UNIVERSITÀ DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

XXX Ciclo

An Agent-Oriented Programming Language for JADE Multi-Agent Systems

Coordinatore:

Chiar.mo Prof. Marco Locatelli

Tutor:

Chiar.mo Prof. Agostino Poggi

Co-Tutor:

Chiar.mo Prof. Federico Bergenti

Dott.ssa Stefania Monica

Dottorando: *Eleonora Iotti*

Anni 2014/2017

Sarà forse un'assurda battaglia ma ignorare non puoi
che l'Assurdo ci sfida per spingerci ad essere fieri di noi
– Francesco Guccini

Don’t panic
– Douglas Adams

Abstract

In this dissertation, a novel approach to program *JADE (Java Agent DEvelopment Framework)* agents and *Multi-Agent Systems (MASs)* is proposed. JADE is a well-known agent platform which provides support to consolidated agent technologies, an agent-oriented distributed architecture, and several tools, both textual and graphical, for the creation of agents and MASs. The work presented in this dissertation originates from the need to assist programmers by means of tools that reduce the complexity and speed up the building of JADE MASs. The need for a simpler and more intuitive way to program JADE MASs in all their parts, from agents and behaviours to ontologies and interaction protocols, has led to the construction of a high level *Domain-Specific Language (DSL)*, explicitly tailored for JADE agent and MASs. The proposed agent-oriented DSL is called *JADEL*, which stands for *JADE Language*. The language is designed to support the effective implementation of JADE MASs in the scope of model-driven development. JADEL provides a simple and clean syntax to describe agents and message passing among agents, behaviours and sub-behaviours, and JADE ontologies with their propositions, concepts and predicates. Moreover, it allows to integrate JADE-native agents, behaviours, and ontologies within JADEL code, which is translated into Java code. One of the main results of the proposed approach is that it permits to clearly put agent-oriented programming methodologies above the object-oriented programming paradigm of JADE programs, decoupling the agent meta-model from the Java meta-model.

Contents

List of Figures

List of Tables

List of Publications

2 Bergenti, F., Iotti, E., and Poggi, A. (2015c). An outline of the use of transition systems to formalize JADE agents and multi-agent systems. *Intelligenza Artificiale*, 9(2):149–161. 28

2 Bergenti, F., Iotti, E., and Poggi, A. (2015b). Outline of a formalization of JADE multi-agents system. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2015)*, volume 1382 of *CEUR Workshop Proceedings*, pages 123–128. 28

3 Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2017a). Agent-oriented model-driven development for JADE with the JADEL programming language. *Computer Languages, Systems & Structures* . . 86

3 Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016b). Interaction protocols in the JADEL programming language. In *Procs. 6^{th} Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*. 86

3 Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016a). A case study of the JADEL programming language. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2016)*, volume 1664 of *CEUR Workshop Proceedings*, pages 85–90. 86

3 Bergenti, F., Iotti, E., and Poggi, A. (2016d). Core features of an agent-oriented domain-specific language for JADE agents. In *Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection*, pages 213–224. Springer International Publishing. 86

Introduction

In this thesis, a novel *agent-oriented programming language* is presented. The language is called JADEL, which stands for *JADE Language*, since it is based on the *Java Agent DEvelopment Framework (JADE)*. In the early design phase, *three* main requirements to meet were identified. In detail,

1. JADEL must follow the *Agent-Oriented Programming (AOP)* paradigm;
2. JADEL must rely on the JADE platform; and
3. JADEL must be described in terms of a *Domain-Specific Language (DSL)*.

Such requirements are determined by a few general considerations, as follows:

1. AOP is a promising programming paradigm that permits developers to think in terms of agents, and that it takes advantage of consolidated agent technologies;
2. JADE is currently one of the most complete, solid and popular agent platform;
3. DSLs allow developers to manage a certain domain, e.g., the JADE agents domain, in a efficient and effective way, to the extent that they provide lighter syntaxes and specific constructs, which are tailored for the target domain; and
4. DSLs support *Model-Driven Development (MDD)*.

Those considerations are far from being sufficient motivations for the development of a novel language from scratch. As a matter of fact, designing and implementing a programming language, yet also a DSL, is a difficult task that requires a substantial effort.

First, from the very early phases of language development, its meta-model and structure must be clear, from main entities to little parts such as single expressions. This implies that the language developer has to understand deeply the domain that the language aims at supporting, and he/she must also delineate precisely target users, whose can be programmers or not (in case of graphical modeling languages, for example). A formalization of the target domain could be useful. There are lots of different approaches that can be suitable to model the domain, which span from informal definitions of main concepts involved, such as use-case or *Unified Modeling Language (UML)* diagrams, to more formal specifications, such as transformation functions or transition systems.

Second, a language developer is demanded to master a high number of implementation details that derive from the use of different technologies, such as lexer and parser generators, code translators and compilers. Moreover, modern languages often came together with other tools such as editors or, better, *Integrated Development Environments (IDEs)*. Hence, developing a language means not only producing a compiler or an interpreter for a newly-defined grammar, but it also means constructing the environment for the language development.

Other problems go beyond the design and implementation phases. For example, maintenance costs have to be correctly evaluated, and a maintenance plan has to be assessed to ensure usability over time. Another important part is the estimation of language capabilities and its actual advantages. Methods for calculating the soundness of a language depend on the type of language, its purposes and motivations, its target domain, and so on. Several metrics have been proposed and used, but they are very related to the language for which they are defined. Only few of these metrics are suitable for almost every language, such as the count of *Lines of Code (LOCs)*. Thus, estimating the impact of the language, in terms of usability and readability, as well as requirements matching and maintenance costs, is often an important exercise.

JADEL responds to some issues which involve JADE development, such as the reduction of the complexity of the framework for new users, and the renovation of the JADE user experience for others.

In the following of this introduction, motivations are largely explained. The ad-

vantages in incorporate AOP in the development process of an application are highlighted, and the improvement that a DSL can provide is discussed. The importance of JADE as leading technology in the agent context is clarified, as well as its limitations which made necessary an enhancement.

Agents and Multi-Agent Systems

AOP is a programming paradigm which puts *agents* as basic building blocks for software development. Agents in *Artificial Intelligence (AI)* are entities which are theoretically capable to perform intelligent actions, i.e., their aim is *to act* in order to achieve a rational behaviour. As a matter of fact, the term *agent* derives from the latin word *agere*, which means to act, or, more generically, to do.

