Component-based Software Architectures and Multi-Agent Systems: Mutual and Complementary Contributions for Supporting Software Development

Victor Noël

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par
l'Université Toulouse III - Paul Sabatier
Discipline ou spécialité :
Informatique

Présentée et soutenue par
Victor Noël
Le *July 18th 2012*

Titre :
Component-based Software Architectures and Multi-Agent Systems:
Mutual and Complementary Contributions
for Supporting Software Development

JURY
Mourad Chabane Oussalah – Professor, Université de Nantes, France (examiner)
Andrea Omicini – Professor, Università di Bologna, Italy (examiner)
Jean-Michel Bruel – Professor, Université de Toulouse, France (president)
Tom Holvoet – Professor, Katholieke Univesiteit Leuven, Belgium (member)
Marie-Pierre Gleizes – Professor, Université de Toulouse, France (supervisor)
Jean-Paul Arcangeli – Associate Professor, HDR, Université de Toulouse, France (co-supervisor)

Ecole doctorale : *Mathématiques Informatique Télécommunications (MITT)*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse (IRIT)*
Directeur(s) de Thèse : *Marie-Pierre Gleizes and Jean-Paul Arcangeli*
Rapporteurs : *Mourad Chabane Oussalah and Andrea Omicini*

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par
Université Toulouse III - Paul Sabatier
Discipline ou spécialité :
Informatique

Présentée et soutenue par
Victor Noël
Le *18 juillet 2012*

Titre :
Architectures logicielles à base de composants et systèmes multi-agents : contributions mutuelles et complémentaires pour supporter le développement logiciel

---

JURY
Mourad Chabane Oussalah – Professeur, Université de Nantes, France (rapporteur)
Andrea Omicini – Professeur, Università di Bologna, Italie (rapporteur)
Jean-Michel Bruel – Professeur, Université de Toulouse, France (président)
Tom Holvoet – Professeur, Katholieke Univesiteit Leuven, Belgique (membre)
Marie-Pierre Gleizes – Professeur, Université de Toulouse, France (directeur)
Jean-Paul Arcangeli – Maître de Conférences, HDR, Université de Toulouse, France (co-directeur)

Ecole doctorale : *Mathématiques Informatique Télécommunications (MITT)*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse (IRIT)*
Directeur(s) de Thèse : *Marie-Pierre Gleizes et Jean-Paul Arcangeli*
Rapporteurs : *Mourad Chabane Oussalah et Andrea Omicini*

Abstract

In this thesis, we explore the various aspects of the mutual and complementary contributions that multi-agent systems (MASs) and component-based software architectures (CBSAs) can provide to each other. In a way, this work is the study of how both worlds can be integrated together, either by supporting MAS implementation using component-based abstractions or by supporting CBSA construction and adaptation using self-adaptive MASs.

As a pragmatic starting point, we study how MAS development is currently done in the field and propose an understanding of the general methodology of development of MASs from an architecturally-oriented point of view. This results in the distinction between two main activities in MAS development. The first one, which we call macro-level design, is concerned with requirements and design choices tackled by multi-agent approaches as a way to decompose the solution in terms of agents and their interactions. The second one, which we call micro-level design, is concerned with requirements and design choices that accompany and more importantly support the result of the first activity to bridge the gap between design and implementation. From this conclusion, we infer that it is necessary to support this micro-level architectural activity with adequate abstractions that favour reuse, separation of concern and maintenance. In particular, an abstraction called "species of agent" is introduced for this purpose. It integrates with traditional component-oriented abstractions and acts both as an architectural abstraction and as an implementation. We define, illustrate, analyse and discuss a component model (SpeAD), an architectural description language (SpeADL) and a design method (SpEArAF) that ease and guide the description and the implementation of MASs using species of agents. This complete answer to the question of MAS development, which is supported by a tool (MAY) to exploit SpeADL with Java, has been applied to many applications in our research team.

Then, by setting back such a solution in the context of the CBSA field, we show how MASs differ and relate to traditional means of development in terms of structural abstractions.

To complete this study, we explore through various experiments how self-adaptive MASs can be used to support the building and the adaptation of CBSAs. Here, the agents and their continuous reorganisation act on one hand as the engine of the construction and of the dynamic adaptation of the architecture, and on the other hand as the runtime container that actually connects these elements together and maintains the architecture alive and working. This makes such an approach, even though it is exploratory and prototypal, a completely integrated solution to architecture building, execution and adaptation. This work opens several interesting research paths to build tools to support the development and the evolution of software architectures at design and at runtime.

Remerciements

Il paraît que l'écriture de cette partie du mémoire de thèse est celle où l'on doit se faire le plus plaisir. Or, ce passage sera certainement le plus lu de ces quelques 200 pages, c'est donc ici que la pression est la plus forte.

Je vais d'ailleurs commencer par remercier ceux qui sont à même d'émettre un avis sur l'intégralité de mon travail, c'est-à-dire chacun des membres de mon jury de thèse. Merci pour l'investissement que vous avez mis dans l'évaluation de mes écrits et de ma soutenance, je suis très fier d'avoir eu votre approbation et vos noms associés à ma production. Parmi ceux-ci, je remercie en particulier ma directrice de thèse, Marie-Pierre, pour m'avoir fait confiance tout au long de ces quatre années et pour avoir su gérer au mieux mon potentiel procrastinateur. Et puis bien sûr, il y a Jean-Paul, qui a, avant tout, eu la patience de me suivre sans faille dans mes délires et élucubrations. Merci pour l'investissement que tu as mis dans chacun de nos nombreux échanges, cela m'a à chaque fois permis de faire des bonds en avant dans mon travail de recherche.

D'une façon générale, je ne peux que remercier l'intégralité de l'équipe SMAC qui applique chaque jour dans la vie réelle les préceptes de coopération utilisés dans leur travaux de recherche. Une aide importante m'a été en particulier apportée par mon pool de cobayes de compét'. Sans vous je n'aurais jamais pu faire de MAY, et tout ce qui se cache derrière, ce qu'il est maintenant (mais préparez-vous, la v3 arrive, il va falloir tout reprendre de zéro ! !). De plus, toutes nos discussions autour du développement des SMA m'ont permis d'avoir un minimum de confiance dans la qualité et l'utilité de mon travail. Dans cette équipe, il est difficile de mettre en place une compartimentation entre collègues et amis, et ces mêmes personnes m'ont aussi grandement accompagné dans des activités de détente au labo et en dehors. La personne la plus méritante est certainement Jérémy avec qui je partage un bureau depuis plusieurs années et qui me supporte tel un moine zen. Et puis, sans ordre particulier, nous retrouvons Valérian, François, Luc, Arnaud, Noélie, Nicolas, Sylvain, Önder, Raja, Simon, Julien, Elsy, Zied et autres stagiaires de passage.

Puisque l'on parle de stagiaires, il est nécessaire de souligner le plaisir que j'ai eu à travailler avec Grégoire. Son travail, complété d'ailleurs par l'aide de Pierre, a énormément nourri le chapitre 7 de ce mémoire. Ce chapitre contient aussi le résultat d'une collaboration avec l'équipe AgentWise de KULeuven qui a eu la gentillesse de m'accueillir pendant 2 mois durant ma thèse. En particulier, ces résultats s'appuient sur les travaux de mon ami Mario qui m'a beaucoup apporté, autant sur le plan personnel que professionnel. L'accueil que

m'ont réservé Tom Holvoet et Danny Weyns m'a aussi été très cher et ces personnes ont nourri certaines des réflexions qui sont exposées dans ce mémoire. Mon passage à Leuven a marqué un tournant dans mon travail de recherche et a en particulier orienté mon attention en direction des architectures logicielles. Un simple merci aussi à Maroussia qui a eu la patience de passer un certain nombre d'après-midi en ma compagnie pour me guider dans l'implémentation d'un modèle SMA d'écoulement de l'eau :)

Maintenant, passons aux choses sérieuses, ou plutôt, à LA chose sérieuse : Christine ! Même si je pars, sachez que vous, votre pancake et le lambda-calcul resterez toujours dans mon cœur ! Plus sérieusement : Christine, Frédéric et Celia, merci pour l'amitié journalière que nous avons partagée durant nos repas et pauses thé. Et surtout, merci pour les modèles que vous avez été tout les trois pour moi autant dans l'enseignement que dans la participation à la vie de l'université. C'est grâce à vous que je me suis (un peu) impliqué dans la collectivité et que je peux me considérer comme ayant une utilité sociale.

Si je sors le nez du boulot, je peux maintenant me tourner vers les nombreux amis que j'ai rencontré à Toulouse. Je ne citerai personne, mais vous vous reconnaîtrez (je suis trop malin hein, comme ça je n'oublie personne ;). Merci pour votre support mais surtout pour m'avoir permis de faire évoluer ma personnalité pendant 4 ans : je pense qu'avec vous, j'ai autant gagné en expérience personnelle que ce que la thèse a pu m'apporter en expérience professionnelle.

Ma famille, que je n'ai pas beaucoup vu ces quatre dernières années : merci mes parents pour m'avoir fait suffisamment intelligent pour arriver jusque-là sans encombres, merci mes petite et grande sœurs pour m'avoir appris à craindre et à respecter les femmes (oui, cela n'a aucun rapport avec la thèse).

Et puis merci à tout ceux qui sont venus me soutenir lors de ma soutenance (je sais que vous n'êtes là que pour le pot !), amis et famille, ainsi qu'à ceux qui m'ont bien aidé pour l'organisation.

Et enfin, pour conclure, ma seconde famille, qui me supporte et me façonne jour après jour. Timomo, en plus de m'avoir fait rencontrer plein de gens supers, jusqu'ici tu m'as appris pas mal de trucs utiles sur la vie. Jojo, merci encore pour ta gentillesse qui cache une sagacité et une écoute dont j'ai pu apprécier les bienfaits ces derniers mois, et puis surtout pour ton débit de blagues (drôles) qui est le plus impressionnant jamais entendu ! Et puis il y a Tibtib, la seule personne au monde qui arrive à me suivre jusque dans mes raisonnements les plus tordus : merci pour tes cours de piano (et puis pour tout le reste).

Contents

List of Figures

List of Tables

Introduction to the Thesis

This thesis recounts the result of my Ph.D. studies on the synergy between Multi-Agent Systems (MASs) and Software Architectures. I spent this time in the SMAC (*Systèmes Multi-Agents Coopératifs*) research team of the University of Toulouse under the supervision of Marie-Pierre Gleizes and Jean-Paul Arcangeli.

In the SMAC team, researchers, among other things, investigate an approach to software engineering called Multi-Agent Systems. This approach and the way it is promoted by people here, under the name of Adaptive Multi-Agent Systems (AMAS), is particularly directed at modelling complex systems and solving complex problems. It is characterised by the building of self-adaptive systems whose global behaviour emerges from the behaviours and interactions of the agents that compose it. In exchange of this "collective intelligence", it becomes difficult to prove the correctness of such systems and the best way of verifying that they work is through experimental evaluations. It thus becomes important to guarantee the quality of the development process that is followed to produce such software in order to have meaningful evaluations. In this context, I participated in a topic of research focused on producing tools, models, methods and development supports useful to support the design and the implementation of such systems.

As far as software engineering is concerned, one important mean of succeeding in the production of a software system is by using software architectures. Software architectures promote a set of practices for organising the development and producing the documentation of software systems from design to deployment, but more pragmatically, they are also instantiated with Component-Based Software Architectures (CBSAs), an approach that tackles the question of considering software architectures only as a composition of software components. This is why the initial objective of this Ph.D. was to contribute to the field of MAS development in order to support the transition between design and implementation by exploiting software architectures. And the more I progressed in my research, and the more I became convinced of the importance of using such a complete approach for design and implementation in the context of MASs.

Inversely, being myself before and everything an advocate of traditional and provable software engineering techniques, the more I participated in and studied the building of self-adaptive MASs, the more I became convinced that such an approach will be a key element of the future of software computing. This gave me even more incentive to improve the quality, productivity and maintainability of the development of MAS-based products. But

more importantly, this brought me to investigate how it is possible to build a conceptual bridge between MASs and software architectures, and how MASs can support the design, the adaptation and the evolution of software architectures.

Thus, the ultimate objective of this work is to see how these two ways of making software can complement and mutually support each other. My desire is to be able to integrate the two by providing an understanding of what links them together and by providing abstractions that make it possible in practice.

This thesis is organised into two parts, each tackling an aspect of the synergy between MASs and software architectures, namely the exploitation of software architectures to support MASs development and the integration of MASs and software architectures. Before presenting the different chapters of these two parts, I present the main motivations and the main contributions of this thesis.

Main Motivations

The first motivation for this thesis is to ease the development of MASs by improving the quality and productivity of the software process. The idea being that as MAS design is a complex activity, people that practice it should have their life simplified as much as possible in order for them to focus on their high-level concerns without bothering with low-level and technical problems. This separation of concerns is important so that MAS development can be more productive but also to obtain an organisation of the development that can be of a better quality.

Strongly linked to that point, the second motivation for this work is to improve the quality and the maintainability of MASs themselves as software products. Indeed, in order to be able to validate correctly the results obtained with MASs, it is necessary to be certain that their implementation actually realises what their design describes. The maintenance and evolution of such software products must also be eased.

A third and last motivation for this thesis is more theoretical and exploratory and is concerned with the understanding of the links between MASs and software architectures, with a focus on Component-Based Software Architectures. On top of the fact it appeared as an interesting subject of discussions during my researches, a farther reached incentive is to study how self-adaptive MASs can provide more adaptivity to software architecture, in their execution but also in their development.

Main Contributions

An analysis of current practices in MAS development pointed up a particularity in the general methodology of MAS development. For MAS, development mainly relies on development supports that are not adapted to the need of the people that build MASs. They are not adapted because every application of MASs has its own specificities, which are not

adequately answered by the tools available. There is thus a need to build an architecture dedicated to the development.

Then, an answer to this need is given in the form of a component-based architectural approach to MAS development. This approach emphasises on the different roles in MAS development and gives a focus on software quality, productivity, reusability, maintainability and other properties of software architecture practices. In particular, I propose a component model that can be used to support the implementation of MASs in practice by following this approach. This model is supported by an iterative and incremental development process and a tool for describing the architectures and implementing them in Java.

Finally, with this contribution as a starting point, I come back to works in the field of software architectures in order to establish our vision of the relations that exist between architectural concepts and MASs. On one hand, I show how our component model can be exploited to use MASs side by side with CBSAs, and on the other hand, I also show how MASs can be used to make software architectures more adaptive.

Plan of the Thesis

Chapter 1 introduces the scientific context both parts of this thesis rely on. It contains an understanding of software engineering, software architectures and MASs that I shaped during my research. It also serves as an introduction to the motivation for the rest of the thesis.

Then this thesis is divided in two parts.

Software Architectures for Multi-Agent Systems. The hypothesis I make in this part of the thesis is that all the matters that are tackled in the field of software architectures also exist while developing MAS-based applications and thus, must be dealt with. In this context, the motivation of this work is to help the development and find ways to improve the quality, maintainability, reusability of produced software. The content of this part is in the continuation of previous works of the SMAC research team (Leriche 2006; Leriche and Arcangeli 2010).

Chapter 2 looks at existing means of development in the MAS field. It analyses them, extracts and identifies shortcomings of current practices by proposing a classification of research works in the MAS field. It proposes a set of methodological and technical challenges to overcome in order to improve MAS development. This chapter is also the result of studying MASs produced in the SMAC team and participating in its research work.

Chapter 3 presents a coherent set of solutions to the previously identified challenges:

- A general methodology of MAS development
- A component model with abstractions adapted to the technical challenges
- A development method to exploit in practice the component model within the scope of the methodology

– A tool that implements the component model

This chapter is the result of applying the various intermediate versions of the contribution in the SMAC team and in the UPETEC [1] start-up company, which is a company that applies the research results of the team for the industry.

Chapter 4 applies the contribution to a real MAS developed by researchers of the SMAC team in the context of a research project. It concludes with some remarks on what would have happened if the contribution had not been used.

Chapter 5 concludes this part by presenting an analysis of the contribution. It positions it in the classification defined Chapter 2 and comes back on the identified challenges to highlight the advantages of its use. It also presents all the academic and industrial works that exploited the contribution and accompanies them with experimental feedbacks from their authors.

Integrating Multi-Agent Systems and Software Architectures. In this part, I am analysing the proposed contribution from the point of view of the software component and software architecture field. In this context the motivation of this work is to see how MASs relate to concepts manipulated in the software architecture field and what role they can play in their dynamic adaptation and construction, in particular for Component-Based Software Architectures (CBSAs). This part is mostly exploratory and contains a lot of ideas I think worth spreading even though they may lake maturity for being published as such. I consider this part as a research agenda from which different exploration paths can be deepened.

Chapter 6 starts from the contribution and positions it with respect to the main concepts used for design in the CBSA field. It shows how the contribution can be re-understood in order to define and build dedicated component-based component models and containers. This actually gives me the possibility to reconsider MAS design and programming into this broader frame. I conclude by advocating for the use side to side of MASs and CBSAs as a mean to design software using adequate abstractions.

Chapter 7 takes a more dynamic and runtime stance. It first justifies the use of MASs as the engine of adaptation of software systems. Then, it proposes, through the study of two examples of self-adapting software architectures, an analysis of the role that MASs plays in it.

One of these examples is the result of a collaboration with the AgentWise research team, which I present next section, on the adaptation of composite web-services. The second example is the result of a Research Master internship on the opportunistic self-composition of components in ambient context using the AMAS approach we promote in the team.

As a conclusion, I advocate for an integrated and self-organising design, construction and runtime adaptation of software architectures. I present my vision of a MAS-based tool that helps to organise the development of software architectures, that supports design decision making and software evolution, and that makes the bridge with the running system.

1. UPETEC (Emergence Technology for Unsolved Problems): http://www.upetec.fr/

Scientific Results

During my Ph.D., I had the opportunity to produce various scientific results and to participate in academic collaborations that are summarised here.

Publications and Collaborations. Since my research subject is in-between two different fields of computer science, namely MASs and Software Architectures, I had the opportunity to present my works to both communities.

In the beginning of my Ph.D., I published a poster in a French-speaking annual event on software engineering (Noël et al. 2009) to present the objectives of my work. In parallel, the results of my research Master thesis, that has no link with this thesis, were presented at LNMPR (*"Logic Programming and Nonmonotonic Reasoning"*), an international conference on logic programming (Noël and Kakas 2009).

I presented the first version of the contribution of this thesis at AT2AI (*"From Agent Theory to Agent Implementation"*), an international symposium dedicated to the development of agent-based systems (Noël, Arcangeli, and Gleizes 2010a). It evolved and was then presented at EUMAS (*"European Workshop on Multi-Agent Systems"*), the main European workshop on MASs (Noël, Arcangeli, and Gleizes 2010b).

Then I presented a more mature version of the contribution at CAL (*"Conférence francophone sur les Architectures Logicielles"*), the main French-speaking conference on software architectures (Noël and Arcangeli 2011). An extended and improved version of the article, and thus the most mature published results of my thesis, appeared in RNTI (*"Revue des Nouvelles Technologies de l'Information"*), a French journal (Noël, Arcangeli, and Gleizes 2012).

During my Ph.D., I visited the AgentWise research team. AgentWise, headed by Tom Holvoet, is part of the DistriNet research group of the Computer Science Department of the University of Leuven in Belgium.

I spent two months with them to work with Mario Henrique Cruz Torres, a Ph.D. there. The result of this work is detailed Chapter 7 and was presented at EUMAS (Cruz Torres et al. 2010a), then at MONA+ (*"International Workshop Series on Monitoring, Adaptation and Beyond"*), an International workshop of the web-service community (Cruz Torres et al. 2010b). I also interacted with Tom Holvoet and Danny Weyns, two leading European figures on the matter of, among other things, the relations between MASs and software architectures.

I participated in the regional research project ROSACE[2]. I first worked on adaptive communication where I took part in the redaction of a book chapter (Lacouture, Rodriguez, et al. 2011). Then I worked on cooperative self-organisation where we used my contribution to develop a prototype of self-organising robots that self-allocate tasks (Georgé et al. 2010; Lacouture, Noël, et al. 2011).

2. ROSACE (Robots and Embedded Self-Adaptive Communicating Systems): http://www.irit.fr/Rosace, 737, funded by the RTRA STAE (Fondation de coopération scientifique, Sciences et Technologies pour l'Aéronautique et l'Espace)

I also participated in the research project AmIE [3] on ambient intelligence and systems. In this context, Jean-Paul Arcangeli and I supervised the internship of a Master student that was presented at UBIMOB (*"Journées Francophones Mobilité et Ubiquité"*), a French-speaking annual event on ubiquity and mobility (Denis et al. 2012). The result of this work is detailed Chapter 7.

Software. The contribution, in particular the component model I propose in this thesis, has been implemented and is presented Appendix B. This tool is named MAY (Make Agents Yourself) and is released under the GNU General Public License (GPL). It is incorporated in the Eclipse IDE (Integrated Development Environment) and provides an editor to describe components and architectures conform to the component model. It generate Java code usable to implement the components in a flexible and typesafe way. It comes together with a library of reusable components released under the GNU Lesser General Public License (LGPL). The website of MAY [4] contains tutorials for installing the tool and using the component library.

Teaching. In my last year of Ph.D., I had the opportunity to present the knowledge I gathered on the design and documentation of software architectures to Master students. It resulted in a 2 hours lecture and a 4 hours tutorial for 60 students. Moreover, MAY, the implementation of the component model, was used to conduct 8 hours of practical tutorials to introduce component-based programming to the same students. The teaching material, in French, that was given to the students can be found on my personal homepage [5].

Editorial Notice

Even though this thesis is the result of my own work, some parts of it could not have been done without the help of a lot of people. Thus, in the rest of my thesis, even though I take responsibility for everything written in it, I switch to a "royal we" to not have to constantly choose between pronouns, which eases the flow of the discourse.

In order to ease the reading, every chapter ends with a summary of my contributions.

3. AmIE (Ambient Intelligent Entities): http://www.irit.fr/AmIE,1003, funded by a scientific program of the University of Toulouse III (Université Paul Sabatier)
4. http://www.irit.fr/MAY
5. http://www.irit.fr/~Victor.Noel/

CHAPTER 1

Engineering Software: Software Architectures and Multi-Agent Systems

> And (repeat after me) we all promise to stop using the phrase "detailed design". Try "nonarchitectural design" instead.
>
> Paul Clements
> Clements et al. (2003)

In this chapter, we introduce background concepts and definitions that are employed in the thesis. The objective is to get a clear picture of the context into which our contribution fits, but also to present works that influenced it.

It starts with a part on software engineering in general as we understand it. Then, we develop the different aspects of the field of software architecture as a mean to do software engineering. Finally, we present Multi-Agent Systems (MASs), also as a mean to produce software.

Software architectures and MASs are two ways of producing software but each tackle different aspects of it. In this sense they are non-competing and non-conflicting. One of the incentives for this thesis is to study how they can be combined together and what advantages they can bring to each other. More particularly, the motivations are, for the first part of this thesis, how software architectures can support the development of MASs and, for the second part, what are the links between MASs and software architectures and how MASs can help to make software architectures more "intelligent". In both parts, the focus is put on how practically these questions can be answered using adequate abstractions for design and implementation.

1.1 Software Engineering

In the field of software engineering, the objective of development could be summarised as producing a software solution to a problem. This definition contains three important concepts: the problem, the solution and the process. We now detail these concepts and what evolve around them during software development.

This section mostly contains definitions that are important from our point of view and, even though they can be considered as subjective, they only act as a set of concepts needed to share a common language for the rest of this thesis, without being axioms or hypothesis.

1.1.1 General Terms

The development starts with a **problem** to solve. There exists several ways of expressing the problem, we will see the most well-known one Section 1.2. The **solution** is the result of answering the problem. A software solution, that we also sometimes call the application or the system, is the main artefact that results from the development. In practice, a software solution is something that, when it is executed, solves the problem it was made for. Ultimately, it takes the form of a source code implementation and is deployed to be used.

An **artefact** is anything that is produced during the development. The solution is a software artefact, but other artefacts are produced during the development, such as documentation of the application or tests. But more importantly, the development itself produces a lot of different design artefacts that describe intermediary solutions to the problem. Indeed, in order to build the solution, the software developers follow, as we are going to see, a **process** that helps to answer the problem by producing these intermediate design artefacts.

1.1.2 Modelling the Solution

In any way, when doing software development, the design artefacts represent the complete solution, or parts of it, using high-level models that are bit by bit refined.

A **model**, for example as defined by Bézivin and Gerbé (2001), is a simplified representation of a system. The system modelled is called the **subject** of study. The model is always made with a specific objective in mind, for example in our case, a model of the system is meant to help its building. Most of the time, the manipulated model abstracts over some of the elements of the solution in order to ease its manipulation and to focus on what we are interested in.

There exists a reflexive relation between models, some of them are called **meta-models** and are meant to describe how to model models that conform to them. They are also sometimes considered as modelling languages. For example, the most known meta-model for building software is the Unified Modeling Language (UML). In the next chapter, the models studied are all meta-models for modelling MASs.

A **semantics** is often associated to a meta-model to describe the meaning of the different types of elements that can be modelled with it. For example, we can talk about the semantics of the concepts provided by a meta-model.

The term **design** is often used both as the process of producing these models of the solution and as the result — *i.e.* an artefact — of this phase of the development. Software development is mainly about producing a design answering the problem, and then about implementing this design. The design potentially undergoes successive refinements, and the implementation is often the translation — for example by programmers, or using model transformation — of the most detailed design into code. This implementation is expressed in terms of the programming abstractions made available to the implementers by the chosen development support.

A **development support** is something that is used to program the solution. Development supports propose models of programming, often called **programming paradigms**, *i.e.* a coherent set of concepts on which the developers can rely on when programming (Van Roy 2009). Such concepts are also called **programming abstractions**. In our opinion — which follows the idea that everything is model — what fundamentally distinguishes models proposed by development supports from models used for design is that the former provide an implementation for the semantics of their model in order to render their instantiation executable. Such implementation of their semantics can be considered as an **abstract machine** that can execute the program expressed using the model (a programming language for example).

1.1.3 Methods and Methodologies

In order to help the design or the implementation, methods are used. A **method** is often seen as the combination of meta-models (to represent the solution) and a complete process, which is a succession of activities that guides the designers towards the solution (Arlow and Neustadt 2005). Methods cover different aspects of the development, among them common ones are: analysis, design, implementation, testing, evolution... Some methods only cover design, other go up to the implementation, some cover analysis or evolution, etc. Methods provide guidelines that help the developer to take decisions in order to find a solution to the problem, or well organise the development.

A term often used in the software engineering field along with the one of method is methodology. As the American Heritage Dictionary of the English Language explains:

> Methodology can properly refer to the theoretical analysis of the methods appropriate to a field of study or to the body of methods and principles particular to a branch of knowledge. [...] In recent years, however, *methodology* has been increasingly used as a pretentious substitute for *method* in scientific and technical contexts, as in *The oil company has not yet decided on a methodology for restoring the beaches*. [...] the misuse of *methodology* obscures an important conceptual distinction between the tools of scientific investigation (properly *methods*) and the principles that determine how such tools are deployed and interpreted.

In this thesis we try to use the terms properly, and more specifically we consider a **methodology** as the set of practices and principles pertaining to a specific field, but also as something that is instantiated by methods of this field. For example Chapter 2, some methods and their models are studied, and Chapter 3 presents a general methodology of MAS development, which is then partially instantiated into a method.

1.1.4 Reuse of Development Artefacts and Tools

Software development is then concerned with producing software and design artefacts that conform to some meta-models by following methods and process, and using development supports. During the development, artefacts can not only be produced, but also used after they were produced in past developments. This **reuse** is often of great importance.

In order to be reusable, an artefact has to be generic in some way: the reusability of an artefact thus ranges from **specific** to **generic**. The concept of generality is actually relative to the point of view: indeed, something that is generic in a context can be specific to the same context in a wider context. In order to denote this kind of context, we talk about domains. A **domain** is a class of problems, applications, etc, that have common characteristics and thus, that can have generic solutions reusable between developments. Also, sometimes, reusable artefacts are not specific to a domain, but to the expertise of a class of developers.

We can then talk about **domain-specific** artefacts, but also about domain-specific methods or development supports. Dedicated is another word for specific, such as "dedicated development support" that denotes a development support dedicated to a specific task or domain. Sometimes artefacts or tools are also said to be specific to a method, a model, a development support, etc.

In particular, development supports are actually most often specific to a domain. We can of course find general programming languages such as the famously known Java, C or Scala. But lately Domain Specific Languages (DSLs) have been put in the light as a mean to ease the development by providing programming abstractions adapted to a specific need (Ghosh 2010). Frameworks are also a good example of domain-specific development support. **Frameworks** are "application generators" that are directly related to a specific domain (Johnson 1997; Markiewicz and Lucena 2001). Their points of flexibility are called **hotspots**. The developer implements them to specify application-specific logic that will be executed by the **frozenspots**, which are the parts of the framework already implemented.

Some methods are made so that use and reuse are both taken into account during the development. Actually, the problem of reusing abstract experiences for specific problems or elements of solution in the form of code artefacts composable together is one of the objectives of software architectures that we now present.

1.2 Software Architectures

As we said, software engineering is mainly about designing a solution that answers an expressed problem. In the software architecture field, the idea is to design the solution as a

structure made of elements and relationships.

A good introduction to the problem of software architecture can be found in the work of Bass, Clements, and Kazman (2003) for example, where they precisely define a **software architecture** of a program or computing system as:

> [...] the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

As we are going to see in this section, elements and relationships between them can be of different natures depending on which part of the architecture has to be expressed and for which purpose. The first chapter of the thesis of Fielding (2000) contains a quite complete and clear survey of the different views and opinions on the matter. It explains the terms used in the field, and presents the different problems that are tackled by works of the domain.

As a general introduction, the software architectures field is mainly motivated by:

- The desire of succeeding in the construction of a software system by making it as close as possible to what it was made for under the constraints of its environment.
- The desire of being able to organise its building, maintain it after production, make it evolve over time when needs and constraints change.

The two founding works on the matter, which each proposes its definition for software architectures, are those of Perry and Wolf (1992) and Garlan and Shaw (1993). Perry and Wolf (1992) qualify the difference between the two by arguing that their work is a properties-based approach, *i.e.* with a focus on characterising the properties of architectures and means to study them, while Garlan and Shaw (1993) work is a type-based approach, *i.e.* with a focus on characterising elements of architectures and means to compose them. Fielding (2000) explains this difference by saying that Perry and Wolf (1992) approach is focused on the architecture as a runtime system, while Garlan and Shaw (1993) approach is focused on architecture description as a mean to model the system and its elements.

These two works are the main origin of what we present now. We look at software architectures from two different points of view: compositional and methodological. The first is about Component-Based Software Architectures, which are practical ways of easing the composition of software, while the second is about means to design software architectures.

1.2.1 Component-Based Software Architectures

A great part of the software architecture field focuses on Component-Based Software Architecture (CBSA). These architectures are composed of elements which can be **components** or **connectors**. Components are the units of computation while connectors are the unit of communication between components. In the following we elude connectors as our work did not rely on this concept: we considered, by oversimplification, that components can replace them in most of the places we would have needed them.

Components are defined by Szyperski, Gruntz, and Murer (2002) as reusable implementations subject to third-party composition with contractually defined required and provided

interfaces, which are the points of interoperability when composing software components together. Component-oriented programming proposes to directly implement component-based architecture using architecture-oriented programming abstractions.

Software components are often defined as conforming to a component model. A **component model**, as defined in a famous survey (Crnković et al. 2011), "defines standards for 1) properties that individual components must satisfy and 2) methods for composing components." Among other, it considers models where components are architectural units as in the definition of Szyperski, Gruntz, and Murer, and these are those that interest us here.

In particular, in this category of component models, **Architecture Description Languages (ADLs)** are means to describe more or less formally a component-based architecture in order to analyse, document or implement it. Some of them are only interested in static properties, others introduce dynamism in the description, and other even include behavioural descriptions of the elements. Medvidovic and Taylor (2000) is a famous survey of existing ADLs.

Components are executed at runtime in **component containers** (Crnković et al. 2011) that enforce the constraints they must respect as well as give them access to their environment and other components. Component containers are also called component frameworks or component platforms. In this thesis, at least in the first part, we prefer to consider frameworks in a more general way as defined previously and keep the term component container for the software that executes the components.

Our own understanding of the relation between these different concepts is as follow. Components and compositions of components can be described with ADLs. Such descriptions are conform to a component model. Once implemented, these descriptions of components can be instantiated at runtime into a component container that takes care of implementing the semantics of the component model. Other relations exist, but they are the subject of Chapter 6 and are not needed before.

To conclude, we present the definition that Bachmann et al. (2000) propose for what they call **architectural components**. They are at the same time architectural abstractions that support design, and implementation abstractions that support reuse and composition. They are "an opaque implementation of a functionality, subject to third-party composition and conformant (*sic*) with a component model."

This shows that components are also a mean to help the design of a software architecture. In that context, components and connectors are not the only means available to define software architectures. The research field built on top of these concepts to more broadly define software architecture practices.

1.2.2 Design of Software Architecture

In order to understand better what architectures are meant to be, we first propose to see why they are built. The primary motivation behind the building of software architecture, even before the problem itself, is what are called the **stakeholders**. They are the people that have any interest in the system to be built: they can be the clients, the users, but also the

developers, the project manager, the shareholders providing money, and even the architect himself. The stakeholders are the ones that express the requirements that drive the building of the software architecture.

Requirements describe what the solution must do and under which constraints, but not how it is actually realised. Requirement engineering is the field concerned with analysis of problems and requirements extractions. Often requirements are said to be either functional, *i.e.* that pertain to the functionality that the system must realise, or non functional. The latter concerns system properties (such as performance, distribution, security, etc.) as well as development properties (costs, organisation, maintainability, reusability, etc.). Of course, requirements can have interdependencies between each other, which complicates their handling.

An architecture answering requirements is said to have the corresponding **quality attributes**. Requirements are strongly linked to architecture and works show well the relationships existing between the two. The way requirements and architectures are linked to each other are well showed in works such as Nuseibeh (2001) that presents the **"Twin Peaks"** model where requirements and architecture can be incrementally refined in parallel.

This helps to understand how architecture and design are related to each other: as Paul Clements says in Clements et al. (2003), architecture is design, but all design is not architecture. In particular, from the point of view of an architect:

> [...] architectural decisions are ones that permit a system to meet its quality attribute and behavioural requirements. All other decisions are non-architectural.

In other words, **architectural decisions** are all the design decisions that answer the requirements. And thus, the architecture can also be defined as the result of taking a given set of architectural decisions. Obviously, this is relative to the point of view taken on the designed element: for the developer of a subsystem of an architecture, its design can be architectural with respect to the requirements of hos element and not of the whole system.

Another motivation for software architectures is that they help to take into account as early as possible these requirements, and that producing a model of an architecture eases its maintenance and evolution. Indeed, as advocated by Clements et al. (2003) designing an architecture cannot be dissociated from producing the documentation that describes this design, records the decisions taken, their rational and the requirements they answer. This helps to build the system, but also to register information on its structure in order to, for example, analyse the impacts of future evolution, allocate tasks to developers or plan costs.

Clements et al. (2003) see the produced documentation as a set of **views** on the architecture of a system that each concerns a part of the system or a point of view on it. Each view is expressed in terms of elements and their relationships, but different types of views exist each with a different meaning. In that work, the three main identified types of view are:

- **Component and connector views**: they describe the structure of the system in terms of runtime entities and their connections.
- **Module views**: they describe the structure of the system in terms of implementation units and their relations, dependencies, etc.

– **Allocation views**: they describe the structure of the system in terms of the links between what is software and what is not, such as hardware, but also teams of developers or deployment units

There exist methods to help to take "good" architectural decisions such as ADD (Attribute-Driven Design) (Bass, Clements, and Kazman 2003) that, roughly, proposes to answer requirements by order of importance, and to design and refine iteratively the architecture until no more requirement have to be answered. Clements et al. (2003) distinguish two types of **refinements**: implementation and decomposition. The first one is about replacing sets of elements of the design by different ones, while the second one is about decomposing elements by zooming in.

Such method relies on other means to help building an architecture that people found to capture their experience for specific domains, requirements, problems, etc. Well-known examples of this are architectural styles, reference architectures, pattern languages or Software Product Lines.

1.2.3 Capturing Experience

As the most famous way of capturing experience in software architectures, **architectural styles** (Abowd, Allen, and Garlan 1993) organise the architecture in a particular way in order for it to have some wanted qualities. Roughly, an architectural style is a way to constrain the elements of an architecture in order to get in exchange some properties and to answer some types of requirements. Architectural styles are formalised in the context of component-based architectures with ADLs. But it is also considered as a mean to drive the documentation of an architecture. For example, in the work of Clements et al. (2003), all types of views are seen as instances of architectural styles. For example, service-oriented architectures, that has become famous lately, are one style of architecture where dynamics and flexibility are emphasised.

Less abstracts, **architectural patterns** provide best practices for a specific family of applications, *i.e.* for a specific domain. Patterns are well-known in the object-oriented field (Gamma et al. 1995) but we are more interested in architectures, and in particular component-oriented architectures, even though patterns can be used for any type of view as with styles.

Here, we consider patterns as more specialised than styles (Monroe et al. 1997): a pattern is more a reusable known composition of a set of elements, while a style only constrains the types of elements and how they can be connected to each other. Architectural patterns and styles can also be differentiated by following the locality criterion (Eden and Kazman 2003): styles are non-local and patterns are local, *i.e.* styles concern the whole architecture while patterns concern parts of the architecture.

There exist works that link up patterns, styles, ADLs, component model, frameworks, etc, but this is going to be the subject of Chapter 6.

Other works focus on applying the knowledge coming from the field of Software Product Lines (SPLs) (Clements and Northrop 2001). Also called domain and application engineering, the objective of product lines engineering is to identify commonalities and variabilities in lines of products in order to reuse as much software artefacts as possible when building an

application. By building an architecture reusable across the domain, one can then build on top of it application-specific architectures that reuse and extends the domain architecture. In the same way, **reference architectures** (Reed 2002) and **pattern languages** (Kerth and Cunningham 1997) regroup sets of architectural patterns.

As we are going to see in the next section, a specific family of architectures — *i.e.* a set of architectures that conform to an architectural style, or more broadly here, to a family of architectural styles — interests us particularly here: Multi-Agent Systems.

1.3 Multi-Agent Systems

As we just said, MASs can be considered as a family of architectures with its own properties and advantages, as it is well explained by Weyns et al. (2004). Research on MASs took a different path than research on software architecture and the two communities are very distinct. Some works tried to reconcile the two, such as the book of Weyns (2010) that covers very well the whole matter of MASs from a software architecture point of view. As it advocates, MASs are used to organise an architecture from a component and connector point of view in terms of agents and their interactions. This thesis also takes a shot at the question and we come back on this last point in the following chapter as well as in the second part.

For now we focus on the point of view of the MAS community that mainly sees MASs as a way to solve problems. In this sense, they are situated at the level of the design in software engineering as presented in the first section of this chapter. We provide here some general definitions and we show the outline of what interests us in the field. The next chapter contains a thorough state of the art on MAS development and implementation.

1.3.1 Agents

There is not one admitted definition for agents and MASs, but some common characteristics can be found in the literature. A commonly used definition for an agent is the following, taken from Ferber (1995):

> An agent is an autonomous physical or virtual entity able to act (or communicate) in a given environment given local perceptions and partial knowledge. An agent behaves in order to reach a local objective given its local competences.

Thus, designing agent-based systems means to build systems made of these entities that interact together using diverse interaction means. What differentiates agent-based systems from MASs, that we present next, is that the latter is founded on methodological concerns that aim at giving the system a behaviour through the definition of the agents' behaviour. Agent-based systems are mostly focuses on the structure of the systems in terms of its elements, the agents, the available means of interaction they can use and possibly the dynamics that drive their internal execution.

1.3.2 Multi-Agent Systems

A MAS is an abstract entity that only exists through its agents, their organisation and their local interactions. Indeed, by interacting together, agents form an organisation, and this organisation is what makes the system to exhibit a global functionality. By changing the organisation of the agents, the system changes the way it realises the functionality (Camps, Gleizes, and Glize 1998). While this can actually characterise any system made of elements connected together, for example a software architecture, in MAS the focus is put on the re-organisation in an autonomous way by the agents. Inversely, more traditional approaches decompose the problem into independents sub-problems and thus address the question in a reductionist way. Most of the time this implies static connections between the elements.

A few words can be dedicated to the terms reductionismand emergence that are going to be used in this thesis, even though we don't really relies on their meaning. Roughly, what we mean by that is that a reductionist decomposition of a problem or a solution is a decomposition in independent sub-elements, which when taken altogether gives the possibility to understand the whole of their composition. On the other hand, emergence characterises the fact that from the elements of a composition, some phenomena can be observed that is not understandable directly from the sum of the elements of the composition. Obviously, the subject is much more complex to that, and in particular these terms are not really opposite, but we think this is enough for understanding the discussions that use these terms.

As Demazeau (1995) says, in MASs, the "operational part of how the solution [of the problem] is found" is taken in charge by the (self-)organisation of the agents of the system and not by the designer of the system. This is what is sometimes called the collective intelligence (Camps et al. 1994; Ferber 1995) or the collective behaviour (Demazeau 1995) of the system.

Organisation is possible through means of interaction that the agents use as well as the environment in which they interact. An important aspect of the interactions is that they are mostly locals. Locality can express itself in terms of social relations between agents, virtual spatial abstractions or symbolic proximity, depending on the problems and on the way chosen to solve it. Thus, a focus has been put in the community on interactions first, and then lately on environments. As we see in the next chapter, there exists a quantity of proposed models to describe a solution expressed using MASs.

The hard part when designing a MAS is to find what is going to be an agent in the system, and what is going to be their behaviours. In a way this is about finding the adequate decentralized coordination algorithms for the agents to organise in an adequate way to solve the problem. This confers adaptability to the system in the sense that the functionality of the system is not decomposed in distinct sub-problems in a reductionist fashion. Instead, the functionality of the system emerges from the interaction of the agents and failure in parts of the system can be absorbed by re-organisation (Capera et al. 2003).

Thus, combined with the models, different approaches exist to design such system. An **approach** is a way to help to design MASs, possibly backed up by a formal or informal theory. Approaches are sometimes accompanied by development methods. For example, we present

next the AMAS approach, which is backed-up by the ADELFE (*Atelier de Développement de Logiciels à Fonctionnalité Emergente*) (Bernon, Camps, et al. 2005) method. ADELFE is also presented and used Chapter 3 to illustrate an aspect of the contribution.

These approaches propose specific ways to design the system by helping to find the right behaviours for the agents. Each of these approaches and methods have their pros and cons. Some of them target specific domains, some other don't.

Then, the question of implementing MASs has had an important coverage by the community. The following chapter is dedicated to this question, we thus don't detail it here.

1.3.3 AMAS: Adaptive Multi-Agent Systems

We briefly present the **AMAS (Adaptive Multi-Agent System)** approach (Gleizes et al. 2008; Georgé, Gleizes, and Camps 2011) to MASs as it is one of the main design approach that we used when applying the contribution of this thesis. In this approach, often referred to as self-organizing MASs, the sole purpose of the agents, who have very local behaviours, is to participate in a self-organizing system from which the collective intelligence emerges. The main engine of the adaptation in this approach is based on a theory of cooperation: this relies on the idea that a system that acts cooperatively, with respect to what it has to do, is adequately functional. In terms of design, it means to identify situations where the system is not likely to be functioning properly (and thus considered not being in a cooperative state) that are called Non-Cooperative Situations (NCSs). When agents find themselves in these NCSs, they try to fix them in order for the system to stabilise in a cooperative state, which corresponds to a state where the system is adequately performing its function.

There exist several mechanisms to describe and handle cooperation used in the AMAS:

- First, the agents have two aspects: the nominal behaviour and the cooperative behaviour. The former describe what the agent do when the system is functioning properly, *i.e.* when the system is in a cooperative state and stabilised. The latter describe how the agents can be cooperative in terms of anticipation or correction of NCS.

- Then, in order to represent diverse degree of non-cooperation, agents use the concept of criticality. The criticality represents how important is a request an agent sends. The difficulty is thus, for the problem domain, to find the best way of representing this criticality. Indeed, agents always favour requests the more critical requests over the others, and thus it is the main way to control the direction into which the system goes.

- Agents communicate with each other and thus have what is called a neighbourhood: the set of agents they communicate with. In a way, this neighbourhood represents the links that exist between the agents, and thus the organisation of the system. When the agents reorganise the system, they actually change the sets of their neighbours. And when they only apply their nominal behaviour, they do so with respect with their current neighbourhood.

By giving agents the capacity to adapt their behaviour and their organisation — always favouring solutions that are the most cooperative—, the system self-organises. Within this approach, the system can self-adapt at three different levels:

- Internal adaptation of the agents
- System reorganisation (modification of the links between agents)
- System evolution (creation and destruction of agents)

All of these adaptations are always triggered by the agents themselves based on their behaviour.

The approach is used Chapter 4 in an application used to illustrate the contribution, but an example of how this really work and how functionality can emerge from the local interactions of the agents is detailed Chapter 7.

1.3.4 Scope

The kind of systems targeted in this thesis are mostly situated (but not restricted to) in the Agent-Based Modelling (ABM) or the Multi-Agent-Based Simulation (MABS) fields. In these fields the focus is really put in MASs as a mean to model a problem or its solution in terms of agents, their interactions and their behaviours. It is said that the entities of the domain of the problem are agentified. This is to be put in opposition to other fields where agents are mainly a way to design elements of the system in terms of intelligent entities that interacts together without a focus on the aforementioned collective intelligence.

1.4 Conclusion

As we see in this chapter, MASs and software architectures each tackles different aspects of software development. In all these works, our particular interests are the means and abstractions that are available to ease the implementation of systems. In software architectures, these are Component-Based Software Architectures (CBSAs) and component models that provide abstractions that act as a bridge between a software architecture in terms of design and concrete implementation of the design. In MASs, these are development supports that provide abstractions that act as a bridge between a MAS design and a concrete implementation of it. We identified two different axis of relationships between them as highlighted in the introduction of this thesis.

First, while software architectures are mainly about organising the development of a complete system in a realistic way, MASs focus on finding a way to solve the problem itself. As we are going to see in the next chapter, MASs have specificities that make them difficult to implement, mainly because of the gap that exists between the models of design and those of the available development supports. This is the subject of the first part of this thesis: how can software architecture practices can be used to ease the development of MASs and more particularly their implementation by exploiting a component-based approach.

Second, software architectures, and in particular component-based software architectures, which are the closest the field has come to implementation, are mainly rooted in static composition of components. More and more interest is expressed towards dynamicity, adaptivity, robustness, etc, by the community. MASs, as an approach with these precise objectives, has the potential to help to tackle this question. This is the subject of the second part of this thesis: how do MASs relate to CBSAs, how can they support the adaptation of such architectures, and as we are going to see, how can the contribution of the first part plays a role in answering both these points.

Part I

SOFTWARE ARCHITECTURES FOR MULTI-AGENT SYSTEMS

CHAPTER 2

State of the Art on MAS Development

Work It Harder, Make It Better
Do It Faster, Makes Us Stronger
More Than Ever, Hour After Hour
Work Is Never Over

Harder Better Faster Stronger
Daft Punk

Encouraged by the numerous evidences of the existence of a gap between design and implementation in MASs development, we thoroughly study and analyse the various existing means of development that exist in the field of MAS development, both for design and implementation. This gives us the possibility to understand the specificities of such systems and the challenges that should be tackled to ease MAS design, implementation and maintenance as well as improve the quality and productivity of the development process.

As it is explained in the next sections, the main way of tackling MAS development can be separated into two important phases.

The first one focuses on designing the system in terms of agents and their environment by defining their behaviours. The MAS community focused a lot of energy on that matter and in the last 10 years a huge quantity of methods and models were produced to support the design (architectural and non-architectural) of MASs. These models and methods target different applications, domains, approaches or problems. Quite complete surveys on the matter of agent-oriented development methods have been proposed (Bergenti, Gleizes, and Zambonelli 2004; Henderson-Sellers and Giorgini 2005).

Then, the second phase focuses on implementing the design. To do so, developers can either use development supports that help them to focus on high-level concepts, or do it by hand using a general programming language. Again, many development supports were proposed by the community and studied in reference surveys (Bordini et al. 2005; Bordini et al. 2006; Bordini et al. 2009).

Unfortunately, as it has been highlighted several times in all these works and as we are going to see, the gap between design and implementation is one of the challenges of MAS development. The main problem here is to make the concepts manipulated at design coincide with those available for implementation. For example, the paper of Bernon, Cossentino, and Pavón (2005), which covers most of the known works on the trends in the field on methods, models and tools, shows and explains this point particularly well.

Thus, always with the motivation brought up by software architectures — take into account functional and non-functional requirements for better software quality, maintainability, reuse and explicit structure — this chapter first gives a general tour of the three different problems of MAS development — design, implementation and the gap between them — in order to spot ways of easing the development and of bridging the aforementioned gap.

