring* that would enable an organization to keep track of its players' performance and *organization-initiated role deactment*.

Another restriction is that only a player can play a role. At first sight, this does not seem like a too big a restriction, if it is a restriction at all. However, an organization could be viewed as a player as well, capable of more than its individual members put together. Therefore, it makes sense to consider the possibility of organizations playing roles in other organizations. Such an organization is a *holon* (simultaneously a whole and a part) in a *holonic MAS*. *Thespian*, and *Thespian4Jade* in particular, are well prepared to accommodate this feature, with the organizations already being agentified.

Bibliography

[1] Michael Wooldridge, *An Introduction to Multiagent Systems, Second Edition*, Wiley, 2009

[2] Gerhard Weiss, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, MIT Press, 1999

[3] Yoav Shoham and Kevin Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, Cambridge University Press, 2008

[4] Aleš Kubík, *Inteligentní agenty: Tvorba aplikačného software na bázi multiagentových systémů*, Computer Press, 2004

[5] Michael Wooldridge and Nicolas Jennings, 'Intelligent Agents: Theory and Practice', in *Knowledge Engineering Review* (Vol. 10 No. 2), Cambridge University Press, 1995

[6] Michael Wooldridge, 'Intelligent Agents: The Key Concepts', in *Proceedings of the 9th ECCAI-ACAI / EASSS 2001, AEMAS 2001, HoloMAS 2001*, pages 3–43, Springer, 2002

[7] Aleš Kubík, 'Agentově-orientované inženýrství: nové paradigma pro tvorbu softwaru?', in *Proceedings of Objekty 2003*, pages 44–59, Vysoká škola báňská – Technická univerzita Ostrava, 2003

[8] Federico Bergenti et al., *Methodologies and Software Engineering for Agent Systems*, Springer, 2004

[9] Lin Padgham and Michael Winikoff, *Developing Intelligent Agent Systems: A Practical Guide*, Wiley, 2004

[10] Gonzalo Génova, 'What is a metamodel: The OMG's Metamodeling Infrastructure', `http://www.ie.inf.uc3m.es/ggenova/Warsaw/Part3.pdf` (accessed April 1, 2012)

[11] OMG, *A Proposal for an MDA Foundation Model*, `http://www.ie.inf.uc3m.es/grupo/docencia/reglada/ASDM/MDA_Foundation_Model_05-04-01.pdf` (accessed April 1, 2012)

[12] Jacques Ferber and Oliver Gutknecht, *Aalaadin: A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems*, Unpublished research report, 1997

[13] Jacques Ferber and Oliver Gutknecht, 'A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems', in *Proceedings of the Third International Conference on Multi Agent Systems (ICMAS 1998), Paris, France*, pages 128–138, IEEE Computer Society Washington, 1998

[14] Jacques Ferber and Oliver Gutknecht, 'Operational Semantics of a Role-based Agent Architecture', in *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL 1999), Orlando, Florida, USA*, pages 205–217, Springer Berlin, 2000

[15] Oliver Gutknecht and Jacques Ferber, 'MadKit: A Generic Multi-Agent Platform', in *Proceedings of the Fourth International Conference on Autonomous Agents (Agents 2000), Barcelona, Spain*, pages 78–79, ACM New York, 2000

[16] Jacques Ferber et al., 'From Agents to Organizations: An Organizational View of Multi-Agent Systems', in *Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering (AOSE 2003), Melbourne, Australia*, pages 214–230, Springer Berlin, 2003

[17] James J. Odell et al., 'Representing Agent Interaction Protocols in UML', in *Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE 2000), Limerick, Ireland*, pages 121–140, Springer Berlin, 2001

[18] H. van Dyke Parunak and James J. Odell, 'Representing Social Structures in UML', in *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE 2001), Montreal, Canada*, pages 1–16, Springer Berlin, 2002

[19] James J. Odell et al., 'The Role of Roles in Designing Effective Agent Organizations', in *Software Engineering for Large-Scale Multi-Agent Systems (LNCS 2603)*, pages 27–38, Springer, 2003

[20] James J. Odell et al., 'Temporal Aspects of Dynamic Role Assignment', in *Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering (AOSE 2003), Melbourne, Australia* pages 201–213, Springer, 2004

[21] James Odell et al., 'A Metamodel for Agents, Roles, and Groups', in *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE 2004), New York City, New York, USA*, pages 78–92, Springer, 2005

[22] Christian Hahn et al., *A Platform-Independent Model for Agents*, German Research Centre for Artificial Intelligence (DFKI), 2007

[23] Christian Hahn et al., 'Interoperability through a Platform-Independent Model for Agents', in *Enterprise Interoperability II: New Challenges and Approaches* pages 195–206, Springer, 2007

[24] Christian Hahn et al., 'A Platform-Independent Metamodel for Multiagent Systems', in *Autonomous Agents and Multi-Agent Systems* (Vol. 18 No. 2), pages 239–266, Springet, 2008