According to a classic subdivision of AI definitions [Russell and Norvig, 2002], there are four distinct categories that describe different purposes of AI. Such categories are obtained by intersecting two axes: the first dimension is divided into *thinking* and *acting*, and the second spaces from *human* to *rational* behaviours. Hence, the four categories are *(i)* thinking humanly; *(ii)* thinking rationally; *(iii)* acting humanly; and *(iv)* acting rationally. Only the last category is covered in this dissertation: as a matter of fact, agents falls in that category.

Due to the inherent complexity of the problem of achieving a rational behaviour through actions, there is not a unique standard definition of agent. On the contrary, many kinds of agents were studied and developed by researchers and practitioners in the field of AI.

As a first definition, *Intelligent Agents (IAs)* are presented. IAs are kind of autonomous entities, which have actuators and sensors, and that affect a certain environment of which they have partial or complete knowledge. Such agents can be computer programs, or they can be physically embedded in their environment.

Sometimes, IAs are also called *Rational Agents (RAs)*, but there are actually few differences between IAs and RAs. In fact, RAs are concept used mainly in AI or game theory to model anything that has goal-oriented behaviours and that can make decisions in order to maximize its utility function. This introduces another concept besides sensors, actuators, and environment: a measure of performance. Also rational

agents can be software programs, but they can also be humans or any other being.

Finally, in the field of Computer Science, there are *Software Agents (SAs)*, called here agents when no ambiguity can arise. Such agents are computer programs which must have certain features to guarantee a high degree of autonomy, reactivity and persistence. Sometimes they are known as *bots*, from *robots*. Examples of software agents are personal assistants, chat bots, Unix-like system daemons, and so on.

Mentioned definitions are generic, and sometimes they overlap. A notable improvement to the concept of agent is the introduction of communicative capabilities, which allow the agent to cooperate or coordinate with other agents. A group of agents defines a distributed system, whose basic units are in some measure autonomous or intelligent, and that exchange messages to each others and collaborate towards common goals. Such distributed systems, which are composed of agents and aim at performing rational actions for achieving a shared objective, are called *Multi-Agent Systems (MASs)*. MASs are widely employed for studying *Distributed Artificial Intelligence (DAI)* problems.

In AOP, agents capabilities are specified with the aid of a dedicated language or infrastructure, such as a framework or a graphical interface. The AOP paradigm is inherently different from the *Object-Oriented Programming (OOP)* one. As a matter of fact, agents are characterized by mental states and they exchange specific types of messages, responding to them in a truthful and consistent way. To this extent, agents are viewed as specialization of objects. Viewing agents as specializations or extensions of objects is somehow intuitive, and points out the active capabilities of agents. Nevertheless, agents and objects abstractions are completely different. As stated in [Baldoni et al., 2016], the behavioral dimension is not sufficient to compare agents and objects, lacking the concepts of *goals* and *environment*. For this reason, the management of agents and MAS often requires the use of specific languages, which focus on agent-oriented abstractions and provide particular constructs and structures.

There are many *Agent Programming Languages (APLs)*, some imperatives, others declarative languages, that follow different *agent models* or meta-models, and that are useful for different purposes. Also, a long list of AOP frameworks, called *Agent Platforms (APs)*, were developed in order to respond to diverse necessity and to target

various applications. A comprehensive but not exhaustive list of APLs and agent platforms is shown in Chapter 1, where different solutions are discussed, according to some classifications and their main characteristics.

The JADE Platform

JADE is a development framework which provides a Java *Application Program Interface (API)*, and graphical tools for building and running complex MASs. JADE first release is dated in the early 2000s, but nowadays JADE is still one of the most popular agent platforms, both in industrial and in academic environments [Kravari and Bassiliades, 2015]. Many factors contributed to its durable success.

First, a critical role was played by its compliance to *FIPA* (*Foundation for Intelligent Physical Agents*)[1] specifications. FIPA is a body for the standardization of software agents which are heterogeneous and that interacts among each others. As a matter of fact, FIPA specifies standards for *Agent Communication Languages (ACLs)*, i.e., the languages which agents use in their interactions, and defines a number of *Interaction Protocols (IPs)* that provide patterns for message passing among agents.

Second, JADE is fully written in Java. This is not surprising, since in the years of JADE launching, Java was a novel and promising technology, and programmers wanted to use it for their applications. It was common opinion that it would have changed radically the way software was built, and it was the technology that marked the growth of the Web. Thus, as a core design decision, JADE completely relies on Java. Just a few projects, e.g., the .NET porting bundled with the source code distribution of JADE, tried to loose the tie that couples JADE and Java.

Furthermore, JADE is currently actively developed, and there are some satellite projects that have been advanced in the meanwhile. The growth of the framework and its extension to other application domains increase and strengthen JADE reliability as software tool for professional use. Just to cite a notable industrial example, JADE has been in daily use for service provision and management in Telecom Italia for more than six years, serving millions of customers in one of the largest and most

[1] `www.fipa.org`

penetrating broadband networks in Europe [Bergenti et al., 2015a].

These factors made JADE success, but there are also important drawbacks. FIPA specifications are already in use, and are still important in MAS development, but *(i)* other standardizations, not necessarily tied with agent-oriented development, are now acquiring significance in this field, mainly because of the request of interoperability among different systems; and *(ii)* JADE is currently not the only agent platform which complies with FIPA standards, and many different proposals come up to be valid alternatives.

The fact that JADE is fully written in Java is somehow a limitation. As a matter of fact, nowadays, the 100% *pure Java* approach of the beginnings becomes less appealing to developers mostly because *(i)* a number of valid alternatives are becoming popular; *(ii)* the minimalistic syntax of Java is perceived as a limitation; and *(iii)* the number of researchers and practitioners advocating the use of DSLs is quickly growing.

Finally, JADE has already more than 15 years of use and it is currently expanding. This rapid growth had increased JADE complexity and had lowered its compliance with AOP. In fact, a good JADE developer is required to master a high number of implementation details, in addition to his/her familiarity with the basics of MAS programming, following the AOP paradigm. Thus, the chances of making mistakes, both in the technical implementation and in the agent design, increase considerably. In addition, the constant growing of JADE in terms of features made it difficult for new developers and students. These difficulties are due to the fact that JADE significantly increased its complexity and now it has a steep learning curve. The main problems in developing agent-based applications and MASs in JADE can be listed as follows.