In the following, we first present a characterisation of existing works that support the design and the implementation of MAS from an architectural viewpoint. This data serves as a basis to present the different important matters that have to be tackled during MAS development. Finally, from this survey of existing means of development for MASs, we propose an analysis of these works and conclude it with a classification that puts the focus on the research works themselves. This helps us to spot important points of research and motivate the contribution presented in the next chapter.

2.1 Characterisation of MAS Meta-Models from an Architectural Viewpoint

In order to analyse the existing works, we first propose to characterise them in order to understand what their meta-models provide in term of architecture. In the following, the term "model" is a central concept used to describe and analyse the works surveyed. As defined previously, a model is used to describe a system, and a meta-model (which itself is a model) is used to describe models.

For a definition of *Model*, see p. 2

We focus here on the meta-models that help us to describe MASs during their design but also implementation using existing development supports. This helps to show the relation between design and implementation, as well as to highlight the specificities of these meta-models. To talk about them and differentiate the two categories, we talk about **MAS meta-models** and **development support meta-models**. They are all detailed Appendix A.

This characterisation is actually exploited in the following sections, we only present the general ideas now. Concerning development support meta-models, we only looked at those with a freely available implementation.

The meta-models are categorised based on which of the three following types of elements of a MAS they can model: internal agent architecture, environment architecture and interaction architecture. The first one focuses on the way agents are architectured in order to behave as a part of a MAS. The second one focuses on the way the environment is architectured: it concerns all the elements of the system that are not considered as agents. The last one is

about mechanisms that let the agents interact with each other or the environment and that lies in-between the two previous elements.

We chose this categorisation as it is well admitted that MAS from an architectural point of view is "designed as a set of autonomous software entities (agents) that are embedded in shared structure (the environment)" as it is for example presented in the Preface of the *Multiagent Systems and Software Architecture (MASSA)* workshop (Weyns and Holvoet 2006) by its organisers. The interaction architecture is introduced because it is of first importance to us as it will be shown in this thesis.

In each category, different features characterise the meta-models. Moreover, a special meta-feature named "definable" is introduced. All of these are explained when needed in the rest of this chapter where the result of the characterisation, presented Table 2.1 and Table 2.2, is exploited.

2.2 Walking Through the Different Aspects of MAS Development

In this section, we propose to look at the different aspects of the state of the art on MAS development that are important with respect to our objectives.

This walk-through is decomposed in two parts. The first part focuses on the different aspects of importance linked to all the works interested in modelling artefacts during the development. We cover here the different MAS meta-models, the different kinds of requirements that are answered and not answered by these meta-models and the different development support meta-models. The second part is about bridging the gap between these different aspects and in particular the design and the implementation.

We restrain ourselves in this section from expressing any judgements on these work: we are more interested in painting a global view of the important aspects of MAS development. This serves Section 2.3 as evidences to infer a critical analysis of the field and an incentive for the contribution.

2.2.1 The Design Point of View

Be it in development methods or not, what particularly interest us are the different MAS meta-models existing in this field that are usable to describe the produced MAS and its structure (by opposition to meta-models used for requirements analysis for example). They concern the description of the agents, their internal organisation, the system organisation, the environment, etc. This is interesting here because they are what is needed to be implemented at one point or another.

As presented previously, Table 2.1 shows the different MAS meta-models and which features they provide for each type of architectural elements. We detail these features now:

- Agents Internal Architecture: how the designer can model the internal architecture of the agents.
  - Goal: behaviour of agents is described in terms of goals and planning.

Table 2.1: Characterisation of Design Models in terms of available Features of Architectural Elements

	Internal Architecture					
Design Model	Goal	UML-like	BDI	Rules	Aptitudes	Recursion
SeSAmUML		X		X		
PASSI2		X				
ASPECS		X			X	Holons
AgentUML			X			
AML			X			
ADELFE		X		X	X	X
A&A						
Macodo						
AGR						
MASQ						
VOWELS						
MOISE+						
Opera						
MESSAGE						
GAIA						
CRIO						
Tropos	X					

	Environment Architecture				
Design Model	Org. Struct.	Org. Dyn.	Situated	Entities	Definable
SeSAmUML				X	
PASSI2					
ASPECS	X	X			
AgentUML					
AML					
ADELFE				X	
A&A					artefacts
Macodo	X	X			
AGR	X				
MASQ					brute space, objects
VOWELS					AEIO bricks
MOISE+	X				
Opera	X	scenes			
MESSAGE					
GAIA					
CRIO					
Tropos	X				

	Interaction Architecture			
Design Model	Messages	Action/Perception	Role Play	Definable
SeSAmUML		X		
PASSI2				
ASPECS	X		X	
AgentUML	X			
AML	X			
ADELFE		X		
A&A	X	Artefacts Use		
Macodo			X	
AGR			X	
MASQ				bodies
VOWELS	X			AEIO bricks
MOISE+			X	
Opera			X	
MESSAGE	X		X	
GAIA	X		X	
CRIO			X	
Tropos	X		X	

    - UML-like: the behaviour of the agents is described using an UML-like abstraction such as sequencing diagrams, state diagram, etc.
    - BDI: the behaviour of the agents is described in terms of Beliefs-Desires-Intentions.
    - Rules: the behaviour of the agents is described using rules.
    - Aptitudes: aptitudes, skills and other kind of mechanisms that agents can refer to in their behaviour can be modelled.
    - Recursion: agents can be recursively modelled as MASs.
- Environment Architecture: how the designer can model the environment of the agents.
    - Organisation Structure: the structure of the agents organisation can be described, often in terms of roles and their relations.
    - Organisation Dynamics: at the environment level, a dynamics for the organisation not managed by the agents themselves can be described.
    - Situated: consider agents are in a situated environment.
    - Entities: entities in the environment with which the agents can interact can be defined.
- Interaction Architecture: which interaction means the designer can use to model the interactions between the agents.
    - Messages: consider agents can communicate using message passing.
    - Action/Perception: consider agents can perceive their environment and act on it.
    - Role Play: consider agents play roles in an organisation.

The *definable* meta-feature has the following meaning: when no feature is available to directly describe an element of the solution, MAS meta-models with a meta-feature provide means to define abstractions adapted to describe the solution instead of using the (potentially inadequate) features of the meta-model. The boundary between a feature and a meta-feature can be fuzzy, but as an example, message passing is typically not a mean to define new abstractions, while an abstraction usable to define new action and perception mechanisms is.

The important point to notice in this characterisation is that the features tackled by these meta-models are **very different depending on the work**. Some of them focus more on the internals of the agents, while others tackle the environment itself. There are some works about the environment as an interaction medium, others try to define a way to describe the organisation the agents operate in. Moreover, in terms of architecture, be it internal or external, MAS meta-models have overlapping features or at different levels of abstraction. But at the same time, not two of them actually share the same set of features, except for a few exceptions. Thus they all cover different parts of what a MAS is, some of them are complementary, while others are exclusive.

A second point, which is not directly highlighted by the characterisation, is that the **semantics of the MAS meta-models are more or less precise**. This is actually linked to the fact that they cover different parts of what a MAS is: a MAS meta-model focused on the

environment does obviously not say anything about the internal architecture of agents, but maybe constrains the way agents can communicate. But moreover, even when a meta-model defines how a specific part of the architecture is structured, it can omit details of the semantics of it. A good example is about all the MAS meta-models defining that an agent has a perceive-decide-act cycle: does it mean the agent is reactive and reacts to every percept that occurs following this cycle, or does it means that there is a general loop where in the perceive step, the agent checks its percepts? This has to be defined at one point or another, but some MAS meta-models don't provide any information on that. We come back on the implication of that matter later.

The third and last point on these works is about the architecture style that MAS meta-models follow. Indeed, **MASs are made of agents and their environment, which are interacting together** in a dynamic and peer-to-peer way. From a software architecture point of view, this is seen as a family of architectural styles — as highlighted Chapter 1 — respected by almost all the MAS meta-models, even though they all focus on different parts of it. Elements of the system are the agents, they have autonomy, but more importantly, they are not directly connected to other particular agents: they only have access to interaction means that give them the possibility to communicate with whoever they know. That last point contrasts particularly with common means of interaction between software elements in other existing approaches for software decomposition. For example, in component-based programming, elements are connected explicitly with connectors, and in object-oriented programming, elements directly know each other and interact only by synchronous calls. This is of importance because this fact has an impact on the implementation as we are going to see.

To conclude, there is a lot of diverse meta-models that are used to design MASs. They are usable to describe the system in diverse ways at different levels, but all with the same kind of architectural style. They do so in a more or less precise way giving some latitude to the designer and developers for choosing a specific semantics for it.

In terms of our objectives of easing the development, something else is of importance: these MAS meta-models are the support of numerous works and MAS approaches, and thus their use cannot simply be avoided. They are tools that researchers and engineers want and should use to build applications based on MASs. This is of first importance since it constrains the solutions that can be proposed to ease the development of MASs.

2.2.2 Diversity of Requirements

The previous section presented MAS meta-models where all of them act as a mean to capture a way to realise a system for a specific problem. At the start of every development, what these systems should do is expressed using requirements. This is the subject of this section.

For a definition of *Requirements*, see p. 7

The first thing to say about these requirements is that **depending on the MAS meta-model used, different kinds of requirements are answered**. Some works see MAS approaches as a mean to provide some quality attributes for the system to be built and analyse them in order to help choosing a development method. For example, Silva et al. (2004) proposes

categories of non-functional qualities that a MAS can have and classifies some development methods in these categories. The preface of the *Multiagent Systems and Software Architecture (MASSA)* workshop (Weyns and Holvoet 2006) also states that MAS are meant to answer quality attributes such as adaptivity, robustness or scalability. This idea is also defended by Weyns (2010). On top of these, which are mostly non-functional requirements, the MAS meta-models are obviously able to describe solutions answering the functional requirements of the application. A work that is well known for emphasising the importance and diversity of requirements is the Tropos method (Giorgini et al. 2005).

Actually, MAS meta-models only answer a subset of them: for example the use cases (which are expressions of functional requirements) involving the user of the application are often not fully answered by the MAS methods and their meta-models themselves. Parts of them are left at the discretion of the developer. Indeed, in MASs, there are requirements that are answered by the MAS itself, such as the main functionality of the application, or the quality attributes previously referred to. But there are also more traditional functional requirements such as Graphical User Interface (GUI) for interacting with the system or the agents, or for monitoring them. There are also other non-functional requirements more concerned with organisation of the development such as modularity, reusability, cost management, etc. Thus, every MAS meta-model in the literature tackles requirements, but **none of them ever answers all the requirements of one application at once**: this means there is always requirements that must still be answered after an approach and its MAS meta-model has been applied.

Moreover, a point often forgotten about requirements in MASs is that **the choice of a MAS meta-model and the resulting design both introduce new requirements** that have to be tackled during implementation. Indeed, as we said earlier, all MAS meta-models make hypothesis on the way agents are internally organised, or on the kind of interaction means they can use, or the interaction language they exploit. When the design is done and implementation has to start, new requirements and constraints have appeared and must be taken into account. Be it using an existing development platform, as we see after, or by building everything by hand, this has to be done. For example, a MAS meta-model can assume that agents are all synchronised in their execution, as it is always done in simulation, for the result of the design to be valid. In a way, these introduced requirements are the expression of the semantics of the MAS meta-model.

To conclude, the MAS meta-models help to tackle diverse requirements and also introduce other new requirements that constrain the implementation. When developing a MAS, a design approach, and thus a MAS meta-model, is chosen and this has an impact on what can be answered, and how it will be answered. Of course, this question of requirements is very common in the software architecture field as being the main reason why an architecture is defined.

What is important to notice from this section is that all of these requirements have to be taken into account, as early as possible, and as easily as possible. This must be remembered to be able to propose an adequate solution for easing the development of MASs.

2.2.3 Commonality of Requirements and Elements of Solution

As we saw, all the requirements that we talked about must be taken into account. What can now be said is that some requirements are mostly common to MASs while other are very specific to some applications. Also, some are very common to software in general, while other are only found in specific sub-domain of the MAS field. For example, adaptivity is often a known requirement answered by MASs, while execution fairness or monitoring of the agent execution is mostly a problem in the simulation domain. Security finds itself more in the web and internet applications, performance in the optimisation domain. Then, as we said earlier, different approaches and MAS meta-models are also introducing new requirements, which are thus specific to them.

In practice, these requirements are answered by some elements of the built solution. These elements can manifest themselves as bit of code such as mechanisms, datatypes but also high-level designs. When one wants to reuse in order to not reinvent the wheel every time an application is developed, one can use code, patterns or other means of reusing knowledge as it was presented Chapter 1. On the other hand, some elements have to be implemented on purpose for the application.

To get a clear vision of the specificity or generality of requirements and elements of solution, we propose a simple classification as following, going from the problem to software development in general:

- Problem/Application/Project: they are specific to the problem at hand, they often concern functionality, but also requirements for the development itself such as modularity and organisational constraints. They are about GUIs, behaviours of the agents, etc.
- Domain: they are concerned by a set of problems or application, examples of domains are optimisation or simulation. They are about monitoring GUIs, scheduling mechanisms, environment management.
- Approach: they are concerned with specific approaches and MAS meta-models, they constraint the implementation for example for the lifecycle of the agents or the way they can interact. An example is rule engines for executing behaviours described using a specific method.
- MAS: they are about MAS in general, for example the adaptivity requirements or the fact that the system is architectured as agents and environment.
- Others: they are anything that is not concerned with the previous ones.

For a definition of *Software Product Line*, see p. 8

Some works focused on applying the SPL approach to tackle this commonality: main works on the matter of MAS Product Line (MAS-PL) are DDEMAS (Girardi and Lindoso 2006), MaCMAS (Peña et al. 2007), Nunes et al. (2011) (based on PASSI) and GAIA-PL (Dehlinger and Lutz 2011). The differences between them are on the phase of design, the types of variability points identified and the methods and techniques they were inspired by.

For a definition of *Architectural Pattern*, see p. 8

A second set of works are about design or architectural patterns. Some of them focus on organisation of agents for specific objectives, while others are about coordination mechanisms, mainly environment-mediated, to tackle some common problems. Huge surveys of existing

patterns in MAS were produced (Oluyomi 2006; Oluyomi, Karunasekera, and Sterling 2007). These works classify them depending on the phases of development they are useful for and the tasks that are tackled in these phases. Some works regrouped sets of patterns in reference architectures or in pattern languages. Most of these works focused on the environment-mediated types of MASs (Weyns, Omicini, and Odell 2006; Viroli et al. 2007). This can be explained by the diversity of interaction means and the few development supports available for implementing this kind of MASs.

As a conclusion, there is a lot of diverse requirements, but a lot of them can be recurring in different applications. The answer to these requirements can be found as **reusable mechanisms, patterns or any other way to capture experience**. This at the design level, but also at the implementation level.

Thus, this is an important point to take into account to ease the development: if possible, development by and for reuse must be possible.

2.2.4 The Implementation Point of View

A common mean of reuse existing known way to implement MAS — and thus by the way to answer some recurring requirements — is to use development supports. The MASs community proposed several platforms, frameworks and programming languages adapted to MAS development.

In a rough way, the link between these three means of implementation is as follow. Programming languages and frameworks are means to declaratively or imperatively describe elements of a MAS. The difference is that languages have programming primitives dedicated to their domain while frameworks provide their programming abstractions to their hotspots in a more general programming language such as an object-oriented one. In order to execute these descriptions, there exist execution platforms for them. In the case of frameworks, it is comprised in their frozenspots. For the languages, it is more like abstract machines that can execute the language or compiler that can transform the language into executable code. Moreover, platforms provide the infrastructure needed for the agents to work, such as communication or spatial interactions for example.

Available agent programming languages are usable to specify agent behaviours with adapted programming primitives. The main origin of programming languages in the MAS field comes from the Agent-Oriented Programming paradigm (Shoham 1993) that proposes to describe agents in terms of their states and mind. The most famous known applications of this approach are the Belief-Desire-Intention (BDI) languages (Bratman 1999).

Table 2.2 shows development supports meta-models categorised as presented previously. Only those with an existing and available implementation are present. Moreover, in the case of development supports relying on others, the name of the development support relied on is written in italics in features it provides.

We now detail the different features existing for each of the types of architecture elements:

– Agents Internal Architecture: how the programmer can implement the behaviour of the agents or elements of its internal architecture.

Table 2.2: Classification of Development Supports in terms of available Features of Architectural Elements

	Internal Architecture						
Dev Model	UML-like	BDI	Aptitudes	Recursion	Event	Scheduling	Definable
JADE						X	
Jadex		X				X	
JACK		X				X	
Jason		X					
J-MOISE+		*Jason*				*Jason*	
S-MOISE+							
Janus	X		Capacities	Holons	X		
MadKit						X	
SeSAm	X						
NetLogo							Methods
Cartago							
MAGIQUE				Hierarchy			Skills
MALEVA						X	Components
Malaca						X	Components

	Environment Architecture						
Dev Model	Org. Struct.	Org. Dyn.	Situated	Msg Ref	Directory	Scheduling	Definable
JADE				X	X	X	
Jadex				*JADE*	*JADE*	*JADE*	
JACK							
Jason				X		X	
J-MOISE+	*S-MOISE+*			*Jason*		*Jason*	
S-MOISE+	X						
Janus	Holons	X		X	X	X	
MadKit	X			X		X	System Agents
SeSAm			X				
NetLogo			X			X	Methods
Cartago			Workspaces				Artefacts
MAGIQUE	Hierarchy						
MALEVA						X	
Malaca				X	X	X	

	Interaction Architecture			
Dev Model	Messages	Action/Perception	Role Play	Definable
JADE	X			
Jadex	X			
JACK				
Jason	X			
J-MOISE+	*Jason*		X	
S-MOISE+				
Janus	X		X	
MadKit	X		X	
SeSAm	X			
NetLogo				Method calls
Cartago		Artifacts Use		
MAGIQUE				
MALEVA				
Malaca	X			

    - UML-like: can execute behaviours described using an UML-like abstraction.
    - BDI: can execute behaviours described in terms of Beliefs-Desires-Intentions.
    - Aptitudes: abstractions to implement unit of aptitude, skill or other mechanisms.
    - Recursion: usable to define agents as MASs.
    - Event: provide abstractions to implement and execute behaviours triggered by events.
    - Scheduling: provide specific scheduling means for executing behaviours, often adapted to the other means of internal architecture implementation.
- Environment Architecture: how the programmer can implement elements of the environment.
    - Organisation Structure: abstractions to describe the organisation structure and take care of its management.
    - Organisation Dynamics: abstractions to describe and execute the organisation dynamics.
    - Situated: to execute and manage situated agents and visualise them.
    - Directory: provide a directory to register and find agents.
    - Scheduling: implement scheduling mechanisms for executing the system in a coherent (not necessarily synchronised) way.
- Interaction Architecture: which interaction means the programmer can use to implement the agents.
    - Messages: provide means for agents to exchange messages.
    - Action/Perception: provide abstractions to describe way of interacting with the environment.
    - Role Play: provide abstractions to take part in an explicit organisation.

The *definable* meta-feature has the following meaning: they are abstractions provided by the development support meta-model to define features instead of defining elements of the solution directly using a given feature. We can clearly see the difference between these two ways through their use when no feature is adapted to the elements to be implemented: a meta-feature can be used to produce their implementation in a direct way, possibly through the definition of a new feature, while without a meta-feature, an existing feature have to be adapted or extended to implement the element.

These meta-models provide abstractions to program a MAS design. They range from very specific such as adapted to a MAS meta-model, to very general such as a general programming language. Others take an orthogonal stance and provide their own meta-model that can be used with a subset of all the available MAS meta-model.

In any way, this classification shows the same kind of conclusion than with the MAS meta-models. They also have overlapping features at different levels of abstraction, and at the same time not two of them actually share the same set of features, except for a few exceptions.

But differently than with the MAS meta-models, the architectural styles followed by the implementation meta-models are much more diverse. For example, in simulation platforms, agents are just behaviours that are executed in a sequential and synchronised way without any independence from each other or the environment in terms of software elements. Other development supports enforce the separation and have a clear distinction between the agents as software entities and the platform as the environment infrastructure as called by Weyns and Holvoet (2006).

What is important here is that when choosing a specific development support meta-model (if one wants to implement its design) the developer have to **rely on its semantics** for implementing his application. This means that there is an effort **to make the concepts manipulated at design and at implementation coincide**.

A second interesting point about development supports is that they have an other objective than just reusing implementation and design. They also are a mean to ease the development by **providing adequate abstractions** to the MAS developer. The MAS developer does want to focus on his design, on his problem, or, should we say, on his business concerns. In the world of MAS design, the user of the MAS meta-models and approaches are not always expert programmers on the subject of concurrency or distribution. He does not want to bother with technical problems — such as synchronisation between agents, spatial movement management, etc — but to work using high-level abstractions. They can be sociologists studying social behaviours, people interested in constraint optimisation, etc. That's why existing development support are used extensively to abstract over the details of the implementation in order to focus on what is their business. They of course are usable to implement the MAS design, but also other things that are linked to any of the requirements they can have. Example of this are the need for having control on how data is shown in simulation.

2.2.5 From Design to Implementation

The previous sections highlight the gap between the concepts manipulated at design and those manipulated at implementation that was implicitly present since the beginning of this walk-through. The gap is well presented by Molesini, Denti, and Omicini (2007): they explain that while the MAS meta-models are interested in agent-oriented solutions, the development support meta-models are more inspired by object-oriented languages and practical concerns. A huge quantity of works try to solve this problem.

Some of them are dedicated to specific MAS and development support meta-models (Amor, Fuentes, and Vallecillo 2005; Sudeikat et al. 2005; Molesini, Denti, and Omicini 2007; Pavón, Gómez-Sanz, and Fuentes 2006; Rougemaille et al. 2009). Others try to compose existing MAS meta-models in order to adapt the process to the development support (Mariachiara et al. 2010). Others provide kind of IDEs to support the implementation of specific MAS meta-models such as INGENIAS Development Kit (IDK) (Pavón, Gómez-Sanz, and Fuentes 2005), which supports INGENIAS, or Shell for Simulated Agent Systems (SeSAm) (Klügl, Herrler, and Oechslein 2003), which supports SeSAmUML.

Finally other works focus on unifying, generalising or composing existing MAS meta-models. Works (Beydoun et al. 2005; Bernon, Cossentino, Gleizes, et al. 2005; Dalpiaz et al. 2008) propose to unify a set of MAS meta-models by finding common concepts in them. By doing so, only one development support adapted to this meta-model can be proposed. Another (García-Magariño 2009) proposes to easily transform between existing MAS meta-models by defining instantiation of meta-concept using powertype modelling. Hahn, Madrigal-Mora, and Fischer (2009), on the other hand, propose a general development support meta-model to which should be provided instantiations depending on the used MAS meta-model. From this general meta-model it is possible to generate an implementation.

To conclude, these works try to answer this matter either by proposing unifying MAS or development support meta-models that can either be used with a generic development support or at least that can be used to generate what is not generic. But as we highlighted earlier, even though these are meant to be generic, MASs meta-models are so diverse that generic solutions to the matter are actually specific in the sense that they only focus on one MAS meta-model. And as we said, each MAS meta-model can not completely answer the requirements of one particular problem to be solved. While this bridges the gap between design and implementation, it only does so in an ad-hoc (although adapted to a MAS meta-model) way.

2.2.6 Practical Observations

All of this has practical implications of the development when combined with our own observations and assumptions on how development is done.

First, as we observed, assumed from existing works in the literature or personally experienced, it seems that the actual practice in MAS development is to use one of the following sequence of tasks:

- Choose a development support before the design, choose a MAS meta-model that fits to it, do the design constrained by these and finally implement the system.
- Choose a MAS meta-model, do the design, choose a development support that fits to it, and finally implement the system.
- Choose a MAS meta-model, do the design and directly and completely implement the system by hand.

In all the cases, the meta-model chosen can be one of those presented previously and the way used to implement the system can be using one of the tools or transformation presented here, or by hand.

What is interesting with that is that in the first two cases this means that **it is needed to either adapt the design to the development support, or to either adapt the development support to the design**. This has several impacts, of course as we just saw for actually bridging the gap, but also on the ease of development itself. There is people that use the existing development support as a mean to have high-level abstraction adequate to their expertise. Inversely, the people that implemented these supports are expert on implementing such

abstractions. They know what are the best way of tackling some of the requirements that are answered by the development support.

These **two different sets of developers** depend on each other: the users of the development support highly depend on their creators to have something adapted to their design. This is particularly important in MAS because of the diversity of available meta-models for designing as well as implementing. Furthermore, all of this is exaggerated by the fact that the concepts at design are not always semantically very precise as it was said earlier.

Also, another fact that makes the previous even more problematic is that methods and approaches try to make the design more and more refined during the development and not only as a first step of it. Indeed, when developing MAS, it has been observed that **the process of implementing and design is a very iterative, agile process**. This is visible in methods where a quantity of iterative phases exists. In the specific case of MAS, the need for such an iterative approach to design has been highlighted by Georgé et al. (2003). In this work, the authors go further than just design and implementation, and talk about **living design**: the idea of running the system and modifying it until it realises the wanted functionality.

In practice, when participating in developments and interacting with people developing MASs, observations, experiences and feedbacks are that people are constantly revising their design during the implementation of their system.

Something else we noticed is the fact that some requirements are taken into account too late in the development process. A very common example of such requirements is the scheduling of the system, the way agents are synchronised, and their execution is ordered. Even though this has an important impact on the behaviour of the system as a whole, this is often kept implicit during MAS design and taken care of at implementation time when it is too late to handle the impacts on the MAS design of a choice of scheduling strategy. Such bad practices are mostly caused by the fact that existing methods does not take such requirements in account as it was highlighted for example by Weyns et al. (2004).

2.3 Analysis: Should We Tweak or Build?!

In this section, we propose an analysis of the previously presented state of the art of MAS development. We give our opinion and criticise the way things are currently done in order to motivate the contribution of this thesis.

From the matter of requirements, which are the input of the development, and moving towards the problem of the gap between design and implementation, we look at the architectural implications of these aspects of MAS development.

With a general point of view, when building a MAS we can differentiate several phases.

The development starts from the requirements and specifications that describe what the system as a whole must do. Based on these requirements, an analysis followed by a design phase are done in order to define a MAS architecture made of agents and their environment. This is where the existing MAS method and their meta-models are used and the result of it is a system described in terms of the agents, their environment and their interactions. It defines

what are the agent behaviours, which interaction means they use, what is the dynamics of the environment and the potential entities existing in it.

At this point of the design, the work of the MAS designer is finished and the MAS architecture has to be implemented:

- Sometimes the concepts used at the level of the MAS architecture can be used directly to program the final application: these programming abstractions, adapted to those needed by the MAS architecture, are provided by a development support. For example, a design expressed in terms of rules "perception implies action" is implementable using a development support where one can express only the rules and not how they are processed.
- Sometimes some adjustments have to be made, either to extend existing development supports (adding or modifying the abstractions) or to connect them to existing software systems. For example, a design can rely on a communication protocol that must be implemented on top of the messages passing abstraction available in the development support.
- Some other times the final application is totally implemented, including what would have been provided by a development support.

The first case does unfortunately never happen, except maybe when building a first prototype, as we did show that every application has its particular requirements. The second case, which we call "**tweaking**", is maybe the most widespread, the idea being to reuse as much code as possible. The third case, which we call "**building**", does actually also often happen, in particular in the ABM field where the design is strongly linked to the domain of the problem and the abstractions needed are thus very specific to it.

2.3.1 The Gap, Again

The explanation for the existence of these two approaches to implementation is that, as we highlighted, not every type of requirement is tackled by every MAS meta-model and some of them are even introduced by MAS meta-models themselves. This is again the question of bridging the gap: how to take all of these requirements into account? We start from this question to detail the two approaches.

"Tweaking". The most widespread solution taken by current practices is to use a MAS meta-model, answering a subset of these requirements, and choose a development support and its meta-model (compatible with the MAS meta-model) that answers another subset of them. Then, the design is done according to the constraints given by the two chosen meta-models and finally the implementation is adapted by hand to answer the remaining requirements.

Such practices often destroy the original intent of the MAS designer by constraining the way his system is designed with the development support. They don't promote adequate abstractions at the level of the expertise of the MAS developer. In a much broader way, every

time any development support is used, either one has to adapt the design to the development support meta-model, or the development support meta-model has to be modified for the specific design. When directly using the development support abstractions in the MAS design, the latter is polluted with unwanted concepts that do not fit the design. When modifying abstractions provided by the development support, there are two possibilities. Either it means going into the inside of the development support, which is not meant to be modified by any MAS developer but mainly by the software authors. Or it means building new abstractions using existing ones, which is not always adapted and introduces complexity in the software produced.

And even if the development support perfectly fits the MAS meta-model, which is the case only when using a development support made for a MAS meta-model and an application domain, as we highlighted before there is much more requirements that can have an impact on the implementation and that need again adaptation of the development support meta-model for the specific problem that is solved. A simple example of that is the integration of the MAS with other systems: that cannot possibly be already present in the development support.

"Building". The other solution available is to build by hand the development support. Obviously, this is a time consuming and inefficient activity. As we highlighted, there is a lot of things that are generic with respect to some aspects of the system to build, and thus with "building" reuse is not easily done.

In a less extreme way, "building" is currently often done when using a development support that covers just a part of the architecture of the system. For example when using a development support focused on the internal architecture of the agents and the MAS design relies on environment-mediated interactions. This part of the implementation is often done using a general programming language, which means that it involves a lot of low-level and complex expertise that every MAS developer does not have. For the sake of completeness, choosing a development support that introduces environment-mediated abstractions still means there can be other things to "build" or "tweak". For example scheduling of the system is often imposed by the chosen development support, and thus is not always adapted. And if it is not imposed, it thus have to be implemented.

So What? The point here is not to say that all the development supports should be thrown away, but that in a general vision of MAS development, existing development supports are too constraining to cover all the possibilities and that there always exists a class of applications that does not fit the meta-models provided by existing development supports.

What we argue here is that **meta-models of development supports should not be considered in the same way than meta-models for MAS design**. While the latter are something that developers want to use, because they answer some of their requirements, the former are something they spend a lot of time to adapt, tweak, modify, extend to fit the rest of their requirements as well as the MAS design. Since it is not possible to have a "one-size-fits-all" development support, it means that it is necessary to **ease the building or tweaking of**

the adapted development support. There is thus a need for defining and implementing such development support in an easy, composable and flexible way, and with reuse. Not surprisingly, that's the admitted goal of software architectures engineering and that's what we are going to do.

Before entering into the details of that point, we briefly come back to our analysis.

2.3.2 Types of Agents and Adequate Abstractions

What we noticed, when doing this state of the art is that if we accept that MAS has to be done using existing MAS meta-models and methods for its design, then something must help to bridge the gap between design and implementation. Moreover this something must do it in a way that it is adapted to all the requirements of the problem, as well as those introduced by the design itself. **This something is what we call a type of agent**. Every application has its own type of agents, with its own specificities but also commonalities with other types of agents. A type of agent is a precise semantics given to the abstractions used to implement the design, inferred from the latter, from all the requirements of the problem to solve, from the expertise of the developer, etc. For example, defining one type of agent is to say that the agents of one application are interacting by exchanging messages, moving between distributed computers where artefacts (as in Agents and Artifacts) are usable with specific interfaces, executed concurrently in a reactive manner to a set of events such as messages received and timers. Of course, nothing prevents a given application to have several different types of agents using overlapping interaction means.

While MAS design is about building an architecture made of agents and their environment, **now the question is about building the architecture that supports the abstractions** used by this design. It means building the types of agents and the platform that makes them live. Actually, every development support studied here is an instance of such an approach, but **we argue there is a need to do that more easily, faster, better and with more reuse** in order to apply it to every MAS development. As we saw, these development supports are based on one meta-model, or, as we can now say, one type of agents. To do so, these supports have their own architecture, made of software entities that are dynamically created, interacting using diverse interaction means, and made of a platform through which this interaction takes place and that contains the environment and all the other tools such as GUIs, monitoring tools, etc. Thus, we need to be able to build a kind of **application-specific development support** adapted to the types of agents of the application to build.

The second point that this application-specific development support must answer, is related to the **different expertises of the MAS developers**. Indeed, we saw that depending on the field or the problem, the developers do not have the same expertise, and we actually noticed that there is two main kinds of developers in the MAS world: those that are concerned with the **business** aspects of the problem they are tackling, and those that are concerned with the **operative** aspects of the solution to build. There is thus a need for having adequate abstractions for the first set of developers. Moreover, this question coupled to the more methodological one of the iterative development bring us another need. This kind of agile

development can be interpreted as needing the development support to ease the modification of certain aspects of the application, while others stay fixed during the whole development.

2.3.3 Two Levels of Architectural Design

These two points make something clear that wasn't before: **there are two different points of view on the architecture of the final application**:

- The one describing the abstractions themselves and their semantics.
- The one describing the MAS using the abstractions.

These abstractions are sometimes very specific to the problem, to the domain or to the approach, and sometimes very generic. More importantly, the first is highly constrained by the second, since the second is the one defining what are the types of agents of the application.

Based on that vision, we can differentiate the following three different major development tasks in the MASs field:

1. Design (architecture) of the system by using abstractions such as agents defined by approaches: this is what we call the **macro-level design**.
2. Design (architecture and implementation) of the abstractions: this is what we call the **micro-level design**.
3. **Implementation** of the system using the abstractions.

Implementation of the abstractions is included into micro-level design because we didn't find the need to separate them in the rest of the thesis.

As a side note, even though the use of the terms macro and micro are sensitive as in the MAS field, we chose these terms for reasons that will be better explained Chapter 6.

Micro-level design and implementation should be particularly differentiated: implementation is about implementing the MAS itself, *i.e.* the macro-level design, for example by using a development support, while micro-level design is about defining and implementing development supports. Actually, what is implemented at micro-level design is are operative concerns and what is implemented during implementation is are business concerns. This is more detailed later.

Within this vision, requirements in MASs are first tackled at the macro-level design of the MAS, by using an appropriate method or approach and its MAS meta-model. But as we highlighted, not every type of requirement is tackled by every MAS meta-model and some of them are even introduced by MAS meta-models themselves. Thus, those that were not answered previously or introduced late are tackled at the micro-level design.

Figure 2.1 shows the development activities corresponding to these phases. The arrow represents the global order of their execution.

2.3.4 Evidences from the Literature

The first evidence of the reality of these needs is the actual research done by the MAS community on implementation.

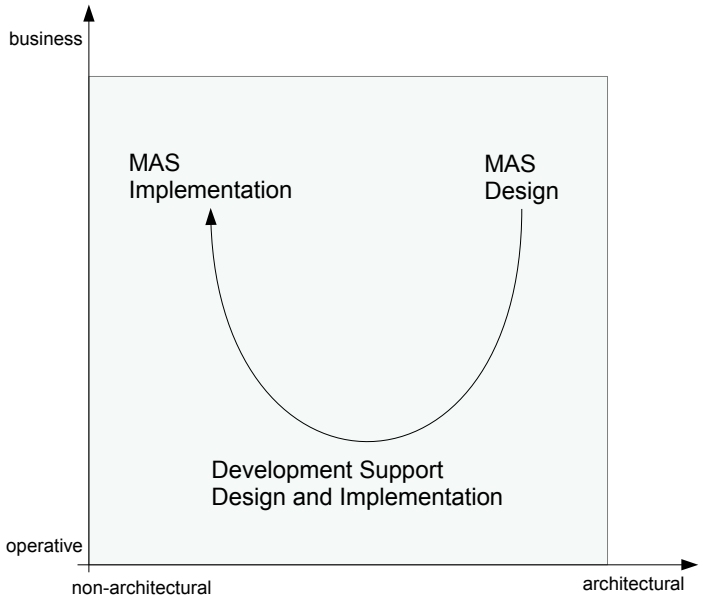


Figure 2.1: Activities in MASs development

When focusing on one development meta-model at a time, the community has provided means and ways to ease the implementation of MASs. Indeed, every time a new type of agents has been defined, either a new development support has been created from scratch, when possible an existing one has been extended, or a transformation from the MAS meta-model to a specific development support has been defined. In all these works, the produced tool was dedicated to a specific type of agents or to a specific domain. These works, see for example Jade, Janus or NetLogo, actually are ad-hoc solutions for specific domains or approaches that ease implementation of the design of MASs.

The problem is even more complex when combining different meta-models together, *i.e.* covering more requirements. A good evidence about that is that the problem of creating a development support for a composition of meta-models is currently so difficult that it is considered a research effort by the community: see for example the JaCaMo, the S-MOISE+ or the SimpA development supports. As we highlighted previously, because every system has its specific requirements, it seems this is actually something MAS developers have to do everyday. This thus shows a real need for easing it.

Then, all the works on patterns (Oluyomi 2006; Oluyomi, Karunasekera, and Sterling 2007) and architectures (Weyns, Omicini, and Odell 2006; Viroli et al. 2007; Weyns 2010) highlight the importance of having an architectural approach and show that every application has its own requirements to satisfy. One particular work on patterns (Schelfthout et al. 2002) caught our attention because it supports what we argue by saying that even if there is a lot of works on bridging the gap, when implementing MAS, there is still some works to do in order to really implement the mechanisms used in the agents and the environment. It makes clear the link between patterns at design and the impacts they have in terms of implementation.

Finally, Weyns et al. (2004) show that existing methods for MASs do suffer from two problems: lake of coverage of the complete architecture of the MAS by the proposed MAS meta-models and abstractions that are inadequate for the level of abstraction needed by non-architectural design. While the conclusions and criticisms are aimed at the methods and the way they tackle software development, this work shows well that there is a need for more ways of going farther in architectural design than what is currently proposed by existing methods and their MAS meta-models. It also confirms the idea that the architectural design of a MAS should take into account the whole system and its requirements, and thus existing development supports are not enough by themselves. Moreover, it supports the idea that some requirements are not taken into account soon enough as we highlighted in our observations Section 2.2.6.

2.3.5 Existing Answers

To support principles of software architectures — better software quality, maintainability and reuse —, which we argue as being necessary to build a development support, few works tried to apply component-based or aspect-oriented approaches. The problem with these is that they only focus on specific type of agents and mainly on the behavioural part of them. The most famous examples of these are Generic Agent Model (GAM) (Brazier, Jonker, and Treur 1999), MALEVA (Briot, Meurisse, and Peschanski 2007), Magique (Routier, Mathieu, and Secq 2001) and Malaca (Amor and Fuentes 2009).

Nevertheless, three other works caught our attention. They try to tackle directly the matter of the environment, but also shows some interesting points for easing the building of adapted development supports. They appear 2.2 as providing what we called the *definable* meta-feature. The first one is Volcano (Ricordel and Demazeau 2002), the second one is CArtAgO (Ricci, Viroli, and Omicini 2007) and the last one is Agent-Environment Interface (AEI) (Behrens et al. 2011).

Volcano is particularly interesting because it makes explicit the need for building a specific architecture supporting the system. Actually, it covers part of our objectives, as the authors highlight, by enabling to build dedicated types of agents for the application, at the agent but also at the environment level. The main problem with this work is actually that no implementation is available and that it seems to be discontinued. Except from the cited article, nothing has been found that clearly explains how things work in detail. It at least comfort us in the idea that it is the right way to go.

Then, CArtAgO, by proposing a general model of interaction of agents in any given environment, coined some interesting abstractions to build, using what they call artefacts, elements of the system adapted to the modelled solution. But artefacts are entities that exist as macro-level abstractions, they are not building blocks for defining new abstractions as it is done with Volcano. Thus, although it is one of the closest solution to the questions we are asking, we are not convinced by it as a general approach to MAS micro-level design, mainly because it still constrains the model of MAS and agents that has to be followed, and because it does not take an architectural approach.

Finally, AEI proposes something different than the previous works because it does not tackle the building of the agents or the environment but just the interface between them. It still provides interesting insight on the problems that arises between the agents and their environment, or should we say, as we are talking about actual implemented software, between the agents and their runtime platform. In particular, we notice that the main problem when connecting several agents to one platform is the fact that the agents have to be kind of "referenced" by the platform in order for the signals coming from it to go to the right agent.

This is mostly that last point about the way things can be implemented to connect agents to their runtime platform that motivates the need for a "good" *definable* meta-feature. This particularity is also highlighted by the patterns proposed in (Schelfthout et al. 2002) or in works about answering visualisation requirements in simulation (Louloudi and Klügl 2011). And of course, this is more generally visible in implementation of development supports or applications.

2.3.6 Revisiting the Different Works

Before exploiting in the next section this analysis to extract challenges to answer for bridging the gap between design and implementation, we first propose a frame into which the state of the art as well as our contribution can now fit.

As we said, there are three different tasks during development: macro-level design, micro-level design and implementation. What interests us in these tasks are their products in terms of software artefacts in order to understand which research work is done for each of them.

We see two levels of understanding where research has been done for these tasks: *base* and *meta*. The main idea behind these two levels is that some works try to solve some challenges — for example applying a MAS approach to a specific problem —, while other works try to propose means to help solving such challenges in a more general way — for example proposing a development method dedicated to a class of problems —.

Table 2.3 shows how all the different works studied in this chapter fit in this classification. In this table, *B1* stands for the *base macro-level design*, *M2* for the *meta micro-level design*, etc. Every pair task/level contains a description of the kind of works that fit in it, then research works are categorised by their general subject of study. The category *Jobs* is about the kind of job researchers and engineers do without being subject to publication.

As a global introduction, this classification does not take into account works on methodologies since it itself actually reflects a methodological vision of MAS development that is not followed by existing works. We now look at each pair one by one to comment them.

For a definition of *Methodology*, see p. 4

Base macro-level design (*B1*) is about producing specific MAS application to solve problems. It is empty, not because there is no work to put in it, but only because they are out of scope of this thesis. Practical applications of MASs can be found in an extensive quantity of conferences, workshops and journals on MASs.

Base micro-level design (*B2*) is about defining and implementing development supports. It is thus concerned with all the development supports and their meta-models presented Section 2.2.4.

Table 2.3: Research Works Organised Following the Proposed Classification

	Base		Meta	
Macro-level	*B1: to design new MASs in order to answer specific problems*		*M1: to define approaches and MAS meta-models to build better systems to support B1*	
			Surveys	Bergenti, Gleizes, and Zambonelli (2004); Henderson-Sellers and Giorgini (2005)
	Jobs	Everyday design work for researchers and engineers	Methods	See Table 2.1
			Requirements	Silva et al. (2004)
			SPL	Girardi and Lindoso (2006); Peña et al. (2007); Nunes et al. (2011); Dehlinger and Lutz (2011)
			Architecture	Weyns (2010)
Micro-level	*B2: to design and implement new development supports and patterns usable to implement B1*		*M2: to define ways of better build artefact such as development supports and patterns usable to implement B1 and/or M1 to support B2*	
			Only Behaviour	Brazier, Jonker, and Treur (1999); Routier, Mathieu, and Secq (2001); Briot, Meurisse, and Peschanski (2007); Amor and Fuentes (2009)
	Surveys	Bordini et al. (2006); Bordini et al. (2009)	Whole System	Ricordel and Demazeau (2002)
	Development Supports	See Table 2.2	Architecture	Weyns, Omicini, and Odell (2006); Viroli et al. (2007); Weyns (2010)
	Jobs	Everyday implementation work for researchers and engineers	Patterns	Schelfthout et al. (2002); Oluyomi (2006); Oluyomi, Karunasekera, and Sterling (2007)
Implementation	*B3: to implement the design from B1 using B2*		*M3: to define means to systematically generate an implementation from B1 and/or M1 and using B2 and/or M2 in order to support B3*	
			M1/B2 to B3	Klügl, Herrler, and Oechslein (2003); Amor, Fuentes, and Vallecillo (2005); Sudeikat et al. (2005); Pavón, Gómez-Sanz, and Fuentes (2005); Beydoun et al. (2005); Bernon, Cossentino, Gleizes, et al. (2005); Pavón, Gómez-Sanz, and Fuentes (2006); Molesini, Denti, and Omicini (2007); Dalpiaz et al. (2008); García-Magariño (2009); Hahn, Madrigal-Mora, and Fischer (2009); Mariachiara et al. (2010)
	Jobs	Everyday implementation work for researchers and engineers	M1/M2/B2 to B2/B3	(Rougemaille et al. 2009)

Base implementation (*B3*) is about concrete implementation of such systems. It is also empty, but only because it concerns implementation of systems, which is not a subject of publication nowadays.

Meta macro-level design (*M1*) is about defining methods to support *base macro-level design*. Thus, it is concerned with all the methods, approaches and MAS meta-models presented Section 2.2.1.

Let's take a pause and first comment these. We can see that meta-models for MAS design (and accompanying methods) in *M1* and for development supports in *B2* are not considered in the same way that it was in the beginning of this chapter. While the former is considered *meta*, the latter is considered *base*: this relates to our argument that it is possible to improve the building of development support at an higher level than just proposing one-shot development supports.

Meta micro-level design (*M2*) is about meta-models to support the building of *base micro-level design*, *i.e.* development supports. It is thus concerned with all the meta-models presented Section 2.3.4 and Section 2.3.5. Not surprisingly, we can also find among them the development support meta-models with the *definable* meta-feature of Table 2.2. Works on patterns provide ways to informally describe recurring solutions to recurring problems. Those on architecture focus more on how to do macro-level and micro-level design, without going up to the implementation itself in a practical way. These lasts are still the most interesting ones the design of MAS and bridging the gap towards implementation, which is not surprising since they follow a software architecture approach. Works in other fields than MASs exist and can also fit in *M2*, but not without adaptations. This matter is actually presented extensively Chapter 6 by relying on the contribution.

Meta implementation (*M3*) is about easing the implementation MASs and automatising the transition between different meta-models. It is separated in two categories depending on which kind of meta-models it bridges. The first one, *M1/B2 to B3*, is concerned with the works surveyed Section 2.2.5. They go from a MAS meta-model to an existing development support (even if this one is a generic programming language). The second one, *M1/M2/B2 to B2/B3*, is concerned with works for a combination of a meta-model for design and a meta-model for defining adapted development supports. They automatise the production of micro-level design but also implementation of the macro-level design, possibly by reusing previously realised piece of micro-level design. The only work present in *M3* is based on a prototype version of the contribution of this thesis. It does not actually completely answer the problem but is a first step in this direction.

2.4 Challenges: Meta Micro-Level Design

From the survey of this classification, we argue that the problem to **the gap from design to implementation can only be tackled by providing complete models for *meta micro-level design*** in order to later automatise the production of software by proposing works for *meta implementation*.

The contribution of this thesis mainly tackles such *meta micro-level design* by helping to build application-specific development support. Actually, what is presented in this thesis fits in several places of the classification. This will be developed Chapter 5, but in few words, the contribution consists of a general methodology of MAS development that covers the whole *meta* axis, a model and a method that fit into *meta micro-level design*, and all the software artefact that can be produced using this model, in particular a library of reusable artefacts, fit into *base micro-level design* as well as *base implementation*.

In order to be able to provide ways to build application-specific development support, a set of challenges have to be answered.

First, because of the way MASs, as a macro-level design, is architectured, there is a specific but broadly defined architectural style that has to be followed: agents as software entities are interacting together using interaction means provided by their runtime platform, which also has the role of executing them. Moreover, they are not only connected to their platform for business interaction means: there is also operative connection between agents and their platform. For example when a GUI shows information from all the agents of a system, or when the agents are all synchronously executed by the a global scheduler as in simulation applications. This is the way things are done in existing development platform, even if there is variations about it depending on the requirements they cover.

To simplify the discourse when talking about these two usages of the abstraction, we name them **interconnection mechanisms**.

By looking at existing answers to the problem — analysed in the previous section —, after studying existing development supports and building actual MASs from scratch, we have identified the following architectural and implementation challenges, tackled in this thesis:

- Define and implement interconnection mechanisms for connecting agents to the runtime platform and vice-versa.
- Define and implement types of agents with specific internal architecture and connected to the runtime platform using the previously defined interconnection mechanisms.
- Provide dynamic creation of instance of these types of agents, as well as their dynamic connection and initialisation.
- Make the reuse of such interconnection mechanisms and architectures easy.

In particular, with respect to the features presented Section 2.2.4, the main difficulty here is to propose a "good" *definable* meta-feature for the interaction architecture. Indeed, it is not about proposing a good abstraction for representing environment and interactions at the macro-level design, but to be able to define new interconnection mechanisms and to compose them in the architecture of the agent. This then supports the implementation of the macro-level design using reusable adapted abstractions. Some of these challenges are illustrated in the next chapter.

Then, more on the methodological side, in order to do this micro-level design and integrate it in a more general vision of MAS development, we propose to answer the following challenges:

- Propose a general vision of MAS development integrating macro-level and micro-level design, and implementation with iterative and incremental development.
- Provide guidelines to identify the requirements that help the micro-level design.
- Provide guidelines to identify abstractions needed to build a dedicated development support for the MAS developer.

Of course, always with architectural concerns in mind, in order to increase the quality of the produced software by applying software architecture practices and to make its development realistic, these challenges have to be considered within the following constraints:

- Take all the requirements into account as early as possible in the development process.
- Improve software quality and in particular reuse, maintainability and evolution.

Recap of the Contributions

⊕ We highlights the particularity of MAS development characterised by the inadequateness of using pre-built development supports.