[25] Cristián Madrigal-Mora et al., 'Implementing Organisations in JADE', in *Proceedings of the Sixth German Conference on Multiagent System Technologies (MATES 2008), Kaiserslautern, Germany*, pages 135—146, Springer, 2008

[26] Cristián Madrigal-Mora and Klaus Fischer, 'Adding Organisations and Roles to JADE with JadeOrgs', in *Proceedings of the International Workshops ATOP 2005 and ATOP 2008*, pages 98–117, Springer, 2009

[27] Matteo Baldoni et al., 'Introducing Ontologically Founded Roles in Object Oriented Programming', in *Proceedings of AAAI Fall Symposium on Roles, An Interdisciplinary Perspective (ROLES 2005), Arlington, Virginia, USA*, AAAI Press, 2005

[28] Matteo Baldoni et al., 'Roles as a Coordination Construct: Introducing powerJava', in *Electronic Notes in Theoretical Computer Science* (Vol. 150), pages 9—29, Elsevier, 2006

[29] Matteo Baldoni et al., 'powerJava: Ontologically Founded Roles in Object Oriented Programming Languages', in *Proceedings of the Twenty-first Annual ACM Symposium on Applied Computing (SAC 2006) Dijon, France*, pages 1414–1418, ACM Press, 2006

[30] Matteo Baldoni et al., 'Interaction between Objects in powerJava', in *Journal of Object Technology* (Vol. 6, No. 2), pages 5–30, ETH Zurich, 2007

[31] Guido Boella and Leendert van der Torre, 'Organizations as Socially Constructed Agents in the Agent Oriented Paradigm', in *Proceedings of the Fifth International Workshop on Engineering Societies in the Agents World (ESAW 2004), Toulouse, France*, pages 1–13, Springer, 2004

[32] Mehdi Dastani et al., 'Enacting and Deacting Roles in Agent and Programming', in *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE 2004), New York City, New York, USA*, pages 189—204, Springer, 2004

[33] Guido Boella and Leendert van der Torre, 'A Foundational Ontology of Organizations and Roles', in *Proceedings of the Fourth International Workshopn on Declarative Agent Languages and Technologies (DALT 2006), Hakodate, Japan*, pages 78–88, Springer, 2006

[34] Guido Boella et al., 'Roles in Coordination and in Agent Deliberation: A Merger of Concepts', in *Proceedings of the Tenth Pacific Rim International Conference on Multi-Agents (PRIMA 2007), Bangkok, Thailand* pages 62–73 Springer, 2009

[35] Matteo Baldoni et al., 'Adding Organizations and Roles as Primitives to JADE Framework', in *Proceedings of the Third International Workshop on Normative Multiagent Systems (NorMAS 2008) Luxembourg, Luxembourg*, pages 95–111, Schloss Dagstuhl – Leibniz Center for Informatics, 2008

[36] Matteo Baldoni et al., 'powerJADE: Organizations and Roles as Primitives in the JADE Framework', in *Proceedings of the Ninth Worskhop 'From Objects to Agents' (WOA 2008) Palermo, Italy*, pages 84–92, Senecaedizioni, Torino, 2008

[37] Matteo Baldoni et al., 'Modeling Organizations and Roles Using a Middleware Jade-Based', Unpublished research report, 2009

[38] Matteo Baldoni et al., 'A Middleware for Modeling Organizations and Roles in Jade', in *Proceedings of the Seventh International Conference on Programming Multi-Agent Systems (ProMAS 2009), Budapest, Hungary*, pages 100–117, Springer, 2010

[39] Klaus Fischer, 'Holonic Multiagent Systems: Theory and Applications', in *Proceedings of the Ninth Portuguese Conference on Artificial Intelligence (EPIA 1999), Évora, Portugal*, pages 34–48, Springer, 1999

[40] Christian Gerber et al., *Holonic Multi-Agent Systems*, Unpublished research report, 1999

[41] Klaus Fischer et al., 'Holonic Multiagent Systems: A Foundation for the Organisation of Multiagent Systems', in *Proceedings of the First International Conference on Applications of Holonic and Multi-Agent Systems (HoloMAS 2003), Prague, Czech Republic*, pages 71–80, Springer, 2003

[42] Michael Schillo and Klaus Fischer, 'Holonic Multiagent Systems', in *Zeitschrift für Künstliche Intelligenz* (No. 3), 2004

[43] Rafael H. Bordini et al., *Multi-Agent Programming: Languages, Platforms and Applications*, Springer, 2005

[44] Rafael H. Bordini et al., *Multi-Agent Programming: Languages, Tools and Applications*, Springer, 2009

[45] Rafael Bordini et al., 'A Survey of Programming Languages and Platforms for Multi-Agent Systems', in *Informatica* (Vol. 30), pages 33—44, Slovenian Society Informatika, 2006

[46] Mehdi Dastani, 'Programming Multi-Agent Systems', in *Proceedings of the 12th International Workshop on Agent-Oriented Software Engineering (AOSE 2011), Taipei, Taiwan*, Springer, 2012?