1. The complexity of the framework requires not only expertise in management of distributed MAS but also a deep understanding of JADE mechanisms;

2. The lack of a fixed agent model sometimes causes unclear or incorrect utilizations of the available agent technologies; and

3. Some procedures and patterns become repetitive or verbose, hence not clear in terms of AOP, due to the gap between the AOP and OOP paradigms.

In summary, factors that made JADE success, namely FIPA compliance, Java as unique programming language for the framework, and the rise of related projects as well as newly-developed features, are now weaker and cause of some problems. Stating the JADE importance in the agent-oriented context, its is not desirable that such problems remain unsolved. On the contrary, there is the necessity of a solution that preserves the stability and reliability of JADE, enhancing at the same time its compliance with AOP, and, hopefully, simplifying the way JADE MASs are build. This dissertation focus on the search for such a solution.

The Role of DSLs in Software Development

DSLs are programming languages that refer to a specific application domain, in contrast to *General Purpose Languages (GPLs)* that are meant for any use case. The definition of what is a *specific domain* and what is not is unclear, and it is often a matter of taste and degree of specialization.

The majority of the modern mainstream languages are GPLs, such as Java, C++, Python, and so on, but there are also very popular languages that are usually classified as DSLs, such as SQL, HTML, EBNF for grammars. Actually, some of these DSLs are not proper programming languages, and they are referred to as markup languages, specification languages or modeling languages. A natural way to delineate the *specificity* of a language with respect to a domain is to put it on a range from very specialized languages, such as EBNF, to very general ones, such as C++.

The number of expressions and constructs the language has, or the number of applications for which it is suitable are not always a good measure for defining DSLs. A DSL can be very little, a small language which does not include many of the features of GPLs, and that is used by a single application. Such DSLs are called *mini-languages*. Other DSLs are meant to be easily integrated with a host language, which is typically a GPL, simply adding to it specific expressions, in a way similar to an application library.

The wide range of technologies and tools involved in the development of a DSL is brightly discussed in [Mernik et al., 2005; Oliveira et al., 2009], where DSLs are clearly marked as important tools to support MDD. Deciding in favor of a DSL in-

stead of an API is not trivial and it depends on many factors. In fact, the API of an application library is written in a certain language, usually a GPL, and it consists in a set of methods or procedures that provide the needed building blocks to make easier the development of a program, by abstracting the underlying implementation. Thus, a GPL in combination with a suitable API behaves as a DSL. The difference is that a DSL can go beyond the notation offered by a GPL, and that a mapping between domain-specific abstractions and library features is not always simple. Briefly, GPLs user-defined operators and functions are often not so direct as DSLs specific expressions and constructs.

The use of a DSL for a specific application domain leads to important benefits. As a matter of fact, the syntax of a DSL is tailored on the specific domain that it describes, with the aid of user-friendly notations that are simpler than the respective general-purpose ones. This facilitates code understanding, and it allows many repetitive and tedious activities to be automated. Moreover, hosting a DSLs into another language ensures applicability and reusability of domain-specific code in real-world scenarios, where interoperability with existing code is essential. On the other hand, APIs are in most cases cheap solution, in terms of development and maintenance of the API itself.

A significant body of research in *software engineering* includes important studies on DSLs and *Domain Specific Modeling Languages (DSMLs)*. Such an approach is part of the so called *Model-Driven Engineering (MDE)*, which aims at driving the whole development process in high-level modeling of entities. In fact, MDE proposes the systematic use of *models* as primary engineering artifacts. Meta-modeling and model transformations are basic approaches in MDE for ensuring interoperability, portability, and reusability of software.

Language Semantics and Formalizations

Programming languages, both DSLs and GPLs, are fundamental tools for every developers. Hence, it is extremely important to assess their capabilities, potential, and semantics, in an almost objective way. Such a task is feasible for programming languages because they do not have the complexity of a natural language, neither they

contain the typical ambiguities of spoken languages. On the contrary, programming languages have precise rules, and they are limited in their expressive power [Felleisen, 1991] and computational power [Turing, 1937].

The art of studying formal languages is the *Mathematical Logic* discipline. In mathematical logic, formal languages are studied by means of a meta-language, which puts a layer between the mathematical model and the actual subject of the study. Such a meta-language is used to define the morphology, the syntax, and the semantics of a formal language. In the programming languages case, a specialization of such a discipline is used. Mathematical models were developed specifically for studying programming languages, and opened a new field of research. In detail, a programming language is defined by means of syntactic categories, that describe the way expressions are build. In other words, syntactic categories specify *how* to write in that language, i.e., which words and phrases are allowed.

Given the way programs have to be written, their meaning is obtained by imposing a semantics. In a classic book on the semantics of programming language, [Winskel, 1993], three different types of semantics are defined, namely, the *operational*, the *denotational*, and the *axiomatic* semantics. All three semantics have different usage and purposes, but it can be shown that they are equivalent. An almost complete reference for the *Structural Operational Semantics (SOS)* is [Plotkin, 2004]. In particular, Plotkin defines *Labeled Transition System (LTS)* for automata, and then it applies the same approach to programming languages.

The operational semantics of a programming language is based on its syntax categories. Given a correct expression, the transition system shows how the expression is evaluated, under which circumstances, and what is the result of the evaluation on the environment. Such a detailed description of the meaning of an expression is useful to avoid mistakes, and to validate the correctness of a program in doing something. Especially for critical software, validating and verifying the semantics of a program or a code fragment is a crucial phase of software development. In general, a formal semantics of a language permits reasoning on language properties, such as type safeness. This is the reason for providing a new language such a formal semantics. GPLs often do not have formal specifications of their semantics, due to their complexity.

Giving a formal semantics for Java-like languages is yet a challenging research problem. A survey on the state of the art in Java formalization can be found in [Alves-Foss, 1999] and in [Joy et al., 1998]. Notably, a formalization of events is provided in [Cenciarelli et al., 1999]. DSLs are often smaller and simpler than GPLs, and giving them a formalization is easier. For this reason, in developing a novel DSLs, a detailed mathematical description of its semantics is expected.

Scope and Structure of the Dissertation

In summary, a number of problems that involve JADE were identified, in particular its compliance with AOP and MAS development, and a possible solution consists in the use of a DSL that can put agent-oriented features of JADE on a higher level of abstraction, in the scope of MDD.

Each of those problems have been analyzed and the proposed solution aims at integrating and enhancing JADE, as well as simplifying the use of the framework.