⊕ We propose a classification of the research works in MAS development highlighting the current lakes.

⊕ We extracts from the state of the art a set of architectural and implementation challenges for building dedicated architectures usable in place of development supports.

CHAPTER 3

Dedicated Micro-Level Software Architectures for MAS Development

Now that ain't working
That's the way you do it

Money for Nothing
Dire Straits

This chapter presents a coherent set of answers to the different needs of MAS development by focusing on the specificities of MASs that were uncovered previously. It proposes architecture-centric methodological solutions to organise development and an approach based on software components to support the design and the implementation of the system.

Our objective here is to ease the development of MASs, ease their maintenance and evolution, as well as increase reuse of produced artefacts. For that, we aim at building systems with software quality in mind by applying well-known software architecture practices.

As the previous chapter concluded, in order to reach these objectives it is needed to propose means to ease what we called micro-level design. Micro-level design is about producing an architecture usable as a development support on which the implementation of the MAS design — also called macro-level design — can rely. Indeed, when producing a MAS, the macro-level design is able to answer only a subset of all the requirements of the application, the rest has to be tackled after the design is done. Depending on the MAS meta-model and approach chosen, as well as the design choices made, the abstractions needed to implement the MAS design vary a lot. We named such abstractions types of agents: they are the bridge between the macro-level design, that relies on them, and the micro-level design that realises them.

This development phase is thus interested in producing such a micro-level design in order to answer the remaining requirements as well as those introduced by the design itself. This micro-level design, once implemented, is then usable to implement the macro-level design.

This highlights the existence of two different roles in the development process, we characterise them as interested in business and operative concerns.

We identified methodological and technical challenges that a solution to this problem must answer:

- First, in order for this solution to be usable and integrable in a more general development process, it is necessary to clearly identify when it should be used with respect to existing MAS methods and MAS meta-models. It is also necessary to be able to identify the requirements that are tackled either by the macro-level design, those tackled by the micro-level design, those that pertain to the business concerns, those that pertain to the operative concerns and how they all impact the architecture.
- Then, MASs are systems that conform to a family of architectural styles where agents — the components —, that can dynamically be created and destroyed, are interacting together and with their environment. For that they use interconnection mechanisms through a runtime platform without being directly connected to each other. This introduces the question of how to define such interconnection mechanisms in order to be able to instantiate agents and connect them to their platform, how to implement them and how to make them reusable.

All of this should be answered in a way that the micro-level design results into an architecture usable as a development support adapted to the expertises of the MAS developers and their business concerns.

In the rest of this chapter, when using the terms agents and (runtime) platform we talk about the runtime entities of the system. As we are going to see, what we propose here are means to describe and implement such entities.

The present chapter is thus organised as follow.

First, we introduce a general methodological vision of MAS development. We introduce terms and concepts that are meant to facilitate communication between stakeholders during MAS development. This methodology gives a frame into which the rest of the contribution fits and identifies the different types of requirements that have to be tackled during MAS development. It is illustrated by being instantiated with the ADELFE development method. The example developed in this illustration also serves as a mean to illustrate the architectural and implementation challenges identified Chapter 2.

Second, we present a component model that supports the micro-level design and answers the identified technical challenges. Third, we propose a method that instantiate the methodology with this component model as a foundation. Then, in order to complete these three main contributions, we present how and what experience could be captured during the exploitation of the approach, and in particular how partially abstract architecture are realised. This manifests itself through reusable software artefacts and patterns.

Chapter 4 develops a complete application of the approach on a real world example, and Chapter 5 discusses the contribution with respect to the objectives and challenges.

3.1 Characterizing MAS Development: Architecture-Centric Methodology

In this section, we propose our vision of the methodology of MAS development. As explained Chapter 1, by methodology we mean the set of principles that pertain to MAS development in general and into which works on design and implementation methods from the field can fit it. Of course, the contribution that follows the presentation of these principles also fits in.

For a definition of *Methodology*, see p. 4

More precisely, in this chapter we present the following things that altogether constitute this methodology:

- We clearly identify the different phases that can exist when designing and implementing MASs and how they relate to each other.
- We provide clear terms to talk about requirements, design and implementation artefacts.
- We identify the different types of requirements and how they relate to the different artefacts produced in the development.

This methodology and the vocabulary that we present here are meant to facilitate communication among stakeholders in MAS development. It also serves as a basis to introduce Section 3.2 the SPEAD (Species-based Architectural Design) model and Section 3.4 SPEARAF (Species to Engineer Architectures for Agent Frameworks) method that together practically answer the challenges collected previously.

We conclude this section by illustrating the methodology with the ADELFE method and illustrate some of the technical challenges identified previously.

3.1.1 Multi-Level Architectural Design

Figure 3.1 shows the kind of process followed in MAS development as considered by this methodology of development. We now describe the different important phases, roles and elements that constitute the methodology used in this process.

As stated previously, during MAS development, requirements are expressed about the software system to develop. These **initial requirements** describe what the solution should comply to.

We introduce two types of views when designing a MAS architecture, each of them taking into account two different subsets of these requirements. These views take an orthogonal stance compared to other architectural views and complete them to describe what interest us here. In the rest of the chapter, when and if needed, we will make it clear which produced design artefacts concern which architectural views.

For a definition of the different *Architectural Views*, see p. 7

The first type of views, called the **macro-level architectural view**, is concerned with expressing the result of the design of the MAS using existing development approaches and methods. Most of the time, this result in the definition of types of agents exploiting interaction means, their behaviours, their states, the elements of their environment, etc. Each method,

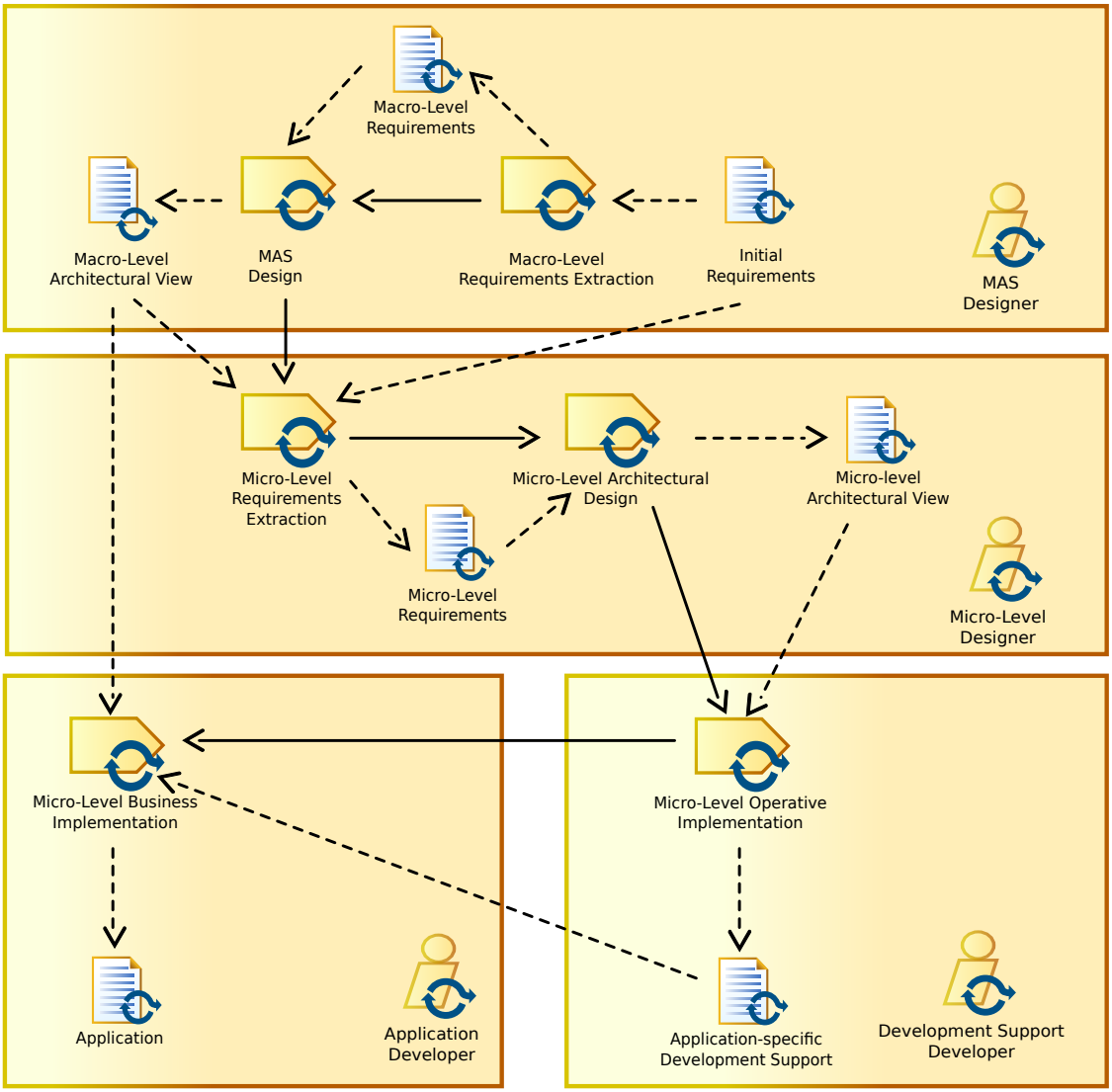


Figure 3.1: Abstract process followed by MAS development in general, described using SPEM

and its MAS meta-model, has its own way of instantiating such a type of view. This view is produced by the MAS developer as a **MAS designer**.

This view records the architectural decisions that answer what we call the macro-level requirements. By definition, the **macro-level requirements** are the requirements that are answered by the MAS design itself. Obviously, depending on the approach or the method, for a same problem, the requirements that can be answered won't be the same. Most of the time, they at least contain the functionality of the system to be produced, quality attributes such as adaptivity, scalability, performance and others that are known to be easily tackled by MAS approaches.

The second view, called the **micro-level architectural view**, is the view introduced here and which is mainly studied in the rest of this chapter. This phase is concerned with expressing the result of the design of the micro-architecture, which is the name we give to the architecture of the system taken from a micro-level architectural point of view. This view is produced by the **micro-level designer**. The differentiation between the macro-level and micro-level views reflects both a **separation of concerns** and a **sequencing of activities** during MAS development.

The micro-level view refines the macro-level view by following the two types of refinements: it decomposes some of the elements of the system, such as the agents with their internal architecture, and details others and their links, such as the interconnection mechanisms and the environment with the platform. This view records the architectural decisions that answer what we call the micro-level requirements.

For a definition of *Refinement*, see p. 8

Micro-level requirements are the requirements that appear to be not yet answered after the MAS design has been done. We can distinguish two different sources for these requirements. The first one is the set of initial requirements, and the second one is the MAS design itself, *i.e.* the macro-level view. This is illustrated Figure 3.2 that we now detail.

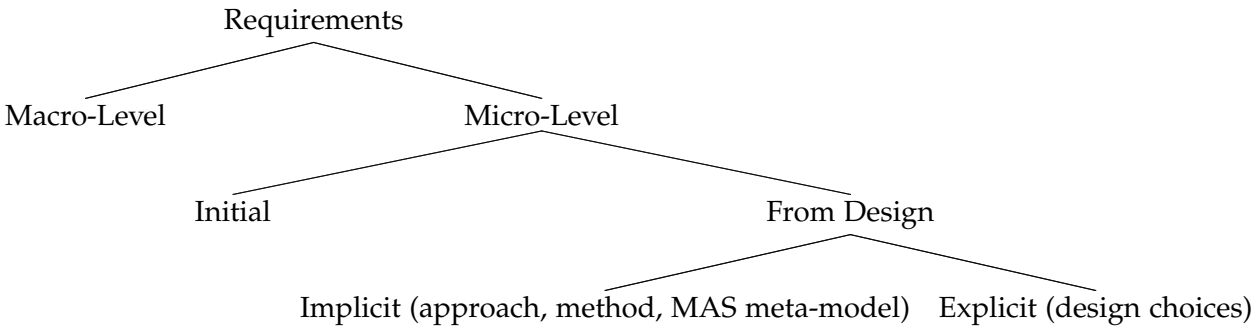


Figure 3.2: Different sources for requirements

On one hand, initial requirements are requirements that must be answered by the application to be built. As we explained previously, a subset of these are tackled by the macro-level view, as macro-level requirements. However, the rest of them are taken care of during the implementation of this design, for example matters such as GUI, connection to external

systems, but also organisational matters such as modifiability or testability requirements.

On the other hand, the MAS design itself introduces many requirements that impact its implementation. For example, the choice of specific interaction means, the way agents are meant to be executed internally or at the system level, the contents of the environment, etc. These requirements can come from the choices of the designer, or can come from the method that was followed. For example, some methods impose that agents have a perceive-decide-act lifecycle, while others only rely on the fact that agents are exchanging asynchronous messages. In the latter case, designers then can choose the way they describe the behaviour of their agents, and thus also impact the way it is implemented.

All of these are requirements taken care of after MAS design has been done, *i.e.* after the macro-level architectural view has been elicited.

3.1.2 Operative and Business Concerns

Orthogonally to these previous matters, in order for the implementation of the system to be eased, we propose to take into account the fact that MAS development has other methodological implications. First, MAS developers are not experts in the matters of the micro-architecture and want to focus on their business and macro-level design. Second, there is an important use of agile and iterative development.

Both concerns result on the following facts:

- The built architecture have parts that are likely to change during development, reflecting changes in the macro-level design, and thus changes in the micro-level requirements.
- These parts must be easily programmable, *i.e.* using adequate abstractions, for the MAS developer and his concerns.

Thus, in order to have adequate abstractions and reduce the impacts on the micro-architecture of changes in the requirements during development, we propose to distinguish the micro-requirements that are likely to change during the implementation — they are those manipulated by the MAS developer — from the requirements that won't. This results into two sets of elements and choices in the micro-architecture that reflect the **separation of concerns** between, respectively, the implementation of the MAS design by the MAS developer and the implementation of what supports it. The idea is to build a micro-architecture that is flexible enough to ease its modification during the life of the application. The question to answer is which requirements are they and how they manifest themselves in the architecture. Obviously, the question of the modifiability quality attribute of an architecture is well studied in the software architecture field, but we focus here on what concerns MAS development and its identified particularities.

As we said earlier, the design resulting from applying a MAS method introduces a set of design decisions. But the design also relies on a quantity of implicit and explicit design constraints and decisions emanating from this very same method.

Thus, some micro-requirements are deeply related to the MAS method used and are not meant to change. The method enforces some constraints on the way the design is made and is

meant to be a fixed point in the design of the system. Furthermore, other micro-requirements are related to the initial requirements that were not tackled by the design: as requirements coming from before the MAS is designed, they are not meant to change either from the point of view of the MAS designer.

Then, apart from these said micro-requirements, what are left are all the requirements introduced by the MAS designer in the macro-level view. And those that are likely to change can be found among them. They are for example requirements about the behaviour of the agents. We call these **business micro-level requirements**, while all the other are the **operative micro-level requirements**. In particular, the second set is supporting, but also constraining, the first set. This is illustrated Figure 3.3.

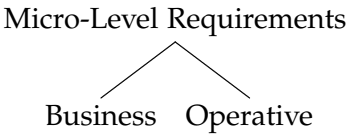


Figure 3.3: Different types of micro-level requirements

While micro-level requirements can be seen as a refinement of the initial requirements, in the context of the micro-level view, business and operative requirements are two distinct categories of requirements that can be used to drive the definition of a micro-architecture usable as a development support dedicated to the application being developed.

In order to materialise these two different sets of requirements in the micro-architecture, the basic idea behind this point is to find and define what are the elements of the architecture that pertain to each of them. The bridge between the two sets of concepts is the idea of programming abstraction: the requirements that are meant to evolve should be expressible as directly as possible using some abstractions.

For a definition of *Programming Abstraction*, see p. 3

For example, if one of the business requirement is to implement the agents' behaviour in terms of messages and sequential handling of received messages, then the programming abstractions must be usable to implement the behaviour in terms of a function taking a new message as input, and must provide a way to send messages in its body. A stated constraint on this abstraction is that this function is called for every message, and that messages are treated sequentially and indefinitely.

We show Section 3.4 how this can be done using component-based architectures and the component model we propose.

3.1.3 Implementation

During implementation, we differentiate:

– The **application developer**, which is often the same person as the MAS developer, that implements the business elements of the architecture using the dedicated programming abstractions.

– The **development support developer** that implements or reuses the operative elements of the architecture.

We don't detail the implementation phases themselves as the design must be sufficiently detailed so that implementation is straightforward. Of course, as we are going to see, the model and process we propose in this chapter are made with that in mind.

3.1.4 Illustrating the Methodology

In order to illustrate the different parts of the proposed methodology, we study how the ADELFE (*Atelier de Développement de Logiciels à Fonctionnalité Emergente*) method can be framed into it. At the end of it, we take the opportunity of this example to illustrate a bit more the architectural and implementation challenges.

We take an example about foraging ants that was developed to present the method in Rougemaille et al. (2009). This paper does not detail some of the requirements and activities: for these we fill the blank using information gathered from the authors. Indeed, the main idea here is that the application of the method only covers part of the presented methodology and some important points must be taken into account in order to implement it completely.

3.1.4.1 Applying the ADELFE Method

ADELFE is broken down in four main phases:

1. Preliminary and Final Requirements: gather requirements and constraints for the application to build, user needs, etc. It is needed to identify the environment in the system and the different elements that populate it, as well as the different use cases for the user of the system. In the case of this example, we take them as they are presented and we organise them in different categories:
   - Requirements
     - **R1**: Develop a simulation of foraging ants to provide a tool for ethologists.
     - **R2**: Evaluate the AMAS (Adaptive Multi-Agent System) approach for defining the behaviours of the system (sic).
     - **R3**: The colony must bring food to the nest as efficiently and quickly as possible.
   - Constraints
     - The environment of the ants is composed of a nest, obstacles, pheromones, patches of food and ants.
     - Pheromones accumulate and self-evaporate with time.
     - Ants can deposit and sense pheromones.
     - Ants have different degrees of perception for obstacles, ants, food and pheromones, but they always know where the nest is located.
     - Ants can carry food in a limited quantity.

    - Ants can be a limited time outside and must rest at the nest.
    - Ants explore the environment, avoid obstacles and harvest food when they find some.
  - Use cases (those are not explicit in the paper)
    - The state of simulation must be viewable in a GUI.
    - Ants creation parameters can be changed by the user of the GUI.
    - The simulation can be stopped, run step by step or accelerated.
  - Entities
    - Passive: obstacles, food and nest
    - Active: ants, pheromones (evaporate)
  - Environment
    - Non-deterministic, accessible (its complete state is viewable for simulation purpose), discrete (grid), dynamic (it is in particular modified by the ants and the pheromones evaporate)
2. AMAS Analysis: determine if the AMAS approach is adapted to the problem and identify which entities can be agents. We will just skip the details of this phase as it is not interesting for us. This phase is also where we identify that ants are actually agents as defined by the AMAS approach: conceptually distributed autonomous interacting entities with limited perception and range of action in an ever changing environment and with an individual goal (harvest food).
3. Design: define the cooperative behaviour of the agents, here the ants, and the behaviour of the environment (for example pheromones evaporation). ADELFE uses a model-driven approach based on a DSML (Domain Specific Modelling Language) named AMAS-ML and UML. The result of this phases is:
  - A diagram called agent diagram that represents the internal structure of the agents in terms of representations, characteristics, skills, aptitudes, actuators, sensors and communication. The relations of this structure to the passive entities of the environment is also given (for example the "mandible actuator" of the ants is connected to "food"). At this point, already some information about how to implement and store data in the agents is given, but only from an internal point of view. Also, representations are implicitly meant to be inferred from percepts, characteristics are intrinsic to the agents but depending on the problem can be modified by the environment or the agent itself, skills and aptitudes are mechanisms that the agent can use internally while actuators, sensors and communication are meant to interact with the outside.
  - A set of behavioural rules decomposed in standard behaviour and cooperative behaviour. These rules are expressed in terms of the agent state at a given time and the actions to undertake when its conditions are met. These conditions are expressed using the previously presented available mechanisms of the agent.

  - Sequence diagrams for detailing how the different mechanisms work and interact with the environment and between agents.

4. Implementation: from the previously defined meta-models, the internal code structure of the agent is automatically generated, the behaviour too and only the different elements of the internal structure such as the implementation of the representations, characteristics, skills, aptitudes, actuators, sensors and communication have to be done. Moreover, the environment itself, the GUI and the interaction means also have to be implemented.

3.1.4.2 Framing ADELFE in the Methodology

Based on this, we can now discuss how ADELFE fits into the proposed methodology.

The requirements, constraints and use cases presented are our initial requirements.

A subset of them is clearly tackled by the method as the macro-requirements, in particular the requirements *R2* and *R3*, the constraints, the entities and the environment.

Already we can spot some future micro-architectural requirements: the use cases and the requirement *R1*. They of course have a partial impact on the MAS design itself, for example for defining which characteristics of the agents must be represented. For example, the fact that it is a simulation and that it must be viewable means that the environment has to be accessible. But there is more to it, for example building the GUI and putting in place all the infrastructure needed to access the state of the environment, the agents, etc.

The macro-level architectural view is clearly the result of ADELFE design step, although it contains too much details about implementation.

From this view, we can actually identify micro-level requirements that were not previously defined:

1. The AMAS approach dictates that agents actually have the following cycle: first perception to update the representations from the percepts, then decision using the behaviour rules to choose which action to do, and then action to actually apply them.
2. Based on that, it means that the parts that are meant to be highly modified are the perception and the rules.
3. The agents have to be scheduled, one choice is to have a global clock and agent execution synchronised, for example one point of synchronisation for each of the step of the agent cycle to ease concurrent modification of the environment.
4. There must be a way to access the internal state of the agents such as the quantity of pheromones and food they have, or other internal informations useful for the system observer.
5. Inversely, there should be a way to modify some of the characteristics of the agents for the ethologists to test different configurations.
6. The environment must implement all of the entities and make their state viewable (such as the quantity of pheromone in one place).

7. The clock must also control pheromones evaporation, for example after agents have acted and before they can perceive.

By making explicit all these micro-level requirements, we avoid taking care of important requirements too late. Of course, it mostly means that some requirements should actually be made explicit by the MAS method used instead of staying implicit like that.

In terms of business requirements, for the developer of the application, it seems the second micro-level requirement is of first importance. The implementation and modification of these parts must be easy.

The micro-level architectural view can then be defined based on that information. The paper does not detail so much the equivalent of this view, but in practice the implementation for the managing the environment as well as the interconnection mechanisms were done by hand. On the other hand, the structure of the agent is automatically generated as well as the code for the rules. This last point concurs with the fact it is one of the business requirements. By exploiting code generation, this illustrates how iterations between design and implementation can happen, while the operative parts of the micro-architecture don't change.

3.1.4.3 Illustrating the Technical Challenges

The identified micro-level requirements shows the complexity in implementing all the interconnections existing between the agents of the system and the runtime platform, in particular the environment managing the pheromones. In particular, a point worth noticing is the interdependencies that exist between the interaction mean to move and the interaction mean to deposit pheromones. Indeed, the latter relies on the position of the agent to implement pheromone depositing. At implementation time, it could first mean to have them implemented together, which doesn't promote reuse, since the move interaction mean could be used alone. It could also mean to implement the clear separation of concerns between the two in the agents (since the position is proper to each agent) and in the environment (since the pheromones are deposited in it).

Such questions also arise for visualising agents and their environment, or for the dependencies that can exist between pheromones evaporation and scheduling of the system.

There is thus needs to manage such complexity and to explicit dependencies between interaction means and more generally for all the interconnection mechanisms. And these needs must be answered both at the agent and platform levels, for mechanisms that exist in-between the two, that must be reusable and composed together easily. This is tackled by the component model we present next.

3.1.5 Conclusion on the Methodology

This methodology is meant to be instantiated by existing methods for MAS development. Current methods mostly focus on the macro-level architectural view, which is not surprising

since this is the main challenge in the MAS field, because these methods mostly do not take a complete architectural stance and of course also because this distinction wasn't coined before.

We now present a component model to ease the design and implementation of the micro-level architectural view and then we continue with a method that relies on this model to guide this design.

3.2 Component-Based Micro-Level Architectures

In order to design and implement the micro-architecture of MASs, we propose to use a component model that answers the technical challenges identified Chapter 2. We recall these here:

- Define and implement interconnection mechanisms for connecting agents to the runtime platform and vice-versa.
- Define and implement types of agents with specific internal architecture and interconnected to the runtime platform.
- Provide dynamic creation of instance of these types of agents and their dynamic connection and initialisation.
- Make the reuse of such interconnection mechanisms and architectures easy.

At runtime, the differentiation between the **agents** and the **runtime platform** executing them is of first importance: the macro-architectural view is made of agents interacting together through an environment, and when refining this view with the micro-architectural view, it is necessary to keep this separation explicit in order to retain the property of the MAS design in terms of interaction, autonomy, dynamics, etc. As we highlighted in the previous chapter, this has an impact on the way to organise the micro-architecture so that the dynamic creation and execution of the agents is possible. In terms of runtime entities, each agent has its own architecture: they are instantiated by and connected to the platform architecture so that interactions between the agents and the environment and between them are possible.

Some ideas are given about how this model is used to support MAS development, but the details are presented Section 3.4 on the SpEArAF method.

This section presents the SpeAD (Species-based Architectural Design) component model incrementally: we start by defining a base component model — named SpeAD$^-$ — resembling existing component model, and then we introduce on top of it the specific abstractions proposed in this work. This has two advantages: first this eases the presentation of the new concepts, second this helps to easily identify the novelty in the contribution.

For a presentation of the relations between *Component Models, ADLs and Component Container*, see p. 6

The general plan followed to present each of the aspects of the model is as follow. First, we present how elements are described used SpeADL (Species-based Architectural Description Language), an ADL we propose. Second, we present how these elements can be implemented using the Java programming language by relying on code generated from the ADL. Finally, we present how these elements can be instantiated and are executed by the component container, which is completely constituted of the generated code.

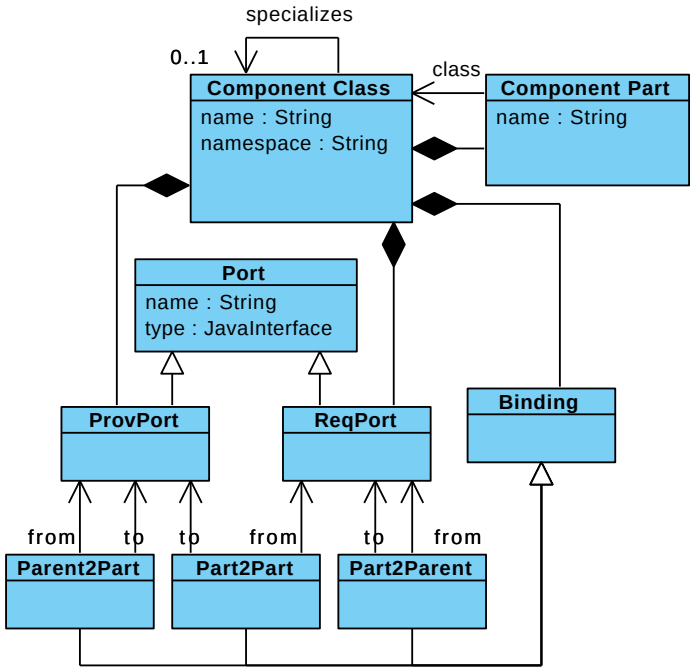


Figure 3.4: Meta-model of the SpeAD$^-$ component model

To situate these produced software artefacts, we can consider component descriptions with their implementation as modules views on the system. Even though compositions of components resemble components and connectors views and are actually usable to document them, they still are implementation units.

For a definition of *Modules Views*, see p. 7

All the concepts that we don't define here are those traditionally admitted in ADLs (Medvidovic and Taylor 2000). In particular we consider a component as an element that can be composed with others inside architectures, itself considered as a composition of components. Such compositions take the form of components called composites. We only introduce a distinction between classes and types of components: a class of component is defined by its description and its implementation. The description plays the role of a type, but also of implementation in the case of composite components. Moreover, depending on the context, if we are talking about implementation or runtime, a component is either considered as a class or an instance.

For a definition of *C&C Views*, see p. 7

Finally, this section focuses on using the component model, details about its implementation are given Appendix B.

3.2.1 The SpeAD$^-$ Base Component Model

The base component model that we present in this section is depicted Figure 3.4. It shows the different elements a software architecture described with SpeAD$^-$ can consist of. The ADL (Architecture Description Language) for SpeAD$^-$ is SpeADL$^-$.

```
import fr.irit.smac.example.interfaces.*

namespace fr.irit.smac.example.client {
  import java.lang.*

  component Client {
    provides runClient: Runnable
    requires serverOps: ServerOperations[String]
  }
}
```

(a) Client component description.

```
import fr.irit.smac.example.interfaces.*

namespace fr.irit.smac.example.server {
  component Server[Param] {
    provides ops: ServerOperations[Param]
  }
}
```

(b) Server component description.

```
package fr.irit.smac.example.interfaces

public interface ServerOperations<T> {
  public void print(T message);
}
```

(c) `ServerOperations` interface.

```
namespace fr.irit.smac.example {
  import java.lang.*

  component CSComposite {
    provides runApp: Runnable = client.runClient

    part client: client.Client {
      bind serverOps to server.ops
    }
    part server: server.Server[String]
  }
}
```

(d) Composite component description with parts and bindings.

Figure 3.5: Components descriptions in SpeADL$^-$ and interfaces description in Java

Note: to define this model, which serves as a basis to introduce the contribution, many design choices were made. We got ideas from other component models, our practical needs and personal preferences. The idea behind it was to ease the development of component-based architecture with a focus on flexibility, ease of use and reuse. Ideally, it could be replaced by an existing component model that has proved its maturity.

Architectures defined with SpeAD$^-$ are made of components connected together with simple connectors. The components externally provide ports, for which they have an implementation, and require ports, that they can use in their implementation.

3.2.1.1 Component Description with SpeADL$^-$

In SpeADL$^-$, a component description is located in a namespace and is composed of a name, a set of type parameters, a set of provided ports, a set of required ports and possibly a configuration. The namespace has the same role that packages have in Java. The component name uniquely identifies a component description in the namespace. For example, Figure 3.5a shows a component named `Client`, residing in the namespace `fr.irit.smac.example.client`, without any type parameters and with one provided port and one required port.

A port is described with a name (that uniquely identifies it in the component description) and refers to an interface type. Theoretically, an interface type is described with a name

and a set of operations, in practice interface types are represented using JAVA interfaces. For example, Figure 3.5c shows the description of a generic interface used Figure 3.5a. In the latter, the type argument of the interface is bound to the `String` JAVA class.

The type parameters of a component description can be used as arguments of the interface type or directly as an interface type. For example, Figure 3.5b shows a component with a type parameter named `Param`, which is passed as an argument to `ServerOperations`.

The configuration of a component is made of parts connected together with simple call-return connectors[1]. A part is described with a name (that uniquely identifies it in the component description) and refers to a component description by its name. The type parameters of the component description can be used as arguments to its parts component descriptions. For example, Figure 3.5d is a component named `CSComposite`, it has one part named `client` whose component description is `Client`, and a part named `server` whose component description is `Server`.

For each of the required ports of a part a binding must be defined. It connects the port to either the provided port of another part, or to a required or provided port of the component description. Moreover, the provided ports of the component description can be delegated to the provided ports of one of its part. All the bindings must respect the types of the ports by following the way type conformance is done in JAVA (including subtyping).

For example, the part `client` has one binding to the `server` part named `server`. It also shows provided port delegation to a part such as with `runApp`.

Finally, it is possible to refine component description using a simple specialisation mechanism. Only component descriptions without parts can be specialized. Component description specializing others can only add parts and bindings but no new ports can be added to the description. When specializing, the previously defined provided ports can be delegated to the added parts, and bindings can refer to them. The idea is that a specialization does not change the external visible properties of the component, but only its internal configuration. The objective of this mechanism is to be able to specialise a component description with different composite components. Specialization does not have any implication on implementation except for the fact that implementation of a component specializing another one can be used in place of the latter.

For example, Figure 3.6 shows a specialization of `Client` by `ClientComposite`, which contains two parts. One of them is bound to one of the required ports defined in the description of `Client`. An implementation of `ClientComposite` can thus be used in place of a an implement of `Client`.

3.2.1.2 Component Implementation with Java

Descriptions of components made with SPEADL$^-$ are then translated to JAVA. The objective of this translation is to be able to implement the component directly without any difference between what was described and what is available in JAVA. Thus the generated code from a description has two important characteristics:

1. The model does not support the definition of custom connectors.

```
component ClientComposite specializes Client {
  provides runClient: Runnable = one.p1

  part one: ComponentOne {
    bind p2 to two.p1
  }
  part two: ComponentTwo {
    bind p2 toThis serverOps
  }
}
```

```
component ComponentOne {
  provides p1: Runnable
  requires p2: Runnable
}
```

```
component ComponentTwo {
  provides p1: Runnable
  requires p2: ServerOperations[String]
}
```

Figure 3.6: Component specialization in SpeADL$^-$

- It has the exact same semantics[2] than the description and acts as an invisible bridge between description and implementation.
- It eases the implementation by well integrating it into the Java language and exploiting the type system to catch errors as early as possible.

In this section, we only describe what is needed to understand how components are implemented. The details of the internal working of the component model and of the transformation from SpeADL (and thus SpeADL$^-$) to Java is described Appendix B.

Each component description becomes a Java class, that we call the description class, with the same name, type parameters and namespace (package in Java). The general idea behind this translation is that the implementation provides a set of methods. These methods return instances of the implementation of the different elements of a component (parts and ports). These instances are used to construct an actual instance of the component.

In the description class, each of the provided ports (that are not already delegated to another port) is translated to an abstract method without parameters named after the port and prefixed by `make_`. The method returns an instance of the type of the port. Moreover, to access the provided ports from within the implementation, a protected method named after the port and returning an instance of the type of the port is also present. To access the required ports, each of them is translated to a protected method without parameters named after the port and returning an instance of the type of the port. Each of the parts is translated to an abstract method without parameters named after the part and prefixed by `make_`. It returns an instance of an implementation of the component description of the part. Moreover, to access the parts from within the implementation, a protected method without parameters named after the part and returning an instance of the `Component` interface of its class description is present. Finally, the description class provides a method (which is not abstract) named `start` that can be overridden to implement initialisation logic.

2. Even though this semantics is limited to the structure and the types of the ports.

```java
package fr.irit.smac.example.client.impl;

import fr.irit.smac.example.client.Client;

public class ClientImpl extends Client {

  private final String name;

  public ClientImpl(String name) {
    this.name = name;
  }

  // abstract method of the parent class
  @Override
  protected Runnable make_runClient() {
    return new Runnable() {

      @Override
      public void run() {
        System.out.println("client before");
        // using the required ports
        serverOps().print("message from "+name);
        System.out.println("client after");
      }
    };
  }
}
```

(a) Client using required port.

```java
package fr.irit.smac.example.server.impl;

public class ServerImpl<T> extends Server<T> {

  @Override
  protected ServerOperations<T> make_ops() {
    return new ServerOperations<T>() {

      @Override
      public void print(T message) {
        System.out.println("server before");
        System.out.println("got: " + message);
        System.out.println("server after");
      }
    };
  }

  @Override
  protected void start() {
    super.start();
    System.out.println("server start");
  }
}
```

(b) Server with type parameters and initialisation logic.

Figure 3.7: Component implementations in Java

Thus, to provide an implementation for a component, one extends this description class. The abstract methods, corresponding to the provided port and the choice of the implementation of the parts, must be implemented, and the `start` method can be overridden if needed. In these implementations, the implementer can use the required ports and access the parts and their provided ports through the protected methods. While the constructor of the implementation must not rely on the required ports and parts, the start method can. If the component must have parameters passed to it at initialization, then they must be added to the implementation.

For example, Figure 3.7a shows an implementation for the component `Client` presented Figure 3.5a. The class `Client` is the description class that has been generated. In order to implement its provided port, a method overrides an abstract method of the latter class. It returns an implementation for the port, here of type `Runnable`. We can notice the use of the required ports in the body of the implementation of the provided port. The implementation also takes a parameter for this instance. Figure 3.7b shows how the `start` method can be overridden and how type parameters translate to Java.

Figure 3.8 shows an implementation of the composite component `CSComposite` presented Figure 3.5d. The method `make_client` is overridden to specify the implementation of the part, where `ClientImpl` is an implementation for the component `Client`. We can also notice the type parameter of `Server` which is bound to `String` as within the description, and the

```
public class CSCompositeImpl extends CSComposite {

  @Override
  protected Client make_client() {
    return new ClientImpl("a simple client");
  }

  @Override
  protected Server<String> make_server() {
    return new ServerImpl<String>();
  }
}
```

Figure 3.8: Component implementation in Java with parts

```
// instantiation of a component implementation
CSComposite.Component compo =
  CSComposite.newComponent(new CSCompositeImpl());

// starting the component
compo.start();

// using a provided port
compo.runApp().run();
```

(a) Program in Java.

```
server start
client before
server before
got: message from a simple client
server after
client after
```

(b) Output.

Figure 3.9: Component instantiation and usage

parameters for the instance of the component `ClientImpl`.

3.2.1.3 Component Instantiation and Runtime Behaviour

While the description class corresponds to the set of the implementations for the component, an interface named `Component` corresponds to the set of the component instances themselves. This interface is present as an inner interface of every description class. At runtime, the instances of the component description are of the type of this interface.

For a definition of *Component Container*, see p. 6

Only components without required ports can be instantiated by hand from Java. The generated code acts as the container of the components and takes care of all the wiring described with SpeADL$^-$. In particular, the description class provides a static method that, given an implementation, returns an instance of its `Component` interface (which only exposes the provided ports). This instantiates recursively all the parts of the component.

Figure 3.9a shows how the implementation of the component `CSComposite` is instantiated and used. Figure 3.9b shows the output of this program.

After a component has been instantiated, it is in an initialised state: it means that all the parts of the components were initialised (the constructors of their implementation were called and instance of the components were created) as well as all the provided ports. At that point, all the wiring is done. The component can then be put in a started state by calling

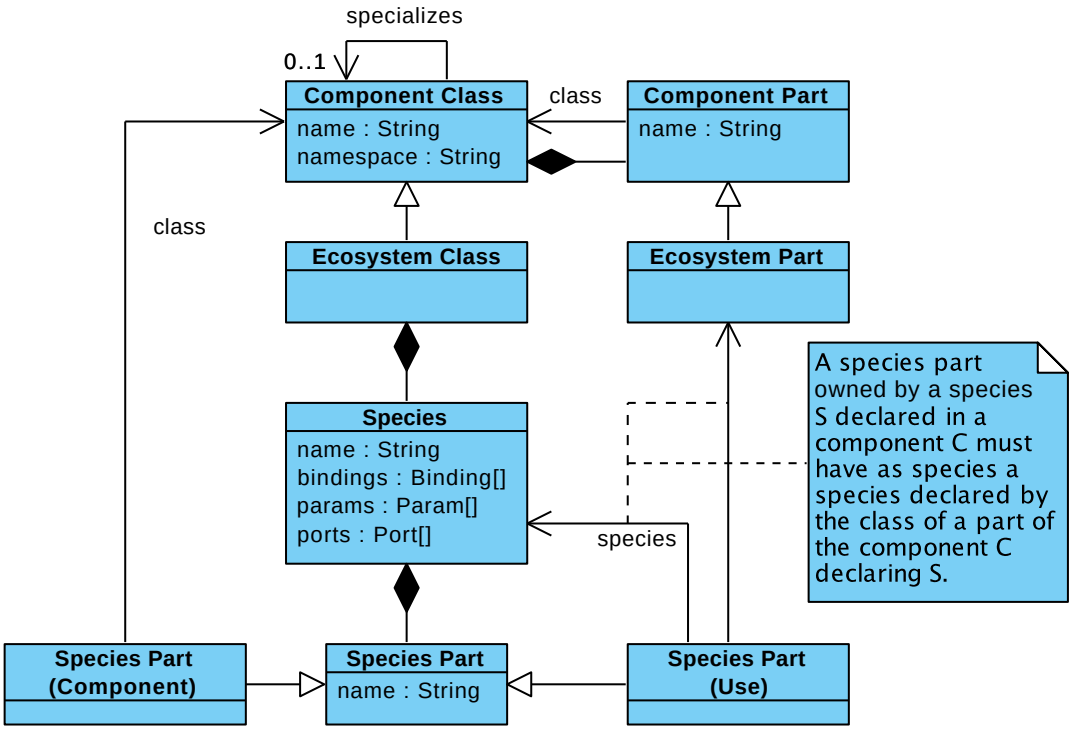


Figure 3.10: Meta-model of the SpeAD component model

its `start` method. This one starts each of the part and then call the `start` method of the implementation.

This state only exists conceptually but does not have any existence except for the fact that the `start` method is called and the provided ports of the component can then be accessed.

3.2.2 The SpeAD Component Model

We now present the SpeAD (Species-based Architectural Design) component model that introduces the abstractions needed to tackle the technical challenges we presented. These abstractions are built on top of SpeAD$^-$.

The SpeAD component model, that we present in this section, is depicted Figure 3.10. The ADL (Architecture Description Language) for SpeAD is SpeADL.

We start with an informal definition of the concepts before moving to the SpeAD component model.

The main abstraction introduced is the species. A **species** is similar to a component description, but while the latter is made to be instantiated inside a static configuration, the former is made to be dynamically instantiated and connected from inside an already instantiated component. Thus instances of species are dynamically created: their ports are connected at runtime by the component container upon creation. Species can only be described inside other component descriptions: we call such components **ecosystems**, which

extend components of SpeAD^{-}. They are the only one capable of creating instances of the species they declare. The species instances then exist inside the instance of these ecosystems.

In the context of MASs, type of agents are described using species, while their ecosystem acts as their runtime platform. More on this matter is given Section 3.4 about the SpEArAF method. However, using this definition of species is not enough to cover the needs the contribution must answer. Instances of species must be connected to their ecosystem: we introduce an additional concept that realises such connection.

In order to understand this concept, we first present a simplified idea of its use and purpose. The idea is that when a species is instantiated, it is desired that its instance is connected to its ecosystem. At runtime, for the instance to be correctly connected, two things are important:

- The elements connecting the instance to its ecosystem must already exist in the ecosystem.
- These elements have to instantiate a part of themselves within the instance in order to establish a link between the two.

This element is then composed of two different "sides": one that is instantiated within the ecosystem, and one that is instantiated for every instance of the species described in the ecosystem.

In the context of MASs, interaction mechanisms and other links between an agent and its platform can be realised using such "two-sides" elements. For example, a mechanism for depositing pheromones as has a side in the ecosystem for managing the pheromones and their evaporation, and a side in the agent for handling its personal stock of pheromones. The side in the agent also has to be aware of the location of the agents in a virtual space, etc. The same is possible for operative interconnection mechanisms between the agents and their platform, such as GUIs for visualising agents, or global control of the execution of the agents. Such other examples are given in the rest of this section as well as Section 3.3.

Back to SpeAD, we can model such "two-sides" elements by using the concepts of species and ecosystem in a recursive way. In order to do so, we propose that when a species is defined, it can "**use**" species of one of the parts of its declaring ecosystem. This means that when an instance of a species is dynamically created, instances of all the species it *uses* are also dynamically instantiated recursively.

Thus, even though the term species was historically introduced as a mean to characterise a type of agent, it does not only refer to that idea. It must also be noted that the term ecosystem used here should not be mixed up with the concepts of *software ecosystem* (Bosch 2009): even if they share an ecological origin, our ecosystem is the product of the development and not the context into which the development happens and is constrained. From this point of view, the term ecosystem is well adapted to characterise the entity that defines species and is responsible of maintaining their instances "alive".

Based on these definitions, we now detail how species and their connections are described in ecosystem, how they are implemented and how they are instantiated and executed.

```
import java.util.concurrent.Executor

ecosystem Scheduler {
  species Agent {
    provides exec: Executor
    provides stop: Do
  }

  requires executor: Executor
}
```

Figure 3.11: Ecosystem description for an interconnection mechanism in SpeADL

As a general introduction, when realising such components, the developer focuses on defining ecosystems, species, providing implementations for the ports of the descriptions and choosing implementation for their parts. The mechanism of instantiation itself is managed by the container and can be directly used from within the implementation of the ecosystem.

3.2.2.1 Species and Ecosystem Description in SpeADL

Like a component description, an ecosystem description is located in a namespace and is composed of a name, a set of type parameters, a set of provided ports, a set of required ports and possibly a configuration. On top of that, it contains a set of species descriptions: only this last point differs from a component description in SpeADL$^-$.

A species description resembles a component description in SpeADL$^-$: it is composed of a name, a set of type parameters, a set of provided ports, a set of required ports and possibly a configuration, although the latter is different from those of SpeADL$^-$. A species configuration have parts, but of two different types: either they are normal part as in SpeADL$^-$, or they *use* and refer to a species description declared in one of the parts of its ecosystem. The bindings follow the same rules as in SpeADL$^-$.

Figure 3.11 shows an ecosystem for an interconnection mechanism that gives the agents internal architecture the possibility to execute tasks. Secondarily, it manages the set of tasks of each agent — which justifies the fact it is modelled using ecosystem-species abstraction — in order for them to be able to stop all their own running tasks. When using such an ecosystem into another one, as in Figure 3.12, the species declared in the latter can *use* the species of the `Scheduler` in order to use the interconnection mechanism it realises. For example `SimpleAgent` *uses* the `sched.Agent` species and binds to it one of its part required port. Any instance of the `SimpleAgent` species can then manage its own set of running tasks.

Next section clarifies some of the advantages of the model with the implementation of `Scheduler`.

Contrary to the implementations of parts in SpeAD$^-$, in SpeAD passing parameters to species implementations can not be handled at implementation time. Since the implementations of species are only declared in the implementation of their ecosystem, when a species *uses* another species it does not manipulate directly its implementation. Thus, it is necessary

```
ecosystem MasExample[Msg] {

  provides send: Send[Msg, DirRef] = msg.send
  provides factory: SimpleFactory[Msg]

  part msg: MessageDirectRef[Msg]
  part sched: Scheduler {
    bind executor to executor.exec
  }
  part executor: ExecutorService

  species SimpleAgent(beha: AbstractAgentBehaviour[Msg], name: String) {

    provides stop: Do
    provides me: Pull[DirRef] = msg.me

    use msg: msg.Receiver(name) {
      bind handler to seq.dispatch
    }

    use exec: sched.Agent

    part seq: SequentialDispatcher[Msg] {
      bind executor to exec.exec
      bind handler to beh.cycle
    }

    part beh: AgentBehaviour[Msg, DirRef] {
      bind send to msg.send
      bind die toThis stop
      bind me to msg.me
    }
  }
}
```

Figure 3.12: Ecosystem description for a simple MAS in SpeADL

to bring species parameters declaration at the ADL level[3]. Species declarations thus also have a set of parameters (with a name and a datatype, here represented using a Java type). These parameters are used when *using* species.

For example the species visible Figure 3.12 have parameters. Some of them are given to the species they *use* as arguments. More details are given how this impact the implementation in the following section.

No specialisation mechanism is introduced for the species. However, component descriptions can be specialised by ecosystem descriptions.

3.2.2.2 Species and Ecosystem Implementation in Java

The translation of SpeADL to Java follows the same approach than with SpeADL$^-$. It only differs on the way species are implemented and instantiated.

In the description class of an ecosystem, each of the species is translated to an inner class, which we call the species description class, named after their name. Each species description

3. It could already have been added in SpeAD$^-$ but wasn't in order to show why it was needed now.

```
public class SchedulerImpl extends Scheduler {

  public class AgentSide extends Scheduler.Agent {

    private final AtomicBoolean run = new AtomicBoolean(true);
    private final Set<Future<?>> s = new HashSet<Future<?>>();

    @Override
    protected Executor make_exec() {
      return new Executor() {
        public void execute(final Runnable command) {
          if (run.get()) {
            final FutureTask<?> f = new FutureTask<Object>(command, null);
            s.add(f);
            eco_executor().execute(f);
          }
        }
      };
    }

    @Override
    protected Do make_stop() {
      return new Do() {
        public void doIt() {
          run.set(false);
          for (Future<?> f : s) {
            if (!f.isDone()) f.cancel(true);
          }
          s.clear();
        }
      };
    }
  }

  @Override
  protected Agent make_Agent() {
    return new AgentSide();
  }
}
```

Figure 3.13: Ecosystem implementation in JAVA for an interconnection mechanism

class have to be extended to provide an implementation for the species. Each of the species is also translated in the ecosystem description class to an abstract method taking as parameters the parameters of the species, named after the species name and prefixed by `make_`. It must return an instance of an implementation for the species. This method gives the ability to the container to automatically create new instances of species.

A species description class is identical to a component description class except that it adds a way to access the provided ports, required ports and parts of its ecosystem, which are prefixed by `eco_`.