[47] Fabio Luigi Bellifemine et al., *Developing Multi-Agent Systems with JADE*, Wiley, 2009

[48] Fabio Bellifemine et al., 'Developing Multi-agent Systems with JADE', in *Proceedings of the Sevent International Workshop on Agent Theories, Architectures and Languages (ATAL 2000), Boston, Massachusetts, USA*, pages 89—103, Springer, 2001

[49] Fabio Bellifemine et al., 'JADE: A FIPA2000 Compliant Agent Development Environment', in *Proceedings of the Fifth International Conference on Autonomous Agents (Agents 2001), Montreal, Canada*, pages 216–217, ACM, 2001

[50] Fabio Bellifemine et al., 'JADE: A Software Framework for Developing Multi-Agent Applications. Lessons Learned', in *Information and Software Technology* (Vol. 50), pages 10–21, Elsevier, 2008

[51] Wikipedia contributors, 'Platform-independent model', *Wikipedia, The Free Encyclopedia*, `http://en.wikipedia.org/wiki/Platform-independent_model` (accessed April 1, 2012)

[52] Wikipedia contributors, 'Platform-specific model', *Wikipedia, The Free Encyclopedia*, `http://en.wikipedia.org/wiki/Platform-specific_model` (accessed April 1, 2012)

[53] Wikipedia contributors, 'Thespis', *Wikipedia, The Free Encyclopedia*, `http://en.wikipedia.org/wiki/Thespis` (accessed April 1, 2012)

[54] Wikipedia contributors, 'Type–token distinction', *Wikipedia, The Free Encyclopedia*, `http://en.wikipedia.org/wiki/Type-token\_distinction`

List of Figures

1.1 An agent interacting with its environment 4

3.1 The representation and conformance relationships [10] 12

3.2 Metalayers [10] 13

3.3 The *Core model* [12] 14

3.4 The *Aalaadin* metamodel [12] 16

3.5 The *O&P* metamodel [21] 18

3.6 *Agent Classifier* and its two specializations: *Agent Physical Classifier* and *Agent Role Classifier* [21] 19

3.7 Examples of physical classifiers forming a class hierarchy [21] . . . 19

3.8 Examples of role classifiers forming a class hierarchy [21] 20

3.9 The association between *Agent Physical Classifier* and *Agent Role Classifier* [21] 21

3.10 The association between *Agent* and *Agent Classifiers* [21] 22

3.11 The *Group* class and its associations [21] 22

3.12 The association between *Group* and *Role* [21] 23

3.13 An example of an agentified group [21] 23

3.14 An example of a non-agentified group [21] 24

3.15 The *Agent Role Assignment* class and its associations [21] 24

3.16 The *PIM4Agents* core model [23] 26

4.1 The Organization model 33

4.2 The Player model 35

4.3 The Protocol model 35

4.4 The Design-time model 37

4.5 The Run-time model 37

4.6 The *Thespian* static model 37

4.7 The *Enact role* protocol 39

4.8 The *Deact role* protocol 40

4.9 The *Subscribe to event* protocol 40

4.10 The *Publish event* protocol 41

4.11 The *Activate role* protocol 42

4.12 The *Deactivate role* protocol 43

4.13 The *Invoke competence* protocol 44

4.14 The *Invoke responsibility* protocol 45

6.1 Stage 1: Role enactment 64

6.2 Stage 2: Role activation 65

6.3 Stage 3: Competence and responsibility invocation 66

6.4 Stage 4: Role deactivation 67

6.5 Stage 5: Role deactment 67

6.6 Stage 1: Role enactment 71

6.7 Stage 2: Role activation 72

6.8 Stage 3: Competence and responsibility invocation 74

6.9 Stage 4: Role deactivation 76

6.10 Stage 5: Role deactment 77

6.11 Stage 3: Competence and responsibility invocation 79

Appendix A

CD-ROM Contents

The contents of the companion CD-ROM are as follows:

- `1-Thesis` — this thesis (in PDF and PS formats) and its zipped LaTeXsource code;
- `2-JavaSE` — Java Platform, Standard Edition (Java SE), Runtime Environment (JRE) 7 Update 3;
- `3-Jade` — Jade, version 4.1.1, and its API documentation;
- `4-Thespian4Jade` — Thespian4Jade and its API documentation;
- `5-Examples` — examples and their API documentation;
- `6-ExampleRuns` — interaction diagrams of the example runs (images in PNG format);
- `ContactDetails.txt` — the thesis author's contact details;
- `ReadMe.txt` — instructions on how to run the examples.