1. The expertise and knowledge of JADE mechanisms is confined in a DSL that provides shortcuts and simple constructs for expressing the most complicated procedures of the framework;

2. The agent model is given by a selection of abstractions of the DSL, which forces the major methodology; and

3. The AOP features and main patterns of communications are enclosed in a limited number of specific expressions that put the agent-oriented concepts in a higher level than object-oriented ones.

The approach used in the preparation and development of JADEL can be summarized as follows. An in-depth study of the state of the art was made, in order to correctly pose JADEL into the wide variety of AOP technologies. In Chapter 1, some of the most popular agent-oriented proposals are described, by listing the main features of each methodology, platform, and language. Such an enumeration does not aim at providing a comprehensive or exhaustive list of AOP approaches. Producing

such a complete survey on AOP is a very difficult exercise, and it is out of the scope of this work. On the contrary, Chapter 1 equips the reader with a quick reference of the most successful AOP proposals. A brief conclusion compares the depicted agent-oriented technologies with the novel proposal discussed in this dissertation.

Then, a formalization of the major concepts of JADE is produced, that permits an analysis of JADE patterns and idioms. Such a work is presented in details in Chapter 2. The formalization is based on the core calculus *Featherweight Java (FJ)*, first presented in [Igarashi et al., 2001]. An extension of such a calculus is proposed, adding to it non-functional features, and using stores and LTS. A subset of important functionality of JADE APIs are selected, and the related patterns and methods are included in the proposed system. The system consists also of some sub-systems, that point out different aspects of JADE programs. For example, a dedicated system is devoted to the managing of events. A final discussion evaluates the outcome of the JADE formalization and delineates future developments and purposes.

In Chapter 3, JADEL syntax is defined, and an operational semantics of the language is provided. Such a semantics is proposed in the same fashion of JADE formalization, highliting differences between the newly-developed JADEL and the platform it was built on, JADE. As a matter of fact, the semantics of JADEL is very important to understand its capabilities and the improvement it give to the direct JADE programming. The next step consists of the actual implementation of the language, and many examples are developed in order to highlight all of its features.

Finally, for evaluating the work on JADEL, examples are used as case-studies and tested in terms of readability, simplicity, and performance. Chapter 4 is devoted to the quantitative and qualitative assessment of JADEL as a programming language in the field of agents and MAS.

A conclusion provides an overview of the present work, the achieved results and spaces for future improvements.

Chapter 1

Agent-Oriented Programming

First Law. A robot may not injure a human being or,
through inaction, allow a human being to come to harm.

– Isaac Asimov

The first rule that is required to meet by JADEL is the compliance with the *Agent-Oriented Programming (AOP)* paradigm. AOP is a programming paradigm first introduced in [Shoham, 1997], which identifies as core abstractions some autonomous and proactive entities, known as *agents*.

Over the years, several languages, frameworks, and platforms, as well as meta-models and programming methodologies, were studied and developed in order to coherently support AOP. The interest in such tools dates back to the introduction of agent technologies and, since then, it has grown rapidly. Nowadays, there are a high number of different alternatives that provide advanced features for the development of agents and *Multi-Agent Systems (MASs)*. A particular mention goes to *Agent Programming Languages (APLs)*, that turned out to be especially convenient to model and develop complex MASs, in contrast with traditional (lower-level) languages, that are often considered not suitable to effectively implement AOP. In this chapter, a survey on agent-oriented methodologies, platforms, and languages is presented, in order to precisely delineate the state of the art in AOP.

1.1 Agent-Oriented Software Engineering

Agent-Oriented Software Engineering (AOSE) is a branch of *software engineering*, which studies and disciplines the available approaches and methodologies involved in the development of agent-oriented applications, agents, and MASs [Bergenti et al., 2004]. A lot of different AOSE methodologies have been proposed to take advantage of AOP. In the scope of such methodologies, some meta-models for MAS were specified. Usually, such meta-models describe only the main elements of a MAS together with the relations among them, using models as primary artifacts, and their transformations as primary phases, in the respective methodologies.

A survey of the state of the art of *Model-Driven Development (MDD)* methodologies in AOSE is provided in [Kardas, 2013], together with an evaluation of described approaches. Model-driven techniques have already proved to be effective to deal with the complexity of MASs design, and many studies in the literature propose model-driven approaches, sometimes by introducing modeling languages based on the *Unified Modeling Language (UML)*. Notably, the work presented in this thesis can be considered as an another approach to *Model-Driven Engineering (MDE)*.

In the rest of this section, the most significant methodologies and meta-models available for AOSE are listed in chronological order and briefly described.

AALAADIN [Ferber and Gutknecht, 1998] is a meta-model based on three main organizational concepts, namely agents, groups, and roles. Agents can be members of groups, and they are atomic entities. Agents can also play a role in such groups. An extension of the AALAADIN meta-model states that an agent can be part of a group or another agent, thus subdividing agents into atomic and not atomic. Roles and groups aim at capturing system requirements, so a precise and concrete definition of the kind of roles and groups played in a MAS is the key for ensuring the effectiveness of the AALAADIN representation. In particular, it allows building MAS with different forms of organizations, such as market-like and hierarchical organizations, by describing coordination and negotiation schema.

Gaia [Wooldridge et al., 2000; Zambonelli et al., 2003] is a well-known methodology for agent-oriented analysis and design. As pointed out by the original authors, Wooldridge et al., the name Gaia comes from the Gaia hypothesis formulated by James Lovelock in the 1970s. Such an hypothesis states that all the living organisms on Earth can be considered as components of a single entity, which co-evolve with their environment and interact among each others for self-regulation. As a matter of fact, the idea of a number of heterogeneous components that cooperate together to achieve a common goal is relevant in the development of a MAS, and the Gaia methodology is founded on this idea. In Gaia, a MAS is viewed as a computational entity that consists of many components (agents) that play a number of diverse interacting roles.

AUML (*Agent UML*) [Bauer et al., 2001] is a set of UML idioms and extensions to the UML class diagrams, which allows the description of software agents in terms of both internal and external behaviours, and it also provides suitable artifacts to define *Interaction Protocols (IPs)*. This approach is closely related to the object-oriented software modeling, and extends it in order to help developers in the passage from object-oriented to agent-oriented technologies.

Tropos [Bresciani et al., 2004] is an AOP methodology, based on the notion of agents with mental capabilities, such as the ability to make plans in order to achieve their goals. The required mental notions follow an agent architecture known as *Belief, Desire, Intention (BDI)* paradigm. Moreover, Tropos covers the activities that precede the first phase of software requirements specification, by means of a early requirements analysis, in order to better understand the environment where agents operate and the kind of interactions among them.