Figure 3.13 shows the implementation of the `Scheduler` ecosystem presented Figure 3.11. For each species, there is an overridden method with the same parameters as the species: it must instantiate an implementation of it. The class `AgentSide` implements for example the species `Scheduler.Agent`. In the specific case of this ecosystem, which realises an interconnection mechanisms, we can see how the ecosystem abstraction can be used: the

implementation exploits the fact that every time a species that *uses* the `Agent` species is instantiated, then an instance of the `Agent` species is also instantiated. This has two main advantages:

- The initialisation logic related to this particular interconnection mechanism can be encapsulated into the implementation without filtrating externally.
- The state related to each instance of this species is encapsulated in it.

Figure 3.14 shows a part of the implementation of the ecosystem `MasExample` presented Figure 3.12. Parts of ecosystem, such as `sched` here in the overridden `make_sched` method, are specified as any other component part and no specific boilerplate code has to be implemented on top of that for connecting it to a species. The overridden method `make_SimpleAgent` illustrates parameter passing to a species implementation. On the other hand, the implementation of the port `factory` shows how to actually create an instance of a species, again with the parameters.

3.2.2.3 Species and Ecosystem Instantiation and Runtime Behaviour

The instances of ecosystems follow the same runtime behaviour as components in SpeAD$^-$ and the instances of species are created from already started instances of ecosystems.

From within an ecosystem implementation, it is possible for every species without required ports to be instantiated using a protected method available in the ecosystem description class. This method takes as parameters the arguments for the species. Instances of species, which are components, can then be started and their provided ports accessed to.

When an instance of a species is created (using the available mechanisms from inside the ecosystem implementation) its parts are created as with components in SpeAD$^-$, and the species it *uses* are recursively instantiated in the same way using the instances of the ecosystem that declares them. At this point, the instance of the species is in the initialised state. Then, by calling its `start` method, all of its parts are started (species and non-species), and the instance is in the started state.

Concerning agent creation, Figure 3.14 illustrates different aspects of it for `SimpleAgent`:

1. The call to the method `newSimpleAgent` inside the implementation of the port `factory` is used to create an instance of the species, the creation of the species it *uses* is handled automatically by the container.
2. We can also see in this port implementation how the provided port introduced in the `SimpleAgent` species is useful to encapsulate the way agents are represented inside the ecosystem and only exposes its messaging reference (using the `MessageDirectRef` ecosystem, not detailed here, that realises message passing using direct references).

We presented the SpeAD component model that can be exploited using SpeADL and Java. We showed how to define ecosystems and species to ease the description of micro-architectures as well as their implementation. It is of course possible with this model to reuse produced code artefacts, this is the subject of the following section.

```
public class MasExampleImpl<Msg> extends MasExample<Msg> {

  // ...

  @Override
  protected Scheduler make_sched() {
    return new SchedulerImpl();
  }

  @Override
  protected SimpleAgent<Msg> make_SimpleAgent(
      final AbstractAgentBehaviour<Msg> beha, String name) {
    return new AbstractSimpleAgent() {
      @Override
      protected AgentBehaviour<Msg, DirRef> make_beh() {
        return beha;
      }
    };
  }

  private abstract class AbstractSimpleAgent extends SimpleAgent<Msg> {

    @Override
    protected Do make_stop() {
      return new Do() {
        @Override
        public void doIt() {
          msg().stop();
          exec().stop();
        }
      };
    }

    @Override
    protected SequentialDispatcher<Msg> make_seq() {
      return new SequentialDispatcherImpl<Msg>();
    }
  }

  private AtomicInteger n = new AtomicInteger(0);

  @Override
  protected SimpleFactory<Msg> make_factory() {
    return new SimpleFactory<Msg>() {
      @Override
      public DirRef create(AbstractAgentBehaviour<Msg> beh) {
        SimpleAgent.Component<Msg> agent = newSimpleAgent(beh,"agent"+n.getAndIncrement());
        agent.start();
        return agent.me().pull();
      }
    };
  }
}
```

Figure 3.14: Ecosystem implementation in JAVA

3.3 Capturing Reusable Experience

During the application of the contribution, several recurring components and patterns were identified.

This section has three main objectives:

1. Show how experience can be captured and reused in the context of the SpeAD model.
2. To illustrate the contribution presented until here.
3. To present a library of components and patterns usable in MAS development.

We first show what can be captured and how, then specific components and patterns are presented. In particular, we introduce a way to capture architectural patterns using component-based architecture templates.

3.3.1 What and How

When developing component-based software architectures, the abstractions that can be reused exist at different levels: interfaces, code and design.

3.3.1.1 Capturing Reusable Interfaces

First, at the finest grain, we have interfaces: they represent points of interoperability between components. Without common interfaces with common semantics, it is not possible to connect components together.

The reusable artefact is the **interface description**, obviously it is a very common practice in software architectures in general. In this work, the Java language is used as a mean to describe interfaces, which are used in component descriptions to specify the types of the ports.

On top of the syntactic description, interfaces described in Java can also be documented to clearly specify their semantics. When defining interfaces, a trade-off must be found between genericity and expressiveness: a too much generic interface has a different meaning in different situations, while a more specific interface has a clear meaning and semantics but isn't usable in a lot of situations. We don't detail how to do such things since they are out of the scope of this work.

3.3.1.2 Capturing Reusable Code

Then, at a bigger grain, we have components themselves: they represent reusable pieces of code that can be composed together in composite components. Reusable components are one of the motivations behind the introduction of software components in the software engineering field and are thus a common practice in software architectures.

For a definition of *Module View*, see p. 7

Reusable artefacts are **components descriptions and implementations**. Along with a component description and implementation, one can find datatypes and interfaces introduced with it. Altogether these artefacts constitute a module that can have dependencies towards

other components. Such dependencies can concern datatypes, interfaces or components defined in other modules.

In our specific case, this is done using the SpeADL and Java languages. When possible, they often exploit type parametrisation for enhancing genericity. Type parametrisation can be exploited in two different ways in SpeAD:

- For datatypes; the type parameters can abstract over the data manipulated by the component as it is often done in Java for example.
- For interfaces; the type parameters can also be used to abstract over interfaces. Obviously, an implementation of such a component has to concretise such types, and if not it cannot implement them. This reduces the possibility of use of the interface, but in specific cases it can be useful for engineering reusable components. In particular, the next section exploits this to describe templates. Advanced use of this approach includes using Java reflection to implement components that provide such interfaces.

3.3.1.3 Capturing Reusable Design

At architectural level, design can be reused: we mean a common composition of components, but described in an abstract way in order to be instantiated in specific situations. A common mean of capturing them in the software architecture field is using architectural patterns.

For a definition of *Architectural Pattern*, see p. 8

Most of the time, patterns are described using textual descriptions in a natural language. But in the specific case of component-based architectures, it is possible to have reusable implementations for patterns. Indeed, a pattern is a kind of composition of components where some of them are kept abstracts and other could well be already implemented. The patterns that interest us are about describing a common organisation of components for achieving a specific functionality or quality attribute. Ideally, we want to reduce the code to develop by reusing parts of architectures while letting other open to the user of the patterns.

We present two ways of doing that. The first one is based on producing partially abstract architectures and the second one is based on using required ports to represent abstract parts of the patterns.

Templates. We introduce the concept of **template**, which is specific to SpeAD. The idea here is to use templates as a mean to implement patterns in the form of partially abstract architectures. As a side note, this is different from templates in ACME (Garlan, Monroe, and Wile 1997) where component and connector templates can be instantiated in architectures and regrouped in styles.

Actually, a template is not an abstraction provided by SpeAD but a way of using the abstractions it provides. Here a template is a reusable abstract description and implementation of a composition of components: it has concrete components with implementations and other components that are left abstract. It is thus actually a component itself. A more detailed discussion of the relations between templates, patterns and other means of reusing design is given Chapter 6.

Templates are exploiting SpeAD in order to be able to abstract over components and interfaces (*i.e.* ports types): by combining the two it is possible to describe compositions where what we call abstract components with abstract interfaces are connected together.

Abstract components are defined as component descriptions with ports and no part. The interfaces of the ports are chosen in order to express the contract its implementation must respect. The programming abstractions that can be used to implement it are the required ports of the component.

When such abstract component is put with other components, this gives us a partially abstract architecture. Using type parametrisation, the types of the ports can be left abstract in order to define generic abstract components and thus templates.

Then, a partially implemented version of the template can be provided. Only the abstract components are left to be implemented — *i.e.* the corresponding methods are not overridden —, while the implementation of the others are provided.

Finally, instantiating the template is just a matter of extending the partial implementation for the template in order to specify the implementation of the abstract components. Implementing the abstract components can be done either directly in Java or by specialising the abstract component with a composite component.

Components. This approach can be put in opposition to the use of required ports to represent the components that must be abstracted over. The idea would be to represent the reusable implementation of the pattern with a completely implemented components. Then, to exploit such a component, one has to describe and implement a component that provides the port that would be composed with the reusable component. Compared to the previous approach, this allows for more flexibility and composability, since there is no constraint on the structure of the composition nor on the structure of the components to implement, but it is thus more complex to use. Instead of encoding a specific design using a composition, this design must be explained in the documentation of the ports. Moreover, when reusing complex composition, the relation between the different components must then be explained, and the user of the reusable composition must be careful to not go out of the tracks when using the component, while with templates, the user is constrained by the composition itself.

This last approach is actually more like component reuse than design reuse, but can achieve the same and is sometimes more adequate for reusing design. Indeed, design reuse with templates is quite limited because of the lake of specialisation support for ecosystem and species.

3.3.2 Components Library

We now present the practical interfaces, components and patterns that are reusable and available in the component library distributed with the implementation of the SpeADL language.

```java
public interface Send<T,R> {
  /**
    * Send message to receiver.
    * Asynchronous and not reliable.
    * Should not block.
    *
    * @param message the message to send
    * @param receiver the receiver of the message
    */
  public void send(T message, R receiver);
}
```

(a) `Send` interface.

```java
public interface ReliableSend<T, R>
  extends Send<T, R> {
  /**
    * Send message to receiver.
    * Asynchronous and reliable.
    * Should not block.
    *
    * @param message the message to send
    * @param receiver the receiver of the message
    * @throws AgentDoesNotExistException when
    * the receiver does not exist in the system
    */
  public void reliableSend(T message, R receiver)
    throws AgentDoesNotExistException;
}
```

(b) `ReliableSend` interface.

Figure 3.15: Interfaces descriptions in Java

3.3.2.1 Interfaces

To start, several interfaces were used during the development of components when applying our approach. We detail some of them here in order to ease the understanding of the rest of the chapter. Obviously this does not consist in a contribution by itself.

We can see two sets of interfaces : the very generic ones and the MAS-oriented ones. In practice, the generic ones are very close to functions types: for example we have `Push` with a method that can take an element as parameter, `Pull` with a method that returns an element, or `Do` with a method that takes no parameter nor return anything. This can be generalised to any number of parameters although we just had the need for those ones. Ideally, one would exploit as much as possible the language standard library to ease interoperability, but Java library is quite limited concerning this kind of need. Languages such as Scala [4] or libraries such as FunctionalJava (FJ) [5] are good alternatives. We chose to use FJ when needed.

The MAS-oriented interfaces are useful to describe messages sending, messages broadcasting, system clock controls, tasks executions, etc. For example we defined `Send`, see Figure 3.15a, to send messages to a receiver (the interface is parametrised by the types of these two datatypes) with the following semantics documented in the comments: the operation must be asynchronous and reliability is not guaranteed. We also defined `ReliableSend`, see Figure 3.15b, that extends `Send` with the following semantics: the operation must be reliable, *i.e.* failure of delivery means the receiver does not exist.

3.3.2.2 Components

We have then reusable components and again, some of them are mostly generic, while others are very specific to MAS.

4. http://scala-lang.org
5. http://functionaljava.org/

In particular, we found the following categories of components:

- Scheduling: in this category we have different components needed for scheduling a system, be it at the platform level or agent level. In the platform level, we have clocks to synchronise the agents of the system in their execution, GUI for controlling the clocks, executors as thread pools, etc. At the agent level, we have ecosystem and species to have a set of agents be scheduled together in a synchronised way or unsynchronised way as needed. The latter also takes care of stopping execution of agents when they are killed.
- Interaction: these components provide means for agent to communicate together, they implement message sending by direct references or by name, but also synchronous call by references, broadcasting (coupled with a mechanism for resolving receivers at the platform level).
- Distribution: here we have components for realising distributed platforms with serialisation, platform naming and referencing, etc.
- Agent dynamics: these components implement some common mechanisms used internally by agents, such a lifecycle like sequential handling of messages, or loops.
- Meta: these components provide some generic mechanisms that are useful to build other mechanisms.

All these components can be used to build architectures, but also to build templates that implement common patterns.

We now detail some components in order to clearly show how they are realised and what are the advantages of using the abstractions introduced with SpeAD. We present an agent dynamics component to execute sequentially a behaviour on messages that arrive concurrently, an ecosystem to give access to a port in an ecosystem to one of its species, two interaction ecosystems to reference agents and one interaction ecosystem to publish and observe values from agents. We conclude with an example of MAS with agents that can publish and observes values by being referenced with names.

Sequential Dispatcher. The first one is a component for handling sequentially messages in an agent. To achieve that, it is made of several parts: a queue to buffer messages arriving concurrently and a mechanism that takes elements (potentially messages in the case of a MAS) sequentially and forwards them to be processed until there is no more. This example shows the use of composite components, reuse of a generic queue component and type parametrisation.

Figure 3.16 shows the description of the component. It was used in the examples shown Section 3.2.2. It has a part that reuses another component for a `Queue` to store the elements. The `dispatch` port, with the interface `Push`, receives the element to sequentially process and implements the mechanism of dispatching by exploiting `queue`, and the two required ports. The `executor` port is needed for executing tasks concurrently (typically, this is bound to a

```
component SequentialDispatcher[Thing] {
  // to give elements to dispatch, can be called concurrently
  provides dispatch: Push[Thing]
  requires executor: Executor
  part queue: Queue[Thing]
  // to handle elements, guaranteed to never be called concurrently
  requires handler: Push[Thing]
}
```

Figure 3.16: Sequential dispatcher description in SpeADL

```
public class SequentialDispatcherImpl<Thing>
  extends SequentialDispatcher<Thing> {

  private AtomicBoolean working = new AtomicBoolean(false);

  @Override
  protected Push<Thing> make_dispatch() {
    return new Push<Thing>() {
      public void push(Thing t) {
        queue().put().push(t);
        if (working.compareAndSet(false, true)) {
          emptyQueue();
        }
      };
    };
  }

  private void emptyQueue() {
    final Thing t = queue().get().pull();
    if (t != null) {
      executor().execute(new Runnable() {
        public void run() {
          handler().push(t);
          emptyQueue();
        }
      });
    } else {
      working.set(false);
    }
  }

  @Override
  protected Queue<Thing> make_queue() {
    return new ConcurrentQueueImpl<Thing>();
  }
}
```

Figure 3.17: Sequential dispatcher implementation in Java

```
/**
  * Simply forward a port from
  * the agents to the infrastructure
  */
ecosystem Forward[I] {
  species Agent {
    provides a: I
  }

  requires i: I
}
```

(a) Description in SpeADL.

```
public class ForwardImpl<I> extends Forward<I> {

  public class AgentSide extends Agent<I> {
    @Override
    protected I make_a() {
      return i();
    }
  }

  @Override
  protected Agent<I> make_Agent() {
    return new AgentSide();
  }
}
```

(b) Implementation in Java.

Figure 3.18: Forward component

thread pool at the platform level or other mechanisms to execute tasks) and the `handler` port is the port to which the elements to be processed are given.

Figure 3.17 shows the implementation for the component. As it can be seen, the part can be accessed as well as the required ports. The implementation chosen for the `Queue` is a concurrent queue in order to respect the specification of the `dispatch` port.

Forwarding from a Species to its Ecosystem. Before presenting interesting components for interconnection, we show first a component for enabling an instance of a species to access a port in its ecosystem.

Figure 3.18a shows its description. A species that uses the `Agent` species of `Forward` is able to access, thought the port `a`, whatever is connected to the port `i` in the ecosystem. The implementation is presented Figure 3.18b. We can notice that under certain conditions, it is possible, when making a port in the implementation of a component (here `Agent`), to directly return another port: this is only possible because the port `i` was already initialised in the ecosystem.

Another example of component for interconnected is the `Scheduler` component presented Section 3.2.2.

Referencing Agents. When agents interact, they need ways to refer to each other. There exist different ways to reference agents in MASs, we implemented two of them: direct references and named references. The idea is to be able to use either of them with interconnection mechanisms such as those presented next.

Figure 3.19 shows the description of the `DirectReferences` ecosystem. It defines a species that can be used in a species to denote that instances of the latter have a reference. The `me` port gives an agent access to its reference, the `stop` port is needed to execute dereferencing of the agent. The `toCall` port is the port that is accessible through a reference using the `call` port available in the ecosystem (and possibly accessible to an agent using the previously presented

```
ecosystem DirectReferences[I] {

  // name is only used for pretty
  // printing the reference with toString()
  species Callee(name: String) {
    // a port provided by the
    // dynamically created component
    // and callable using the reference
    requires toCall: I
    // the dynamically created
    // component reference
    provides me: Pull[DirRef]
    // to call when the dynamically
    // created component is stopped
    provides stop: Do
  }

  // to call a dynamically created
  // component by reference
  provides call: Call[I,DirRef]
}
```

Figure 3.19: Direct references mechanism description in SpeADL

`Forward` ecosystem). We present an example of that after we present the last interconnection mechanism.

Figure 3.20 presents the implementation of the `DirectReferences` component. We don't detail all of the implementation, but present the idea to illustrate advantages of the SpeAD component model. First `DirRef` is an empty marker interface for a reference and this is the only type that is visible from outside of the implementation. Thus, agents implementations that manipulate such reference can only see an object without any particular method. This implies that casts has to be done inside this implementation, this is a price to pay for hiding implementation [6]. The implementation of `DistRef` is responsible of making a bridge between the mechanism implemented in `call` at the ecosystem level and the port `toCall` at the species level.

We also present briefly another referencing component that can play the same role as `DirectReferences`. This mechanism is map referencing where the agents are referenced not with a direct reference but with a key that maps to them.

Figure 3.21a shows its description. Here we can notice two different species, they represent each a different way of getting the key for mapping, either as a parameter of the species or through a required port. Except from that, their behaviour is similar, and equivalent to `DirectReference`. We don't detail the implementation but it is available in the library of components.

Publishing Observable Values. We now present an interconnection mechanism at a higher level of abstraction than referencing.

6. Cleaner ways of solving this problem exists, but would require to change the ADL and are out of scope of our work.

```
public class DirectReferencesImpl<I> extends DirectReferences<I> {

  @Override
  protected Call<I, DirRef> make_call() {
    return new Call<I, DirRef>() {
      public I call(DirRef ref) throws RefDoesNotExistsException {
        if (ref instanceof RefImpl) {
          // needed to not expose externally
          // a datatype parameterised by I
          RefImpl realRef = (RefImpl) ref;
          return realRef.call();
        } else throw new RefDoesNotExistsException();
      }
    };
  }

  @Override
  protected Callee<I> make_Callee(String name) {
    return new CalleeImpl(name);
  }

  private class CalleeImpl extends Callee<I> {

    private final RefImpl me;
    private CalleeImpl(String name) {
      this.me = new RefImpl(this, name);
    }

    @Override
    protected Pull<DirRef> make_me() {
      return new Pull<DirRef>() {
        public DirRef pull() { return me; }
      };
    }

    @Override
    protected Do make_stop() {
      return new Do() {
        public void doIt() { me.stop(); }
      };
    }
    public I p_toCall() { return toCall(); }
  }
  private class RefImpl implements DirRef {

    private CalleeImpl ref;
    private final String name;
    private RefImpl(CalleeImpl ref, String name) {
      this.ref = ref;
      this.name = name;
    }
    private void stop() {
      // allow for garbage collection and reliability checks
      this.ref = null;
    }
    private I call() throws RefDoesNotExistsException {
      if (ref != null) return ref.p_toCall();
      else throw new RefDoesNotExistsException();
    }

    @Override
    public String toString() {
      return this.name + (this.ref == null ? "(stopped)" : "");
    }
  }
}
```

Figure 3.20: Direct references mechanism implementation in Java

```
ecosystem MapReferences[I,K] {

  species Callee(key: K) {
    requires toCall: I
    provides me: Pull[K]
    provides stop: Do
  }

  species CalleePullKey {
    requires toCall: I
    requires key: Pull[K]

    provides me: Pull[K]
    provides stop: Do
  }

  provides call: Call[I,K]

}
```

(a) Map references

```
ecosystem ValuePublisher[T,K] {

  species PublisherPush {
    provides set: Push[T]
    provides get: Pull[T]
  }

  species PublisherPull {
    provides get: Pull[T]
    requires getValue: Pull[T]
  }

  requires call: Call[Pull[T],K]
  provides observe: ReliableObserve[T, K]
}
```

(b) Value publishing and observing

Figure 3.21: Interconnection mechanisms descriptions in SpeADL

```
public interface Observe<V,R> {
  public Option<V> observe(R ref);
}

public interface ReliableObserve<V, R> extends Observe<V, R> {
  public V reliableObserve(R ref)
    throws RefDoesNotExistsException;
}
```

Figure 3.22: The `ReliableObserve` and `Observe` interfaces in Java

Figure 3.21b shows the description of such a component. As we can see, it relies on the existence of a mechanism to call a reference but for a specific type of port linked to its function. It allows, through the `observe` port, to observe an agent using its reference, whatever it is (abstracted by the `K` type parameter). There exist two ways of publishing a value: either with `PublisherPush` that stores the value or either with `PublishPull` that pulls the value from the agent.

Figure 3.23 shows the implementation of the component. It only implements the logic behind its functionality while relying on its required ports to realise the referencing. Figure 3.22 shows the interfaces provided by the component. We can notice the use of the `Option` class from FJ that is used to represent container of values that can either be empty or full. It is used in the implementation of the `observe` port of the component.

Composing the Interconnection Mechanisms. Figure 3.24 shows an ecosystem using the two interconnection mechanisms we defined previously for a MAS where the agents that can observe each other using the publishing mechanisms and are named using `String`

```
public class ValuePublisherImpl<T, K> extends ValuePublisher<T, K> {

  @Override
  protected ReliableObserve<T, K> make_observe() {
    return new ReliableObserve<T, K>() {
      public Option<T> observe(K ref) {
        try {
          return Option.some(self().observe().reliableObserve(ref));
        } catch (RefDoesNotExistsException e) {
          return Option.none();
        }
      }

      public T reliableObserve(K ref)
        throws RefDoesNotExistsException {
        return call().call(ref).pull();
      }
    };
  }

  @Override
  protected PublisherPull<T, K> make_PublisherPull() {
    return new PublisherPull<T, K>() {
      @Override
      protected Pull<T> make_get() {
        return new Pull<T>() {
          public T pull() {
            return getValue().pull();
          }
        };
      }
    };
  }

  @Override
  protected PublisherPush<T, K> make_PublisherPush() {
    return new PublisherPush<T, K>() {

      private T value;

      @Override
      protected Push<T> make_set() {
        return new Push<T>() {
          public void push(T thing) {
            value = thing;
          };
        };
      }

      @Override
      protected Pull<T> make_get() {
        return new Pull<T>() {
          public T pull() {
            return value;
          }
        };
      }
    };
  }
}
```

Figure 3.23: Value Publishing interconnection mechanism implementation in Java

```
namespace namedPublish {

  component ObservedBehaviour {
    provides cycle: Do
    requires changeValue: Push[Integer]
  }

  component ObserverBehaviour[Ref] {
    provides cycle: Do
    requires observe: Observe[Integer,Ref]
  }

  ecosystem NamedPublishMAS {

    provides create: NamedPublishMASFactory

    part refs: MapReferences[Pull[Integer],String]
    part observeds: ValuePublisher[Integer,String] {
      bind call to refs.call
    }
    part observers: Forward[Observe[Integer,String]] {
      bind i to observeds.observe
    }

    part executor: ExecutorService
    part schedule: Scheduled {
      bind sched to executor.exec
    }
    part clock: Clock {
      bind sched to executor.exec
      bind tick to schedule.tick
    }
    part gui: SchedulingControllerGUI {
      bind control to clock.control
    }

    species Observed(name: String, beha: AbstractObservedBehaviour) {

      use sched: schedule.Agent {
        bind cycle to beh.cycle
      }
      part beh: ObservedBehaviour {
        bind changeValue to observed.set
      }
      use ref: refs.Callee(name) {
        bind toCall to observed.get
      }
      use observed: observeds.PublisherPush
    }

    species Observer(beha: AbstractObserverBehaviour[String]) {

      use sched: schedule.Agent {
        bind cycle to beh.cycle
      }
      part beh: ObserverBehaviour[String] {
        bind observe to observer.a
      }
      use observer: observers.Agent
    }
  }
}
```

Figure 3.24: An ecosystem for agents referenced by names and observing each other values in SpeADL

```
public interface NamedPublishMASFactory {

  public void createObserver(AbstractObserverBehaviour<String> beh);

  public void createObserved(String name, AbstractObservedBehaviour beh);
}
```

Figure 3.25: The `NamedPublishMASFactory` interface in Java

references. It also contains mechanisms for scheduling the agents in a synchronised way. `ExecutorService` is the equivalent of the Java standard library `ExecutorService`. `Scheduled` executes concurrently all the agents of the system, `Clock` is a clock responsible of instructing the `Scheduled` component to do one step of execution, and `SchedulingControllerGUI` control the `Clock` with a GUI. All of these components are available in the library.

The composition of the previously presented mechanisms is done at the ecosystem level (between `refs`, `observeds` and `observers`) and at the species level (between `ref`, `observed` and `observer`). As we can see, only what is needed for a species is included in its definition: the `Observer` species does not publish anything and thus does not use the `PublisherPush` species. With this, different combinations are possible depending on the need.

The ecosystem provides a way to create new agents of each species, this is possible through the use of the `create` port with the `NamedPublishMASFactory` Java interface presented Figure 3.25.

The implementation, Figure 3.26, shows the species related part of the implementation. The rest of the implementation only reference the implementation of each of the parts, which are all available in the library. We now focus on the `Observed` species as they are both similar. The only information that has to be provided is how to construct one instance of the implementation of the species. It relies on a component for the behaviour of each of the species. A partial implementation of these to ease behaviour definitions is provided by the class `AbstractObservedBehaviour`. Thus, this example also shows a simple template for a MAS where the agents behaviours are the hoptspots.

An example of the use of such an ecosystem is given Figure 3.27.

3.3.2.3 Templates

To illustrate the use of templates to implement patterns, we present now an example. Because the specialisation mechanism used in SpeAD is not usable for ecosystem and species, we don't present any template for interconnection mechanisms. Those we previously presented can also actually be used together to form architectural patterns as explained Section 3.3.1.3.

ADELFE Agents Template In the AMAS approach, when applied using the ADELFE method, agents all have the same kind of internal architecture. In this example, templates are used to provide pre-made agent architectures for easily starting new AMAS-based applications.

```
public abstract class AbstractObserverBehaviour<Ref> extends ObserverBehaviour<Ref> {

  @Override
  protected Do make_cycle() {
    return new Do() {
      public void doIt() {
        behaviour();
      }
    };
  }

  protected abstract void behaviour();

}

public class NamedPublishMASImpl extends NamedPublishMAS {

  // ... parts make_* ...

  // ... make_Observer ...

  @Override
  protected Observed make_Observed(String name, final AbstractObservedBehaviour beh) {
    return new Observed() {
      @Override
      protected ObservedBehaviour make_beh() {
        return beh;
      }
    };
  }

  @Override
  protected NamedPublishMASFactory make_create() {
    return new NamedPublishMASFactory() {

      public void createObserver(AbstractObserverBehaviour<String> beh) {
        Observer.Component agent = newObserver(beh);
        agent.start();
      }

      public void createObserved(String name, AbstractObservedBehaviour beh) {
        Observed.Component agent = newObserved(name,beh);
        agent.start();
      }
    };
  }
}
```

Figure 3.26: Implementation of `NamedPublishMAS` and one of the behaviour class in JAVA

```
public class Test {
  public static void main(String[] args) {
    NamedPublishMAS.Component mas =
      NamedPublishMAS.newComponent(new NamedPublishMASImpl());
    mas.start();

    mas.create().createObserved("agent1", new AbstractObservedBehaviour() {
      @Override
      protected void behaviour() {
        int v = new Random().nextInt(10);
        System.out.println("observed: changing value to "+v);
        changeValue().push(v);
      }
    });

    mas.create().createObserver(new AbstractObserverBehaviour<String>() {
      @Override
      protected void behaviour() {
        System.out.println("observer: observing value of "+
                           observe().observe("agent1").orSome(-1));
      }
    });
  }
}
```

Figure 3.27: Use of `NamedPublishMAS` in Java

Figure 3.28 shows an example of a template for the kind of internal architecture used by AMAS agents. It has to be composed with other components to have a fully complete architecture: this template focuses on the lifecycle of the agent and constrains the way its behaviour is implemented. Its implementation is shown Figure 3.29. In particular it mixes a reusable component named `Buffer` with the components `AbstractPerception` and `AbstractDecision` that are meant to be implemented by the user of the template. `Buffer` is available in the library, it buffers the calls to a port until they are released (see the implementation of the port `cycle`). This template can be combined with a component that manages the knowledge of the agent and a component that executes the behaviour.

3.4 The SpEArAF Method

We now present SpEArAF (Species to Engineer Architectures for Agent Frameworks) that instantiates the methodology by using SpeAD as a mean to build the micro-architecture.

SpEArAF does not rely on any specific MAS design method to produce the macro-architectural view, it only relies on the fact that after doing this part of the design, the MAS is at least described in terms of:

– Types of agents: depending on the approach, different types of agents can be defined for a given application, and for each of them the running application has several instances of it. Obviously there is always at least one type of agents. They may have difference in the way they have to be internally organised, their dynamics, their behaviours, etc.

```
import fr.irit.smac.may.lib.interfaces.*
import fr.irit.smac.may.lib.components.meta.*

// an abstract component
component AbstractPerception[Sensors, Knowledge] {
  provides perceive: Do
  requires sensors: Sensors
  requires kb: Knowledge
}

// an abstract component
component AbstractDecision[Knowledge, Actuators] {
  provides decide: Do
  requires kb: Knowledge
  requires actuators: Actuators
}

// a concrete component
component ActionBuffer[Actuators] {
  provides act: Do
  provides actuators: Actuators
  requires realActuators: Actuators
}

// Knowledge: operations to access and modify knowledge
// Sensors: operations to access sensors
// Actuators: operations to access actuators
component AMASAgent[Knowledge, Sensors, Actuators] {
  // to implement in order to specify how to update knowledge
  // based on sensors
  part perception: AbstractPerception[Sensors, Knowledge] {
    bind sensors toThis sensors
    bind kb toThis kb
  }

  // to implement in order to specify which actions to do
  // based on knowledge
  part decision: AbstractDecision[Knowledge, Actuators] {
    bind kb toThis kb
    bind actuators to action.port
  }

  // already implemented, it buffers the actions to execute them
  // after decision has been done
  part action: Buffer[Actuators] {
    bind realPort toThis actuators
  }

  requires actuators: Actuators
  requires sensors: Sensors
  requires kb: Knowledge

  provides cycle: Do
}
```

Figure 3.28: Template of an AMAS agent architecture in SPEADL

```
public abstract class AbstractAMASAgentImpl<K,S,A> extends AMASAgent<K,S,A> {

  private final Class clazz;

  /**
   *
   * If I is actually MyPortInterface,
   * then this class should be used with
   * new BufferImpl(MyPortInterface.class)
   * Just omit type parameters of the port interface
   *
   * @param clazz is the class of the concrete I
   */
  public AbstractAMASAgentImpl(Class clazz) {
    this.clazz = clazz;
  }

  @Override
  protected Do make_cycle() {
    return new Do() {
      @Override
      public void doIt() {
        perception().perceive().doIt();
        decision().decide().doIt();
        action().release().doIt();
      }
    };
  }

  @Override
  protected Buffer<A> make_action() {
    return new BufferImpl<A>(clazz);
  }
}
```

Figure 3.29: Implementation of the template for an AMAS agent architecture in Java

– The interaction means they use: strongly linked to the types of agents, but also to the environment, agents are meant to use interaction means. There is again at least one interaction means, but several approaches include the use of different interaction means at the same time. Different types of agents can use different interaction means for example.

– The runtime platform: the result of design can also introduce some environmental elements, such as spaces or organisations that have their own dynamics. In term of runtime software element, the runtime platform is what mediates the interactions, let agents be created and let them live.

For example, Section 3.1.4 that illustrates the methodology with the ADELFE method, the type of agents identified are the ants. They are defined using a perceive-decide-act lifecycle that is executed by the platform for all the agents concurrently in a loop. They use an interaction mean to move in a 2D space, an interaction mean to deposit and sense pheromones. The runtime platform has a GUI to visualise the state of the 2D space, the position of the agents and the quantities of pheromones in the different cells.

In the following we map the concepts of types of agents, interaction means and runtime platform to those available in SPEAD in order to realise a partially abstract micro-architecture. We then present an iterative and incremental design process for building this micro-architecture. We conclude with guidelines that are useful when applying the method in practice.

3.4.1 Component-Based Architectures for MASs

The platform is implemented using an ecosystem: its sub-components represent different mechanisms at the environment level, such as 2D plan, extra-agent organisation dynamics, but also scheduling, distribution, visualisation GUI, etc.

Types of agents are implemented using species and are created and maintained "alive" by the ecosystem: components in the species represent internal mechanisms and dynamics such as lifecycles, adaptation, GUI, capabilities, knowledge management etc.

Interconnection mechanisms are implemented using a combination of ecosystem and species, used inside the environment ecosystem and composed into the different types of agents species. They can be sensors, actuators, messages passing and other interaction means, but also scheduler, visualisation and other operative links between the agents and the platform.

As we said, we propose to describe the micro-architecture as an architecture with holes, that we also call partially abstract architecture: some components of the architecture are provided implemented, while others are left abstracts for implementation by the MAS developer. The way to describe such architecture using templates is presented Section 3.3. The basic idea behind this point is to find what are the concrete and the abstract components of the architecture. The programming abstractions, as presented Section 3.1, that the MAS developer can use to focus on his business concerns are then provided by the architecture in the form of abstract components. Concrete components would be the rest of the architecture, not meant to be modified by the MAS developer.

3.4.2 Iterative and Incremental Micro-Architectural Design

The following gives a process to build the partially abstract micro-architecture that can then be used as a development support.

This process is driven by the requirements. When the process starts, the requirements are those we identified in the methodology Section 3.1, but the more the micro-architecture get refined, the more detailed requirements are elicited and are what need to be implemented at one point or another. This process is mainly inspired by the ADD (Attribute-Driven Design) method for the iterative refinement of the architecture and the "Twin Peaks" model for the incremental definition of both requirements and architecture.

For a presentation of *ADD*, see p. 8

The iterated cycles of the process are separated in two phases: species and ecosystem. Every step helps identifying new elements to define the architectures. They must be repeated until no more requirement is present. Each phase of a cycle starts with a set of requirements

For a presentation of the *Twin Peaks Model*, see p. 7

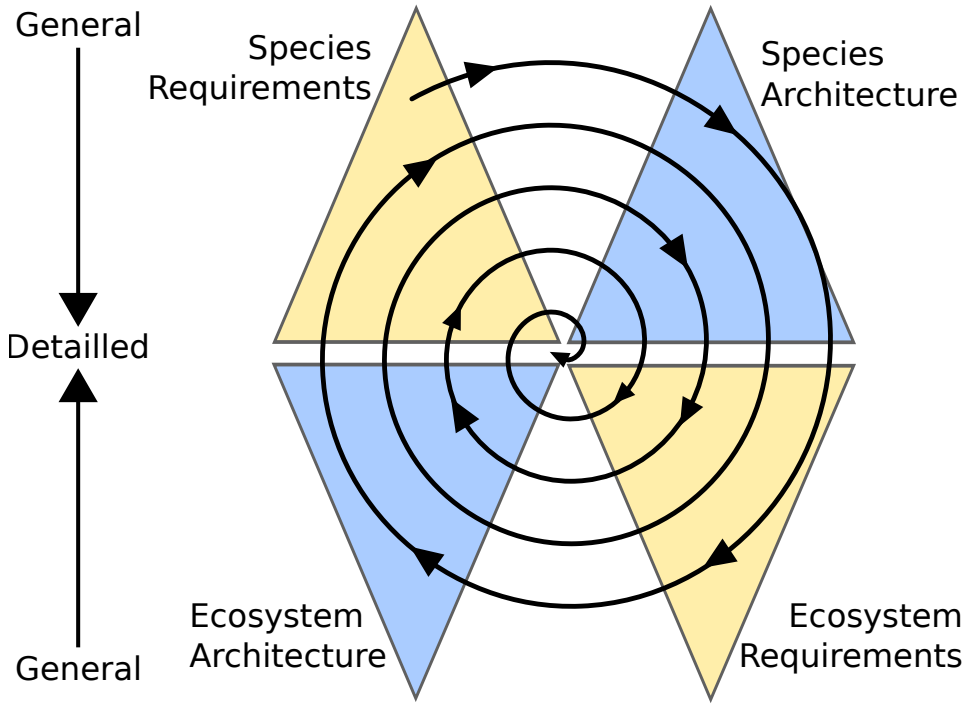


Figure 3.30: The SpEArAF Method.

and potentially ends with new requirements. At the beginning, we start with all the requirements, business and operative, then more requirements for the micro-architecture get elicited during the design. The more detailed the design gets, the more the requirements that are left to be answered are operative ones.

Figure 3.30 illustrates this process.

Species Phase. The focus is on each of the species internal architecture: we don't describe a species but the main component that is its internal architecture.

The first step is to identify mechanisms used by the agents to interact with the platform. It means any mechanism that may need the platform to operate. In the first cycle, this is mostly about interaction means with the MAS environment and other agents. This step sets the external boundary of the species architecture. This takes the form of required and provided ports of the species internal architecture.

The second steps is to identify the abstract component that the MAS developer uses to program its agents. They are mainly elicited from the business requirements by taking the form of abstract component definitions that are parts of the species internal architecture. This step sets the internal boundary of the species.

The third step is to define or refine the internal architecture of the species based on these two boundaries. In the first cycle, this is mostly about the internal dynamics of the agents. This step introduces a set of elements used for the actual execution of the agents. This may introduce new external and internal boundaries.

All the external boundaries are requirements towards the ecosystem for the species to properly operate.

Ecosystem Phase. Starting from the external boundary of each of the species, as well as the other requirements, a new version of the ecosystem architecture can be defined. For example, at the beginning, the dynamics of the platform should be devised.

For every species internal architecture, a species is defined and must be refined to link it to the ecosystem. New elements are introduced in the ecosystem and in the species. In order to realise the links between the ecosystem and the species, some of these elements are themselves ecosystems whose species are used in the agents' species and connected to its internal architecture ports. Most of the time, these elements are reused, but when needed, new one may have to be defined. These mechanisms, because they are meant to be part of other species, may have required and provided ports to be usable. In complex cases, the current process can be applied recursively to them.

In the same way than the species internal architectures, it is advised to start with business requirements and materialise them using abstract components at the ecosystem level.

All of this can add new requirements towards the species internal architectures.

After this phase, a cycle ends and it may be needed to go back to start a new cycle and so on until all the requirements are answered.

3.4.3 Additional Guidelines

We now present several guidelines that were discovered during the application of such a process to design the micro-architecture of MASs. These are informally detailed.

What should be a Species Based on the macro-level architectural, several types of agents are identified with their interaction mechanisms and other explicit and implicit requirements related to how agents at runtime are connected to the runtime platform. But other entities identified at the macro-level design may also be modelled with species at micro-level design. The main guidelines is to see if the type of entities concerned is likely to be:

1. Created dynamically.
2. Interacting with agents (or other entities, depending on the MAS approach) using interaction mechanisms such as those used by agents.

Decomposition. When decomposing the architecture in different parts, the question of where to put some of the components and their functionality occurs. For example, should the GUI controlling a MAS be inside the ecosystem with the species or outside the ecosystem in a composite that contains the ecosystem and other components.

As a general guideline, it seems that it is preferable to keep the ecosystem focused on MAS mechanisms, and thus any element not directly concerned with the agents and their interactions would gain to be composed with the ecosystem in a component representing the application itself.

Actually, often in MAS approaches, MASs are considered to exist inside a computational environment that gives feedback to it or gets results from it. This environment is not aware of the fact it is interacting with a MAS but sees it as a blackbox with clearly defined interfaces. But on top of the agents and their platform, what is considered the MAS also contains parts aware of the fact they are interacting with a MAS without themselves being part of the MAS. A common way of decomposing this kind of systems is:

1. One ecosystem for the MAS itself exposing MAS-oriented ports.
2. One component containing it and representing the part of the system aware of the fact it is a MAS and exposing MAS-agnostic ports.
3. One component for the actual final application that contains it.

Mapping between Views. It is a good practice, when applying the process, to keep track of the mapping between the elements of the micro-architectural view and the macro-architectural view as it is often recommended in software architecture practice.

In particular, in the context of MAS development, this is particularly important to record the mapping between types of agents and the corresponding species.

3.5 Conclusion

We presented in this chapter a coherent set of answers to the challenges identified previously.

First, we detailed our vision of the methodology of MAS development as a direct response to the analysis made Chapter 2. By separating the design of the MAS itself — called macro-architectural design — from the design of the architecture that support its implementation — called micro-architectural design —, this highlighted architectural and implementation challenges for building the latter.

The SpeAD component model we present next answers such challenges while taking into account our desire of easing the development and improving the quality and reuse of software produced during MAS development. By choosing a component-oriented approach, it is possible to build an architecture adapted to the needs of the application to be built, but also to the concerns of the MAS developer. This thus gives the possibility, when doing macro-architectural design to focus on the business concerns of the application. At the same time, by making explicit the micro-architectural design and all the requirements it answers, this helps to make a bridge between design and implementation.

Finally, to build such micro-architecture, the SpEArAF development method positions the exploitation SpeAD into the general proposed methodology and provides guidelines to incrementally and iteratively use it.

To better understand how all of this can be used in combination with existing MAS development methods and how practically it is applied, next chapter presents an example of a real MAS-based application for a research project with industrial partners. This application ends with a quick discussion on the advantages of using our contribution, and a complete

analysis of the contribution is then presented Chapter 5. It is accompanied by a presentation of academic and industrial works using our contribution and a positioning with respect to other research works on MAS development.

Furthermore, Chapter 6 proposes a positioning of the contribution with respect to existing works in the software architecture field.

Summary of the Contributions

- ⊕ We propose a methodology of MAS development that allows to take into account all the specific types of requirements that exist in MAS development.
- ⊕ We propose the SpeAD (Species-based Architectural Design) component model that enables to design and implement the micro-level architecture of a MAS.
- ⊕ We propose the SpEArAF (Species to Engineer Architectures for Agent Frameworks) method that instantiates the methodology using SpeAD.
- ⊕ We present guidelines on how to reuse knowledge and experience using SpeAD.
- ⊕ We present reusable components that are part of a component library.

CHAPTER 4

Application

En mai, fais ce qu'il te plaît.

Proverbe français

In this chapter, to illustrate the whole lifecycle of the development of a MAS, we now apply SPEARAF with SPEAD after following a macro-level design method on an example. This was completely applied in practice to implement this example using MAY (MAKE AGENTS YOURSELF), the tool presented Appendix B that supports the SPEAD component model.

This example is taken from a real world application developed in our research team for a research project with industrial partners named GAMBITS[1]. The objective of this project is to provide training to maritime surveillance operators. These operators use a software system that assists them to supervise maritime zones and detect suspicious behaviours. In order to train them to the software, the project exploits a serious game approach. A serious game is "a mental contest, played with a computer in accordance with specific rules that uses entertainment to further government or corporate training, education, health, public policy, and strategic communication objectives" (Zyda 2005). In our case, the type of game used is a tower defence game were the objective is to protect a zone from enemies by placing defence towers on a map. For example, the enemies are the lawbreaker boats and the towers are static radars, mobile radars or patrols.

The objective of our research team in the project is to provide an intelligent self-adaptive MAS able to control the game in order to adapt at runtime its difficulty and behaviour to the level of the operators in training. The idea is that such a game provides a lot of informations on a running session, from which it is possible to infer the learning success of the player. Based on that information and on domain-specific constraints elicited by the trainers, the system must control continuously the parameters of the game so that the learning curve of the player is progressive and that some wanted educational objectives are reached.

1. Game-Based Intelligent Strategies: http://www.gambits.fr/, financed by the French DGCIS (*Direction Générale de la Compétitivité, de l'Industrie et des Services*).

The MAS approach used to design this MAS is the AMAS (Adaptive Multi-Agent System) approach and the ADELFE method. Here, we present the results of the diverse phases of the development from the point of view of our methodology and not from the point of view of the ADELFE method.

For a presentation of the *AMAS approach*, see p. 11

In few words, the main reason behind the choice of the AMAS approach to solve this problem is (on top of the obvious fact that the approach is promoted by our research team) that, as we are going to see, the problem is not precisely specifiable. Indeed, using an approach that agentifies the domain entities of the problem is well adapted to the solving of this kind of complex problem. We don't discuss here the legitimacy of using such an approach or method, as it is out of scope of this work, but the ADELFE method has a step to check that point. During the presentation of the macro-level architectural view, we don't detail the rationale behind every design choices.

For a presentation of *ADELFE*, see p. 50

The presented MAS actually controls an open-source game that has the same characteristics than the project game. Using a game that wasn't the one the system was originally made for actually adds a bit of stress on the development, which can only give more credit to the reusability of the solution built with our development tools.

4.1 Context and Requirements

In the following, we call "system" the software to build, including the MAS, but also anything needed to answer the requirements.

We divide the set of initial requirements expressed at the beginning of the development in several categories: the environment of the system, the functional and non-functional requirements pertaining to the system to build, the non-functional requirements pertaining to the development and its organisation. On top of that, in each of these categories, we present diverse constraints. At the same time, we introduce the vocabulary used in the domain by emphasising it in *italics*.

For a definition of *Initial Requirements*, see p. 45

4.1.1 Environment of the System

The system interacts with a *game platform* responsible of executing the game. For a given *session* of training, the system controls, by communicating with the game platform, how the session evolves, which are the objectives to attain, etc.

Such interactions between the system and the game platform must be done through the network.

In the project, the game platform is divided in two parts: the game engine and the behaviour of the entities of the game. In practice here, the game platform is a simpler tower defence game in one block where the objective is to protect a zone from invasions of enemy waves of different types by placing defence towers of different types on a map.

The system is controlled by the *game trainer* through a GUI. The system under development is called the Virtual Game Designer (VGD) as its role is to assist the game trainer.

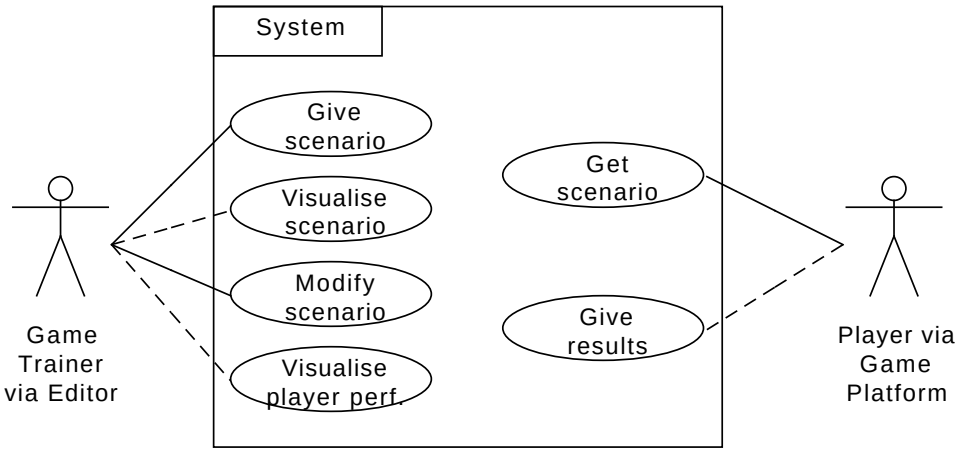


Figure 4.1: Use Cases for the VGD in UML

The *player* only interacts with the VGD through the use of the game platform. The VGD thus does not have any information on the player other than his actions in the game visible through the scenario accessed and updated by the game platform.

4.1.2 Functional and Non-Functional Requirements for the VGD

The VGD manipulates a scenario under the supervision of the game trainer in order to maximise the educational objectives expressed by him.

The *scenario* is the set of informations needed by the game platform for a given session of training. In other words, it is a big quantity of interdependent parameters that influence the execution of a session. Depending on the design of the VGD, the scenario has to be modelled in a specific way that we thus detail later.

In our case, parameters can be enemy strength, enemy types ratio, tower range, etc, and objectives can be to get high scores, to survive a certain number of waves, etc. In any way, the VGD controls the game platform by continuously updating the scenario and retrieving informations on the player through it.

Use Cases. The most easier things to formally specify are the interactions of the system with its environment, *i.e.* the game trainer through the interface of the VGD and the player through the game platform. Figure 4.1 shows such use cases for the system. On top of that, there is other functional and non-functional requirements for the system.