Prometheus [Winikoff and Padgham, 2004] is a methodology for building agent based systems, which consists in three phases, namely system specification, architectural design, and detailed design. The scope of Prometheus is to be complete and detailed as well as easily understandable, as target users are both undergraduate stu-

dents and industrial practitioners that do not have a previous background in AOP. Thus, the methodology responds both at educational and industrial needs. Moreover, in [Gascueña et al., 2012], the MDE paradigm and some meta-modeling techniques of MDD are analyzed, using Prometheus for validating their proposal.

ADELFE A methodology for adaptive MAS is proposed in [Bernon et al., 2005], which is called *Atelier de Développement de Logiciels à Fonctionnalité Emergente (ADELFE)*. It uses both UML and AUML notations to specify three phases, namely the requirement, the analysis, and the design workflows. ADELFE is not general, as Gaia or Tropos, but it is meant for adaptive MAS, i.e., software used to manage open systems or unpredictable environment situations. For this reason, the requirement workflow, which can be found also in the Tropos methodology, is important for modeling the environment, while the analysis identifies the agents as entities of the system, and the design workflow the agent model.

INGENIAS Some approaches that adopt MDD in the scope of AOSE can be found in [Pavón et al., 2005, 2006; Fuentes-Fernández et al., 2010; Gómez-Sanz et al., 2010], where the *Engineering for Software Agents (INGENIAS)* methodology and related tools were discussed. INGENIAS is used to define MAS specifications as the main development artifacts, and it provides model-to-text tools to transform specifications into executable code and documentation. Agents have mental capabilities and they are goal-oriented, but they do not follow the BDI paradigm. Rather, they are characterized by mental states. Agents can also take roles and participate to interactions.

PASSI (*Process for Agent Societies Specification and Implementation*) [Cossentino, 2005] is a methodology for designing MAS step-by-step from the early requirement phases to the actual implementation, by means of UML notations. It proposes five models, namely system requirements, agent society, agent implementation, code, and deployment.

PIM4Agents A general-purpose meta-model called *Platform-Independent meta-model for Multi-Agent Systems (PIM4Agents)* is presented in [Hahn et al., 2009]. PIM4Agents is an enhancement of a *Platform-Independent Meta-Model (PIMM)* for MASs. In the PIM4Agents meta-model, modeling concepts are grouped into seven viewpoints, namely Multiagent, Agent, Behavioral, Organization, Role, Interaction, and Environment.

VigilAgent Prometheus and INGENIAS were used as basis for another model-driven technique called VigilAgent [Gascueña et al., 2014], which uses model-to-model and model-to-text transformation to ease interoperability between these two methodologies.

1.2 Agent-Oriented Frameworks and Platforms

Several tools have been provided over the years to support the effective construction of agents and MASs, in the scope of AOP, and taking advantage of AOSE methodologies. Many of these tools are software development frameworks, which offer *Application Program Interfaces (APIs)* for helping developers in building agent-oriented solutions for their applications. Such frameworks, which are specifically targeted for programmers, often co-exist with other technologies, such as visual interfaces that permit the management of agent-oriented concepts. It is not uncommon that such graphical interface are meant to be easy to use for agent-oriented domain experts, although they are not professional developers. A comprehensive tool for developing and running complex MASs, with the aid of both application libraries and complementary interfaces is called *Agent Platform (AP)*.

The rest of this section is devoted to the description of a selection of those frameworks and platforms, in alphabetical order. Only the most popular platforms are cited, but a complete survey can be found in [Kravari and Bassiliades, 2015].

INGENIAS Development Kit The INGENIAS methodology takes advantage of a tool for developing MAS in a model-driven approach. Such a tool is called *INGENIAS*

Development Kit (IDK) [Pavón et al., 2005] and it consists of a visual editor, produced by *INGEME* (*INGENIAS Meta-Editor*), and of the *INGENIAS Agent Framework (IAF)*. The visual editor permits to process MAS specifications into source code, HTML documents, and other related artifacts.

JaCaMo (*Jason, CArtAgO, Moise*)[1] [Boissier et al., 2013] is a multi-agent framework, which uses Jason (described later in this section) for programming autonomous agents, *Common Artifacts for Agents Open framework (CArtAgO)* [Ricci et al., 2006] for the environment artifacts[2], and Moise [Hübner et al., 2010] for MASs organizations[3]. It is a combination of these three different technologies, that were put together in order to obtain a comprehensive approach for developing complex MASs. The BDI agents of Jason cooperate inside a shared environment defined by CArtAgO artifacts, following a programming agent model called JaCa. Moise provides an organizational meta-model to describe groups and roles for structuring agents, a way to define scheme, missions, and goals, and a norm entity for constraining agent behaviours.

JACK Intelligent Agents [Howden et al., 2001; Winikoff, 2005] is a Java agent platform that brings the concept of *Intelligent Agent (IA)* into a mature commercial software. It uses the BDI paradigm and provides several different tools, such as the *JACK Plan Language (JPL)* and the *JACK Development Environment (JDE)*, which strengthen its relevance and its usability in industrial contexts. JACK is currently actively developed and maintained, and many extensions were released since its first launch. Notably, JACK was used as a solid base for testing and validating the Prometheus methodology. A model transformation which compile PIM4Agents models into source code for JACK is shown in [Hahn et al., 2009].

[1] `jacamo.sourceforge.net`
[2] `cartago.sourceforge.net`
[3] `moise.sourceforge.net`

JADE (*Java Agent DEvelopment Framework*)[4] [Bellifemine et al., 2005, 2007, 2001] is an agent platform and a software framework for building MAS by taking advantages of a wide and comprehensive API, entirely written in Java. Currently, JADE is one of the most popular *FIPA* (*Foundation for Intelligent Physical Agents*) compliant agent platform, both in academy and industry [Kravari and Bassiliades, 2015]. A JADE methodology was defined in [Nikraz et al., 2006]. A process for model transformation from models into source code for JADE in the scope of the Gaia methodology was proposed in [Moraitis and Spanoudakis, 2006], and another transformation from PIMP4Agents models to JADE sources was illustrated in [Hahn et al., 2009]. *WADE (Workflows and Agents Development Environment)* [Bergenti et al., 2012; Caire et al., 2008] can also be considered a tool to transform graphical workflows into executable JADE code.