Functional. The VGD gives the possibility to the game trainer to modify the scenario, including the educational objectives, through a GUI. It applies the modification of the game parameters based on the scenario. It proposes modifications for the scenario in relation to the observed behaviour of the player in order to maximise the expressed objectives.

Non-Functional. The propositions of the VGD must be given in real time. The modification of the parameters must be done in real time. The VGD must self-adapt its control to the player's behaviour.

Comments. As we can see, the requirements for the system itself are difficult to specify precisely. Indeed, we don't have any formal way of specifying what it means to maximise the expressed objectives in another way than saying, for example, that the score goes up, or what it means to adapt to the player in another way than saying, for example that the parameters should be modified in order for the game to be easier and that the score goes up. Such a specification is useless if we were to decompose the system in sub-parts following a reductionist approach. Here, as we are going to see, the idea here is to model the problem itself and let its various entities self-organise to find a good solution. This is actually one of the reasons the AMAS approach is used to tackle such a problem.

For a reminder on *reductionism*, see p. 10

4.1.3 Non-Functional Requirements for the Development

Even though the presented solution is using a simpler game than in the project, this change happened only at the end of the development in order to have a working prototype. Thus the development followed the organisation of the project that we now present with its requirements and constraints.

The development team is organised in three groups. One of them focuses on the VGD, with one developer, whose work we describe here. Another group is responsible for the realism of the behaviour of the game platform (exploiting the parameters of the scenario) and the last one is taking care of the game engine itself.

The main requirement for the development is about tuning: the developer needs to visualise the internal execution of the VGD to support its tuning during the development. Such visualisation must be removable when working with the other teams during integration. As we are going to see, this manifests itself in a particular way when using MASs.

Other requirements actually appear later in the development. Indeed, the methodology of MAS we propose relies on the fact that after the MAS design is done, it is needed to extract new requirements from it in order to tackle implementation. They are thus detailed when needed.

4.2 Macro-level Architectural Design

As we said, the approach chosen is the AMAS approach with the ADELFE method. In this approach, the idea is to agentify the entities of the domain problem and give them the behaviours that will drive them to collectively find a solution.

The first need is thus to extract the micro-requirements the approach answers, and then precisely define the domain problem, which here is the scenario and its adaptation. Based on that, the MAS design can be done.

4.2.1 Macro-Level Requirements Extraction

Some of the requirements presented Section 4.1 are directly be answered by the MAS design: they form the set of the macro-level requirements.

For a definition of *Macro-Level Requirements*, see p. 47

These requirements are:

- The functional requirements, except for the GUI interactions with the game trainer and the scenario transmission to the game platform.
- The non-functional requirements, in particular those about adaptivity.

All the rest is not directly tackled by the MAS design except for the fact that the state of the agents must be easily accessible for tuning: for example it means it is not possible to distribute the agents on a lot of machines. Happily, this is not needed here, but this shows the kind of implication some requirements can have on the design.

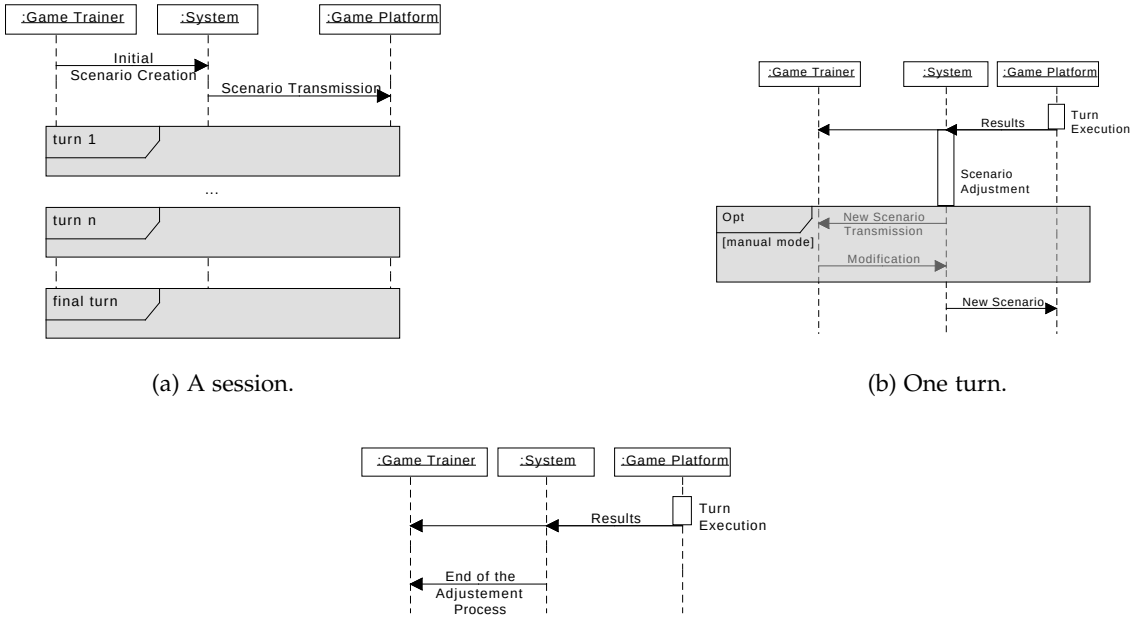
4.2.2 Problem Domain Model

A scenario is composed of *parameters* whose values exist inside a range specific to it. The values of these parameters define how the session goes on. For example, some parameters for the game platform can be the speed of the enemy entities and the defences efficiency.

On top of these low-level informations, the scenario is also composed of higher-level informations that influence the orientations the scenario should take during the session. They are used by the game trainer to define the objectives to attain and under which conditions. The difference is made between the *measured criteria*, which represent information coming from the game platform, and the *criteria*, which represent aggregations of the other domain entities. The concept of criterion is not directly represented in the game platform itself but is expressible using the low-level parameters and the other, possibly measured, criteria. For example, a criterion for our game platform can be the ratio of a specific type of enemy entities. The measured criteria are typically used to represent things like the score, *i.e.* the number of killed entities per wave.

Then, some *dependencies* between parameters and between parameters and criteria are used to express how their values should evolve during the scenario. They are expressed through the use of a matrix that says in which direction (positive or negative) the evolution of a parameter should influence another one. The actual strength of the influence is something that can evolve at runtime and is controlled by the VGD. The dependencies are domain-specific knowledge expressed by the game trainer. An example of dependencies is that the score is inversely dependent to the speed of the enemy entities.

All these informations are provided by the expert. The system manipulates and controls them to make the player reach the wanted objectives. *Constraints* can be expressed on each criterion and parameter: they can for example be desired values, ranges, etc. Objectives are constraints on criteria that can't be directly controlled. They must be enforced by the system and trigger its self-adaptation.



(a) A session.

(b) One turn.

(c) The final turn.

Figure 4.2: Sequences diagram for the VGD and its environment interactions in UML

4.2.3 Temporal Interactions of the VGD with its Environment

Thus, during a session, for a given scenario:

- Some values are updated, based on:
  - What changed in the game platform (parameters and measured criteria).
  - Desires of the game trainer (constraints).
- The parameters values are read by the game platform.

On the side of the VGD, the different entities are encapsulated by the agents and live in the same memory space and process than the MAS. They can thus be accessed directly by the game trainer GUI. Differently, the interactions with the game platform should be decoupled in order for the system to work through a network.

Figure 4.2 shows a specification of the interaction of the system with its environment.

4.2.4 Multi-Agent System

For a definition of *Macro-Level Architectural View*, see p. 45

We now describe the macro-level architectural view, the result of the application of the ADELFE approach. The solution is described in terms of its structure — types of agents, environment and interaction means —, and in terms of the agents' behaviour.

We actually present the behaviour of the entities first, as it is the way it is done in practice. We don't describe it in details since, except for what is needed to understand how the system is working, as this does not influence so much the micro-architectural design.

The criteria, parameters and constraints are modelled as agents and interact to compute or change their value with respect to the dependencies in order for the constraints to be enforced as much as possible. Dependencies represent who knows who in the system: for each agent, it is its neighbourhood. All the agent-entities have their value, either computed (parameters and criteria), coming from the game platform (measured criteria) or coming from the game trainer (constraints). They try to change it, either by influencing other agents that may have an impact on them, or by directly changing them if they can.

As we said Chapter 1, in the AMAS approach, the engine of adaptation is the cooperative behaviours of the agents. One way to represent that is by using the concept of criticality: when agents have requests, they associate a criticality to them in order for the answering entities to be able to know which is the more critical request to answer first. Of course the difficulty is to find the best way to compute the criticality of a request. We don't detail that here.

In our case, constraints generate more or less critical requests by expressing that they are not enough enforced and in which direction the other entities must change so they can. At the same time dependencies are also followed in order to request change between parameters and criteria, while propagating criticality. The most critical requests are answered in priority until the system stabilizes. Since there are continuously new values for measured criteria, the system continuously self-adapts and reorganizes to tend towards the adequate solution.

The ADELFE method organises the internal of the agents in several parts: the knowledge, the skills and aptitudes, the nominal behaviour and the cooperative behaviour. Agents follow an iterated perceive-decide-act lifecyle where the behaviours are executed in the decision based on the knowledge and choosing actions to execute. Knowledge is manipulated by perception and decision. Decision and action uses the skills and aptitudes.

Available action means are sending messages and changing own value. Available active perception mean is observing other agents. Available passive perception means are receiving messages and having own value externally changed.

Without entering into the details, the behaviour of the parameters, criteria and measured criteria is to receive requests asking them to change their value, observing other agents and choosing to change or not their values as well as request other agents to change their. Concerning the constraints, they observe other agents and request for them to change their values.

All of this is what is produced by the application of the AMAS approach and the ADELFE method. They are some of the requirements needed to build the micro-architecture. These micro-level requirements are those that are explicitly introduced by the design, but a lot of information is missing to be able to implement it: what is the exact semantics of the interaction means, how are agents executed, in what order, when does update of values happen, how information is accessed by the GUIs, etc.

For a definition of *Micro-Level Requirement*, see p. 47

4.3 Micro-Level Requirements Extraction

We first present what are the assumptions made here, then the supplementary design choices that must be taken to go towards implementation. All of these will form the set of implicit micro-level requirements. By making them explicit now, we prevent ourselves to discover them too late during implementation, and we can already see if there is things to change in the macro-level design to be able to answer them.

4.3.1 Assumptions Made during the Design

To be executed, agents have to alternate between perception/decision and action repeatedly. Also, in order for the agents to know each other, it is necessary to give them the list of their neighbours (inferred from the dependencies). This must be done at agent creation time, and as we are going to see, depending on the way references of agents are implemented, this can have impacts on the way this initialisation is done.

Furthermore, the agents need to interact, for that they use:

- Messages passing: asynchronous communication, uses references for agents, messages are received in the mailbox of the agents that they can consult.
- Observation: synchronous interaction, uses references for agents, read access on the values of the agents.

In both cases, the agents must know the other agents to interact with them.

Finally, depending on the type of agents, some are observable, some are not, some receive messages, some only send them, etc. Precisely, parameters are observed and receive messages; measured criteria are observed, send and receive messages; criteria are observed, observe, send and receive messages; and constraints only send messages and observe. It could be relevant to use all the mechanisms for the first three in case they become needed during the evolution of design, but the constraints, because they do not have any value, cannot be observed.

4.3.2 Supplementary Design Choices

On top of these assumptions made during design, other choices have to be made. Some of them are actually linked to the initial requirements, while others are linked to the macro-level design.

Scheduling. About the execution of the agents, the choice made is to schedule the agents and the system in the following order in a loop:

1. All the agents execute one step of perception/decision.
2. All the agents execute one step of action.
3. Values are sent to the game platform.
4. Values are updated from the game platform.

This is needed because the macro-level design relies on the fact that no action is applied before every agent has taken a decision.

Game Trainer GUI. The GUI used by the game trainer to modify the system in real time only interacts with the system to get and update values. The way the GUI manipulates the values is directly through the references of the entities objects encapsulated by the agents without any decoupling. This GUI is also responsible for triggering the sending of the new scenario to the game platform.

Tuning GUI. In AMAS, the driver of the adaptation being the criticality, in term of tuning, it is necessary to have an easy way of visualising the state of criticality of the agents and of the system (which is the maximum of the criticality of every agents). This means that we need a GUI that shows that information on top of the entities values. This GUI must be easily removable from the system.

4.4 Micro-Level Architectural Design

Based on all these design choices and requirements, which form the set of micro-requirements, we can now design the micro-architecture of the system. From this set, we extract precise specifications of the types of agents and their requirements for implementation. In particular we categorise the requirements as business or operative. They will drive the building of the partially abstract micro-architecture so that it is easily usable by the MAS developer to work on his MAS design.

For a definition of *Micro-Level Architectural View*, see p. 47

For a definition of *Business and Operative Requirements*, see p. 49

4.4.1 Operative Requirements

Different GUIs have to be integrated with the system: for tuning and for the game trainer interactions. The first one has to be removable and needs access to internal values from the agents, the second one only needs access to the domain entities that are shared and encapsulated by the agents.

Interactions with the game platform must be doable through the network and should thus be decoupled as much as possible from the system execution. They only concern the domain entities encapsulated by the agents.

The system is synchronised; the agents' execution and the updates of values are alternatively executed by the platform:

- Update of values must be buffered before they can be sent to agents.
- Agents must be externally triggered by the platform.
- Their behaviour must be separated in two different steps.

The initialisation of the system must inject in the agents their list of neighbours.

There are four different types of agents: parameters, criteria, measured criteria and constraints. They all use either or both messages sending and observation as described previously. They all are synchronised at the two different steps in the same way.

They are all operative requirements since their implementation is not of the concern of the MAS developer.

4.4.2 Business Requirements

To implement his design, the MAS developer needs to describe the behaviours of the agents. In our case, this is mostly about describing what to do when agents have to perceive and decide, and what to do when agents have to act. The developer also wants to be able to choose the implementation of the exchanged messages.

In the first phase, the developer wants to be able to access the mailbox as well as perceive other agents using their references (independently of the implementation of such references). In the second phase, he wants to be able to send requests and/or change their value.

This should be usable to implement directly his design without bothering about operative details. He wants to exploit mechanisms to manage neighbours and manage the values. Possibly he wants to implement the mechanism to construct new requests for all their neighbours.

There is no business requirements for the runtime platform.

4.4.3 Incremental Design

For a presentation of the SPEARAF *method*, see p. 82

We present now the resulting design in an incremental way using the SPEARAF method. We start from the species, we focus on the criteria agent type as it is the most complete one and the constraint agent type as it is the most different from the other three.

4.4.3.1 Cycle 1, Species Phase

In this cycle, we focus on the constraint agent type. The chosen implementation for the messages is `Request` provided by the developer, not detailed here, it contains informations that requests between agents must contain. The agent references, for messaging as well as observing (not visible in this cycle but that can be different than those for messaging), are not chosen yet and stay abstract.

External Boundaries. The interaction means needed by the agents are message passing and observation. We already know they should also provide two steps for execution. This is represented by the required and provided ports of `ConstraintAgentComponent` Figure 4.3.

Internal Boundaries. The abstract components for the agents should permit to answer the business requirements: implement the two steps of behaviour by exploiting the available mechanisms. This is represented by `ConstraintBehavior` Figure 4.3.

```
component ConstraintBehavior {
  requires criticality: Pull[Double]
  requires sendRequest: Push[Request]
  requires whichWay: Pull[VariationWay]
  requires getObservedValue: Pull[Double]

  provides perceiveAndDecide: Do
  provides act: Do
}

component ConstraintAgentComponent[MsgRef] {
  provides perceiveAndDecide: Do = beh.perceiveAndDecide
  provides act: Do = beh.act

  requires getValue: Pull[Double]
  requires send: Send[Request, MsgRef]
  requires die: Do

  part beh: ConstraintBehavior {
    bind criticality to criticalityManager.criticityValue
    bind sendRequest to requestSender.sendRequests
    bind whichWay to criticalityManager.wichWay
    bind getObservedValue toThis getValue
  }
  part criticalityManager: CriticalityManager {
    bind value toThis getValue
  }
  part requestSender: RequestSender[MsgRef] {
    bind neighboursInfo to neighboursMan.neighboursInfo
    bind send toThis send
  }
  part neighboursMan: NeighboursManager
}
```

Figure 4.3: Micro-architecture, Cycle 1: Constraint Agents Internal Architecture

Internal Architecture. Figure 4.3 shows a choice of decomposition with the behaviour itself separated from the management of neighbours, from the management of the criticality and from the building and sending of requests. All the components not prefixed by `Constraint` are actually used in other species internal architectures.

4.4.3.2 Cycle 1, Ecosystem Phase

Figure 4.4 presents the result of this step that we now comment.
For each of the types of agents a species is defined.

Interactions. To implement their interaction means, we choose to use the components available in the library, they are both described Chapter 3, Section 3.3.2:

- `AsyncReceiver` that implements asynchronous messages reception: it proposes a species that puts messages in a mailbox accessible to the agent.
- `ValuePublisher` that implements direct synchronous concurrent access to a value managed by an agent: it proposes a species `PublisherPull` that delegates the management of the data to the internal architecture of the agent.

To work, they must be coupled with a mechanism to manage agent references and to use them. As presented Chapter 3, there exist two of such mechanisms: `MapReferences` and `DirectReferences`. The first one is easier to use because the references do not need to be created before being used as they are strings, while the second is more efficient as it uses direct references, but must be created before being used. For various reasons, the choices are to use named references with string (corresponding to the entities names in the scenario) for observation and direct references for messaging.

These mechanisms enable to receive messages and be observed. In order to send messages and to observe, the complementary mechanisms are available from the previously presented components at the ecosystem level. We choose to use the `Forward` component to give access to them to the agents.

Scheduling. For the execution, there exists an interconnection mechanism in the library to give control of the agent execution to the platform named `Scheduled`. We use one instance of it respectively for the perceive/decide phase and for the act phase. Since they are used by all the species, we factorise them together in an ecosystem shown Figure 4.6. While we are on the design of the execution aspect of the system, we use the component `Clock` to synchronise the system. We introduce the component `AgentSequencer` to decompose one cycle of the system in two steps. We use the component `ExecutorService` to execute these in a thread pool; and finally we introduce the component `ClockController` presented Figure 4.5. This last one answers the requirement of removable GUI by providing two different implementations, one without GUI and one reusing the library component `SchedulingControllerGUI` providing a GUI. The species `Agent` of the ecosystem `Scheduling` encapsulates all of that.

Scenario Management. For creating agents from a scenario, we introduce the interconnection mechanisms `ScenarioManager` shown Figure 4.7 that itself introduces the need for ports to create new agents in the ecosystem and to add neighbours to agents. It encapsulates the logic of creation of agents: in particular it needs to have access to the agents references (through the port `me` for example) in order to get and give them to the agent they must know. In practice, the implementation of the `loadScenario` port creates every agent of the scenario, while their direct references are stored upon creation, then, when all agents are created, the needed references are injected in each agent using their species port `addNeighbour`.

On top of that, it provides ports to update agents values.

Tuning GUI. We introduce an interconnection mechanisms named `TuningGUI` responsible of encapsulating everything linked to this concern in order to make it easily removable. It is included as a part named `gui` in `ScenarioManager`, shown Figure 4.7, since its concerns are linked to those of the scenario. The species is responsible of receiving debugging information from the agents, showing it in the GUI, and the ecosystem is responsible of aggregating it. The species can be added only to the species we want to visualise, in this case, the species `Constraint` of `ScenarioManager`.

```
ecosystem Proto {
  provides agentFactory: AgentFactory

  part mrVP: MapReferences[Pull[VariableValue],String]
  part vp: ValuePublisher[VariableValue,String] {
    bind call to mrVP.call
  }
  part vr: meta.Forward[ReliableObserve[VariableValue,String]] {
    bind i to vp.observe
  }
  part drR: DirectReferences[Push[Request]]
  part receive: AsyncReceiver[Request,DirRef] {
    bind call to drR.call
  }
  part sender: meta.Forward[Send[Request, DirRef]] {
    bind i to receive.deposit
  }
  part sched: Scheduling
  part scenario: ScenarioManager[DirRef] {
    bind agentFactory toThis agentFactory
  }

  species CriteriaAgent(el: ICriteria, name: String) {
    provides stop: Do

    part arch: CriteriaAgentComponent[DirRef,String] {
      bind send to sender.a
      bind me to drR.me
      bind die toThis stop
      bind getValueOf to vr.a
      bind incoming to receive.getAll
    }
    use sched: sched.Agent {
      bind act to arch.act
      bind pAd to arch.perceiveAndDecide
    }
    use drR: drR.Callee(name) {
      bind toCall to receive.toCall
    }
    use receive: receive.ReceiverBuf
    use sender: sender.Agent
    use mrVP: mrVP.Callee(name) {
      bind toCall to vp.toCall
    }
    use vp: vp.PublisherPull {
      bind getValue to ???
    }
    use vr: vr.Agent
    use scenario: scenario.Criteria(el) {
      bind addNeighbour to ???
      bind me to drR.me
    }
  }
  species ConstraintAgent(el: IScenarioElement, name: String) {

    part arch: ConstraintAgentComponent[DirRef] {
      bind send to sender.a
      bind die to sched.stop
      bind getValue to valueObserver.getValue
    }
    part valueObserver: SpecificValueObserver[String] {
      bind getValueOf to vr.a
    }
    use sched: sched.Agent {
      bind act to arch.act
      bind pAd to arch.perceiveAndDecide
    }
    use scenario: scenario.Constraint(el) {
      bind addNeighbour to arch.addNeighbour
    }
    use sender: sender.Agent
    use vr: vr.Agent
  }
}
```

Figure 4.4: Micro-architecture, Cycle 1: MAS Ecosystem

```
component ClockController {
  requires control: interfaces.SchedulingControl
  provides startRunning: Do
  provides stopRunning: Do
}

component GraphicalClockController specializes ClockController {
  part gui: SchedulingControllerGUI {
    bind control toThis control
  }
}
```

Figure 4.5: `Clock` component and its specialisation in SpeADL

```
ecosystem Scheduling {
  species Agent {
    provides stop: Do

    requires pAd: Do
    requires act: Do

    use scheduledPAndD: scheduledPAndD.Agent {
      bind cycle toThis pAd
    }
    use scheduledAct: scheduledAct.Agent {
      bind cycle toThis act
    }
  }

  provides stop: Do = executor.stop
  provides startAgents: Do = clockController.startRunning

  part scheduledAct: Scheduled {
    bind sched to executor.exec
  }
  part scheduledPAndD: Scheduled {
    bind sched to executor.exec
  }

  part executor: ExecutorService
  part sequencer: proto.AgentSequencer {
    bind tickPerceiveAndDecide to scheduledPAndD.tick
    bind tickAct to scheduledAct.tick
  }
  part clock: Clock {
    bind sched to executor.exec
    bind tick to sequencer.globalTick
  }
  part clockController: ClockController {
    bind control to clock.control
  }
}
```

Figure 4.6: Micro-architecture, Cycle 1: Common Scheduling Ecosystem

```
ecosystem ScenarioManager[MsgRef] {

  part gui: TuningGUI

  species Criteria(el: ICriteria) {
    requires addNeighbour: Push[AgentNeighbourInformation]
    requires me: Pull[MsgRef]
  }

  species Constraint(el: IScenarioElement) {

    use gui: gui.Agent(el)
    provides logActivity: Push[AgentLog] = gui.logActivity

    requires addNeighbour: Push[AgentNeighbourInformation]
  }

  provides updateMeasuredCriteria: Push[IMeasuredCriteria]
  provides updateParameter: Push[IParameter]
  provides loadScenario: Push[IScenario]

  requires agentFactory: AgentFactory
}
```

Figure 4.7: Micro-architecture, Step 1, Ecosystem Cycle: Scenario Manager Ecosystem

This component needs that the values to visualise are pushed by the agent itself, a required port in their internal architecture should thus be added in a following step.

New Requirements. In the end, we notice that the description that new requirements towards the internal architecture of the species were introduced. First in the form of required ports by `vp.PublisherPull`, `scenario.Criteria` and `scenario.Constraint`. But also in the form of the provided port `logActivity` for `scenario.Constraint` that should be exploited by the internal architecture of the species. Interestingly, new requirements in the micro-level architecture are not only materialised as required ports, but also as provided ports.

4.4.3.3 Cycle 2, Species Phase

The second step is to refine the internal architecture of the species to add the needed port from the previous cycle. On top of that, we add a required port for the behaviour and the internal architecture of the species named `logActivity` that must be bound in the following ecosystem cycle to `TuningGUI` through `ScenarioManager`. Since the tuning GUI can be activated for any species, this required port should be added to all of them.

4.4.3.4 Cycle 2, Species Phase

As an answer to the previously added requirement ports, for the species `ConstraintAgent`, we bind it to the port provided by the use `scenario`. Inversely, for the species `CriteriaAgent` where we didn't add the `TuningGUI` species, it is needed to add something that provides a fake port that does nothing when called. We choose to take care of that directly in

```
ecosystem ScenarioManager {

  // ...

  species Criteria(el: ICriteria) {
    part voidp: meta.Void[Push[AgentLog]]
    provides logActivity: Push[AgentLog] = voidp.port

    requires addNeighbour: Push[AgentNeighbourInformation]
    requires me: Pull[MsgRef]
  }

  // ...
}
```

Figure 4.8: Micro-architecture, Cycle 2: Scenario Manager Ecosystem

`ScenarioManager`. It takes the form of a component such as `Void` that have a generic port doing exactly that. Figure 4.8 shows the new species.

To complete the ecosystem, the game trainer GUI must be added as well as a component to communicate with the game platform. In particular, they would exploit ports provided by the component `ScenarioManager` to get and update the values of the agents.

As advised in the previous chapter, it is better to keep these concerns outside of the MAS ecosystem and uses MAS-agnostic ports and interfaces to manipulate entities of the scenario without knowing that they are agents. Such ports should be added at that point of the design, possibly introducing new requirements for the ecosystem.

4.4.4 Complete Design and Implementation

From this design, the implementation should be as straightforward as possible. Obviously, in practice, roundtrips between implementation and micro-level architectural design were done to get the solution presented here. In particular, this was useful to find out which was the best decomposition in components and which were the exact requirements of each of the components.

The complete architectural description and implementation is available as an example on the website of the MAY tool.

4.5 Conclusion

In theory, once the proposed design is finished, the MAS developer can focus on implementing the behaviours of his agents. In practice, the whole design was done by the same developer. But an effort was done to actually explicitly differentiate business and operative elements of the architecture, mainly because operative components were already available from the library, but also as a way to make the business part of the implementation as close as possible to the MAS design.

Several conclusions can be extracted from this example, in particular with respect to what would have had happened if the contribution wasn't used:

1. If SPEAD had not been used, then it would have been much more difficult to keep the mapping between the macro-level design and the micro-level design, in particular for types of agents. Furthermore, in order to reuse interaction means, most certainly an existing development support would have been used. In particular its integration with the domain-specific logic for initialising the MAS and its agent realised by `ScenarioManager` would have been much more difficult in our opinion. Then, the composition of the two means for interaction is most certainly not provided by any existing development support for MAS, and thus two solutions would have had been possible:
   a) Compose two development supports, which wouldn't have been too much complex, but still not straightforward.
   b) Only use messages, which would most certainly, from our observations of current practices, what would have been done.

   Finally, the specific way the system is scheduled would have had to be either implemented by triggering agents using messages, as we already witnessed in the past, or by manipulating directly the objects implementing the agents from the code of the platform. The latter case would have introduced a lot of implicit synchronisation logic at different places.
2. If the general methodology of MAS development that we propose had not been applied, we think, as we often witnessed in practice and from what we assumed from the works we read in the field, that the implementation would have started just before micro-level requirements extraction presented Section 4.3. Moreover, because macro-level requirements would not have had been clearly identified, it would have been more difficult to identify which were the requirements that needed to be tackled after the MAS design. Thus, as we saw, what was uncovered in these sections have been useful to explicit assumptions that were made on the MAS design itself and thus to ease the implementation. This also makes the whole design much more readable and analysable, and most certainly, evolutions of the MAS design are easier to apply to the implementation.

This concludes the example of application of the approach, the next chapter draws more general conclusions on the contribution.

Summary of the Contributions

- ⊕ We propose a complete application of the methodology of MAS development using SPEAD and SPEARAF.
- ⊕ We exploit the reusable components of the library.
- ⊕ We propose a short analysis of the advantages of using our contribution.

CHAPTER 5

Positioning, Analysis and Experimental Feedbacks

In this chapter, we present an analysis of the contribution presented Chapter 3 and Chapter 4. We position the whole contribution and the example with respect to the classification of the works presented Chapter 2. Then, we present how the contribution and in particular the proposed component model answers the challenges identified Chapter 2. This analysis is discussed to extract the motivations and advantages of using our component model. Finally, we present academic and industrial works that used our contribution and detail some experimental feedbacks from our users.

5.1 Positioning the Contribution

Chapter 2, we presented a classification that presented six classes of research works about MAS development. On the other hand, our contribution is decomposable in elements, each potentially positionable in the classification. To that, we add the example as well as the method used for its development. This gives us the following set of elements to position:

For a detailed presentation of the *Classification*, see p. 37

- The architecture-centric methodology, Section 3.1
- The SpeAD component model, Section 3.2
- The guidelines for capturing and reusing experiences, Section 3.3.1
- The library of components produced with SpeAD, Section 3.3.2
- The SpEArAF design method, Section 3.4
- The ADELFE method and the AMAS approach, Section 1.3.3
- The macro-level design of the VGD, Section 4.2
- The micro-level design of the VGD, Section 4.4
- The implementation of the VGD, Section 4.4.4

Table 5.1: Contributions positioned in the proposed classification

	Base	Meta
Macro-level	*Macro-level design of MASs applications*	*Methods, approaches, models and tools to support macro-level design*
	VGD macro-level design	ADELFE AMAS
Micro-level	*Micro-level design, including in particular development supports for implementation of MAS*	*Methods, approaches, models and tools to support micro-level design*
	Library of components VGD micro-level design	SpeAD Reuse guidelines SpEArAF
Implementation	*Implementation of macro-level design using the micro-level design*	*Methods, approaches, models and tools to support this implementation*
	VGD implementation	

Table 5.1 summarizes the different classes of research works and shows the result of this positioning. Obviously, the methodology doesn't fit into the classification since the latter is extracted from the vision expressed by the methodology.

5.2 Analysis

The proposed methodology answers directly the specificities of MAS development we identified Chapter 2. But it is necessary to question the SpeAD component model, because it is far enough from the challenges we identified. We thus present here an analysis of the different aspects of the component model in order to justify its relevance. We illustrate it using the example developed Chapter 4. We don't evaluate the method itself as it relies entirely on the component model concerning the identified challenges.

We recall the identified architectural and implementation challenges:

- Define and implement interconnection mechanisms for connecting agents to the runtime platform and vice-versa.
- Define and implement types of agents with specific internal architecture and interconnected to the runtime platform.
- Provide dynamic creation of instance of these types of agents and their dynamic connection and initialisation.
- Make the reuse of such interconnection mechanisms and architectures easy.

For a definition of *Architectural Component*, see p. 6

Following the characterisation of architectural components presented by Bachmann et al. 2000, we look at two aspects of the abstractions introduced in SpeAD: architectural and implementation. The former is meant to see the species and ecosystem as a design abstraction for answering architectural objectives and describing the architecture. The latter is meant to

see the species and ecosystem as means of implementing component-based architectures with respect to their role as architectural abstractions.

For both of these aspects, we look at two points of view: agents and interconnection. The first one is about answering the need to realise types of agents, while the second one is about answering the need to realise interaction and interconnection mechanisms. This gives us the possibility to differentiate between a species in an ecosystem as a type of agents, and a species in a ecosystem as an architectural element located inside agents.

We focus on the MAS aspect of the solution and do not comment the possible advantage of the model for component-oriented programming.

5.2.1 Architectural Abstraction

From the agent point of view, species are first and foremost a way to explicit the types of agents at the architectural level. Indeed, in MASs, it is obvious that having agents is an important design choice and thus should be made explicit in the software architecture of the built system. They explicit the existence of dynamically created entities that interact together using specific interaction means. Furthermore, by doing so, the species drives the definition of the rest of the architecture, *i.e.* the ecosystem and the species internal architectures. It also helps to identify what must be in the agents or outside of them, what is going to be duplicated in every agent and what is centralised in the runtime platform.

From the interconnection point of view, obviously, species play the same role of making explicit the use of specific interaction mechanisms and the existence of needed links between the agents and the runtime platform. In a way, such explicitly specified interconnection mechanisms are a guarantee of the autonomy of the agents, which is an important feature of the MASs and often forgotten point during implementation. Indeed, when implementing MASs, it is easy to introduce runtime dependencies between what is inside the agents and what is in the platform: for example environment entities directly manipulating data in the agents, which could breaks the assumption made at design. Thus, by having species use species of their ecosystem parts, the places were the autonomy of the agents is threatened are clearly identified. The `TuningGUI` in the previous chapter is a good example of this: without such abstraction, the answer to this requirement would have most certainly be hidden in the implementation of the system. This is particularly important also when implementing prototypes Furthermore, this abstraction enables to clearly identify reusable and composable components realising recurring interconnection mechanisms. The examples of interconnection mechanisms in the previous chapter are self-explaining.

Finally, the composition of these interconnection mechanisms in species to represent types of agents act as a way to encapsulate the definition of a type of agents behind a clearly defined interface. Indeed, everything not explicitly needed outside of the description of a type of agents does not have to filtrate out, and in particular not to the components that wants to create new instances of agents. A good example of this is the status of the reference of agents, which are used internally in the species as well as in the used species, but not needed to create an agent. In particular, this separate the concern of creating an agent and

implementing this creation. Indeed, they do not have to know how the agents that are created are linked to the ecosystem and what is their internal architecture.

5.2.2 Implementation Abstraction

From the interconnection point of view, the species is a way to implement, either directly or by composition of other species, a $1 - N$ link. It contains an implementation for the unique part, for the duplicated part, and more importantly for the initialisation of the latter. Indeed, by encapsulating the initialisation of new elements meant to exist in agents, it abstract over the dynamic creation of these elements. Using the ports, it gives the possibility to explicit the needed mechanisms to work with the other dynamically created elements of the species architecture. For example, the references mechanism composed with the messages receiving mechanism let the definition of the internal architecture be independent of the way these mechanisms are realised. In a way, it is a mean to implement the functionality required by the internal architecture of agents by abstracting over their multiplicity. It is a way to implement individual-centric mechanisms so that the user does not bother with questions about dynamicity or multiplicity. Again, `TuningGUI` is a good example of that.

From the agent point of view, the species allows to compose interconnection mechanisms. Describing which ones are used and how they are connected is enough to have a working implementation for the species. Moreover, it abstracts over the dynamic creation of instances of the species as well as the creation of all of the species used. Furthermore, such composition of interconnection mechanisms enables to easily add or remove elements from the species without changing the rest of the system. This is well illustrated by the `TuningGUI` which is easily removable.

In other words, what our model introduces are abstractions that are a way of realising a kind of recurring pattern that answers the recurring problem of dynamically creating and connecting entities to a platform. This is particularly interesting in the case of agents creating other agents. Because the agent construction mechanisms is taken care in each of the interconnection mechanisms, composed together in an ecosystem through the definition of a species and hidden behind a port definition, there is then no need to manage the fact that the creating entity is itself an agent that was created dynamically. Implementing such things using for example object-oriented abstractions is difficult because the links between the created entity and the platform would have had to be handled by the creating entity.

5.2.3 Why and When to Use SpeAD

Thereby, ecosystem and species are seen as abstractions that are useful, in the context of MASs, for expliciting architectural choices, enforcing them and implementing them. Even though this is of first importance for developing software in a realistic way, this does not completely justify the use of the specific component model proposed here for any kind of applications.

Indeed, to build MASs, a lot of existing development supports provide means to connect agents together using a diversity of interconnection mechanisms. The precepts presented in this thesis could be applied with these without reinventing these very common interconnection mechanisms as we did in the previous chapters. This also questions the interest of building a complete platform for every application.

For a definition of *Agent-Based Modelling*, see p. 12

But as we showed in the state of the art and with the example, there is sometimes the need for building such interconnection mechanisms adapted to the MAS design. The `TuningGUI`, that we used a lot as an example in the previous section, is a good motivation: it shows a development-specific requirements answered easily thanks to the abstractions. This point also manifests itself in ABM (Agent-Based Modelling), which is one of the type of MASs we are particularly interested in. Indeed, for this class of application, the MAS design is strongly influenced by the domain model.

In all these situations, our model provides what is needed to build such adapted interconnection mechanisms. But even more importantly, it makes possible to compose them in a coherent and adapted way, while at the same time making such mechanisms reusable. From our point of view, this last point is maybe the most challenging technical objective that the model answers.

Thus, coupled with the need for following software architecture principles, all of this justify the need for starting back from a cleaner and clearer development model for MAS engineering.

5.3 Experimental Applications and Users Feedbacks

We now present the different academic and industrial works that used our tools and approaches, and present some feedbacks we got.

Our contribution was mainly applied internally in the SMAC research team, but also in the UPETEC start-up company.

For a presentation of *SMAC and UPETEC*, see p. xxii

As it was said in the introduction, our contribution has been been published in two versions before we ended up with the contribution in this thesis. For each of these, we released tools, mainly focused on component-based engineering for agents and MASs. These tools promoted the use of the methodological concerns we expressed in this thesis.

The first version of the contribution (Noël, Arcangeli, and Gleizes 2010a; Noël, Arcangeli, and Gleizes 2010b) is mainly focused on the methodological separation between macro-architectural design and micro-architectural design with a tool that supported the definition of component-based internal agent architectures. The tool was used in several works and projects with industrial partners such as:

- Biological simulation (Bonjean, Bernon, and Glize 2009).
- Dynamic ontology construction (Sellami and Camps 2012) in the context of the national DYNAMO [1] project.

1. DYNAMic Ontology for information retrieval: `http://www.irit.fr/dynamo/`

- Naval surveillance (Georgé et al. 2009) in the context of the national SISMARIS [2] project.
- Self-organising team of robots self-allocating tasks (Lacouture, Noël, et al. 2011) in the context of the regional ROSACE [3] project.
- Distributed constraint optimization (Kaddoum 2011).
- Cabling optimisation for planes in the context of the regional SMART [4] project.

Then, the second version of the contribution (Noël and Arcangeli 2011; Noël, Arcangeli, and Gleizes 2012) introduced the species and ecosystem components abstractions. It is very close to what is presented here and is fully compatible with it. This version of the contribution is broadly and currently used: the tool implementing it is the current main and most stable version of it. In particular, it was used:

- In the context of research projects with industrial partners such as the ID4CS [5], GAMBITS [6], ORIANNE [7].
- By Ph.D. students of the team in applications such as dynamic management of context in ambient systems (Guivarch, Camps, and Péninou 2012), self-composed method fragments (Bonjean et al. 2012), adaptive combustion motor calibration, adaptive energy regulation, multi-disciplinary constraint optimization, etc.
- By a Master student during its internship that resulted in the system for opportunistic self-composition of ambient components (Denis et al. 2012) presented Chapter 7.
- To introduce component-based programming to Master students.
- In the context of a Master student project for university schedule timetabling using MASs.

Moreover our methodology of MAS development was used to organise the development of the system for dynamic workflow adaptation (Cruz Torres et al. 2010b; Cruz Torres et al. 2010a) presented Chapter 7.

It would have been interesting to do an evaluation, with our users, of the proposed contribution, of its implementation itself and of the systems that were produced using it. Unfortunately, this has not been possible, but

The general feedback is positive, once a time of learning has passed. Indeed, they all are discovering software architectures as a design approach and component-oriented programming as an implementation approach.

The main advantages expressed by the users are about reuse and maintenance. The library of components along with the help of people (including us) familiar with the approach are

2. Système d'Information et de Surveillance MARitime pour l'Identification des comportements Suspect: http://www.sismaris.org/
3. RObots et Systèmes Auto-adaptatifs Communicants Embarqués: http://www.irit.fr/Rosace,737
4. SMart-hARness Technologies: http://www.irit.fr/SMART,1178
5. Integrative Design for Complex Systems: http://www.irit.fr/id4cs/
6. GAMe-Based IntelligenT Strategie: http://www.gambits.fr/
7. Outil numéRIque pour le mAquettage de foNctions de coNtrôle motEur: http://www.irit.fr/ORIANNE,1176

very useful in starting a new application. Once people have a working base, they become more independent and build their software on top of it. For example, we got a feedback of someone happy to be able to easily change the way their system was scheduled with as little actions as possible. Or another told us they built their own set of reusable components for rapid prototyping and debugging.

Concerning teaching, the tool was well accepted in the classroom and proved to be robust enough to pass the "student crashtest".

5.4 Conclusion

We presented a positioning of the contribution with respect to other works for MAS development. We draw an analysis from which we drew a conclusion on the advantages of using our component model. We presented the academic and industrial works that applied our approaches and used our tools.

From all of that we concluded that what we propose in this first part of the thesis is useful to MAS development and positively accepted by people that actually design and implement MASs. We are confident, from the feedbacks we got from them, that the tools we gave them helped them in producing better software.

As a conclusion, we also have the feeling, through our observations, that applying the methodological aspects of our approach helps the design of MASs itself. Indeed, by applying such software architecture principles to the development of a MAS-based application, it clarifies what is the MAS really answering and what relations it has to the rest of the software being built. Even if it is most certainly too soon to conclude that, in our opinion, all of this is particularly useful when presenting results to other researchers that are familiar with the work we do in the team.

We also take this opportunity to raise the attention on research works published lately in notorious interdisciplinary journal (Ince, Hatton, and Graham-Cumming 2012; Morin et al. 2012). They argue that systematically making the implementation freely available with research papers would be beneficial to the research community. We think that our contribution goes towards such objective for the following reasons:

- It makes the design and the implementation of MASs much more accessible to external observers by emphasising on architectural documentation.
- It proposes a frame to which the diverse people can refer to when talking about MASs as a software.
- It makes reuse and sharing in the community easier.

Summary of the Contributions

⊕ We position the contribution as well as the produced software artefacts in the classification presented Chapter 2.

- ⊕ We analyse the contribution with respect to the architectural and implementation challenges identified Chapter 2 using a recognized frame of characterisation for architectural components.
- ⊕ We show that the contribution was used in many different works and that we got positive feedbacks about it.

Part II

INTEGRATING MULTI-AGENT SYSTEMS AND SOFTWARE ARCHITECTURES

CHAPTER 6

MASs and CBSAs Side by Side: Component-based Component Containers

> More is not better (or worse)
> than less, just different.
>
> *The paradigm paradox*
> Peter Van Roy (2004)

In this chapter, we look back at the proposed approach of producing dedicated development support and at the SpeAD component model from the point of view of the software architecture field. Indeed, even though the contribution of this thesis is inspired by many works from a diversity of fields, its main innovations are at first only meant to answer the challenges identified Chapter 2. The contribution of the first part of this thesis — namely the definition of partially abstract component-based architectures to produce development supports dedicated to applications and the definition of explicit dynamically created entities connected to a runtime platform — can thus now be positioned with respect to existing works to see in which points they are alike or different.

All of this brings us to reinterpret the contribution with respect to component models and containers, and all the works evolving around Component-Based Software Architectures (CBSAs). In particular we:

- revisit MASs design approaches as a family of architectural styles and design paradigms.
- reconsider MASs as components in CBSAs.

Based on that we show how SpeAD helps to achieve horizontal integration between MASs and components-based architectures and gives the possibility to software system designers to choose and combine relevant design approaches when building applications.

Secondarily to that, in order to reach this conclusion, we explore the relations that exist between frameworks, components models, component containers, architectural patterns, architectural styles, architectures and components. This brings us to propose a way of defining and using dedicated component containers that can be implemented using SpeAD.

The content of this chapter is exploratory and mostly exists to draw some, in our opinion interesting, axis of research on the general matter of design and implementation of development supports for the component-oriented development field. Although, some of the works tackling such questions in that field try to get as formal as possible, here we only are manipulating abstract, and thus sometimes imprecise, concepts in order to reinterpret concepts from the MAS field in the broader context of software architectures.

In a way the present chapter has a focus on the relations between architectural concepts and MASs, but considering the latter at design, while next chapter considers MASs at runtime.

6.1 Related Works in the Software Architecture Field

6.1.1 Reusing High-Level Design

In this section, we first compare our work to Software Product Lines and then to object-oriented frameworks. We actually don't rely on the ecosystem and species abstractions, but only on the idea of producing partially abstract architectures with component-oriented abstractions.

6.1.1.1 Software Product Line

One of the objectives of the contribution presented Part I is to reuse experience along different domains, applications, problems, etc, of MASs. In this sense, our work comes within the scope of Software Product Lines (SPLs) and ultimately could result in easing the development of SPLs for MASs.

For a reminder on *Software Product Lines*, see p. 8

Nevertheless, while SPLs propose to define reusable architectures from which dedicated architectures are built, our current contribution only focuses on the reuse of components to build dedicated architectures. From this point of view, the experience reused is only at the level of components and not architectures.

For a presentation of *Template*, see p. 69

On the other hand, as we noted Chapter 3, it is possible to reuse design using templates. However, this approach has some shortcomings, which are mostly concerned with the SpeAD component model. Indeed, in its current state, it is not possible to specialise ecosystems nor species. This thus means that the model is not able to represent reusable ecosystem with abstract components in the species.

For example, if building a MAS development support where the agents behaviours are left open, this would be represented with an ecosystem where the behaviours of the species are abstract components and where the implementation makes concrete all the other components of the ecosystem and the species. But it is not possible to use such an ecosystem and its implementation directly with concrete behaviours for the agents without modifying either

Table 6.1: Mapping between frameworks and partially abstract architectures concepts

Frameworks Concepts	**Partially Abstract Architecture Concepts**
Framework	Composite component with a partial implementation
Hotspot	Abstract component (without implementation)
Frozenspot	Concrete component (with an implementation)
Hotspot Contracts	Provided ports and interfaces definitions
Hotspot Programming Abstractions	Required ports and interfaces definitions

its description or its implementation. A wanted feature would be to be able to specialise the reusable ecosystem without modifying it and to be able to specialise its species in order to specify their behaviours.

6.1.1.2 Object-Oriented Frameworks

On a different note, but still in the domain of reusing experience, code and design, frameworks are known to be a way of implementing reference architectures for SPL (Greenfield and Short 2003). Frameworks, as presented by Markiewicz and Lucena (2001) or Johnson (1997), are partially implemented object-oriented application that can be reused to create full-fledged applications: they are application "generators". They are also considered as in-between code and design: high-level design can be reused along with code. They have hotspots and frozenspots: the former are the places where application-specific code goes, while the latter are generic parts already implemented by the framework. Implementing the hotspots implies respecting the contracts the framework imposes on them, but in return allows to exploit programming abstractions the framework provides.

Frameworks have always been promoted as a way to reuse code using object-oriented abstractions. In a similar way, as highlighted in the previous section, Chapter 3 explains how partially abstract architectures can be modelled using templates: components left abstract in a composition of components.

We thus think that **partially abstract architectures are a way to model component-based frameworks** where abstract components represent the concepts of hotspots and their constraints. And thus, SpeAD is usable to describe and implement such component-based frameworks.

Table 6.1 shows the mapping between the concepts of the two. In order to implement an hotspot, one must implement its corresponding component provided ports. The framework gives access to its functionalities through programming abstractions that are represented by the required ports of the abstract component. And the concrete components are the frozenspots of the framework.

Even though we don't study in detail this way of defining and building frameworks, we think that it is interesting enough to be highlighted and commented. Our opinion on the matter is that such an approach eases the description and the exploitation of frameworks

because:

- It uses *explicit* provided interfaces to define the hotspots.
- It uses *explicit* required interfaces to define the programming abstractions usable to program the hotpots.
- It takes the advantages of component-oriented abstractions for *reuse and separation of concerns* to build frameworks.
- It makes possible to more easily define *hierarchically* frameworks and *compose* them together.

To conclude on the last point, as it was coined in several works (Johnson 1997; Bachmann et al. 2000; Crnković et al. 2002), components and frameworks are conceptually reifiable in the sense that frameworks are themselves usable as components and are at the same time containers for components. We think that our proposition of component-based frameworks makes this feasible in practice with real component-oriented abstractions. Indeed, it is possible with it to describe frameworks that are themselves components, possibly with hotspots kept open. They can thus be composed together in composites, but also used as the building blocks of other component-based frameworks. Obviously more studying should and must be done on the matter, first to ease the use of such approach, for example using SpeAD, but also in the MAS field as SpeAD has limitations regarding partially abstract ecosystems and species.

6.1.2 Dynamic Component Creation and Connection

In this section, we look at the fact that SpeAD species are descriptions of classes of components and that these components instances can be created dynamically. We position this abstraction with those proposed in ADLs (Architecture Description Languages) and in more general programming languages. We focus on means to describe declaratively and structurally how components can be dynamically created because for other means, our contribution does not radically differ.

For a reminder on *Architecture Description Languages*, see p. 6

In ADLs, a good example of such mean is the *direct dynamic instantiation* mechanism offered by Darwin (Magee and Kramer 1996). We focus on this one since we consider it a good stereotype of what is generally provided in ADLs. In these works, the idea is that it is possible to describe that an element of an architecture can be dynamically instantiated by another component. The latter triggers the creation of instances whenever it wants by using the mechanism provided by the component model. Both dynamically created components and creator components are at the same level in a composition.