Jadex [Braubach et al., 2005] is a software framework that implements a BDI-based reasoning engine. It combines declarative and imperative approaches by using a specification language based on *XML* (*Extensible Markup Language*) to define beliefs, goals and plans, and by using Java as procedural language to implement plans. Although Jadex does not introduce a specific syntax besides using XML, it underpins an APL. The Jadex framework can be used on top of different agent platforms, even if it was originally intended to work on top of JADE. Jadex is intended for practical and commercial use[5].

Janus [Cossentino et al., 2007; Galland et al., 2010][6] is an open-source software platform for developing agent-based applications. It is fully implemented in Java, but Janus agents can also be written by using its specific APL. As a notable feature, Janus supports the development of particular types of agents, namely holons and recursive agents.

[4] `jade.tilab.com`
[5] `www.activecomponents.org`
[6] `www.janusproject.io`

Jason [Bordini et al., 2007][7] is an agent platform that provides an interpreter for an extension of the AgentSpeak(L) abstract programming language. AgentSpeak(L) is a formally defined language for producing BDI agents. As a matter of fact, Jason is based on the BDI architecture. As an addition to the AgentSpeak(L) features, it supports agent interactions and MAS organizations. It is an open-source software fully developed in Java and it offers support for IDEs, such as Eclipse. For distributing a MAS over the network Jason can be used side by side with JADE.

1.3 Agent-Oriented Programming Languages

APLs are a class of programming languages that usually rely on a specific agent model, which is often formally defined, and they aim at providing specific constructs to adopt such a model at a high level of abstraction, thus ensuring a correct and effective use of the underlying agent technology.

Nowadays, APLs are widely recognized as crucial tools in the development of agent technologies and represent an important topic of research [Bordini et al., 2006; Bădică et al., 2011]. The transparency of the agent model, simplicity and ease of use are some of the characteristics which made the success of such languages among developers. As a matter of fact, those languages are especially convenient to model and develop complex systems, and they allow developers to expedite the creation of agents and MASs.

Despite these similarities, the features of various APLs may differ significantly, concerning, e.g., the selected agent mental attitudes (if any), the integration with an agent platform (if any), the underlying programming paradigm, and the underlying implementation language.

Some classifications of relevant APLs have already been proposed [Bădică et al., 2011; Bordini et al., 2006] to compare the characteristics of different languages and to provide a clear overview of the current state of the art of APLs. For example, [Bădică et al., 2011] classify APLs on the basis of the use of mental attitudes. According to such a classification, APLs can be divided into: AOP languages, BDI languages, hy-

[7] `jason.sourceforge.net`

brid languages, which combine the two previous classes, and other languages, which fall outside previous classes. It is worth noting that such a classification recognizes that BDI languages follow the AOP paradigm, but it reserves special attention to them for their notable relevance in the literature. The study in [Bordini et al., 2006] proposes a different classification, where languages are divided into declarative, imperative, and hybrid. Declarative languages are the most common because they focus on automatic reasoning, both from the AOP and from the BDI points of view. Some relevant imperative languages have also been proposed, and most of them were obtained by adding specific constructs to existing procedural programming languages. Finally, the presence (or absence) of a host language is an important basis of comparison among APLs.

Due to the long history of the research on agents and MASs, which lasts various decades, several APLs have been designed and implemented. In the rest of this section, some of the most important languages are cited in chronological order.

AGENT0 AOP was first introduced in [Shoham, 1993], together with his AGENT0 language [Shoham, 1991], as a first example of the application of the AOP paradigm. The work of Shoham is very important because it affected most of the developments of APLs, thus opening a wide and promising research area. In detail, Shoham describes a framework in which agents have a mental state and computation is seen as sequence of collaborative and/or competitive interactions among agents. Mental states of agents in AGENT0 are divided into two categories: beliefs and commitments. Such states change over time, which is represented as a sequence of discrete steps whose granularity is specified by the programmer. Computation in AGENT0 is described by means of an agent program, which describes and governs the behaviour of agents. First, an AGENT0 agent program initializes beliefs and commitments, then commitment rules referring to future actions are given. The life cycle of an agent is a loop in which incoming messages are processed, beliefs and commitments are updated, and actions are executed.

PLACA The direct descendant of AGENT0 is the language called *PLAnning Communicating Agents (PLACA)* [Thomas, 1993]. It extends the capabilities of AGENT0 by providing improved syntax and new mental categories. Its major improvement with respect to AGENT0 is that agent communications is lighter in terms of the amount of needed messages. Actually, the contents of message refer to high-level goals [Bădică et al., 2011] rather than to single actions. This choice allows reasoning on desired results of actions, thus easily adding planning capabilities to agents. Just like AGENT0, PLACA has experimental nature and it was not meant for practical use.

Concurrent METATEM [Fisher, 1994] is an APL based on temporal logic that can be used to program agents and MASs. In Concurrent METATEM, the life cycles of an agent and of its actions are described by sets of rules. Such rules can be divided into temporal ones and non-temporal ones. Temporal rules can be parted into three main categories, namely: start rules, step rules, and sometimes rules. In Concurrent METATEM, agents act asynchronously and the interactions among them are based on message passing. The structure of a METATEM agent provides a context set, which can contain another agent, or a group of agents.

AgentSpeak(L) An important example of a classic APL is AgentSpeak(L), whose syntax and semantics were formalized in [Rao, 1996]. The proposed formalization is based on the BDI agent model, and it uses first-order logics and Horn clauses to define a language to declaratively write agent programs with no need of low-level implementation details. The abstractions of beliefs, goals, events, and actions are formalized in first-order logics and they are used to define plans and intentions. An agent is described as a tuple which collects all such characteristics. The first widely usable implementation of AgentSpeak(L), called Jason [Bordini et al., 2007], has become very popular recently.

3APL (*An Abstract Agent Programming Language*) [Hindriks et al., 1999b] is a BDI APL which includes features of both imperative and logic programming lan-

guages. It provides a set of abstractions that are used in the development of MASs composed of agents with reasoning capabilities. The agents of 3APL have mental states that consist of collections of goals and beliefs. Agents have the ability of modifying their mental states by means of sets of practical reasoning rules which provide plans to achieve goals. Java and Haskell implementations of 3APL have been made available recently[8].

KLAIM A programming language for managing mobile agents into a networks is *Kernel Language for Agents Interaction and Mobility (KLAIM)* [De Nicola et al., 1998]. Processes, like data, can be moved from one computing environment to another, and this made KLAIM suitable for programming mobile code applications. A formal operational semantics of the language is provided, defining a type system to control access violations and determine operations in a locality, and focusing on mobile agents coordinations.