In Darwin, dynamic component creation is triggered by other components that requires a special type of port. In order to keep the architecture valid — *i.e.* that all required ports of components are connected to the provided port of another one — it is possible to define how the instantiable type of component is connected to the rest of the architecture. As Darwin authors point out, only the required ports connections of a dynamically instantiable component can be described, as there wouldn't be any way to describe how provided ports are

connected (since this would mean requiring multiple provided ports). The way to alternatively do that is to use what they call *service references* that bring down the concept of ports at the programming level and thus are not handled at the description level.

Inversely, with SpeAD, the relation between the created component — instance of a species — and the creating component — the ecosystem declaring the species — is the one of contained and containing. Thus, the dynamically created component is connected to the rest of the world through its ecosystem, which, by the choice of *uses* of its parts in the species, constrains the way the instantiated component can interact with others. Obviously, this is completely different from what is done in traditional ADLs since the *uses* of the species describe the interaction means available to it and not just how it is connected. We come back on that point Section 6.4 to show what can be expressed with that.

Here, the important point is that we define how instances of species are created and connected by relying on the semantics of their *uses*. This gives the possibility to be much more expressive than traditional ADLs, while guaranteeing that the composition is valid by construction. In exchange, the point where uncertainty is present is the interconnection mechanism, the *use*, and its semantics. We come back on that point Section 6.6 to discuss why and how to use such solution instead of others.

When looking at component-oriented programming as an evolution of object-oriented programming, this resembles a lot non-static inner classes (Gosling et al. 2005) from object-oriented languages such as Java: these inner classes can only be instantiated from an existing object. We can thus say that with species, we ported the inner class concept to component-oriented programming.

6.2 Relations between Component-based Architectural Concepts

In order to prepare to what is presented in the rest of this chapter, we now present our vision of **the relations existing between architectural concepts such as frameworks, components models, component containers, architectural patterns, architectural styles, architectures and components**. As with the rest of the thesis and explained Chapter 1, we elude connectors. Moreover, in this section we completely put SpeAD aside.

Figure 6.1 illustrates such relations that we now detail. Next chapter comes back on these relations in a runtime context in the next chapter.

Monroe et al. (1997) present *styles* as providing specialized design languages for specific classes of systems. They define a set of authorised types of components and constraints that an architecture must *instantiate* to be well-formed with respect to the style. But they can also constrain and be *instantiated* by more abstract architectures expressed as *patterns*.

As presented by Bachmann et al. (2000), architectural styles and *component models* are similar in that they try to answer the same problems in the sense that they provide abstractions adapted to a specific need. More precisely, it seems that architectural styles are more diverse, and that there exists a specialisation relation between them as defended by Fielding (2000). We consider that existing component models *allow* to express architectures that instantiates one or

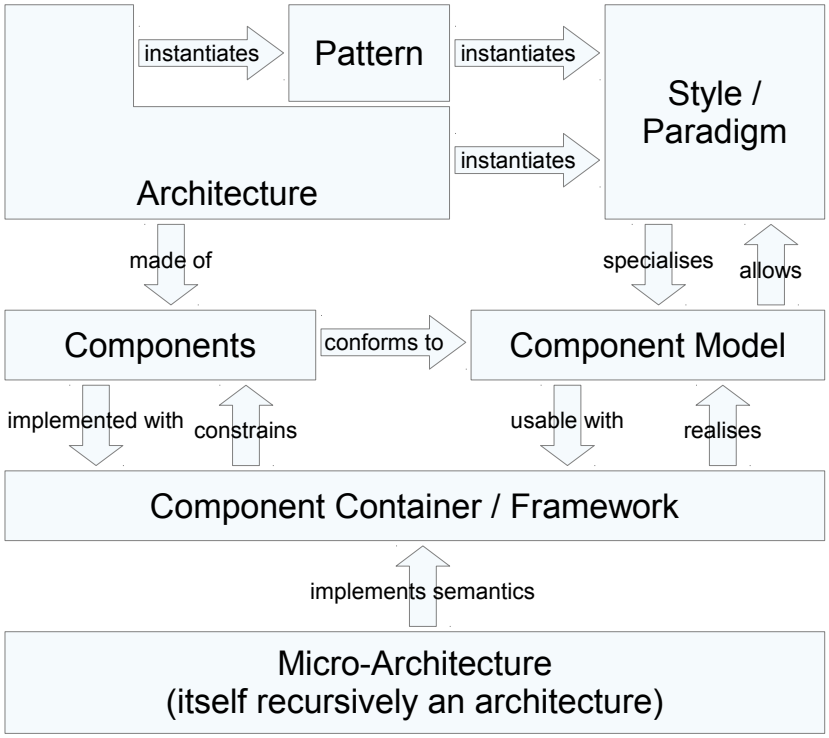


Figure 6.1: Relations between frameworks, components models, architectures, components, architectural patterns, architectural styles and paradigms

several styles, but that styles do not all directly correspond to a component model. However, a style can be a *specialisation* of a style that directly corresponds to a component model. Thus, styles and component models both define abstractions usable to express architectures, and the elements of these architectures, the *components*, *conforms* to the corresponding component model.

Frameworks, as development supports, *realise* component models (and thus styles) (Johnson 1997; Bachmann et al. 2000) in the sense that they permit *to use* the component model. In other words, using a framework, one can *implement* components that conforms to a model/style. The hotspots of the frameworks are used to express such implementation.

But frameworks, as runtime entities, realise *component containers* (Bachmann et al. 2000) of component models. Using a framework, one can execute components within the constraints of this model. The implementation of the framework, the frozenspots, is the implementation of the component container. Such an implementation is realised by an architecture, internal to the framework, called *micro-architecture* [1] (Johnson 1997; Monroe et al. 1997; Crnković et al. 2002) in order to differentiate it from the architectures built using the frameworks.

For a reminder on *Paradigms*, see p. 3

To these relations, we add another one between paradigms and component models, and thus styles. A component model can be seen as a dedicated programming and design

1. As we are going to see below, the use of the same term Part I is not accidental.

paradigm, or specialised design language as said above. Like a paradigm, a component model defines a coherent set of programming abstractions. The components that conform to the component model are implementable using such a paradigm using the abstractions provided by the hotspots of the framework/container. The container acts as the abstract machine that executes such programs/components.

6.3 Defining and Using Dedicated Component Models and Containers

We now look at what must be done when one wants **to define a component model and the container that allows to use it to implement components**. The need for such a thing is developed in the next chapter where we use self-organising MASs as the semantics of diverse component models.

We rely on the relations between component-based architectural concepts identified in the previous section to help the identification of the important elements to define. The reader can relate to Figure 6.1 for a better understanding of the concepts manipulated here. Again, in this section we put SpeAD aside.

This section proposes our own understanding of how to:

- Define a component model.
- Define a component container/framework, *i.e.* define the hotspots to implement components.
- Use them for programming and executing components that conform to and rely on the model and container.

As we said in the introduction, this is done in a quite abstract way that cannot be considered a formal demonstration, but only an informal presentation of our ideas.

A simple example is used to illustrate these different activities. Next chapter shows more complex examples that exploit them.

6.3.1 Defining the Component Model

It is first needed to define the dedicated component model and its semantics. There exist a lot of works in the field proposing definition for the semantics of component model, we don't dare to concurrent them, but we only present what interests us here in order to define component containers next. As a basis for discussion, by semantics, we could mean "the set of component types, their interfaces, and, additionally a specification of the allowable patterns of interactions among component types" as proposed by Bachmann et al. (2000), but we also mean, inspired by the definition of architectural styles, the constraints that the components must conform to. But what interests us the most here is the semantics of the composition: what are the means of expressing how components are composed, following which dynamics, etc.

For example, a traditional event-based component model could be defined as allowing that components, executed each in their own process, are defined with provided and required ports to send and receive events, and are composable in static configurations. This then means that components must only be implemented in terms of their ports, reactions to and sending of events, or in terms of compositions of components.

6.3.2 Defining the Component Container

As we previously highlighted, a good way of realising and implementing a component model is to build a component framework, also called component container. In other words the component framework/container is the development support usable to describe and implement the components, including the description of how they are composed together.

The idea is to define clear hotspots to represent how to program the components, but also to potentially "configure" the container itself. These hotspots should be elicited from the component model defined previously.

The general idea is to have what we call **component hotspots** to describe each type of components that can be programmed with the container. Indeed, as we said, a component conforming to a component model must comply to some constraints and are allowed to use some patterns of interactions. For example, for a traditional ADL there could be two different component hotspots:

- Both would give the possibility to define the set of provided and required ports and a way to identify the component.
- One would give:
  - The possibility to implement the different provided ports.
  - Access to the required ports.
- The other would give the possibility to express how components, themselves expressed with the hotspots, are composed with each other.

On top of that, there can exist **container hotspots**. Indeed, depending on the context, there could be specific things to express about the container itself and not the components that live inside it. For example for a component model that let distributed components communicate together transparently, this could contain the information needed for every instances of the container to know each other.

6.3.3 Implementing the Component Container

Then, based on the defined hotspots, the container/framework can be implemented to realise the semantics of the component model. This corresponds to the building of the micro-architecture of the component container, or in other words, implementing the frozenspots. We don't detail here how to implement containers, but this approach is obviously consistent with how object-oriented languages can be used to build frameworks, and thus component frameworks/containers. Section 6.4 shows how this is also actually practicable with the abstraction introduced in SpeAD.

6.3.4 Using the Component Container

Based on that container, one can now realise an application that relies on the defined component model. In particular, it clearly separates the concerns of composition or adaptation provided by the model from those of the application and its business objectives. Examples of such applications are given in the next chapter.

6.3.5 Going Further

To conclude, all of this is just an introduction to the question of defining and using dedicated component models, but a lot of things can be investigated in this domain. In particular, we think about the links with existing works and in particular DSLs (Domain Specific Languages), which is a proven solution to make the use of a dedicated language programmatically practicable (Ghosh 2010). They are an interesting alternative to frameworks for defining and building component containers.

For a definition of *Domain Specific Language*, see p. 4

6.4 Building Dedicated Component-Based Component Containers

Section 6.1 identifies in SpeAD the role of species and ecosystem respectively as contained and containing as well as the fact that it is usable to build component-based frameworks. Section 6.2 examines the relations between architectural concepts to build and reuse software. Section 6.3 shows how one can define and use dedicated component models and containers. We can now get a new way of interpreting the things we can build with SpeAD.

Indeed, the main idea of this section is that by defining species with abstract components, we actually realise the micro-architecture of component containers/frameworks for component models. In particular, the *uses*, possibly coupled with parts, realises the implementation of the allowable "interaction patterns" for the components that conform to the component model.

In other words **we are building dedicated component-based component containers**.

Several works defend a vision where component models, programming paradigms or architectural styles are things that can be built in order to be dedicated to a specific need.

(1) Bachmann et al. (2000) say that "what is needed is a technique for constructing component models from a kit of model fragments known to support particular quality attributes".

(2) Van Roy and Haridi (2004) say that programming paradigms are actually a composition of programming concepts adapted to some needs in term of expressiveness.

(3) Fielding (2000) proposes to define styles as incrementally built by adding constraints in order to get particular qualities for the architectures that instantiate it.

Our opinion is that the composition of several species into one composite species with the *use* relation corresponds to such idea. Indeed, we can see the *uses* as the fragments of (1), as providing the concepts of (2) or as providing the qualities of (1) and (3). The internal architecture of a species is a component of (1), a program of (2) and conforms to a style of (3). The ecosystem of the species is the container of (1) or the abstract machine of (2).

```
ecosystem DirectReferences[I] {

  // name is only used for pretty
  // printing the reference with toString()
  species Callee(name: String) {
    // a port provided by the
    // dynamically created component
    // and callable using the reference
    requires toCall: I
    // the dynamically created
    // component reference
    provides me: Pull[DirRef]
    // to call when the dynamically
    // created component is stopped
    provides stop: Do
  }

  // to call a dynamically created
  // component by reference
  provides call: Call[I,DirRef]
}
```

(a) Direct references mechanism.

```
// an ecosystem realising
collection ports
ecosystem CollectionInteger[I] {
  species Indexed {
    // a port provided by the
    // dynamically created component
    // and made accessible by
    // the collection port
    requires p: I
    // to get the index of the dynamically
    // created component in the
    // collection port
    provides idx: Pull[Integer]
    // to call when the dynamically
    // created component is stopped
    provides stop: Do
  }

  // to access the port of a
  // dynamically created component
  // using its index
  provides get: MapGet[Integer,I]
  provides size: Pull[Integer]
}
```

(b) Collection port.

Figure 6.2: Interaction mechanisms definition in SpeADL

If we now focus on the relation with (1), as we said, by the choice of a composition of *uses* into a species, we can define dedicated component models.

As promised Section 6.1, we present two examples of what can be expressed using our species and their *uses*. They are proposed as ecosystems, with their species, can be *used* inside other species to give them specific "interaction patterns" similar to those that can be found with traditional ADLs or programming paradigm.

The first one was actually already presented Chapter 3: it is the `DirectReferences` component, whose description is reproduced Figure 6.2a. It realises a limited version of the interaction mean used in the object-oriented programming paradigm. An instance of a species that *uses* `Callee` is then accessible through a `DirRef` reference. Then it is possible, using this reference and the `call` port provided by the component `DirectReferences`, to make a synchronised call to the port bound to `toCall` in the dynamically created component instance.

The second one shows how we can realise the concepts of collection ports, *i.e.* a port that gives access not to one component but to a set of component indexed by integers. This type of ports would give a way of solving the question raised Section 6.1.2 by Darwin of connecting the provided port of dynamically created component instances. Figure 6.2b depicts the description of such an ecosystem called `CollectionInteger`. An instance of a species that *uses* `Indexed` is then accessible through the `get` port of the `CollectionInteger` component using an index. This is an interesting example because this concept of collection port is

traditionally encoded directly in the component model of ADLs, while here it is defined using only the abstraction provided by SpeAD.

Obviously these relations and the examples are an over-simplification of the reality, but as with the previous sections, we don't go into the details of all these relations and how a model like SpeAD can help build component models. In our opinion, this matter should be studied more thoroughly, and a lot of things could be said, explored and improved about it. In particular, we think there exist links to identify with existing works that have equivalent objectives (Hofer and Ostermann 2010; Loiret et al. 2011).

6.5 Revisiting MASs Design as a Family of Paradigms

Obviously, a perfect instantiation of what is said in the previous section, when going back to the main subject of this thesis, are agents and MASs. As we said Chapter 1, MASs are a family of architectures: they are conform to a family of architectural styles characterised by an emphasis on direct peer-to-peer interactions, environment-mediated interactions and reorganisation. One example of a style of this family is the one that was defined in the example developed Chapter 4 where:

- Agents interact using messages passing and values observation.
- Agents are scheduled together in two synchronised separate steps.

In this vision, then, building a dedicated development support as we advocate is actually defining the precise architectural style, or component model, needed for programming the agents. In that case, the agents are the components at runtime, and their behaviours are the implementation of the components. In another words, designing and implementing a MAS is, amongst other things, building a dedicated abstract machine for processing the agents' behaviours, programmed in a programming paradigm dedicated to the MAS design. For example, the dedicated programming paradigm of the previous example gives the possibility to:

- Express the two steps of the behaviour of the agents.
- Access their mailbox.
- Publish observable information.
- Send messages.
- Observe some other agents.

This thus shows an interesting point about **MAS approaches: they actually are not one design paradigm, but a family of such**. Indeed, every type of agents (and thus species) introduces its own set of interaction means and semantics for the execution dynamics of the agents' behaviours.

In particular, it is worth noticing that the choice of this dedicated programming paradigm is strongly linked to the choice of the approach as well as the choices made during macro-level design. Indeed, if we look at systems built with an approach such as ADELFE as in the

For a definition of *ADELFE*, see p. 50

example above, the choice of the types of agents is very dependent on the domain problem and the dynamics of the agents is mainly enforced by the approach itself. This point of view on MASs approaches as paradigms should be explored to extract knowledge on the impacts the choices at macro-level design have on the micro-level design.

More precisely, in terms of programming paradigm, **the contribution presented in the first part of this thesis is actually aimed at defining a dedicated MAS programming paradigm and its corresponding container**. In that context, this better explains why the words **micro-level architectural design** is well adapted to denote the design of the architecture that provides the abstraction needed to implement the macro-level design.

Finally, this reinterpretation of MASs as architectures defined using dedicated component containers, and the fact that such component containers are themselves defined using SpeAD enables us to see a MAS as a component that can take part in a software architecture. Indeed, with SpeAD it is possible to use such containers/frameworks as components, and thus to build MASs that are externally considered as components. Such components can then be composed with traditional components in composites.

6.6 Horizontally Integrating CBSAs, MASs and Other Paradigms

As we showed here, it is possible to integrate the two approaches in an easy and well organised way using our abstractions. One can exploit the SpeAD component model to define traditional component-based architectures where some of the components are actually themselves MASs. But the opposite is also possible, as it is actually one of the motivations of the first part of this thesis, and the different elements composing a MAS can themselves be described using static and verifiable component-based architectures.

To conclude we now advocate for using design paradigms that are adapted to the needs, and in particular MASs approaches, as they are our primary interest, when dynamicity or adaptivity is required. When it is necessary to build static architectures whose reliability should be formally verified, component-oriented architectures described and implemented with traditional ADLs should be used. When it is necessary to build very dynamic, emergent, adaptive architectures whose reliability can be validated by experimentation, MASs should be used.

As we illustrated it, other kinds of dedicated component models and containers than those for MASs can be built using this approach. And thus, different component containers can live side to side and be composed using the abstractions provided by SpeAD. We characterise that as an **horizontal integration** between different components, including MASs, each possibly with their own dedicated component model.

However, it obviously looks like this vision of dedicated component-based component models and container is aimed at building component models different from what is traditionally done. For example, building something usable to define configurations of components as in ADLs would be using a sledge-hammer to crack a nut. Within such a context, it would

be even more difficult then to tackle adaptivity and complex behaviours of dynamicity in component-based architectures.

This is why we don't advocate for adapting architectures by extending existing and traditional ways of defining compositions of components. As we just said, one should use design paradigms adapted to their problems. But, the horizontal integration that we propose here keeps clear and hermetic walls between component-based architectures and MASs. The next chapter thus proposes to study how both approaches can be integrated vertically and having software components self-organising like agents in a MAS. In particular, it presents two experiments introducing dedicated component models and containers whose semantics is inspired by MASs approaches.

Summary of the Contributions

- ⊕ We position the contribution of Part I with respect to related works in the software architecture field.
- ⊕ We propose an integrated understanding of the different concepts used in the CBSA (Component-Based Software Architecture) field.
- ⊕ We propose a way to define and use dedicated component models and containers.
- ⊕ We show how the SpeAD component model can be used to build dedicated component-based component containers.
- ⊕ We show how MASs fit in this vision and how MASs as design approaches can be considered a family of design paradigms.
- ⊕ We show why the SpeAD component model well adapted for horizontal integration of MASs and CBSAs.

CHAPTER 7

From Self-Composing Components to Self-Designing Software Architectures

In the previous part of this thesis, we focused our interest on the development of MASs, which resulted, among other things, into the definition of a novel component model named SpeAD. We studied this model in the previous chapter, which directed our interest towards the realisation of dedicated component containers.

In the present chapter, we completely put aside the concern of micro-level MAS development and the SpeAD model. Instead, we now focus on the use of MASs as a mean to ultimately build self-designing software architectures.

We present here experiments of self-adaptive Component-Based Software Architectures (CBSAs) with various degrees of adaptivity. This brings us to conclude on the interest of generalising such approaches for software architectures, and not limited to CBSAs.

In opposition to the previous chapter, our concern here is clearly to vertically integrate CBSAs approaches with MASs ones. In other words, we want to use MASs as a support for connecting components to each other, but also to organise their composition in a way adapted to functional or non-functional objectives. To do so, we exploit the idea that agents can act as containers for components at runtime. The important point here is that our objective is not to replace components with agents, which would result on applying MAS approaches as they are normally applied, but rather to see how the principles and abstractions used in MASs can support the construction of CBSAs and maintain them in activity with respect to some requirements.

Such an idea has already been proposed in the past and declined in different ways. What mainly differentiate what we propose here from other works are the following points:

– We exploit agents as part of a MASs in the sense that the emerging collective intelligence is the engine of the self-adaptation of the system. In particular, as we are going to see, this has the advantages of having adaptive systems that can handle underspecified compositions.

- We integrate our solutions with the idea of building dedicated component models and containers:
  - In terms of design and implementation, it eases the understanding and the identification of clear points of variability of the solutions. This ensures that the solutions are actually applicable in practice.
  - In term of runtime architecture, agents really encapsulate the components and mediate their interactions. The process of adaptation is not only conceptual but is integrated with the executed system.
- We identify what features our solutions provide and how they relate to functional and non-functional requirements.

The chapter is organised as follow. First we present the different arguments that justify the use of the MAS technology to adapt software architectures and systems. Based on that, we present how agents and MASs can be used in this context. Then, we detail two experiments that each instantiates different aspects of this vision.

For a presentation of *AgentWise*, see p. xxiii

The first was developed in collaboration with the AgentWise research team. It proposes to adapt a composition of web-services using MASs organisational coordination mechanisms. It shows a simple degree of adaptivity where the focus is put on non-functional requirements.

The second was developed with a Research Master student during an internship we supervised (Denis 2011). In the context of ubiquitous computing, it proposes to let components existing in a distributed environment opportunistically self-compose to form applications that are not *a priori* specified. It shows a higher degree of adaptivity where the focus is not only put on non-functional requirements, but also on functional ones.

We conclude the chapter by extracting from these experiments an analysis of the links between self-adaptivity, emergence, functional and non-functional requirements. This brings us to generalise the content of this chapter to the design of software architectures in general and not only limited to CBSAs.

It must be noted that in this chapter, we consider compositions of, on one hand, components, and on the other hand, services. In this work, they are both examples of CBSAs in the sense that services and components participate in a composition of runtime elements. But as highlighted by Hock-koon and Oussalah (2011), component-based and service-oriented software **engineering** are two different aspects of the design of software architectures. Obviously, this difference has impacts on how elements of each approach can be composed together. More importantly, the motivations that drive such a composition are different in each case. Nevertheless, we think this does not contradict with what is presented in this chapter.

7.1 MASs for Self-Adaptive Software Architectures

7.1.1 Why MAS

As we previously highlighted, as practitioners of MAS approaches, we are particularly convinced by the validity and legitimacy of using approaches that provide collective intel-

ligence. The global idea behind what is presented in this chapter is that instead of relying on traditional and reductionist ways of composing elements to form a software architecture, we should look at more emergent ways of doing so. Such a concern has been expressed in the software engineering field for systems considered as coalitions (Sommerville et al. 2011). What characterises such systems is that the different elements that compose them were not produced together as a whole in order to be composed into one given application. We think this shows a kind of underspecification in the sense that it is difficult to know in advance, for example when building the components, what is going to be the problem answered by their composition and how it should be solved efficiently.

For a reminder on *Collective intelligence, reductionism and emergence*, see p. 10

Indeed, even though some functional or non-functional specifications are attached to the components, the services, or any other elements used in a composition, there is still underspecified requirements pertaining to the composition itself or to the way it should be executed. This underspecification is one of the main motivation for the current chapter and the experiments we detail in it.

But furthermore, simply the problem of having self-adaptive software architecture with respect to clearly identified requirements has also been highlighted in several works. We don't detail them here, but only selected two of those that identify points we consider important:

- The need for self-managed systems without external intervention (Van Roy 2007).
- The need for being able to distribute the "algorithm" that drives the adaptation of the system (Sykes, Magee, and Kramer 2011).

Because in MASs the control is decentralised by nature and because they showed their efficiency for self-adaptivity and self-organisation for solving underspecified problems (Di Marzo Serugendo, Gleizes, and Karageorgos 2011), all these aspects of the problem are particularly well supported by MASs.

7.1.2 MAS-based Containers

The main idea in this proposition is that since systems are made of compositions of components, there is thus a need for choosing the right composition depending on the functional and non-functional requirements[1]. Instead of doing this step completely by hand, we propose to use agents as a mean to assist in their composition, or at least in their adaptation once it has been done or specified. In this vision, agents acts as the container of the components, but the mapping between agents and components does not have to be one to one. For example, in the following experiments, the runtime components are encapsulated in agents, but there exist other agents in the system participating in the collective intelligence.

Components can then interact together without taking care of selecting or choosing other components. We thus relate this idea to the building of a component model and container whose semantics is realised by a MAS. Indeed, the fact that decentralised coordination mechanisms are introduced into the components of the system implies that the way to

1. Even though a decomposition normally answers some requirements itself, we consider here already defined components.

define such elements must change. It is not possible, with this way of doing, to continue to implement components as it is traditionally done. In exchange of the profit gained by the self-organisation, it is necessary to coin what are the implications in terms of the components definitions. For example, if components can find themselves in situation where they are connected to nothing, a way of expressing how it will behave must be available to the component developer. Of course, a lot of work has to be done to be able to attain a good ratio between adaptability and ease of definition. We don't cover such question here: we only present what we found in the experiments we use to illustrate the approach.

The main idea is to enable as much separation as possible between the adaptation and the business logic of the element. By business logic, we mean the real role of the component in the system, independently of its self-organising behaviour as an agent.

Thus, in order to describe our solution, we follow the approach proposed in the previous chapter, Section 6.3, to define and use component models and containers. The container takes care of the adaptation matter, potentially in a decentralised way, and enforces the component model that was defined. By expressing the proposition like that, we clearly identify the role of the system that realises the self-adaptation, with respect to the role of the components that takes part in such system. In another words, such an approach is useful to clearly identify the points of variability in the solutions proposed.

This abstract approach is to be instantiated for specific settings, MAS approaches and problems, and we present in this chapter two examples of such things. For each of them, an example of application that can use such container is shown. We identify the semantics of the model and the abstractions provided by the container.

7.2 The CASAS Experiment: Non-Functional Adaptation

The globalised world of business has created new demands for the architecture of distributed applications. These demands were shifted again with the creation of globally distributed supply chains (Pereira 2009). The service-oriented computing paradigm provides concepts satisfying the demands in this distributed environment (Papazoglou et al. 2007).

Service-oriented computing considers that services are provided by business organisations through the use of clearly defined interfaces. From a third-party point of view, the focus is put on using several services from different partners in order to answer its own needs. The most famous way of defining and accessing services are web-services. Web-services are defined using a specific interface format called WSDL (Web Service Description Language) that gives the possibility to express what are the operations proposed by a service. Then they are accessed through the internet, and more precisely using the abstractions provided by the web such as HTTP.

Nowadays, complex business processes are modelled as composite services using the *de facto* standard for service composition BPEL (Business Process Execution Language). Such compositions are also often called workflows as they describe a sequence of partner's services invocations to represent a complete business process. However, BPEL is not suited to work

in very dynamic environments, leading to research on how to adapt processes written in this language. BPEL can deal with primitive forms of adaptation, using `Dynamic Partner Links` and `Endpoint References` (Gudgin, Hadley, and Rogers 2006). However, it is hard, if not impossible, to model the adaptations and its constraints required by a composite service using only BPEL. We focus on the problem of adapting a composite service in order to deal with global constraints, such as the End-to-End Quality of Service (QoS), and the problem of preventing SLA (Service Level Agreement) violations. In a few words, a QoS can be specified with a SLA that describe the service provided by a partner and is composed of SLOs (Service Level Objectives) that specify what can be measured to characterise the SLA.

We add another level of abstraction to the composite service model to solve the adaptation problem. This layer, called the organisation layer, explicitly represents the interactions between all the services participating in the composition, the adaptation constraints and the expected behaviour of the composition.

To link with the objectives introduced in this chapter, we define a component model named CASAS (Composable, Adaptive, Service, Agent System) that can represent such a composition and its adaptation. We thus focus here on the self-adaptation of a predefined composition in the form of a workflow. This self-adaptation takes the form of the selection of the best partners at a given time with respect to some global QoS. The components here are traditional web-services provided by partners that are not under our control: thus in practice, the components are not the web-services themselves but bridges to the real web-services.

We first present a scenario for an example application that motivates the existence of CASAS. Then we present CASAS by following the proposed process for defining component models and containers.

7.2.1 Scenario

To illustrate the core ideas of this experiment, we use a simple scenario from the Supply Chain Management domain based on interviews with the industrial partners of the DiCoMAS project [2].

A fourth-party logistics company (4PL) takes care of its clients' logistic procedures such as the transportation of materials between the client's factories. A 4PL has contracts with many carriers, called third-party logistics companies (3PLs), that do the actual transportation. The 4PL's goal is to save time and money for its clients, by optimising transportation and business processes.

Each time our example 4PL receives a transportation request, it creates a transportation plan using an Advanced Planning System (APS). The transportation plan is composed of many activities, each representing a transportation that should be made between two locations. The planning system splits the original transportation request in several sub-transports, because, normally, 3PLs are specialised in specific geographical regions. Finally

2. DiCoMAS (Distributed Collaboration using Multi-Agent System Architectures) is a SBO (*Strategisch BasisOnderzoek* – Strategic Basic Research) project funded by IWT (*Instituut voor de Aanmoediging van Innovatie door Wetenschap – Technologie in Vlaanderen* - Belgium)

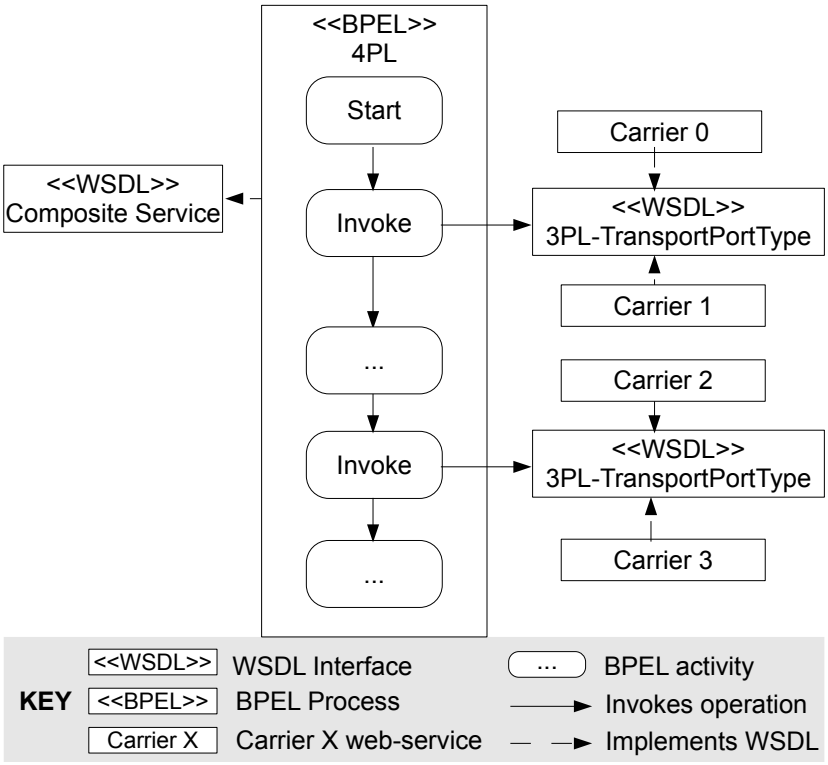


Figure 7.1: Transportation plan deployed in a BPEL engine

the transportation plan is written as a BPEL process and, after a first selection of 3PLs, deployed in a BPEL engine.

To execute this process, the 4PL's BPEL engine invokes the selected 3PL's web-services, informing them about the constraints, such as time constraints, monitoring constraints, etc.

How can we meet the quality requirements of executing a BPEL process within a specific time frame, even in the presence of partner failures, is the problem that we want to solve.

This problem is illustrated by our example 4PL, that needs to do the transportation within a limited time period, as specified in the plan, but sometimes a partner that previously committed to do a transportation has problems and is not able to execute its part in the plan. For example, a small carrier, that has just one truck, is assigned a sub-transportation, but, suddenly, has a broken truck that needs to be repaired. In this situation, the 4PL needs to find another carrier capable of doing the transportation for that specific sub-transportation, preferably within the same quality constraints.

Figure 7.1 depicts a simplified transportation plan. The transportation plan is deployed on a BPEL engine and is seen as an internal service within the 4PL, simply called `Composite Service` here. Each `Invoke` activity in the transportation plan is executed by an external service, through the `3pl-TransportPortType` interface, which is implemented by partner companies web-services. Each 3PL must comply to the specified QoS, in this case the time to do the transportation.

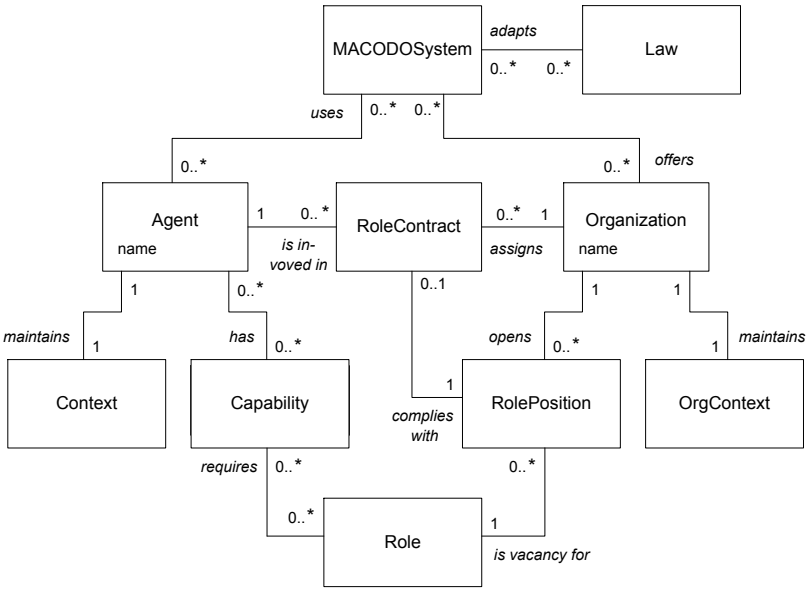


Figure 7.2: Domain Model of the Macodo Context-Driven Organisational Model (Weyns, Haesevoets, and Helleboogh 2010)

We now present the organisational concepts used in our solution and the mapping between these concepts and those of the problem. It then serves as a basis to present the CASAS component model, its semantics and the constraints it introduces on the components.

7.2.2 The Macodo Organisation Model

The MASs research community has a body of knowledge on Organisational Models. In this community, there are two distinct visions regarding these models:

– Organisation being a first class entity, with its properties, states, laws (Weyns, Haesevoets, and Helleboogh 2010)

– Organisation mainly as a process, composed by a set of steps to be taken by different actors (Dignum 2009b)

MASs Organisational Models cope with collaboration between autonomous entities, called agents, working and interacting together, cooperatively or not, to achieve an organisation goal (DeLoach 2009). In our work we rely on the Macodo (Middleware Architecture for COntext-driven Dynamic agent Organizations) Organisation Model, which defines the organisation as a first class entity with its own dynamics and separated from the participating agents.

The Macodo Organisation Model copes with context-driven dynamic organisations. With it we can model complex collaborations between different entities, the agents, and specify the rules that will trigger actions to adapt these collaborations (Weyns, Haesevoets, and Helleboogh 2010).

Figure 7.2 depicts the domain model of the Macodo Organisation Model. The main concepts of the model are: `Organisation`, which contains roles and role positions; `OrgContext` which keeps the context information needed by the organisation; `Role`, which constrains the behaviour expected from the agents; and `Context`, which is the contextual information needed by the organisation.

When joining an `Organisation`, an `Agent` takes a `RolePosition` in order to play a `Role`: this is represented by a `RoleContract`.

An `Organisation` encapsulates organisation rules used to adapt the collaboration. It actively inspects its context (`OrgContext`), *i.e.* the set of all the participating agent contexts, to enforce the organisation rules that need to deal with global constraints. A `Role` constrains the agent behaviour towards the organisation, requiring the agent to provide a set of capabilities and context. Using the `Roles` and agents' `Context`, the `Organisation` allows or denies the participation of the `Agents` in the collaboration.

There are two possible ways for an organisation to adapt and a well balanced combination of the two is the key to an adaptive organisation:

1. The organisation can access the context information of the agents and check if the rules are being satisfied. That way, the adaptation is triggered by the organisation that keeps monitoring the collaboration between the agents. If one rule is not satisfied, it can change the state of a role or open a new role position, so that another agent can try to play that role in the organisation.
2. The agents can have a pro-active behaviour and monitor their own state. They can actively decide to leave or join an organisation. When an agent leaves an organisation, the organisation changes the state of the played role, opening a new role position, leading to its adaptation.

7.2.3 Mapping Macodo Organisations to Composite Services in BPEL

A BPEL process consists of:

– The BPEL code, which defines the execution flow.

– WSDL interfaces for the different consumed services.

– WSDL interface for the provided service (the composite service itself).

A composite service specified in BPEL is made of a set of activities that are executed in a specific order. One special type of activity is declared using the `Invoke` construct, which invokes an operation in a partner link web-service. In BPEL the communication with other web-services is done through the `PartnerLinks`, which define the relation between the BPEL process and partner web-services. Partner web-services are referenced by their `Port Type`, which is a set of abstract operations defined in WSDL.

We established a mapping between Web-services and Macodo Organisation concepts in order to create the organisation layer. This mapping is illustrated in Table 7.1.

One BPEL process corresponds to one analogous `Macodo Organisation`. For each `Partner Link` specified in the BPEL process, we have a `Role Position` in the organisation. We have a

Table 7.1: Mapping between web-services and Macodo concepts

Web-services	**Macodo**
BPEL process	Organisation
PortType	Role
PartnerLink	Role Position
SLA	Capability
SLO	Agent Context

`Role` for each `Port Type` in the BPEL process. The SLAs are specified in terms of required `Capabilities` and, finally, SLOs are specified as `Agent Context`.

As we are going to see, these informations are then used to describe under which constraints the composition should adapt.

7.2.4 Defining the CASAS Component Model

The CASAS component model uses this mapping and the BPEL process description to define the structure, such as the number of `Roles` and `Role Positions` of an organisation. Each BPEL process instance provides the correct flow of activities needed by the composite service and can be seen as the organisation functional behaviour. The agents operate the runtime binding to the real service providers and act as their representatives in the system.

Based on this mapping, separately from the BPEL process definition, the organisation provides a way to define adaptation rules that deal with the runtime adaptation that can occur during the process execution. For that, the agents provide context information to the organisation. They also contain monitoring mechanisms and have pro-active behaviour to trigger adaptations (*e.g.* an agent can monitor its SLOs and predict that an SLA will be broken).

Figure 7.3 shows all the entities that collaborate in the system to create an adaptive composite service. It shows a BPEL process, the adaptation rules, the `Macodo Organisation`, and the `Agents` that can participate in the organisation. The leftmost `Agent` is taking the `position2 RolePosition`, the other `RolePosition`, called `position1`, is not taken by any `Agent`. An `Agent` taking a `RolePosition` means that the agent is playing a specific `Role` in one `Macodo Organisation`. When the BPEL engine invokes an operation in a `PartnerLink`, the CASAS system intercepts and redirects that invocation to the right `Agent`, which will, in turn invoke the operation on the actual web-service.

From a service-oriented point of view, the CASAS system is responsible for dynamically providing the best partner services to the workflow instances. This selection is done in compliance to the SLA of each partner web-service. The BPEL engine doesn't know about the changes that can happen to the partner web-services, since it communicates with endpoints provided by the CASAS system. The CASAS system makes the connection between the BPEL engine and the real partner web-services, acting as a type of evolved proxy.

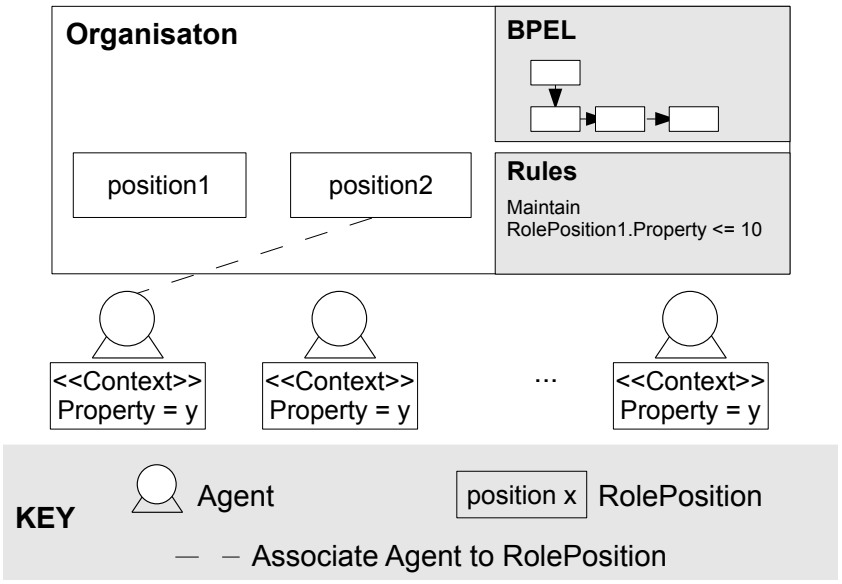


Figure 7.3: Conceptual solution integrating Macodo organisations, BPEL, and agents

When the service realised by a workflow is requested, CASAS instantiates it with an organisation and the agents that represent the web-service that can participate in the workflow. Thus from an agent-oriented point of view, the MAS is composed of the agents and their environment, *i.e.* the organisation that regulates their interactions. The MAS, explained in the next section, is transparent to the BPEL process and to the partner web-services, being used just internally by the CASAS system.

7.2.5 Multi-Agent System

The agents are responsible for joining the organisation if they can take open role positions. They play their role when asked by the organisation and maintain a context that can be consulted by the organisation when required. There is one agent for each existing web-service that can be used in the workflow. Describing the MAS is decomposed in describing the organisation behaviour and the agents behaviours.

Behaviour of the Organisation. The organisation uses the agent's context and its own context to continuously enforce the adaptation rules. It accepts, refuses and revises `RoleContracts` based on this context information. If one agent does not comply with the organisation rules anymore, the organisation closes its role position and opens a new one.

It also gives agents the possibility to join and leave positions, as well as play their role and thus to participate in the workflow.

Behaviours of the Agents. There are two parts in the agent behaviour:

– Social behaviour for joining and leaving the organisation

– Monitoring behaviour for managing the context

An agent that receives an event for an open position will try to start a contract if he has the capability of playing this role, *i.e.* he has a matching `PortType` and the required SLA. If the agent decides to take a role position already taken, if the organisation refuses to give the position to him, or if the position is closed while playing it, then the agent waits for new open positions.

The monitoring behaviour is strongly dependent to the type of context required by the organisation.

7.2.6 Defining the CASAS Component Container

Thus, from the previous sections, we can extract the definition of the different types of hotspots of our component container. There exists two types of components in this component model: organisation and agent. There is no container hotspots.

Organisation Hotspot. For a given organisation, *i.e.* a given workflow, it is needed to give:

– The provided interface definition in WSDL.
– The functional behaviour for the organisation: the workflow definition in BPEL.
– The adaptation rules that have to be enforced. For example they can take the form of functions per role position responsible for checking that the QoS indicators of the agents (its context) satisfies the defined SLA for the partner (a threshold). Or could be more complex as accepted by the Macodo system.

Agent Hotspot. This hotspot is very simple as most of the adaptation is done at the organisation level. Moreover, the agents mostly play the roles of bridge towards the real services.

What should be provided for a service is:

– The provided interface definition in WSDL.
– A reference to the web-service they represent.
– An implementation for extracting the context information.

7.2.7 Implementation of the CASAS Component Container

We don't detail the implementation of the CASAS component container, but details can be found in Cruz Torres et al. (2010a).

The prototype was written in Java, according to the described architecture. To handle the service-oriented concerns, CASAS rests on the Apache ServiceMix ESB (Enterprise Service Bus). In particular, this ESB provides the Apache Ode BPEL engine to execute the workflows and the Apache CXF component to access external web services. The ESB provides mechanisms that can be used by CASAS to intercept messages exchanged in the ESB and redirect them to the correct recipients.

Thus, the system is usable directly with real web-services and workflows.

7.2.8 Using the CASAS Component Container

If we instantiate the proposed container with the scenario presented earlier, this gives us the following. The delivery time of the 3PLs are the context of our agents. The threshold that shouldn't be exceeded is determined by the APS for each role position.

The BPEL workflow and WSDL interfaces presented Figure 7.1 are provided with the organisation hotspot. The adaptation rules are expressed per role position as not exceeding the threshold.

Dynamically, for each web-service that can play in the workflow, the creation of an agent is parametrised by its interface and an implementation for computing the context information using the operations of the interfaces.

Thus, when web-services goes online or offline, their corresponding agents comes in or out of the system. They try to join the organisation and depending on their context information they are accepted or not.

7.2.9 Discussion and Possible Evolutions

As we can see, the current state of the proposition is very primitive and doesn't show so much intelligence. But, as the auditors of the web-service community workshop, where this work was presented (Cruz Torres et al. 2010b), told us, what is interesting in that solution is the abstractions it provides. The community is in need for such high-level abstractions in the context of business processes where workflows actually represent collaborations between several business partners. This experiment shows that it is possible to do so.

We see several evolutions that can be applied to this work. First, related to the Macodo model, the more it is evolved, and the more advanced the adaptation our system provides can be. For example, it would be interesting to specify global constraints for the whole organisation, and not only per role position. Furthermore, the Macodo model is capable of creating role positions, expressing relations between them and how the the organisation can evolve: in that context, Macodo could be used to model the reorganisation of the sequence of activities by moving away from the static aspects of workflows as they are currently considered. Second, in terms of the mapping between organisational concepts and workflow concepts, we think it could be interesting to have agents not only act as bridges to the web-services, but also as bridge from the web-services participating in the organisation. Finally, interactions directly between agents could also be interesting. The next experiment illustrate particularly well this last point in a different context.

It must also be noted that this work was continued with a focus on decentralised coordination mechanisms by the AgentWise task force (Cruz Torres and Holvoet 2011a; Cruz Torres and Holvoet 2011b).

7.3 The Greg Experiment: Opportunistic Composition

Ambient systems are complex systems composed of interacting (potentially mobile) artificial and humans entities. They are distributed, open, decentralised, heterogeneous, dynamic, etc. Ambient intelligence reflects the capacity of the ambient environment to provide relevant and adequate services to the humans living within. As it was imagined it 20 years ago (Weiser 1991), we are now faced with a new computing environment that asks for new techniques to build, deploy and maintain applications in production.

In this context, mainly because of mobility and openness, we can imagine that systems could opportunistically self-construct only because communicating devices are present simultaneously in the same space. New applications (or software systems) can emerge simply because the entities that compose them are in interaction at a given time. For example, a multimedia application such as "watching and controlling a movie" can result from the composition between a remote control, a media box, a display device and an audio device. The sole presence of these hardware devices, at a given time and in a given space, should allow a system to self-construct without any human intervention. We can thus imagine an application that autonomously self-composes, directly from its elements initiative, without having the composition expressed by a designer or a user beforehand, or even without having the different components made on purpose for such a composition.

The objective of this work is to explore such question of autonomous self-composition and to evaluate its feasibility.

As we imagine it, self-composition (at the initiative of the components themselves) raises original questions compared with more traditional design and programming approaches and their context:

- There is no client that express a need nor any design that tries to answer them. There is thus no validation that can be done in its traditional meaning (showing that a product answers the initial needs). There is neither any specification of the product, thus no possible verification of the conformity of the product with respect with these requirements.
- There is no designer that creates the compositions and that controls them.
- There exist human users that themselves are part of the system and that can play a role in the acceptance of the applications that dynamically appear.

The question that interests us here is relative to the opportunistic composition of applications that can be characterised as emergent in an ambient context. But once created, such application must also be dynamically adapted, controlled and maintained in a functional state. In this context, we look at how this can be automated and self-adapted.

As with the previous experiment, we first present scenario that motivates it, then the concepts manipulated, then we propose a solution that we describe using the process we proposed for building and using component models and containers.

7.3.1 Motivating Scenario

This scenario takes place in the context of multimedia ambient application. We imagined the following.

(1) David is in his living room and his objective is to watch a HD video on his TV by choosing one of the movies available on the disk of its internet media box. His TV is on standby and his hi-fi system is off. He starts the interface to access multimedia content from his smartphone. The system finds two mediacenter (one from his internet box and one included in its TV) and proposes to David to make a choice. David choose the mediacenter of its internet box. A browser appears on its screen. He selects the video he wants to watch and validates his choice. The video appears on the TV and the sound comes out from the TV integrated speakers. Since he wants to profit from the better quality of his hi-fi system that provides 5.1 surround sound, he turns it on and the sound switches to the speakers of the hi-fi system.

(2) Later in the evening, David turns off the TV (the movie is automatically paused) and moves to his room because he wants to finish the movie in his bed. He turns on his computer and use his smartphone to start the movie again. The sound comes out from the computer speakers. He turns on his headphones and puts them on. The sound switches to them.