JAL The JACK platform [Winikoff, 2005] provides an environment to build MASs in which agents are based on the BDI paradigm. The *JACK Agent Language (JAL)* is an APL whose host language is Java. JAL adds to Java features of logic languages and specific statements which allow the creation of plans. Moreover, JAL supports the creation of agent teams and organizations, thus ensuring a high level of modularity. JACK and JAL are intended for practical and commercial use[9].

CLAIM *A Computational Language for Autonomous Intelligent and Mobile Agents (CLAIM)* [El Fallah-Seghrouchni and Suna, 2003] is an agent language that supports agent mobility. CLAIM agents are entities embedded in a bounded environment, and they have lists of sub-agents, so that they are organized hierarchically. CLAIM agents have two types of reasoning capabilities, namely forward and backward reasoning, which represent the reactive and the goal-driven behaviours of agents, respectively.

[8] `www.cs.uu.nl/3apl`
[9] `aosgrp.com`

Message passing among agents is allowed and, in addition, specific messages are used by the underlying system to support agent mobility.

GOAL (*Goal-Oriented Agent Language*) [Hindriks et al., 2000; Hindriks, 2009; Hindriks and Dix, 2014] is an APL that incorporates the concept of *declarative goals*. It aims at reducing the gap between agent logic and agent programming frameworks. As a matter of fact, the latter are mainly focused on agents that perform tasks in order to realize their plans, instead of agents that have a final goal to be realized. Thus, GOAL agents are rational agents, and their actions depend on their beliefs and goals. GOAL provide programming construct to structure the agent decision-making process, and to define its mental states, i.e., beliefs and goals.

SEA_L The *Semantic web-Enabled Agent Language (SEA_L)* [Demirkol et al., 2012; Challenger et al., 2016b,a] is a DSL for modeling and developing MASs in the scope of the Semantic Web. SEA_L aims at overcoming common limitations of traditional frameworks for MASs when working with the Semantic Web. In particular, it targets domain experts by putting the agent meta-model at a platform-independent level, from which developers can obtain *OWL* (*Web Ontology Language*) models, and generate code for the Jadex BDI engine. The introduction of this textual DSL can be found in [Demirkol et al., 2013]. Both SEA_L syntax and semantics are formally specified. A *DSML*, called *Semantic web-Enabled Agent Modeling Language (SEA_ML)* [Challenger et al., 2014], is also available for graphical modeling of MASs.

SARL [Rodriguez et al., 2014] is one of the latest entries in the plethora of APLs[10]. It is a general-purpose imperative language with an intuitive syntax, and it can be considered platform-agnostic, even if it is commonly used with the dedicated agent platform Janus. One of its main features is the support for the creation of holonic agents. SARL provides a syntax to declare agents and some constructs for handling

[10] `www.sarl.io`

events, that can be also user-defined. Moreover, SARL permits the definition of agent capabilities.

PROFETA (*Python RObotic Framework for dEsigning sTrAtegies*) [Fortino et al., 2015; Fichera et al., 2017] is a software framework for programming BDI-based autonomous robots or agents. It is inspired by AgentSpeak(L), but, as a notable feature, PROFETA offers a unique environments for both imperative and declarative constructs, in order to ease the definition of the behaviour of a robot. As AgentSpeak(L), PROFETA has goals, actions, beliefs, and plans. Such an extension to the AgentSpeak(L) semantics is completely implemented in Python, allowing developers to program high-level robot behaviours within a single runtime environment.

1.4 The Scope of JADEL in AOP

In this chapter, some of the most significant and impacting agent-oriented methodologies, agent-oriented platforms and frameworks, and agent-oriented languages were listed and described. A plethora of metodologies have been developed mainly because adapting object-oriented analysis to agents and MAS has many disadvantages, due to the significant differences between the two abstractions. The JADE methodology [Nikraz et al., 2006] points out such drawbacks, and shows an approach for JADE programming that is suitable for the agent paradigm, although JADE is entirely written in an object-oriented language. Nevertheless, the JADE methodology covers only the analysis and the design phases of MAS development, lacking the formalization of the implementation phase. The work on JADEL aims at simplifying the implementation of MASs, and it would improve the JADE methodology by providing the developer with an effective tool to approach that phase.

Moreover, JADEL comes with a formalization of the main concepts of JADE, and rules that establish JADEL semantics. Such an approach is not new in agent technologies history, rather it was used from the earlier studies on AGENT0, when an operational semantics was specified in [Hindriks et al., 1999a]. Then, other semantics were developed for APLs. METATEM and 3APL have both a formal specification, de-

scribed in [Wooldridge, 1997] and in [Hindriks et al., 1998]. The work on METATEM includes also an analysis of its temporal rules, [Fisher, 1996]. Also, a verifiable semantics for agent communication languages was provided, in [Wooldridge, 1998]. AgentSpeak(L) speech-act based communication was analyzed in [Vieira et al., 2007]. A more recent work, [Getir et al., 2014], shows a DSLs for agents in the Semantic Web context, together with a precise formalization of the language, and its transformation functions. The request for formalizations of APLs is not unusual, since the strength that such an analysis provide to a language is known, for example, for verifying important properties and helping programmers in avoid mistakes. Other studies, such as [Damiani et al., 2012], that explains advantages in introducing the type-checking in JaCaMo, and [Ricci and Santi, 2013], that formalizes an algorithm for verifying types on that platform, illustrates well that a formalization could be useful also for agent-oriented platforms, beside APLs. The type-checking problem applied to MASs is discussed in [Baldoni et al., 2014a] and in [Baldoni et al., 2014b].

The present work deals with the problem of a formalization of the JADE platform, before the actual definition of JADEL syntax and semantics. Then, the newly-developed APL is informally described, and an assessment of its capabilities is done, following some of the methodologies in [Challenger et al., 2016a] for the evaluation of agent-oriented DSLs.

Chapter 2

JADE Agents and MASs

Second Law. A robot must obey the orders given it by human beings
except where such orders would conflict with the First Law.