(3) It is 10pm and someone tries to call him. Because David has his headphones on, and he is watching his screen, the phone does not ring but a feedback is sent to the tv. Because of that, the movie is put on pause and a visual feedback is shown on screen.

The first part (1) of this scenario should validate that our solution is able to create applications, that it can resolve problems such as the choice between several boxes, and that it can propose always the best composition with respect to the user needs (5.1 sound better than tv speakers).

The second part (2) should validate the dynamic of the composition (moving between places, new devices appearing) after it has been created.

The last part (3) should validate that the system can handle conflicts between resources (feedback of the phone versus movie already playing on it).

It is important to notice that the applications that are built during this scenario are not *a priori* specified. The user only expresses his needs through the use of devices and the components self-compose until they form a working composition.

Figure 7.4 shows an example of a composition that should be self-built by the system according to this scenario. There is several components composed together in a given configuration and thus forming an application. We now present the domain model and the way such components are structured.

7.3.2 Modelling the Problem Domain

In an ambient environment, we can find **devices** such as a TV set, a media box, etc. **Resources** are physical entities directly linked to a device, for example the screen of a

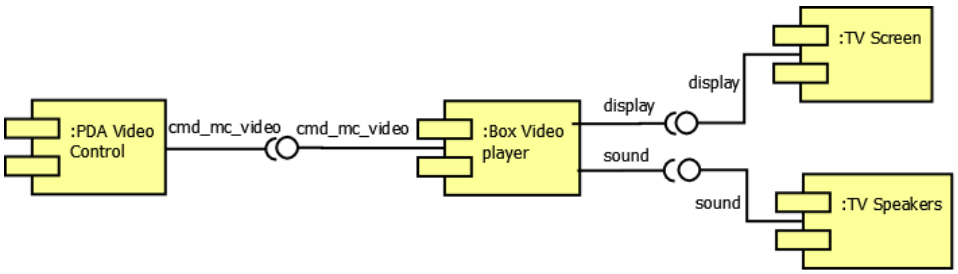


Figure 7.4: Composition example

smartphone, the CPU of a media box, etc. These resources have several **non-functional properties**, such as the audio quality of a speaker, or the video qualities a screen can display, etc.

Devices can embed **software components**, such as the applications of a smartphone, the video player of a media box, etc. Components uses resources to function. The kind of components that interest us are providing services through their **provided ports**. Ports have a name and an interface. For implementing their provided port, they require services through their **required ports**, potentially with specific **non-functional constraints**. A component is able to provide a service only if all its required ports are "resolved", *i.e.* all its required ports are connected to another component that provides it. A **component instance** is to a component what an object is to a class: during the life of the ambient system, it is possible for a same component in a device to have several instances. Such a component instance is the elementary building block of our self-composed applications. A resource can be shareable (several component instances can use it at the same time) infinitely or up to a certain point (for example a CPU). An **application** is the result of the composition, in a distributed ambient environment, of several component instances whose required ports are resolved.

One of the objective of our system is to define a est of autonomous agents able to build and connect dynamically the instances of the components. However, its role doesn't end with that objective: it continues during the execution of the application, manages and reconfigures dynamically the compositions that it maintains depending on diverse **events**. These events come from the environment (new devices, change of location of a device, etc), but also from the middleware that connects in practice the instances together to let them communicate (interrupted connections fro example).

Finally, we simplify the problem by considering that:

– Each components provides only one port.

– Interoperability is not taken into account: the semantic matching of interface is reduced to the simplest string name matching.

7.3.3 Defining the Greg Component Model

Our solution relies on the use of the AMAS (Adaptive Multi-Agent System) approach to assemble in a dynamic and autonomous way the components of the ambient environment. On

For a definition of *AMAS*, see p. 11

top of the fact that such systems seems *a priori* adequate to tackle such question, as presented in the introduction of the chapter, we think that in a general way, AMAS are particularly adapted to the development and the design of application whose specification is not or not completely known.

As we presented Chapter 1, in the AMAS approach, the engine of adaptation is the cooperative behaviours of the agents. This behaviour is elicited by identifying NCSs (Non-Cooperative Situations) and by exploiting the concept of criticality.

Cooperative agents have a nominal and a cooperative behaviour. In our solution, the nominal behaviour of the agents is the business concerns of the components they encapsulate. For example, the nominal behaviour of an agent containing a media player component is to actually play the video. In order to apply their nominal behaviour, the agents must be in a state that makes it possible. Obviously if a media player has nowhere to send video, it cannot operate properly. This is where the cooperative behaviour is used. The cooperative behaviour, expressed in the form of atomic and individual NCS detections and solving, takes care of putting the system in a usable state. There exist different types of NCSs but we don't detail them here.

Of course, depending on the business concern of the encapsulated component, the way to solve a NCS can vary. In the following, we present the cooperative behaviours of the agents, and when needed we explain how it can vary depending on the business concern of the agents. This later serves as a basis to define the component container and its hotspots.

It must be noted that this solution is perfectible and its main objective is to show the feasibility of using such a bottom-up approach to self-composition.

7.3.3.1 Types of Agents

In our system, we have two types of agents: **class agents** and **instance agents**. The idea is that in each device, there is one class agent responsible for each software component that can be instantiated. When needed, by interacting with other agents as we are going to see, such an agent can create instance agent of its components.

These instance agents are responsible of playing their role in a given application, by resolving conflicts and dependencies with other instances in the system. Instances can be completely, partially or not resolved: it means their required ports are, respectively, all, not all or not connected to another instance.

7.3.3.2 Neighbourhood

Devices are on a network, and thus can "see" each other using traditional broadcast means used on local networks, such as the IP protocol. When a device is started in a given environment, then for all the software components in it, their class agent is created and thus are in the physical neighbourhood of other agents in their range. Inversely, if the device is stopped or stops working for any reason, then the class and instance agents disappear and thus disappear from the physical neighbourhood of other agents. If the device goes out of range of other devices, the same happens.

A class agent is interested in a subset of its physical neighbourhood: the requires neighbourhood contains class agents that have interfaces it requires. These neighbourhood are maintained internally by each class agent, while the physical neighbourhood is induced by the physical network. A class agent also have its **instances neighbourhood** composed of all the instance agents it created.

An instance agent has a **potential neighbourhood** of instance agents that it can potentially connect to. It also knows its class agent.

7.3.3.3 Agents Behaviours

We now detail the different informations managed by the agents, the events that can happen and what are the agents actions. These "behavioural rules" are induced from the Non-Cooperative Situations that were identified but not detailed here.

In our solution, the questions of constructing the application and adapting it are considered the same problem. For example, either at construction or after a composition has been fully constructed, an instance that finds itself with one of its port not resolved anymore will handle the problem in the same way.

Instance Behaviours. An instance agent is responsible of a component instance. Its main objective is to find the "best" instance agents to resolve its required services.

Every time an instance is requested to be potentially used by another instance, it is able to compute its **capability level** with respect to the request. Such a capability level is computed from the non-functional properties of the resources used by the instance agent, the non-functional constraints of the request and from the best capability level of its resolved required ports. Indeed, for each of its required ports, an instance agent maintains its potential neighbourhood (using various ways that we describe after) and store the capability level of each of the instance it can use. For example, a video player trying to play a 5.1 video could have a capability of 80% if its required port for streaming sound is not resolved to an instance that can handle 5.1. And thus, the "best" instance for a given required port is the instance with the best capability.

On the other hand, an instance agent also have a **criticality level**. This level represents how critical is an agent in resolving its required ports. It is used to solve conflicts over resources and to drive the reorganisation by having agents leaving their current role in order to answer more critical requests. The main difficulty here is to find the best way to compute such a criticality. First, it is partially inherited from other instances that requested its service. But this information also varies with the business concerns of the components. For example, a request emanating from an instance agent that was created after a user launched an application can have a criticality that increases until it is not answered, while a request from a component created automatically by the system wouldn't change.

There is 10 behavioural rules that are roughly decomposed in three different aspects: maintaining the potential neighbourhood, handling connection to other instances and resolving conflicts over resources.

Maintaining the potential neighbourhood:

1. After it is started, it regularly sends requests for its required ports along with the interface type and the non-functional constraints.
2. When it receives a request for potentially using its provided port, it answers positively with a capability level computed functions of the request.
3. When it receives an answer for one of its required ports or when there is a capability update, it updates its potential neighbourhood and their capabilities.
4. When its capability changes, it propagates it to the instance using it, which in turn is likely to do so, which can trigger reorganisation somewhere in the system.

Managing connections to other instances:

5. When another instance request to connect to it:
   - (1) If it is in a completely resolved state:
     - (2) If there is no conflict with another instance on its needed resources, then it accepts.
     - Else, for each resources it needs, if there is a conflict with other instances:
       - It asks them to leave the resources with its criticality as information.
       - When a positive answer comes back, go to (2).
   - Else, for each of the unresolved ports:
     - It notifies the "best" instance, along with its criticality level, that it wants to connect to it.
     - When a positive answer come back, go to (1).
6. When there is a new "best" instance agent to connect to and it is already connected:
   - The agent notifies the new one, along with its criticality level, that it wants to connect to it.
   - When a positive answer comes back, it notifies its previous "best" that it disconnects it.
7. When its criticality changes, it propagates it to the agents it is connected to, which in turn are likely to do so, which can trigger reorganisation somewhere in the system.
8. When it receives a disconnection notification for a required ports (from an agent or from the middleware), it notifies the instance using it that it is disconnecting its provided port, and it updates its criticality (and thus propagates it).
9. When it receives a disconnection notification for its provided port (from an agent or from the middleware), it updates its criticality (and thus propagates it).

Resolving conflicts over resources:

10. When it receives a request to leave a resource and if its criticality is lower than the requester:

- It leaves the resource.
- It notifies the requester.
- It updates its capability (and thus propagates it).
- It notifies the instance using it that it is disconnecting its provided port.

At different times, the components can have a behaviour that varies depending on their business concerns. As previously highlighted, to compute their capabilities or to compute their criticality. But also, it is needed to specify the components behaviours when they find themselves in a non working state. Indeed, when a component is executed and the implementation tries to use the required ports after being called on their provided port, everything is connected. But if there is a failure at one moment, the component must define how it should be handled in its particular case.

For example, with required ports that get disconnected, a video player would just go into pause, while a remote control would show an error message. Actually, with the way things are current handled, in other all the cases, any instance we could think of would terminate. Then when a new composition would be formed, new instances would take the relay.

A more complex and mature solution would most certainly introduces more interesting points of variability. For example the computation of the capability could be business-specific and would help drive the composition towards something that better answer the users needs.

Class Behaviours. Class agents are much more simple.

After it is started, it regularly broadcasts the existence of its provided port.

When it receives a broadcast about the existence of a provided port with the same interface as one of its required ports, it stores it in its requires neighbourhood.

When it receives a request for potentially using its provided port, it creates an instance and forwards the request to it.

Depending on the business concern of the components, the reason for creating instances are different. For example an instance of a component for a user interface is created when the user launch the application.

7.3.4 Defining the Greg Component Container

Thus, from the previous sections, we can extract the definition of the different types of hotspots for our component model.

Since all the dynamics of the model is managed from inside the agents, and thus the components, the container would only have some configurations about the resources of a given device.

For each of the component, it is needed to provide:

- An implementation for its provided port, that can use the required ports.
- An implementation for computing the criticality (or a part of it) of the component.
- An implementation for the different hooks corresponding to specific actions to execute when special events happen:

- When getting disconnected for provided port.
- When no instance is found to resolve the required ports (and thus when they are disconnected).

7.3.5 Implementation of the Greg Component Container

We don't detail the implementation of the container but information about it can be found in Denis (2011).

The prototype is actually realised using SpeAD and SpEArAF. A home made simulator was built for the networks, the devices and the resources.

7.3.6 Use of the Greg Component Container

The only thing that must be predetermined beforehand and that should be shared by every implementation of components conform to the container is the meaning and normalisation of the criticality. This is one prerequisite of the AMAS approach. There could be a standard that define its meaning, for example that defines which are the default values for a user requested action, for an automatically created component, for an urgency, or things like that.

In the scenario, there are two types of ports: streams such as video, audio, etc, and call-return. The latter are the ports prefixed by `cmd` Figure 7.4.

We now show how a component for a video player is implemented using the component container.

Its provided port can accept commands to control the execution of the video, *i.e.* play, pause, stop, etc. It has two required ports: one to stream sound and one to stream video. Its implementation is a process that streams video and sound when the play signal is sent to its provided port, stops streaming when the pause signal is received and terminates when the stop signal is received.

The criticality function is the default one for automatically created components. The implementation of what to do when the provided port is disconnected is to terminate. The implementation of what to do when required ports are not resolved, is to put the video on pause.

It is a cooperative behaviour towards the users but also towards other components of the system in the sense that:

- If it is not controlled by anything, then it does not have any reason to exist.
- If it is still controlled but can't stream, then it should not play the video, which would be against user's desire.

7.3.7 Discussion and Possible Evolutions

What this system proposes is enough to demonstrate that it is possible to have components that self-compose in a physical and conceptual distributed context. On the other hand, we have a lot of ideas on how to improve it.

First, concerning the ways to express users needs in terms of components behaviours, it seems relevant to consider that a component providing a port is as much as important than a component requiring a port. For example, a user putting headphones on means a new provided port is announced in the system, and thus, the user need is that this provided port should be connected.

Then, even though components should be encapsulated by agents, it would much more interesting to have agents encapsulating each of their ports to introduce even more local and emergent behaviour. For example, this would be useful for expressing dependencies between ports that should be connected together to a same device.

7.4 Discussion: MASs for CBSAs

We presented two MASs-based component containers. In order to conclude this chapter we comment them and discuss their differences. This gives us the opportunity to draw some conclusions on the links between functional and non-functional requirements, self-adaptation, emergence and the aforementioned underspecification. Furthermore, we also conclude on the advantages of explicitly defining the proposed solutions as component models and containers.

7.4.1 Adaptation and Emergence

In the CASAS experiment, the functionality of the composition is completely specified beforehand using a workflow description. The system provides the "intelligence" that chooses the right partners. Indeed, the organisation, by the definition of adaptation rules, enforces the global constraints of the composition and drives the reorganisation. But furthermore, as we explained, the agents can, by themselves, decide to leave the organisation, when individual constraints are not respected. Thus, in this system, what drives the reorganisation and thus the self-adaptation of the system is clearly the non-functional requirements expressed by the Service Level Agreements and Service Level Objectives.

In the Greg experiment, the functionality of the composition is not specified at all, but this doesn't mean the functionality emerges from nothing. Indeed, because the elements of the composition are functional components, the relation between the self-built application functionality and the functionality of the elements is purely reductionist. However, the composition process itself can be said to be emergent: indeed, what emerges from the interaction of the agents is the selection of the components, and in which configuration. And then some conflicts at a point of the composition is propagated to another place and can trigger reorganisation. Thus, the functional requirements drive the self-composition of the components and the non-functional requirements guide it towards the most adapted one. The ways this can be controlled is through the notion of cooperation, expressed by the criticality, that implies some specific choices by the agents at some points, and the capabilities that represent the level of satisfaction of the functional requirement. Thus, to conclude, in this system, what drives the reorganisation is:

- The functional requirements expressed by the required ports of the components and their interfaces.
- The non-functional requirements expressed by the capabilities of the components and their non-functional constraints.
- The user needs that are some kind of functional requirements not formally specified, expressed in a cooperative way by the notion of criticality.

The two first points are what, in our opinion, enables the self-composition of components in a completely decentralised way. That last point is what, in our opinion, introduces an adaptation to the underspecification of the system highlighted in the beginning of this chapter.

In the end, the two systems are not so far from each other in their objective because they both optimise the selection of the components to compose. To better understand the links between the two, we can use the differentiation coined by Bouziane, Pérez, and Priol (2008) between workflows as defining a temporal relation between the elements of the composition and component-based compositions as defining a spatial relation between the elements of the composition. Indeed, in the first system, the functionality is described using a temporal relation between the elements, while in the second system, their composition is spatial. Obviously, the first one seems to be much more complex to handle in terms of self-composition and reorganisation of the sequence because of the complexity of the semantics of such composition. But what this also highlights is that actually in the second case, this temporality is hidden in the component and expressed through the way a component is implemented and uses its required ports. This just opens the discussion on the matter, and we think this should be better explored to better understand how we can better adapt these kinds of software architectures at runtime. Moreover, as highlighted in the introduction of the chapter, this discussion should most certainly also take into account the differences between component-based and service-based approaches to composition.

7.4.2 Component Models and Containers

Concerning the definition of the component models and containers, these examples show that when using a semantics for the component models that is very different to what is traditionally done in the software architecture field, there are impacts on the information needed to be provided for the implementation of the components. It supports the argument that if new methods of composition are used, then it would be vain to keep using always the same traditional abstractions to define components. In particular this is visible with the second solution where the concept of cooperation leaks out at the business level. We think this point is strongly linked to the question of underspecification as it is one of the main reason we novel different semantics for the component models.

It also shows that by expressing these details about the container hotspots, it clarifies for a given component model the points of variability and eases its documentation.

But moreover, by using the component container abstraction to realise such system, we built systems that integrate together the decision and its actual execution. Indeed, as we said,

what emerges from these systems is the choices of the components to use, and the choices of how to connect them. The system applies these choices in order for the system to be executed in direct. This is interesting because such an approach makes the bridge between, on one hand, the concepts and the reasons behind the existence of a particular organisation, and, on the other hand, the elements of this organisation and their actual composition.

7.4.3 Better Adaptation for CBSAs

Finally, we think, as it is highlighted by the methodological precepts of Agent-Based Modelling as instantiated by the AMAS approach, that a good definition of the problem domain is what is important for having a good self-adaptive system that answers its requirements in an adequate way. There is thus most certainly more works to do to spot the right abstractions for representing components and their dependencies. We think in particular about the work of Desnos et al. (2007) that raises and answers some of these questions by proposing the same kind of compositions than the second experiment, except for the underspecification of the user needs, but in a totally centralised way.

7.5 Towards Human-Assisted Self-Designing Software Architectures

To illustrate how MASs could be used to support the self-adaptation of systems, we presented the idea of building MAS-based containers. This approach is principally based on a runtime adaptation and only for components. On the other hand, in the previous chapter, Section 6.2, we presented how the concepts used in architectures are related to each others. In particular, it highlighted that a given component model allowed a number of architectural styles instantiated by architectures and patterns.

In this section, we propose to first give an understanding of the relations between the state of a CBSA at runtime and the architectural styles and patterns that it instantiates. Then, we generalise that idea to software architectures in general as a mean to support the design of software systems.

7.5.1 Styles and Patterns that Emerge

If the elements of an architecture can self-organise at runtime, then at a given time, their organisation is in a particular configuration. Such a configuration is possibly an instantiation of an architectural style or pattern. For example, a set of components could all be using the same other one: this would instantiate the client-server style.

And thus, the runtime architecture has several qualities that are induced by this configuration, or in another words, the runtime architecture answers some non-functional requirements. Inversely, we could imagine that a wanted quality would drive the self-organisation so that an adapted configuration of components is reached by the self-adaptive architecture.

The main problem with such an idea is that most of the architectural styles that exists constrains so much the way components are implemented that it becomes difficult, for an already defined component, to be used in different styles. There must still exist some examples of styles and components for which this approach would work, but this is a question we won't answer here.

More interesting in our opinion, if we generalise this vision to software architectures building, we think that creating self-designing architectures is more realistically feasible.

7.5.2 Generalisation to Architectural Views

Indeed, nothing prevents this approach to be applied to all the aspects of software architectures such as the design, the decision making, the application of design choices to actually build the architecture, the matching between the views, the solving of conflicts, etc. The idea would be to have a tool that helps elicit the different views of a software architecture by answering feedbacks from the designer as well as from a repository of knowledge either domain-specific or generic.

For a definition of *Architectural Views*, see p. 7

By using the designer as a way to give feedback to the tool, it becomes much more easy to automatise the adaptation of all the structures of the system in a coherent way. Indeed, the tool would then provide the intelligence needed to manage the application of the modifications or to keep the architecture coherent, while the designer would act as the driver of the self-organisation.

From the point of view of the elements of the architectures, *i.e.* modules or components, they would act as the components of the previously presented experiments. They would try to form coherent structures that answer the expressed functional and non-functional requirements. But they would do so from different views at the same time. One of them could for example be a decomposition view, and the modules could try to find the better decomposition with the help of the designer feedbacks. In parallel, we could have layered view that would enforce the constraints expressed by the designer, or maybe found by the system itself. Then, we could have a mapping view between modules and components, components views, etc. This could even go to the point where one of the views is the actual running system made of the real instantiated components, encapsulated by agents as in our previous examples.

Such a system could be used to assist in the decision making for designing the architecture, but also in the actual application of the changes to the running system. Knowledge about known patterns and styles could also be made available to such a system to better support the design.

Approaches for assisting evolution has already been proposed in different works such as Le Goaer et al. (2008) or Garlan et al. (2009). What differentiate our proposition from these are, on one hand, the links that exist between the runtime and design-time self-organisation, and, on the other hand, the fact that the system would be totally integrated to tackle together all the aspects of the software architecture. Such a tool would then gracefully make the bridge between design, deployment and maintenance.

Summary of the Contributions

- ⊕ We highlight the reasons that motivate the use of technologies such as MASs for the adaptation of software architectures.
- ⊕ We propose MAS-based component containers to have self-adaptive self-composing Component-Based Software Architecture.
- ⊕ We illustrate that vision with an experiment that adapt a composition of services.
- ⊕ We illustrate that vision with an experiment that let components opportunistically self-compose in a completely decentralised and distributed way.
- ⊕ We present an understanding of the relations between self-adaptation, emergence, functional and non-functional requirements, and highlight the role of the underspecification in a the adaptation of a composition.
- ⊕ We present challenges that these experiments raise.
- ⊕ We present the idea of a support tool for self-designing architecture that tackles in an integrated way the design of the software architecture up to its its deployment, maintenance and evolution.

BACK

CHAPTER 8

Conclusions and Perspectives

The initial objective of this Ph.D. was to contribute to the field of MAS development in order to support the transition between design and implementation. In order to reach such results, we uncovered several elements of contributions in the field of MASs and software architectures.

We propose to present these contributions from four different points of views that each pertain to a specific field or combination of field. The balance in size between the first part and the second part is not reflected by these four points of view. Indeed, the first part, which is the biggest of the two, answers the two main motivations of this thesis, that is improving the development process followed in MASs as well as its products, the MASs themselves. Then, on the other hand, the second part answers the third motivation, that is understanding the relations that exists between MASs and software architectures with a focus on CBSAs. This is mainly due to the fact that this part is mostly exploratory and results on set of axis of research that have to be deepened.

To close this thesis, we present some open questions and perspectives of research.

8.1 Contributions of the Thesis

8.1.1 Software Architectures for Multi-Agent Systems

Encouraged by the numerous evidences of the existence of a gap between design and implementation in MASs development, we thoroughly studied and analysed the various existing means of development that exist in the field, both for design and for implementation. This brought us to point up a particular specificity in MASs development: all the requirements that are expressed initially when developing a MAS are not all answered by the design, and thus must be answered at implementation. Moreover, the choice of a given approach to MAS development, and there are many, introduces supplementary requirements that also must be answered at implementation. On top of that, the choices made during the design by the MAS

designer are themselves new requirements towards the implementation. The combination of all those requirements makes each development very specific to the problem answered, to the MAS developer, to the chosen design approach, etc. The main conclusion here is that it is not possible to have a "one-size-fits-all" development support that would support every development and its specificities. The second conclusion is that every time a MAS design is being produced, the requirements that have an important impact on the implementation can be expressed as the definition of the types of agents of the application being developed. Such types of agents describe what are the interaction means that the agents use, what are their internal dynamics, but also more operative constraints that are important for the design and the implementation. We thus extracted challenges for the design and the implementation of such types of agents in order to better support the implementation of the MAS design itself.

To answer that, we proposed a coherent set of answers that takes its inspiration from software architectures as a mean to organise the development, and Component-Based Software Architectures (CBSAs) as a mean to support the development with quality, productivity and maintainability in mind. It takes the form of our own understanding of the general methodology of MAS development. This methodology separates the development into two phases. The first one is what we call the macro-level architectural design, focused on applying a MAS approach to define the system in terms of agents, their behaviours, their interactions, etc. The second one is what we call micro-level architectural design, focused on building an architecture adapted to the specific types of agents of the development and providing the abstractions needed to implement the MAS design in a straightforward way.

In order to ease the construction of such a micro-level architecture as well as make reuse and maintenance possible, we proposed next a novel component model, named SpeAD (Species-based Architectural Design), that introduces specific abstractions, namely species and ecosystem, that are usable to define and implement the types of agents of a given application and their computational environment. We showed how SpeAD can be used to model common interaction means as well as more operative ones. We also proposed a method, named SpEArAF (Species to Engineer Architectures for Agent Frameworks) that instantiates the general methodology of MAS development with SpeAD. To illustrate all of that, we presented a real MAS-based application from a research project developed in our research team. We analysed the contribution with respect to the state of the art, to the previously identified implementation challenges and to experimental feedbacks from users to conclude on the advantages of using our contribution for MAS development.

8.1.2 Component-Based Software Architectures

From this contribution, we proposed to compare it to existing works from the software architecture field, and in particular the Component-Based Software Architectures (CBSAs) field. This brought us to draw a set of ideas on the production of development supports for CBSAs without considering MASs at all.

To support the proposed ideas, we presented an integrated understanding of the different concepts used in the CBSA field, namely frameworks, components models, component

containers, architectural patterns, architectural styles, architectures and components.

We first compared our use of the proposed component model with object-oriented frameworks to conclude that our model could be used to define component-based frameworks. In particular, this idea is used in the first part to provide reusable architectures with partial implementation for MAS development.

Then, we compared the abstractions we introduced in our model to the dynamic instantiation of components as it can be found in works of the field, and then, by relating to several works in the literature, we showed how our component model could be used to build dedicated component-based component containers.

These results are of course still in the rough and we thus tried to show for each which were the directions to explore.

8.1.3 Multi-Agent Systems and Component-Based Software Architectures Side by Side

In the light of the previous results, we then re-examined MASs development with respect to CBSAs.

The main conclusion we drew is that, coherently with the conclusions of the state of the art, MASs approaches are not one design paradigm, but a family of such. And linking that to our component model and CBSAs, when doing micro-architectural level design, we are actually building models and containers for components that are the agents.

In this context, a MAS from a macro-level architectural point of view is seen as a CBSA, but more interestingly, because of the fact that such a CBSA is implemented using a component container realised with our component model, a MAS is also seen as a component that can be used together with more traditional components.

This brought us to conclude that our component model was well adapted for horizontal integration of MASs and CBSAs, which is important for choosing a relevant design paradigm depending on the type of problems that appear in a same application.

8.1.4 Multi-Agent Systems for Software Architectures

Independently from the previous set of results, we also investigated how MASs, as an approach for designing self-adaptive complex systems, can be used to support the construction of self-adaptive software architectures. We focused most of that investigation towards CBSAs. Using two experiments, we presented how we could first adapt a composition of services using organisational abstractions for MASs, and on the other hand, how we could let components in an ambient environment self-compose opportunistically by physically distributing the control and let adapted compositions appear.

The main new challenges supported by these experiments was to use agents as the runtime containers of the components. An important point here is that the agents do not replace components but support the adaptation of their composition. The MAS formed by these agents, and potentially others, was providing the self-organisation needed to adapt the system

to the non-functional and functional requirements, but also to the unspecified needs of the users.

Using these experiments as a starting point, we proposed the idea of pushing the integration between MASs and software architectures further to produce self-design software systems where MASs would support decision making for design, adaptation of the system at runtime, and even evolution of the design and of the system in an integrated way.

8.2 Open Problems and Perspectives

As we can see from this summary of the contributions, a subset of them are already perspectives of further research. But on top of that, there exist open problems that are closer to the contribution we proposed in the first part of this thesis.

We mainly see three main types of perspectives related to MAS development: methodological, architectural and evaluative. We conclude this section with one open problem that connects all the different perspectives together.

8.2.1 Methodological Perspectives

As it is highlighted in the state of the art of this thesis and in the methodological aspect of the contribution, MASs have specificities in term of methodology of development. This matter should be investigated even more. First, it is necessary to deepen the question of how current practices and methods instantiate this methodology and which are the implications it has on the development. Then, the previous point should be a starting point to better integrate tools and models in the context of model-driven engineering. Furthermore, the field of requirements engineering specialises in studying requirements in software engineering. Our contribution could most certainly be improved by seeing how they both relate.

This brings up the matter of the macro-level architectural design and its documentation. Indeed, we noticed that MAS researchers do not always know how to document and describe their designs to other researchers. We think there are guidelines, linked to the methodology of MAS development, that could be elicited on how to properly present a MAS design in a complete way.

More on the side of proposing dedicated architectures and development supports to the MAS developers, an interest has grown in the MAS development community towards method fragments to build tailored development processes for a given the problem. With a focus on MAS macro-level architectural design, it would be interesting to investigate how our approach for building tailored micro-level architectures is linked to it.

Finally, on the matter of reusing design and implementation products, the field of Multi-Agent Systems Software Product Lines has investigated ways to build reusable architectures for lines of products. Again, it would be interesting to see how our contribution integrates with these works and can contribute to the question.

8.2.2 Architectural Perspectives

We proposed an architectural solution to the question of the micro-level architectural design. In that context, it is necessary to continue to investigate how the architectural abstractions proposed answer the challenges, and how they can do it better. In particular, we think about the various research directions developed Chapter 6 on reusing design and patterns as code using templates.

Of course, on a technical side, the tool that supports the application of our contribution can also be improved by providing better ease of use. The component library that was produced must also be further extended and documented. In particular, we think there is a strong need for tools helping to find adequate reusable components when developing MASs since a great diversity of mechanisms usable by agents exists.

8.2.3 Evaluation Perspectives

One of the regret we have with the contribution is the fact that it has not been evaluated as thoroughly as possible. We think it is necessary to clearly evaluate what its different aspects provide to MAS developers, but also how the tool we provide actually succeeds in supporting its application.

APPENDIX A

Design and Implementation MAS Meta-Models

A.1 Agent and MAS Meta-Models

AGR. AGR (Agents, Groups, Roles) (Ferber, Gutknecht, and Michel 2004), previously called AALAADIN, is focused on describing the organisation of the agents partitioned in groups where agents take roles. No assumption is made on the internal structure or capabilities of the agents except that they communicate by messages inside groups. The difference is made between the structure of the organisation and its concrete realisation by agents at runtime.

MOISE+. MOISE+ (Hübner, Sichman, and Boissier 2002) is a meta-model based on the notion of roles and organisation, with a focus on the definition of laws and norms to regulate the organisation. Agents are able to reason about the organisation, but apart from that, no assumption is made on their internal working.

OperA. With the same objectives as MOISE+, OperA (Organization per Agents) (Dignum 2004) proposes to design the organisation of the agents and define norms and law.

Macodo. Macodo (Middleware Architecture for COntext-driven Dynamic agent Organizations) (Weyns, Haesevoets, and Helleboogh 2010) is a meta-model based on the notion of roles and organisation, with a focus on the organisation as a first class runtime entity responsible of enabling and enforcing its laws and norms.

CRIO. The CRIO (Capacity, Roles, Interaction, Organisation) meta-model, present in the ASPECS method (Cossentino et al. 2010) along with holons, is an evolution of the RIO meta-model (Hilaire et al. 2000). In this meta-model, it is proposed to represent the organisation

and its roles, but it distinguishes from other works by integrating these concepts with the one of holons.

AEIO. The VOWELS approach (Demazeau 1995; Da Silva and Demazeau 2002), which proposes the AEIO (Agent, Environment, Interaction, Organisation) meta-model, was one of the first to underline the importance of interaction and its modelling to build MASs. This approach tackles the matter of coordinating agents activity by proposing an integrated coordination meta-model for all the four components of MASs: Agent, Environment, Interaction and Organisation instead of focusing on one at a time.

A&A. The A&A (Agents and Artifacts) meta-model (Ricci, Viroli, and Omicini 2008), strongly linked to the SODA method (Molesini et al. 2006), proposes to represent artefacts. Artefacts are abstractions to represent means for the agents to interact, perceive and act. Situated agents use the artefacts that abstract whatever is present in the environment.

ORA4MAS. Based on A&A and MOISE+, ORA4MAS (Organisational Artefacts for Multi-Agent Systems) (Hübner et al. 2010) proposes to use artefacts to represent an organisation and to give to the agents the power to perceive and act on it.

MASQ. The MASQ (MASs based on Quadrants) meta-model (Stratulat, Ferber, and Tranier 2009) tries to put concepts of organisations and environments together. It proposes to model four aspects of MASs: interior-individual (what happens inside the agents), exterior-individual (how the agents interact with the outside), interior-collective (how the environment is considered by the entities participating in it) and exterior-collective (how the environment is by itself). This meta-model stays at an abstract and high level of representation of these different aspects of MASs.

SeSAmUML. SeSAmUML (Shell for Simulated Agent Systems UML) (Oechslein et al. 2001), with a focus on MABS, extends UML to describe a simulation in terms of agents and their behaviours (with activity diagrams), their environment and its resources.

IODA. IODA (Interaction Oriented Design of Agent simulations) (Kubera, Mathieu, and Picault 2011) is an approach and a meta-model focused on interactions as first-class entity for MABS (Multi-Agent-Based Simulation) where everything is agent. The argument is that interactions should be modelled and realised independently of the agents acting in them (what they call interaction polymorphism). Agents (and resources which are considered as agents because they can take part in interactions) are situated in spaces. Interactions are modelled using conditions, initiators and effects and agents can chose between the possible interactions to initiate when their turns come.

FIPA. The FIPA (Foundation for Intelligent Physical Agents) standards (Poslad and Charlton 2006) are the most known and used meta-model in the MAS field. Their meta-model is focused on the agents themselves. They communicate by messages, without any explicit environment defined at all or particular action and perception capabilities. They also propose an Agent Communication Language (ACL) for the agents and a reference infrastructure with message transport, execution, agent and service directories, etc. A lot of other meta-models rely on the FIPA meta-model to model the agents.

GAIA. In GAIA (Zambonelli, Jennings, and Wooldridge 2003), a MAS is described as agents taking roles in an organisation. The design results in the definition of the agents and their behaviours. The agents interact using messages and protocols with a FIPA-like architecture.

INGENIAS. INGENIAS (Pavón, Gómez-Sanz, and Fuentes 2005) proposes the same kind of abstractions than GAIA and uses UML to model them. It also has FIPA-like agents.

PASSI. The PASSI2 method (Cossentino and Seidita 2009), an evolution of the PASSI method (Cossentino 2005), proposes the same kind of meta-models than GAIA with FIPA-like agents.

ADELFE. The ADELFE (*Atelier de Développement de Logiciels à Fonctionnalité Emergente*) method (Bernon, Camps, et al. 2005) is based on the AMAS (Adaptive Multi-Agent System) approach (Gleizes et al. 2008). It proposes the AMAS-ML meta-model to define the behaviour of the agents in terms of rules. The meta-model also defines the environment of the MAS as being made of passive and active entities.

For a presentation of the *AMAS approach*, see p. 11

Tropos. Tropos (Giorgini et al. 2005) models agents as taking part in an organisation that is extracted from the requirements of the problem to solve. Agents interact by taking roles in the organisation and their behaviour is modelled using plans extracted from the organisation.

AUML. AUML (Agent UML) (Bauer, Müller, and Odell 2001) extends UML to model protocols for multi-agent interactions. It was later extended in two different works (Bauer 2002; Huget 2003) to model the behaviours of the agents. Both describe agents in terms of behaviour, states and actions, and link it to the roles of the protocols. AUML has been used in numbers of other methods, mainly for describing interactions.

AML. AML (Agent Modeling Language) (Trencansky and Cervenka 2005) also extends UML to model MAS in terms of agents, resources and environment. It dissociates behavioured, mental and socialized entities to describe these elements of the system.

Actors The actor meta-model (Hewitt 1977; Agha 1986) — which has not originated from the MAS field but inversely that may be considered the first ancestor of the MAS approach to design — proposes to model actors (the equivalent of agents) as entities that can receive

messages and treat them sequentially and reactively. The meta-model is very precise and has strong theoretical formal foundations. It is often used as a means to implement agents that exchange messages, mostly in reactive MASs.

A.2 Development Support Meta-Models

A.2.1 Languages

In this category we can find the AgentSpeak(L) language (Rao 1996) supported by the platforms Jason (Bordini, Hübner, and Wooldridge 2008), the JAL language supported by JACK (Howden et al. 2001), the 3APL language (Hindriks et al. 1999), the Jadex platform (Pokahr, Braubach, and Lamersdorf 2005), and so on.

All of these languages and corresponding platforms focus on the internal architecture of the agents, and are to be integrated with other platforms providing the execution platform for the agents (except for the Jadex that is tied to JADE).

A.2.2 Frameworks and Platforms

Existing frameworks (and their platforms) are providing means to implement a MAS using a given meta-model. These meta-models either directly correspond to a method's meta-model or are provided by the framework.

Follows a list of these platforms and the meta-models they provide.

MadKit for AGR. MadKit (Gutknecht and Ferber 2001) provides means for the agents to live in an AGR-like world. In particular it provides agent life-cycle management, messages transport and provides them identifier to communicate. These services are actually realised using agents called system agents (by opposition to application agents) that are able to interact with the core kernel of the platform by using event-based hooks. The platform can thus be extended by defining system agents exploiting these special mechanisms.

JADE for FIPA. JADE (Bellifemine, Poggi, and Rimassa 1999) is a very famous Java platform used as the implementation target of a lot of methods to handle agents execution and interaction. It provides means to execute the behaviours of the agents in a concurrent way (with primitives to trigger behaviours depending on the message received). It also provides means to construct messages using the FIPA ACL and exchange messages with other agents thanks to a message transport service. It provides the agents with identifier. It is usable to have several networked agents communicating together. Then it also provides directory services according to the FIPA meta-model that are realised by agents (thus they can be accessed by messages even if the framework provides shortcuts to ease their use). The platform can not really be easily extended except by providing more advanced behaviour management as in the Jadex case.

CArtAgO for A&A. CArtAgO (Common ARTifact infrastructure for AGents Open environments) (Ricci, Viroli, and Omicini 2007) is usable to realise artefact-based environment in JAVA. It distinguishes the workspace (where agents and artefacts are situated), agents' "bodies" and artefacts themselves. The agent "mind" (which can be any platform focused on internal agent architecture, for example a Jason bridge is distributed with CArtAgO) controls the body using its effectors and sensors. The bodies gives access to the artefacts in the workspaces. In particular sensing dynamics is decoupled by the body: events from artefacts are collected by sensors, and the body explicitly asks for the collected percepts. Artefacts are referenced by an identifier provided by the platform and can generate events and provide the way to use interface (which are described in terms of their function and how to use them, for now represented as strings without semantics). Because of the use of artefacts, CArtAgO can actually be extended by defining new artefacts and new ways to use them. In a way, CArtAgO provides a means to define any abstraction by reformulating it in terms of artefacts. This is done for example for the implementation of ORA4MAS in JaCaMo presented later.

SimpA for A&A and activity-oriented behaviours. SimpA (Ricci, Mirko, and Giulio 2011) is based on CArtAgO and on top of it provides means to describe the agents behaviours by atomic and structured activities that are automatically scheduled by the platform.

S-MOISE+ for MOISE+. S-MOISE+ (Hübner, Sichman, and Boissier 2006) is usable to realise MOISE+ kind of organisations. Part of S-MOISE+, the OrgBox, gives agents access to the organisation primitives such as joining groups, accepting goals or roles, etc. Then a special agent is responsible of executing these primitives and changing the organisation while verifying constraints of MOISE+ are respected. The SACI [1] middleware is used to let agents communicate by messages, but theoretically any other message transport service can be used. Nothing is available to extend S-MOISE+ except for the internal architecture of agents that is not pre-specified.

J-MOISE+ for MOISE+ within BDI agents. J-MOISE+ (Hübner, Sichman, and Boissier 2007) is usable to manipulate and reason about MOISE+ abstraction (from S-MOISE+) from within Jason. It lets Jason handle communication middleware and relies on S-MOISE+ for handling the organisation itself.

JaCaMo for ORA4MAS and BDI agents. JaCaMo is the surname given by its author of the combination of Jason, CArtAgO and MOISE+. The main platform here is an implementation of MOISE+ with CArtAgO as presented in ORA4MAS.

Janus for ASPECS. Janus (Gaud et al. 2009) is a platform providing the concepts developed in CRIO and Holons meta-models and that is mostly FIPA-compatible. It contains a kernel where agents are represented either as atomic agents or holons. Several execution mechanisms

1. http://www.lti.pcs.usp.br/saci/

exist either using one thread by agent or a more lightweight mechanisms. It provides also identification, messages transport and directory services to respectively refer to, communicate with and find other agents. It should be noted that communication receivers are roles and not directly on agents' addresses. A part of the kernel is responsible of maintaining the organisation in terms of roles, groups, etc. For the agents, if they are atomic, they are implemented by a set of capabilities that are invoked (synchronously or asynchronously) depending on the role they play in the groups they are in. The roles are triggered based on received messages.

MAGIQUE. Magique (Bensaid and Mathieu 1997; Routier, Mathieu, and Secq 2001) is a platform which permits the construction of agents by gathering reusable units of code representing skills. Agents are actually made of agents and requests for performing actions are sent to the set of their contacts (which are their sub-agents) without explicit receiver specified. Thus, the set of skills of an agent can change dynamically but the inversely availability of skills at runtime is not guaranteed.

JEDI for IODA. JEDI (Kubera, Mathieu, and Picault 2008) is a platform for IODA with euclidean spaces and discrete time. It takes as input the interactions definitions as well as agents description and applies a generic algorithm to update the agents' states, makes them perceive their neighbours, choose an interaction and apply the chosen interaction. Agents act sequentially but in randomized order.

MALEVA. MALEVA (Briot, Meurisse, and Peschanski 2007) is a meta-model of software components for the building of complex behaviours of agents by composing elementary ones. MALEVA targets the behaviour of the agents with an explicit notion of control flow, without taking care of the environment.

Malaca. Malaca (Amor and Fuentes 2009) exploits a combination of components and aspects to define the internal mechanisms of agents. Interactions are done using message passing.

NetLogo. NetLogo (Wilensky 1999), very used in MABS, proposes to model a MAS using agents and patches. Agents can be of different races (kind of classes or types) and move between patches that represent places in a situated environment. Agents are defined using methods that can be executed by a general system loop definable by the developer. NetLogo is very generic and powerful, kind of object oriented language where the objects are situated and scheduled by the system. NetLogo introduces a set of tools for easing the visualisation and the execution of all the agents of the system in a synchronous way.

GAMA and GAML. GAML (GAMA Modeling Language), the language behind the GAMA (Gis & Agent-based Modelling Architecture) simulation platform (Taillandier et al. 2010), proposes to describe a situated environment where species (class of agent that can be

instantiated) are described in terms of actions they can do, behaviour (described with one of the following formalisms: imperative, final state machines and etho-modeling framework, which is a task-based behaviour meta-model) and state. The environment can be multiple and described either with grids or continuous spaces. The platform can then execute the agents according to their species and take care of synchronisation between agents and environments, while presenting visual informations about the simulation. GAMA can be extended by providing new internal architectures for the behaviour of the agents.

JavAct. JavAct (Arcangeli et al. 2004) is an implementation of the actor meta-model with mobility. Actors are implemented as pattern matching behaviour executed depending on the messages received. They have the simple actor programming abstractions such as "become" to change their behaviour, "send" to send an asynchronous message, "move" for mobility and "die" for disappearing.

APPENDIX B

Implementation of SpeAD: Make Agents Yourself

In this appendix, we present a brief description of the implementation of the SpeADL component model. It is available as a tool named MAY (Make Agents Yourself) that we present. It takes the form of a transformation from SpeADL to Java and the generated code is the one used to implement components as described Section 3.2.

B.1 Make Agents Yourself

SpeADL was completely implemented as a textual editor and code generator. This tool is named MAY (Make Agents Yourself) and is released under the GNU General Public License (GPL)[1]. The implementation relies on *Eclipse* and the *Xtext*[2] framework. The editor provides syntactic colouration, automatic completion, type verification for components bindings (including the parametric types and the Java types).

All the code generated by the tool does not have to be modified and every implementation is done through class extension as presented in this chapter. Thus, with it component programming is flexible and incremental: component descriptions are considered as code, every modification on it is reflected on the generated code, and the implemented code is directly impacted by these updates through the generated class.

Figure B.1 shows an instance of Eclipse with a SpeADL description on the left and a component implementation on the right.

1. http://www.irit.fr/MAY
2. http://www.eclipse.org/Xtext/

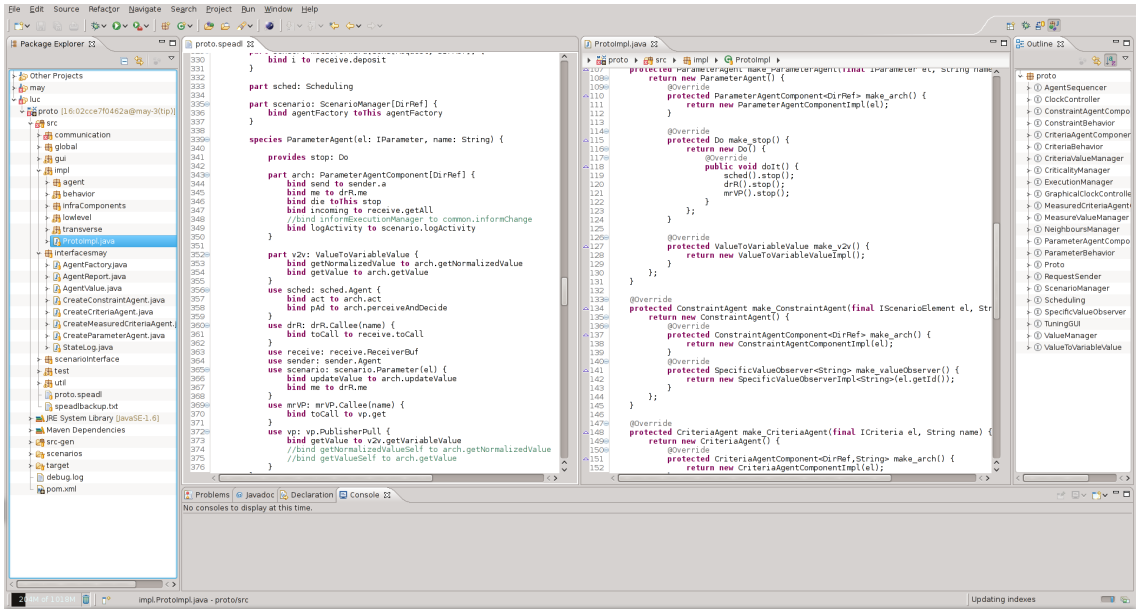


Figure B.1: MAY in Eclipse

```
ecosystem EcosystemX {
  provides portX: InterfaceX
  requires portY: InterfaceY

  part partZ: EcosystemZ

  species SpeciesY
}
```

Figure B.2: Component description for a component named `EcosystemX` in SpeADL

B.2 From SpeADL to Java

Figure B.3 shows the set of classes generated from the SpeADL description shown Figure B.2. The formalism used is UML2 with some adaptations. The dependency labelled `is` between `SpeciesY` and `EcosystemX` means that the way the class `SpeciesY` is generated follows the same schema as `EcosystemX` (except for having species itself obviously). The `$` character represents containment of classes (inner classes): for example `EcosystemZ$Component` is an inner class of the class `EcosystemZ`. The two classes `SpeciesY` and `BridgePartZImpl` would be duplicated respectively for every species and parts of the ecosystem. Since the species follow the same schema than ecosystem for the generated classes, the species's uses are represented in the same way than the parts.

`EcosystemX` is what we previously called the description class. It is an abstract class that must be extended to implement the component. It contains all the abstract methods that must be implemented to specify an implementation for the parts, the provided ports and the species. It also contains methods to access the required ports, to access the parts as

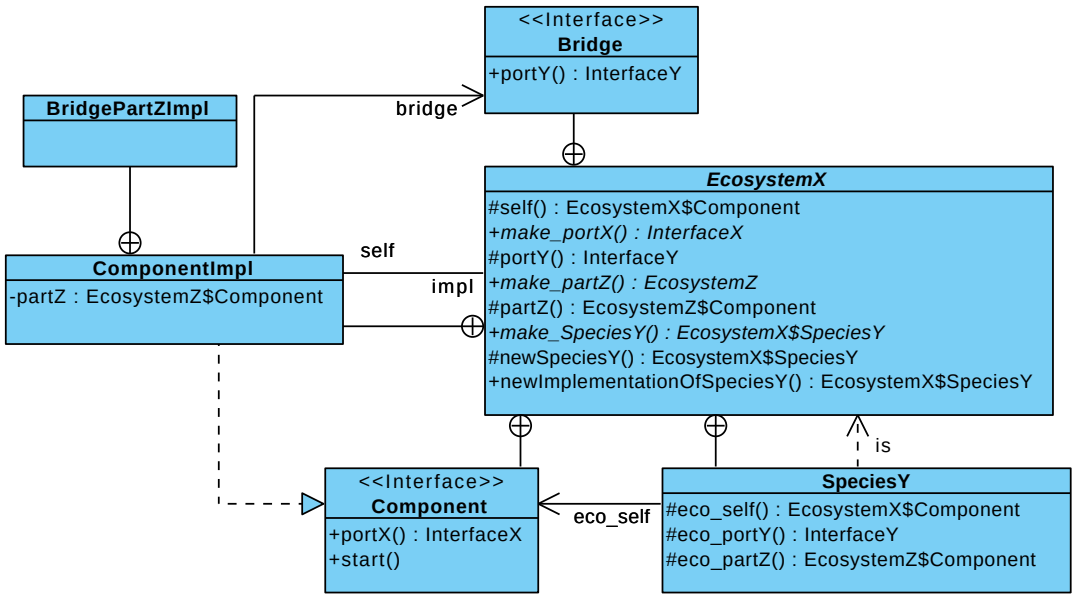


Figure B.3: Generated classes from SpeADL to Java for a component named `EcosystemX` in UML2

components, to access its provided ports and to create new instances of its species. Finally it contains the `start` method that can be overridden if needed.

The `EcosystemX$Bridge` interface represents the set of required ports of the component. It is meant to be implemented by composites that contain parts of the type of this component.

By opposition to `EcosystemX` that represents an implementation for the component, the inner interface `EcosystemX$Component` represents the component itself. It contains the provided ports of the component as well as the `start` method used to start the component. Internally to `EcosystemX`, a class, named `EcosystemX$ComponentImpl`, actually implements this interface to realise the configuration of the components such as the parts and the bindings.

For each of the parts an implementation of its bridge is present and realises the bindings to other parts or to ports of `EcosystemX`. For example with `partZ`, the class `EcosystemX$ComponentImpl$BridgePartZImpl` implements `EcosystemZ$Bridge`, which is not represented here but similar to `EcosystemX$Bridge`. Indeed, `EcosystemZ`, the generated classes for the component used in `EcosystemZ` as `partZ`, follows the same schema than `EcosystemX`.

The abstract class `SpeciesY`, that we previously called the species description class, represents an implementation for the species. The rest of its related inner classes — `Component`, `ComponentImpl` and `Bridge` — are not represented here but follow the same schema than `EcosystemX`. On top of the methods present in the description class, some additional methods are present in order to get access to the species' ecosystem during implementation. The method `newImplementationOfSpeciesY` returns an instance of the species and initialises its uses implementation recursively.

Author's Bibliography

French Journals

Noël, Victor, Jean-Paul Arcangeli, and Marie-Pierre Gleizes (2012). « Une approche architecturale à base de composants pour l'implémentation des Systèmes Multi-Agents ». In: *Revue des Nouvelles Technologies de l'Information*. In press (cit. on pp. xxiii, 114).

International Conferences and Workshops

Cruz Torres, Mario Henrique, Victor Noël, Tom Holvoet, and Jean-Paul Arcangeli (2010a). « MAS Organisation at your Composite Service ». In: *Proceedings of the 8th European Workshop on Multi-Agent Systems (EUMAS'10), Paris (France)*. Université Paris Descartes (cit. on pp. xxiii, 114, 143).

Cruz Torres, Mario Henrique, Victor Noël, Tom Holvoet, and Jean-Paul Arcangeli (2010b). « MAS Organisation at your Composite Service ». In: *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond (MONA+), Ayia Napa (Cyprus)*. Ed. by Dimka Karastoyanova, Raman Kazhamiakin, and Andreas Metzger. ACM, pp. 33–39 (cit. on pp. xxiii, 114, 144).

Georgé, Jean-Pierre, Marie-Pierre Gleizes, Francisco Garijo, Victor Noël, and Jean-Paul Arcangeli (2010). « Self-adaptive Coordination for Robot Teams Accomplishing Critical Activities ». In: *Advances in Practical Applications of Agents and Multiagent Systems, 8th International Conference on Practical Applications of Agents and Multiagent Systems (PAAMS 2010), Salamanca (Spain)*. Ed. by Yves Demazeau, Frank Dignum, Juan M. Corchado, and Javier Bajo. Vol. 70. Advances in Soft Computing. Springer, pp. 145–150 (cit. on p. xxiii).

Lacouture, Jérôme, Victor Noël, Jean-Paul Arcangeli, and Marie-Pierre Gleizes (2011). « Engineering Agent Frameworks: An Application in Multi-Robot Systems ». In: *Advances in Practical Applications of Agents and Multiagent Systems, 9th International Conference on Practical Applications of Agents and Multiagent Systems (PAAMS 2011), Salamanca (Spain)*. Ed. by Yves Demazeau, Michal Pechoucek, Juan M. Corchado, and Javier Bajo Pérez. Advances in Intelligent and Soft Computing. Springer (cit. on pp. xxiii, 114).

Noël, Victor, Jean-Paul Arcangeli, and Marie-Pierre Gleizes (2010a). « Component-Based Agent Architectures to Build Dedicated Agent Frameworks ». In: *Proceedings of the 7th International Symposium "From Agent Theory to Agent Implementation" (AT2AI-7), Vienna (Austria)*. Ed. by Robert Trappl. Austrian Society for Cybernetic Studies, Apr. 2010, pp. 483–488 (cit. on pp. xxiii, 113).

Noël, Victor, Jean-Paul Arcangeli, and Marie-Pierre Gleizes (2010b). « Between Design and Implementation of Multi-Agent Systems: A Component-Based Two-Step Process ». In: *Proceedings of the 8th European Workshop on Multi-Agent Systems (EUMAS'10), Paris (France)*. Université Paris Descartes, Dec. 2010 (cit. on pp. xxiii, 113).

Noël, Victor and Antonis Kakas (2009). « Gorgias-C: Extending Argumentation with Constraint Solving ». In: *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (System Descriptions) (LPNMR 2009), Potsdam (Germany)*. Ed. by Esra Erdem, Fangzhen Lin, and Torsten Schaub. Vol. 5753. Lecture Notes in Computer Science. Springer, Sept. 2009, pp. 535–541 (cit. on p. xxiii).

French Conferences and Workshops

Denis, Grégoire, Victor Noël, Jean-Paul Arcangeli, Sylvie Trouilhet, and Charles Triboulot (2012). « Composition opportuniste et ascendante à base d'agents coopératifs ». In: *8èmes Journées Francophones Mobilité et Ubiquité (UBIMOB 2012)*. In press (cit. on pp. xxiv, 114).

Noël, Victor and Jean-Paul Arcangeli (2011). « Frameworks, architectures et composants: revisiter le développement de systèmes multi-agents ». In: *Conférence Francophone sur les Architectures Logicielles (CAL), Lille (France)*, pp. 23–32 (cit. on pp. xxiii, 114).

Noël, Victor, Sylvain Rougemaille, Jean-Paul Arcangeli, Jean-Pierre Georgé, Frédéric Migeon, and Stephane Dudouit (2009). « MAY : Make Agents Yourself. Un générateur d'API agent à base de composants ». In: *Journées du GDR GPL (Session Outils et Posters) (GDR GPL), Toulouse (France)*. Ed. by Yves Ledru and Marc Pantel. IRIT Press, Jan. 2009, pp. 262–263 (cit. on p. xxiii).

Books Parts

Lacouture, Jérôme, Ismael Rodriguez, Jean-Paul Arcangeli, Christophe Chassot, Thierry Desprats, Khalil Drira, Francisco Garijo, Victor Noël, Michelle Sibilla, and Catherine-noel Tessier (2011). « Mission-Aware Adaptive Communication for Collaborative Mobile Entities ». In: *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*. Ed. by Maria Manuela Cruz-Cunha and Fernando Moreira. IGI Global. Chap. 64, pp. 1056–1076 (cit. on p. xxiii).

Bibliography

Abowd, Gregory, Robert Allen, and David Garlan (1993). « Using Style to Understand Descriptions of Software Architecture ». In: *Proceedings of SIGSOFT'93: Foundations of Software Engineering*. Software Engineering Notes 18(5). ACM Press, Dec. 1993, pp. 9–20 (cit. on p. 8).

Agha, Gul (1986). *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press (cit. on p. 169).

Amor, Mercedes and Lidia Fuentes (2009). « Malaca: A Component and Aspect-Oriented Agent Architecture ». In: *Information & Software Technology* 51.6, pp. 1052–1065 (cit. on pp. 36, 38, 172).

Amor, Mercedes, Lidia Fuentes, and Antonio Vallecillo (2005). « Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA ». In: *Agent-Oriented Software Engineering V*. Ed. by James Odell, Paolo Giorgini, and Jörg Müller. Vol. 3382. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 93–108 (cit. on pp. 28, 38).

Arcangeli, Jean-Paul, Vincent Hennebert, Sébastien Leriche, Frédéric Migeon, and Marc Pantel (2004). *JavAct*. `http://javact.org/`. Institut de Recherche en Informatique de Toulouse, Toulouse University. Toulouse, FR. (cit. on p. 173).

Arlow, Jim and Ila Neustadt (2005). *UML 2 and the Unified Process*. 2nd Edition (cit. on p. 3).

Bachmann, Felix, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau (2000). *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition*. technical CMU/SEI-2000-TR-008. Software Engineering Institute, Carnigie Mellon University (cit. on pp. 6, 110, 122–125, 127).

Bass, Len, Paul Clements, and Rick Kazman (2003). *Software Architecture in Practice*. 2nd. Boston, MA, USA: Addison-Wesley (cit. on pp. 5, 8).

Bauer, Bernhard (2002). « UML Class Diagrams Revisited in the Context of Agent-Based Systems ». In: *Agent-Oriented Software Engineering II*. Ed. by Michael Wooldridge, Gerhard Weiß, and Paolo Ciancarini. Vol. 2222. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 101–118 (cit. on p. 169).

Bauer, Bernhard, Jörg Müller, and James Odell (2001). « Agent UML: A Formalism for Specifying Multiagent Software Systems ». In: *Agent-Oriented Software Engineering*. Ed. by Paolo Ciancarini and Michael Wooldridge. Vol. 1957. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 109–120 (cit. on p. 169).

Behrens, Tristan, Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Jürgen Dix, Koen Hindriks, Jomi F. Hübner, and Alexander Pokahr (2011). « An Interface for Agent-Environment Interaction ». In: *Proceedings of International Workshop on Programming Multi-Agent Systems (ProMAS-8)*. Springer (cit. on p. 36).

Bellifemine, Fabio, Agostino Poggi, and Giovanni Rimassa (1999). « JADE - A FIPA-Compliant Agent Framework ». In: *4th Proceedings of International Conference on the Practical Applications of Intelligent Agents*, pp. 97–108 (cit. on p. 170).

Bensaid, Nourredine and Philippe Mathieu (1997). « A Hybrid and Hierarchical Multi-Agent Architecture Model ». In: *In Proceedings of PAAM'97*, pp. 145–155 (cit. on p. 172).

Bergenti, Federico, Marie-Pierre Gleizes, and Franco Zambonelli, eds. (2004). *Methodologies and Software Engineering for Agent Systems*. Klüwer Academic Press (cit. on pp. 17, 38).

Bernon, Carole, Valérie Camps, Marie-Pierre Gleizes, and Gauthier Picard (2005). « Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology ». In: *Agent-Oriented Methodologies*. Ed. by Brian Henderson-Sellers and Paolo Giorgini. Idea Group Publishing, pp. 172–202 (cit. on pp. 11, 169).

Bernon, Carole, Massimo Cossentino, Marie-Pierre Gleizes, Paola Turci, and Franco Zambonelli (2005). « A Study of Some Multi-agent Meta-models ». In: *Agent-Oriented Software Engineering V*. Ed. by James Odell, Paolo Giorgini, and Jörg Müller. Vol. 3382. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 62–77 (cit. on pp. 29, 38).

Bernon, Carole, Massimo Cossentino, and Juan Pavón (2005). « An overview of current trends in european aose research ». In: *Informatica* 29, pp. 379–390 (cit. on p. 18).

Beydoun, Ghassan, Cesar Gonzalez-Perez, Graham Low, and Brian Henderson-Sellers (2005). « Synthesis of a generic MAS metamodel ». In: *SIGSOFT Softw. Eng. Notes* 30 (4 May 2005), pp. 1–5 (cit. on pp. 29, 38).

Bézivin, Jean and Olivier Gerbé (2001). « Towards a Precise Definition of the OMG/MDA Framework ». In: *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*. IEEE Computer Society, pp. 273–280 (cit. on p. 2).

Bonjean, Noélie, Carole Bernon, and Pierre Glize (2009). « Engineering Development of Agents using the Cooperative Behaviour of their Components ». In: *MAS&S @ MALLOW'09, Turin*. Ed. by Giancarlo Fortino, Massimo Cossentino, Marie-Pierre Gleizes, and Juan Pavón. Vol. 494. CEUR Workshop Proceedings (cit. on p. 113).

Bonjean, Noelie, Marie-Pierre Gleizes, Christine Maurel, and Frederic Migeon (2012). « Forward Self-Combined Method Fragments ». In: *Workshop on Agent Oriented Software Engineering (AOSE), Valencia, Spain, 04/06/2012-08/06/2012*. Springer (cit. on p. 114).

Bordini, Rafael H., Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, J. Leite, G. O'Hare, Alexander Pokahr, and Alessandro Ricci (2006). « A Survey of Programming Languages and Platforms for Multi-Agent Systems ». In: *Informatica* 30, pp. 33–44 (cit. on pp. 17, 38).

Bordini, Rafael H., Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, eds. (2005). *Multi-Agent Programming: Languages, Platforms and Applications*. Vol. 15. Multiagent Systems, Artificial Societies and Simulated Organizations. Springer (cit. on p. 17).

Bordini, Rafael H., Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, eds. (2009). *Multi-Agent Programming: Languages, Tools and Applications*. 1st. Springer Publishing Company, Incorporated (cit. on pp. 17, 38).

Bordini, Rafael H., Jomi F. Hübner, and Michael Wooldridge (2008). *Programming multi-agent systems in AgentSpeak using Jason*. Vol. 8. Wiley-Interscience (cit. on p. 170).

Bosch, Jan (2009). « From software product lines to software ecosystems ». In: *Proceedings of the 13th International Software Product Line Conference*. San Francisco, California: Carnegie Mellon University, pp. 111–119 (cit. on p. 62).

Bouziane, Hinde L., Christian Pérez, and Thierry Priol (2008). « A software component model with spatial and temporal compositions for grid infrastructures ». In: *Euro-Par 2008–Parallel Processing*, pp. 698–708 (cit. on p. 154).

Bratman, Michael E. (1999). *Intention, Plans, and Practical Reason*. Cambridge University Press, Mar. 1999 (cit. on p. 25).

Brazier, Frances M. T., Catholijn M. Jonker, and Jan Treur (1999). « Compositional Design and Reuse of a Generic Agent Model ». In: *Applied Artificial Intelligence Journal* 14, pp. 491–538 (cit. on pp. 36, 38).

Briot, Jean-Pierre, Thomas Meurisse, and Frédéric Peschanski (2007). « Architectural Design of Component-Based Agents: A Behavior-Based Approach ». In: *Programming Multi-Agent Systems*. Ed. by Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. Vol. 4411. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 71–90 (cit. on pp. 36, 38, 172).

Camps, Valérie, Bernard Carpuat, Marie-Pierre Gleizes, Pierre Glize, André Machonin, Christine Piquemal, Jo Link-Pezet, Christine Régis, and Sylvie Trouilhet (1994). « Vers l'intelligence artificielle collective ». In: *Journée Systèmes Multi-Agents du PRC-GRD I.A., Paris, 16/12/94-16/12/94* (cit. on p. 10).

Camps, Valérie, Marie-Pierre Gleizes, and Pierre Glize (1998). « A self-organization process based on cooperation theory for adaptive artificial systems ». In: *Problems of Evolution in Real and Virtual Systems: Proceedings of the First International Conference on Philosophy and Computer Science, November 2-4, 1998*. Krakow, Poland: Jagiellonian University, University of Mining, and Metallurgy (cit. on p. 10).

Capera, Davy, Jean-Pierre Georgé, Marie-Pierre Gleizes, and Pierre Glize (2003). « The AMAS Theory for Complex Problem Solving Based on Self-organizing Cooperative Agents ». In: *International Workshop on Theory And Practice of Open Computational Systems (TAPOCS)*. IEEE Computer Society Press, pp. 389–394 (cit. on p. 10).

Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford (2003). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley (cit. on pp. 1, 7, 8).

Clements, Paul and Linda Northrop (2001). *Software product lines*. Addison-Wesley (cit. on p. 8).

Cossentino, Massimo (2005). « From requirements to code with the PASSI methodology ». In: *Agent-Oriented Methodologies*. Ed. by Brian Henderson-Sellers and Paolo Giorgini. Idea Group Publishing, pp. 79–106 (cit. on p. 169).

Cossentino, Massimo, Nicolas Gaud, Vincent Hilaire, Stéphane Galland, and Abderrafiâa Koukam (2010). « ASPECS: an agent-oriented software process for engineering complex systems ». In: *Autonomous Agents and Multi-Agent Systems* 20 (2 2010), pp. 260–304 (cit. on p. 167).

Cossentino, Massimo and Valeria Seidita (2009). *PASSI 2 - Going Towards Maturity of the PASSI Process*. Tech. rep. 09-02. ICAR-CNR, Dec. 2009 (cit. on p. 169).

Crnković, Ivica, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan (2002). « Specification, implementation, and deployment of components ». In: *Commun. ACM* 45.10 (Oct. 2002), pp. 35–40. DOI: 10.1145/570907.570928 (cit. on pp. 122, 124).

Crnković, Ivica, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron (2011). « A Classification Framework for Software Component Models ». In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 593–615. DOI: 10.1109/TSE.2010.83 (cit. on p. 6).

Cruz Torres, Mario Henrique and Tom Holvoet (2011a). « Composite service adaptation: a QoS-driven approach ». In: *Comsware* (cit. on p. 144).

Cruz Torres, Mario Henrique and Tom Holvoet (2011b). « Towards robust service workflows: a decentralized approach ». In: *CoopIS - Cooperative Information Systems* (cit. on p. 144).

Dalpiaz, Fabiano, Ambra Molesini, Mariachiara Puviani, and Valeria Seidita (2008). « Towards Filling the Gap between AOSE Methodologies and Infrastructures: Requirements and Meta-model ». In: *9th Workshop "From Objects to Agents" (WOA 2008) – Evolution of Agent Development: Methodologies, Tools, Platforms and Languages*. Ed. by Matteo Baldoni, Massimo Cossentino, Flavio De Paoli, and Valeria Seidita. Palermo, Italy: Seneca Edizioni, Nov. 2008, pp. 115–121 (cit. on pp. 29, 38).

Da Silva, Joao Luis T. and Yves Demazeau (2002). « Vowels co-ordination model ». In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*. AAMAS '02. Bologna, Italy: ACM, pp. 1129–1136 (cit. on p. 168).

Dehlinger, Josh and Robyn R. Lutz (2011). « Gaia-PL: A Product-Line Engineering Approach for Efficiently Designing Multi-Agent Systems ». In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (cit. on pp. 24, 38).

DeLoach, Scott A. (2009). « OMACS: A framework for adaptive, complex systems ». In: *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. Ed. by Virginia Dignum. IGI Global, pp. 76–98 (cit. on p. 139).

Demazeau, Y. (1995). « From interactions to collective behaviour in agent-based systems ». In: *Proceedings of the First European conference on cognitive science*. Saint Malo, France, Apr. 1995, pp. 117–132 (cit. on pp. 10, 168).

Denis, Grégoire (2011). « Composition autonome et émergence au moyen d'agents-composants coopératifs ». MA thesis. Université Paul Sabatier, June 2011 (cit. on pp. 134, 152).

Desnos, Nicolas, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay (2007). « Automated and Unanticipated Flexible Component Substitution ». In: *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering*. Ed. by Heinz Schmidt, Ivica Crnkovic, George Heineman, and Judith Stafford. Vol. 4608. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 33–48. DOI: 10.1007/978-3-540-73551-9_3 (cit. on p. 155).

Dignum, Virginia (2004). « A Model for Organizational Interaction: Based on Agents, Founded in Logic ». PhD thesis. Universiteit Utrecht (cit. on p. 167).

Dignum, Virginia, ed. (2009a). *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global.

Dignum, Virginia (2009b). « The Role of Organization in Agent Systems ». In: *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. Ed. by Virginia Dignum. IGI Global, pp. 1–17 (cit. on p. 139).

Di Marzo Serugendo, Giovanna, Marie-Pierre Gleizes, and Anthony Karageorgos, eds. (2011). *Self-organising Software*. Natural Computing Series. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-17348-6 (cit. on p. 135).

Eden, Amnon H. and Rick Kazman (2003). « Architecture, design, implementation ». In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. May 2003, pp. 149–159. DOI: 10.1109/ICSE.2003.1201196 (cit. on p. 8).

Ferber, Jacques (1995). *Les systèmes multi-agents : vers une intelligence collective*. InterEditions (cit. on pp. 9, 10).

Ferber, Jacques, Olivier Gutknecht, and Fabien Michel (2004). « From Agents to Organizations: An Organizational View of Multi-agent Systems ». In: *Agent-Oriented Software Engineering IV*. Ed. by Paolo Giorgini, Jörg Müller, and James Odell. Vol. 2935. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 443–459 (cit. on p. 167).

Fielding, Roy Thomas (2000). « Architectural Styles and the Design of Network-based Software Architectures ». PhD thesis (cit. on pp. 5, 123, 127).

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. A. Wesley (cit. on p. 8).

García-Magariño, Iván (2009). « Towards the Coexistence of Different Multi-Agent System Modeling Languages with a Powertype-Based Metamodel ». In: *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*. Ed. by Juan Corchado, Sara Rodríguez, James Llinas, and José Molina. Vol. 50. Advances in Soft Computing. Springer Berlin / Heidelberg, pp. 189–193 (cit. on pp. 29, 38).

Garlan, David, Jeffrey M. Barnes, Bradley Schmerl, and Orieta Celiku (2009). « Evolution styles: Foundations and tool support for software architecture evolution ». In: *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. Oct. 2009, pp. 131–140. DOI: 10.1109/WICSA.2009.5290799 (cit. on p. 156).

Garlan, David, Robert T. Monroe, and David Wile (1997). « Acme: an architecture description interchange language ». In: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*. Ed. by J. Howard Johnson. IBM, p. 7 (cit. on p. 69).

Garlan, David and Mary Shaw (1993). « An introduction to software architecture ». In: *Advances in Software Engineering and Knowledge Engineering*. Publishing Company, pp. 1–39 (cit. on p. 5).

Gaud, Nicolas, Stéphane Galland, Vincent Hilaire, and Abderrafiâa Koukam (2009). « An Organisational Platform for Holonic and Multiagent Systems ». In: *Programming Multi-Agent Systems*. Ed. by Koen Hindriks, Alexander Pokahr, and Sebastian Sardina. Vol. 5442. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 104–119 (cit. on p. 171).

Georgé, Jean-Pierre, Marie-Pierre Gleizes, and Valérie Camps (2011). « Cooperation ». In: *Self-organising Software*. Ed. by Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. Natural Computing Series. Springer Berlin Heidelberg, pp. 193–226. DOI: 10.1007/978-3-642-17348-6_9 (cit. on p. 11).

Georgé, Jean-Pierre, Gauthier Picard, Marie-Pierre Gleizes, and Pierre Glize (2003). « Living Design for Open Computational Systems ». In: *International Workshop on Theory And Practice of Open Computational Systems (TAPOCS at IEEE 12th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003) (TAPOCS), Linz, Austria, 09/06/2003-11/06/2003*. http://www.computer.org: IEEE Computer Society, June 2003, pp. 389–394 (cit. on p. 30).

Georgé, J.-P., J.-P. Mano, Marie-Pierre Gleizes, M. Morel, A. Bonnot, and D. Carreras (2009). « Emergent Maritime Multi-Sensor Surveillance Using an Adaptive Multi-Agent System ». In: *Cognitive systems with Interactive Sensors (COGIS), Paris*. SEE/URISCA, Nov. 2009 (cit. on p. 114).

Ghosh, Debasish (2010). *DSLs in action*. Manning Publications Co. (cit. on pp. 4, 127).

Giorgini, P., J. Mylopoulos, A. Perini, and A. Susi (2005). « The Tropos Metamodel and its Use ». In: *Informatical journal* (cit. on pp. 23, 169).

Girardi, Rosario and Alisson Lindoso (2006). « An Ontology-Driven Technique for the Architectural and Detailed Design of Multi-agent Frameworks ». In: *Agent-Oriented Information Systems III*. Ed. by Manuel Kolp, Paolo Bresciani, Brian Henderson-Sellers, and Michael Winikoff. Vol. 3529. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 124–139 (cit. on pp. 24, 38).

Gleizes, Marie Pierre, Valérie Camps, Jean-Pierre Georgé, and Davy Capera (2008). « Engineering Systems Which Generate Emergent Functionalities ». In: *EEMMAS 2007*. Ed. by Danny Weyns, Sven A. Brueckner, and Yves Demazeau. Vol. 5049. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence). Springer, pp. 58–75 (cit. on pp. 11, 169).

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley (cit. on p. 123).

Greenfield, Jack and Keith Short (2003). « Software factories: assembling applications with patterns, models, frameworks and tools ». In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '03. Anaheim, CA, USA: ACM, pp. 16–27. DOI: 10.1145/949344.949348 (cit. on p. 121).

Gudgin, Martin, Marc Hadley, and Tony Rogers (2006). *Web Services Addressing 1.0 - Core*. World Wide Web Consortium, Recommendation REC-ws-addr-core-20060509. May 2006 (cit. on p. 137).

Guivarch, Valérian, Valérie Camps, and André Péninou (2012). « Amadeus : sensibilité au contexte et adaptation dans les systèmes ambiants par une approche multi-agent adaptative ». In: *8èmes journées francophones Mobilité et Ubiquité (UBIMOB 2012)*. Ed. by Stéphane Lavirotte and Makhlouf Derdour (cit. on p. 114).

Gutknecht, Olivier and Jacques Ferber (2001). « The MadKit Agent Platform Architecture ». In: *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*. Ed. by Tom Wagner and Omer Rana. Vol. 1887. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 48–55 (cit. on p. 170).

Hahn, Christian, Cristián Madrigal-Mora, and Klaus Fischer (2009). « A platform-independent metamodel for multiagent systems ». In: *Autonomous Agents and Multi-Agent Systems* 18 (2 2009), pp. 239–266 (cit. on pp. 29, 38).

Henderson-Sellers, Brian and Paolo Giorgini, eds. (2005). *Agent-Oriented Methodologies*. Idea Group Publishing (cit. on pp. 17, 38).

Hewitt, Carl (1977). « Viewing Control Structures as Patterns of Passing Messages ». In: *Artificial Intelligence* 8.3, pp. 323–364 (cit. on p. 169).

Hilaire, Vincent, Abder Koukam, Pablo Gruer, and Jean-pierre Müller (2000). « Formal Specification and Prototyping of Multi-Agent Systems ». In: *In ESAW '00: Proceedings of the First International Workshop on Engineering Societies in the Agent World*. Springer Verlag, pp. 114–127 (cit. on p. 167).

Hindriks, Koen V., Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer (1999). « Agent Programming in 3APL ». In: *Autonomous Agents and Multi-Agent Systems* 2 (4 1999), pp. 357–401 (cit. on p. 170).

Hock-koon, Anthony A. and Mourad Oussalah (2011). « The Product-Process-Quality Framework ». In: *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011*. IEEE, pp. 20–27 (cit. on p. 134).

Hofer, Christian and Klaus Ostermann (2010). « Modular domain-specific language components in Scala ». In: *Proceedings of the 9th international conference on Generative programming and Component engineering*. GPCE'10. Eindhoven, The Netherlands: ACM, pp. 83–92. DOI: 10.1145/1868294.1868307 (cit. on p. 129).

Howden, Nick, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas (2001). « JACK Intelligent Agents - Summary of an Agent Infrastructure ». In: *In 5th International conference on autonomous agents* (cit. on p. 170).

Hübner, Jomi F., Olivier Boissier, Rosine Kitio, and Alessandro Ricci (2010). « Instrumenting multi-agent organisations with organisational artifacts and agents ». In: *Autonomous Agents and Multi-Agent Systems* 20 (3 2010), pp. 369–400 (cit. on p. 168).

Hübner, Jomi F., Jaime S. Sichman, and O. Boissier (2007). « Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels ». In: *International Journal of Agent-Oriented Software Engineering* 1.3, pp. 370–395 (cit. on p. 171).

Hübner, Jomi F., Jaime S. Sichman, and Olivier Boissier (2002). « A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems ». In: *Advances in Artificial Intelligence*. Ed. by Guilherme Bittencourt and Geber Ramalho. Vol. 2507. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 439–448 (cit. on p. 167).

Hübner, Jomi F., Jaime Simão Sichman, and Olivier Boissier (2006). « S-moise+: A middleware for developing organised multi-agent systems ». In: *COIN I, volume 3913 of LNAI*. Springer, pp. 64–78 (cit. on p. 171).

Huget, Marc-Philippe (2003). « Agent UML Class Diagrams Revisited ». In: *Agent Technologies, Infrastructures, Tools, and Applications for E-Services*. Ed. by Jaime Carbonell, Jörg Siekmann, Ryszard Kowalczyk, Jörg Müller, Huaglory Tianfield, and Rainer Unland. Vol. 2592. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 49–60 (cit. on p. 169).

Ince, Darrel C., Leslie Hatton, and John Graham-Cumming (2012). « The case for open computer programs ». In: *Nature* 482.7386 (Feb. 23, 2012), pp. 485–488. DOI: 10.1038/nature10836 (cit. on p. 115).

Johnson, R.E. (1997). « Frameworks = (components + patterns) ». In: *Communications of the ACM* 40.10, pp. 39–42 (cit. on pp. 4, 121, 122, 124).

Kaddoum, Elsy (2011). « Optimisation sous contraintes de problèmes distribués par auto-organisation coopérative ». PhD thesis. Université de Toulouse, Toulouse, France, Nov. 2011 (cit. on p. 114).

Kerth, N.L. and W. Cunningham (1997). « Using patterns to improve our architectural vision ». In: *Software, IEEE* 14.1, pp. 53–59 (cit. on p. 9).

Klügl, Franziska, Rainer Herrler, and Christoph Oechslein (2003). « From Simulated to Real Environments: How to Use SeSAm for Software Development ». In: *Multiagent System Technologies*. Ed. by Michael Schillo, Matthias Klusch, Jörg Müller, and Huaglory Tianfield. Vol. 2831. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1097–1098 (cit. on pp. 28, 38).

Kubera, Yoann, Philippe Mathieu, and Sébastien Picault (2008). « Interaction-Oriented Agent Simulations : From Theory to Implementation ». In: *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*. Ed. by Malik Ghallab, Constantine Spyropoulos, Nikos Fakotakis, and Nikos Avouris. IOS Press, pp. 383–387 (cit. on p. 172).

Kubera, Yoann, Philippe Mathieu, and Sébastien Picault (2011). « IODA: an interaction-oriented approach for multi-agent based simulations ». In: *Autonomous Agents and Multi-Agent Systems* 23 (3 2011), pp. 303–343 (cit. on p. 168).

Le Goaer, Olivier, Dalila Tamzalit, Mourad Chabane Oussalah, and Abdelhak-Djamel Seriai (2008). « Evolution styles to the rescue of architectural evolution knowledge ». In: *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*. SHARK '08. Leipzig, Germany: ACM, pp. 31–36 (cit. on p. 156).

Leriche, Sebastien (2006). « Architectures à composants et agents pour la conception d'applications réparties adaptables ». PhD thesis. Université Paul Sabatier, Toulouse, France (cit. on p. xxi).

Leriche, Sébastien and Jean-Paul Arcangeli (2010). « Flexible Architectures of Adaptive Agents : the Agent-Phi Approach ». In: *International Journal of Grid Computing and Multi-Agent Systems* 1.1 (Jan. 2010), pp. 51–71 (cit. on p. xxi).

Loiret, Frédéric, Romain Rouvoy, Lionel Seinturier, and Philippe Merle (2011). « Software Engineering of Component-Based Systems-of-Systems: A Reference Framework ». In: *14th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'11)*. Ed. by Springer. June 2011, pp. 61–65 (cit. on p. 129).

Louloudi, Athanasia and Franziska Klügl (2011). « Towards a Generic Connection Between Agent Behaviour and Visualisation ». In: *Proceedings of the 9th European Workshop on Multi-Agent Systems (EUMAS'11)*. Maastricht University (cit. on p. 37).

Magee, J. and J. Kramer (1996). « Dynamic structure in software architectures ». In: *ACM SIGSOFT Software Engineering Notes*. Vol. 21. 6. ACM, pp. 3–14. DOI: 10.1145/250707.239104 (cit. on p. 122).

Mariachiara, Puviani, Massimo Cossentino, Giacomo Cabri, and Ambra Molesini (2010). « Building an agent methodology from fragments: the MEnSA experience ». In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. ACM, pp. 920–927 (cit. on pp. 28, 38).

Markiewicz, Marcus Eduardo and Carlos J. P. de Lucena (2001). « Object Oriented Framework Development ». In: *Crossroads* 7 (4 July 2001), pp. 3–9. DOI: http://doi.acm.org/10.1145/372765.372771 (cit. on pp. 4, 121).

Medvidovic, N. and R.N. Taylor (2000). « A classification and comparison framework for software architecture description languages ». In: *Software Engineering, IEEE Transactions on* 26.1 (Jan. 2000), pp. 70–93 (cit. on pp. 6, 55).

Molesini, Ambra, Enrico Denti, and Andrea Omicini (2007). « From AOSE Methodologies to MAS Infrastructures: The SODA Case Study ». In: *8th International Workshop "Engineering Societies in the Agents World" (ESAW'07)*. Ed. by Alexander Artikis, Gregory O'Hare, Kostas Stathis, and George Vouros. Workshop Notes. Athens, Greece: NCSR "Demokritos", Oct. 2007, pp. 283–298 (cit. on pp. 28, 38).

Molesini, Ambra, Andrea Omicini, Enrico Denti, and Alessandro Ricci (2006). « SODA: A Roadmap to Artefacts ». In: *Engineering Societies in the Agents World VI*. Ed. by Oguz Dikenelli, Marie-Pierre Gleizes, and Alessandro Ricci. Vol. 3963. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 49–62 (cit. on p. 168).

Monroe, R.T., A. Kompanek, R. Melton, and D. Garlan (1997). « Architectural styles, design patterns, and objects ». In: *Software, IEEE* 14.1, pp. 43–52. DOI: 10.1109/52.566427 (cit. on pp. 8, 123, 124).

Morin, A., J. Urban, P. D. Adams, I. Foster, A. Sali, D. Baker, and P. Sliz (2012). « Shining Light into Black Boxes ». In: *Science* 336.6078 (Apr. 13, 2012), pp. 159–160. DOI: 10.1126/science.1218263 (cit. on p. 115).

Nunes, Ingrid, Carlos de Lucena, Uirá Kulesza, and Camila Nunes (2011). « On the Development of Multi-agent Systems Product Lines: A Domain Engineering Process ». In: *Agent-Oriented Software Engineering X*. Ed. by Marie-Pierre Gleizes and Jorge Gomez-Sanz. Vol. 6038. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 125–139 (cit. on pp. 24, 38).

Nuseibeh, B. (2001). « Weaving together requirements and architectures ». In: *Computer* 34.3 (Mar. 2001), pp. 115 –119. DOI: 10.1109/2.910904 (cit. on p. 7).

Oechslein, Christoph, Franziska Klügl, Rainer Herrler, and Frank Puppe (2001). « UML for Behavior-Oriented Multi-Agent Simulations ». In: *in Multi-Agent Systems LNAI; Second International Workshop of Central and Eastern Europe on MultiAgent Systems, CEEMAS 2001*, pp. 26–29 (cit. on p. 168).

Oluyomi, A. (2006). « Protocols and patterns for agent-oriented software development ». PhD thesis. University of Melbourne (cit. on pp. 25, 35, 38).

Oluyomi, Ayodele, Shanika Karunasekera, and Leon Sterling (2007). « A comprehensive view of agent-oriented patterns ». In: *Autonomous Agents and Multi-Agent Systems* 15 (3 2007), pp. 337–377 (cit. on pp. 25, 35, 38).

Papazoglou, Michael P., Paolo Traverso, Schahram Dustdar, and Frank Leymann (2007). « Service-Oriented Computing: State of the Art and Research Challenges ». In: *Computer* 40.11, pp. 38–45. DOI: 10.1109/MC.2007.400 (cit. on p. 136).

Pavón, Juan, Jorge J. Gómez-Sanz, and Rubén Fuentes (2005). « The INGENIAS Methodology and Tools ». In: *Agent-Oriented Methodologies*. Ed. by Brian Henderson-Sellers and Paolo Giorgini. Idea Group Publishing, pp. 236–276 (cit. on pp. 28, 38, 169).

Pavón, Juan, Jorge J. Gómez-Sanz, and Rubén Fuentes (2006). « Model Driven Development of Multi-Agent Systems ». In: *Model Driven Architecture – Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Vol. 4066. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 284–298 (cit. on pp. 28, 38).

Peña, Joaquin, Michael G. Hinchey, Manuel Resinas, Roy Sterritt, and James L. Rash (2007). « Designing and managing evolving systems using a MAS product line approach ». In: *Sci. Comput. Program.* 66 (1 Apr. 2007), pp. 71–86 (cit. on pp. 24, 38).

Pereira, Jorge Verissimo (2009). « The new supply chain's frontier: Information management ». In: *International Journal of Information Management* 29.5, pp. 372–379. DOI: 10.1016/j.ijinfomgt.2009.02.001 (cit. on p. 136).

Perry, Dewayne and Alexander L. Wolf (1992). « Foundations for the Study of Software Architecture ». In: *ACM SIGSOFT Software Engineering Notes* 17, pp. 40–52 (cit. on p. 5).

Pokahr, Alexander, Lars Braubach, and Winfried Lamersdorf (2005). « Jadex: A BDI Reasoning Engine ». In: *Multi-Agent Programming: Languages, Platforms and Applications*. Ed. by Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. Vol. 15. Multiagent Systems, Artificial Societies and Simulated Organizations. Springer, pp. 149–174 (cit. on p. 170).

Poslad, Stefan and Patricia Charlton (2006). « Standardizing Agent Interoperability: The FIPA Approach ». In: *Multi-Agent Systems and Applications*. Ed. by Michael Luck, Vladimír Marík, Olga Štepánková, and Robert Trappl. Vol. 2086. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 98–117 (cit. on p. 169).

Rao, Anand (1996). « AgentSpeak(L): BDI agents speak out in a logical computable language ». In: *Agents Breaking Away*. Ed. by Walter Van de Velde and John Perram. Vol. 1038. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 42–55 (cit. on p. 170).

Reed, P (2002). « Reference Architecture: The best of best practices ». In: *The Rational Edge* (cit. on p. 9).

Ricci, Alessandro, Viroli Mirko, and Piancastelli Giulio (2011). « simpA: An agent-oriented approach for programming concurrent applications on top of Java ». In: *Science of Computer Programming* 76.1 (Jan. 2011). Ed. by Pascal Poizat Mirko Viroli Carlos Canal, pp. 37–62 (cit. on p. 171).

Ricci, Alessandro, Mirko Viroli, and Andrea Omicini (2007). « CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS ». In: *Environments for Multi-Agent Systems III*. Ed. by Danny Weyns, H. Parunak, and Fabien Michel. Vol. 4389. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 67–86 (cit. on pp. 36, 171).

Ricci, Alessandro, Mirko Viroli, and Andrea Omicini (2008). « The A&A Programming Model and Technology for Developing Agent Environments in MAS ». In: *Programming Multi-Agent Systems*. Ed. by Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff. Vol. 4908. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 89–106 (cit. on p. 168).

Ricordel, Pierre-Michel and Yves Demazeau (2002). « Volcano, a Vowels-Oriented Multi-agent Platform ». In: *From Theory to Practice in Multi-Agent Systems*. Ed. by Barbara Dunin-Keplicz and Edward Nawarecki. Vol. 2296. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 742–742 (cit. on pp. 36, 38).

Rougemaille, Sylvain, Jean-Paul Arcangeli, Marie-Pierre Gleizes, and Frédéric Migeon (2009). « ADELFE Design, AMAS-ML in Action: A Case Study ». In: *Post-Proceedings of the International Workshop on Engineering Societies in the Agents World (ESAW 2008)*. Vol. 5485. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence). Springer, pp. 97–112 (cit. on pp. 28, 38, 50).

Routier, Jean-Christophe, Philippe Mathieu, and Yann Secq (2001). « Dynamic Skills Learning: A Support to Agent Evolution ». In: *Proceedings of the Artificial Intelligence and the Simulation of Behaviour, Symposium on Adaptive Agents and Multi-agent systems (AISB'01)*. Ed. by Daniel Kudenko and Eduardo Alonso (cit. on pp. 36, 38, 172).

Schelfthout, K., T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns (2002). « Agent Implementation Patterns ». In: *Proceedings of the Workshop on Agent-Oriented Methodologies, 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pp. 119–130 (cit. on pp. 35, 37, 38).

Sellami, Zied and Valérie Camps (2012). « DYNAMO-MAS: A Multi-Agent System for Building and Evolving Ontologies from Texts (short paper) ». In: *International Conference on Practical Applications of Agents and Multiagent Systems (PAAMS), Salamanca, 28/03/2012-30/03/2012*. Vol. 155. 4240. Springer, pp. 283–286 (cit. on p. 113).

Shoham, Y. (1993). « Agent-oriented programming ». In: *Artificial intelligence* 60.1, pp. 51–92 (cit. on p. 25).

Silva, C., P. Tedesco, J. Castro, and R. Pinto (2004). « Comparing agent-oriented methodologies using NFR approach ». In: *IEE Seminar Digests* 2004.916, pp. 1–9 (cit. on pp. 22, 38).

Sommerville, I., D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. McDermid, and R. Paige (2011). « Large-scale Complex IT Systems ». In: *ArXiv e-prints* (Sept. 2011). arXiv:1109.3444 [cs.SE] (cit. on p. 135).

Stratulat, Tiberiu, Jacques Ferber, and John Tranier (2009). « MASQ: towards an integral approach to interaction ». In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*. Ed. by Carles Sierra, Cristiano Castelfranchi, Keith Decker, and Jaime Sichman. AAMAS '09. International Foundation for Autonomous Agents and Multiagent Systems, pp. 813–820 (cit. on p. 168).

Sudeikat, Jan, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf (2005). « Evaluation of Agent-Oriented Software Methodologies – Examination of the Gap Between Modeling and Platform ». In: *Agent-Oriented Software Engineering V*. Ed. by James Odell, Paolo Giorgini, and Jörg Müller. Vol. 3382. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 126–141 (cit. on pp. 28, 38).

Sykes, Daniel, Jeff Magee, and Jeff Kramer (2011). « FlashMob: distributed adaptive self-assembly ». In: *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*. Ed. by Holger Giese and Betty H. C. Cheng. ACM, pp. 100–109 (cit. on p. 135).

Szyperski, C., D. Gruntz, and S. Murer (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press (cit. on pp. 5, 6).

Taillandier, Patrick, Alexis Drogoul, Duc-An Vo, and Edouard Amouroux (2010). « GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control ». In: *The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)*. India (cit. on p. 172).

Trencansky, Ivan and Radovan Cervenka (2005). « Agent Modeling Language (AML): A Comprehensive Approach to Modeling MAS ». In: *Informatica* 29, pp. 391–400 (cit. on p. 169).

Van Roy, Peter (2007). « Self Management and the Future of Software Design ». In: *Electronic Notes in Theoretical Computer Science* 182. Proceedings of the Third International Workshop

on Formal Aspects of Component Software (FACS 2006), pp. 201 –217. DOI: 10.1016/j.entcs.2006.12.043 (cit. on p. 135).

Van Roy, Peter (2009). « Programming Paradigms for Dummies: What Every Programmer Should Know ». In: *New Computational Paradigms for Computer Music*. Ed. by Gérard Assayag and Andrew Gerzso. France: IRCAM/Delatour (cit. on p. 3).

Van Roy, Peter and Seif Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Mar. 2004 (cit. on pp. 119, 127).

Viroli, Mirko, Tom Holvoet, Alessandro Ricci, Kurt Schelfthout, and Franco Zambonelli (2007). « Infrastructures for the environment of multiagent systems ». In: *Autonomous Agents and Multi-Agent Systems* 14 (1 2007), pp. 49–60 (cit. on pp. 25, 35, 38).

Weiser, Mark (1991). « The Computer for the 21st Century ». In: *Scientific American* 256.3, pp. 66–75 (cit. on p. 145).

Weyns, Danny (2010). *Architecture-Based Design of Multi-Agent Systems*. 1st. Springer Publishing Company, Incorporated (cit. on pp. 9, 23, 35, 38).

Weyns, Danny, Robrecht Haesevoets, and Alexander Helleboogh (2010). « The MACODO organization model for context-driven dynamic agent organizations ». In: *ACM Trans. Auton. Adapt. Syst.* 5 (4 Nov. 2010), 16:1–16:29 (cit. on pp. 139, 167).

Weyns, Danny, Alexander Helleboogh, Elke Steegmans, Tom De Wolf, Koenraad Mertens, Nelis Boucké, and Tom Holvoet (2004). « Agents are not part of the problem, agents can solve the problem ». In: *Proceedings of the OOPSLA Workshop on Agent-Oriented Methodologies*, pp. 101–102 (cit. on pp. 9, 30, 36).

Weyns, Danny and Tom Holvoet, eds. (2006). *Multiagent Systems and Software Architecture (MASSA)* (cit. on pp. 19, 23, 28).

Weyns, Danny, Andrea Omicini, and James Odell (2006). « Environment as a first class abstraction in multiagent systems ». In: *Autonomous Agents and Multi-Agent Systems* 14.1, pp. 5–30. DOI: 10.1007/s10458-006-0012-0 (cit. on pp. 25, 35, 38).

Wilensky, Uri (1999). *NetLogo itself*. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL. (cit. on p. 172).

Zambonelli, Franco, Nicholas R. Jennings, and Michael Wooldridge (2003). « Developing multiagent systems: The Gaia methodology ». In: *ACM Trans. Softw. Eng. Methodol.* 12 (3 July 2003), pp. 317–370 (cit. on p. 169).

Zyda, Michael (2005). « From visual simulation to virtual reality to games ». In: *Computer* 38.9 (Sept. 2005), pp. 25 –32. DOI: 10.1109/MC.2005.297 (cit. on p. 91).

Victor Noël

Component-based Software Architectures and Multi-Agent Systems: Mutual and Complementary Contributions for Supporting Software Development

Thesis Supervisor:
Marie-Pierre Gleizes — Professor, University of Toulouse, France
Jean-Paul Arcangeli — Maître de Conférences, HDR, University of Toulouse, France

PhD defended June 15, 2012 at IRIT - Université Paul Sabatier

Abstract

In this thesis, we explore the various aspects of the mutual and complementary contributions that multi-agent systems (MASs) and component-based software architectures (CBSAs) can provide to each other. On one hand, we define, illustrate, analyse and discuss an architecture-oriented methodology of MAS development, a component model (SpeAD), an architectural description language (SpeADL) and a design method (SpEArAF) that ease and guide the description and the implementation of MASs. This complete answer to MAS development is supported by a tool (MAY) and has been applied to many applications. On the other hand, we explore through various experiments how self-adaptive MASs can be used to support CBSAs. The agents and their continuous reorganisation act both as the engine of the construction and of the dynamic adaptation of the architecture, and as the runtime container that practically connects its elements together.

Keywords:

Discipline: Informatique

Institut de Recherche en Informatique de Toulouse — UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31 062 Toulouse Cedex 4, France

Victor NOËL

Architectures logicielles à base de composants et systèmes multi-agents : contributions mutuelles et complémentaires pour supporter le développement logiciel

Directeurs de thèse :
Marie-Pierre Gleizes — Professeur, Université de Toulouse, France
Jean-Paul Arcangeli — Maître de Conférences, HDR, Université de Toulouse, France

Thèse soutenue le 15 juin 2012 à l'IRIT — Université Paul Sabatier

Résumé

Dans cette thèse, nous explorons les diverses contributions que les systèmes multi-agents (SMA) et les architectures à base de composants (CBSA) peuvent mutuellement et complémentairement s'apporter l'un à l'autre. Dans un premier temps, nous définissons, illustrons, analysons et discutons une méthodologie du développement des SMA, un modèle de composants (SpeAD), un langage de description d'architecture (SpeADL) et une méthode de conception (SpEArAF) qui facilitent et guident la description et l'implémentation des SMA. Cette réponse complète au développement des SMA est assistée par un outil (MAY) et a été appliquée à un grand nombre d'applications. Dans un second temps, nous explorons à travers divers expériences l'aide que peuvent apporter les SMA auto-adaptatif aux CBSA. Les agents et leur réorganisation continuelle jouent à la fois le rôle de moteur de la construction et de l'adaptation dynamique de l'architecture, mais aussi du conteneur qui connecte ses éléments en pratique.

Mots-clés :

Discipline : Informatique

Institut de Recherche en Informatique de Toulouse — UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31 062 Toulouse Cedex 4