– Isaac Asimov

As a second requirement, JADEL must be founded on the *Java Agent DEvelopment Framework (JADE)*. JADE is an *Agent Platform (AP)*, which provides an *Application Program Interface (API)* and graphical tools to ease common tasks in the creation of agents and distributed *Multi-Agent Systems (MASs)*. JADE does not undertake a specific *Agent-Oriented Software Engineering (AOSE)* methodology, and it does not force developers to employ a specific agent model. Nevertheless, it has a well-defined architecture, and its features rely on precise structures. Hence, it is necessary to deeply analyze such architecture and structures, and formalize them in a suitable way to infer JADEL concepts. The formalization is based on the use of *Labeled Transition Systems (LTSs)*. This work presents the first attempt at formalizing JADE agents and MASs, but other formalizations of agent-based systems can be found in the wide literature of *Agent Programming Languages (APLs)*. The proposed transition system provides an operational semantics [Plotkin, 2004] for the life cycle of an agent which describes the agent main loop, picking up the procedures from user-defined agent classes. Thus, the entire formalization relies on Java, and the

major feature of that language have to be defined with a coherent notation. An outline of the proposed formalization was shown in preliminary works [Bergenti et al., 2015c,b].

Next sections are structured as follows. First, a brief introduction on JADE entities and on the architecture of a JADE application is provided. Then, a complete formalization of JADE agents and MASs is illustrated, starting from a description of the notation and background definitions. Finally, a complete example of a computation of a MAS is shown.

2.1 JADE Architecture and Concepts

The JADE main abstraction is the *agent*. JADE agents are multi-tasking, yet single-threaded, entities that live in the JADE runtime environment. The `Agent` base class offered by the core JADE APIs can be extended by developers in order to create their own agent classes. The creation of an agent is done by instantiating the desired agent class, i.e., a class subclass of `Agent`, in the scope of a JADE runtime environment and by providing the desired local name of the agent. An agent is characterized by an execution state that can change during its life cycle. For example, the execution state of an agent at the beginning of its life cycle is *active*, and it can become *suspended* or *waiting*, to reflect its actual runtime state. An agent life cycle is divided into three main phases, namely, a start-up phase, a control loop, and a final, or take-down phase. During start-up, the agent initializes. Such an initialization phase is primarily used to provide an initial list of actions, which the agent can use in next phases. In fact, during the control loop, the agent chooses the next action to perform in an autonomous way, and its list of available actions can change dynamically. Moreover, an agent can sense interesting events during its life cycle, and normally it reacts to them. The JADE runtime environment manages such events by means of a hidden event queue which does not necessarily require the aid of the developer.

A particular type of event is the reception of a message. Such an event does not trigger a change of agent state, but the agent takes into account the received message by storing it into a message queue. Then, the agent can activate and perform actions

in order to read or reply to messages, in their incoming order. Actions are encapsulated into JADE *behaviours*. JADE APIs offer the `Behaviour` class, whose purpose is to provide JADE agents with tasks that can be executed. As a matter of fact, developers can implement their own behaviours by subclassing that `Behaviour` class. Implementing a behaviour consists in the definition of an action, which can be added to the list of available actions of the agent. In other words, JADE provides behaviours as structured entities which represent tasks and actions that an agent can perform during its life cycle. An agent holds a list of available behaviours for its entire life cycle and it chooses behaviours to perform actions whenever needed. Each action maintains a state, which can be `active` or `blocked`, and it refers to the actual state of the behaviour within the agent list of available behaviours. A notable distinction among behaviours is given by two subclasses of `Behaviour` that are provided directly by JADE, namely the `CyclicBehaviour` and the `OneShotBehaviour` classes. Such a distinction causes cyclic and one-shot behaviour actions to adopt different semantics when they are chosen by the agent. The action of a one-shot behaviour is performed only once, because such a behaviour is removed from the list of the agent immediately after the first execution of its action. Conversely, the action of a cyclic behaviour can be performed repeatedly, because such a behaviour is removed from the list of the agent only upon explicit request.

Besides agents and behaviours, another abstraction is needed to develop useful JADE agents. Such an abstraction is the so called (*communication*) *ontology*. Ontologies support the semantics of agent communication for specific problems, and they can be used to reason on messages. Although there are no constraints on the definition of the actions of behaviours, actions are often used for message passing, i.e., to send and/or receive messages which comply with a known ontology.

A JADE MAS is composed of several agents that interact with each others and that perform actions, sending and reading messages by using a common ontology. Figure 2.1 shows a typical JADE interaction. Message are exchanged among three agents, and vertical lines show agents life cycles from initialization to take-down.

The architecture of a JADE application is organized in terms of *platforms*, which are also subdivided in *containers*. The latter are actually the running instances of

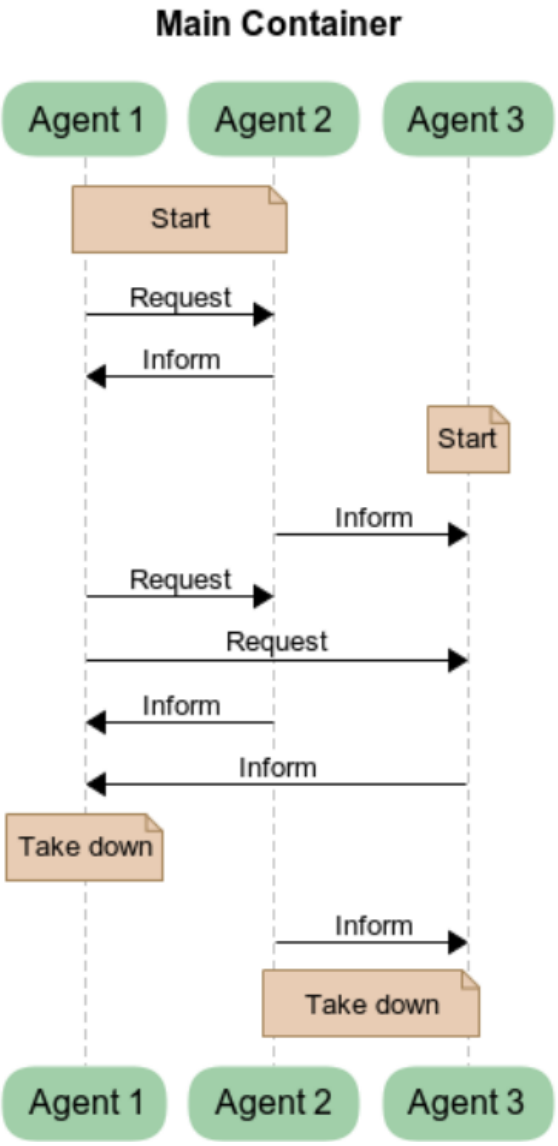

Figure 2.1: An example of interaction among three JADE agents.

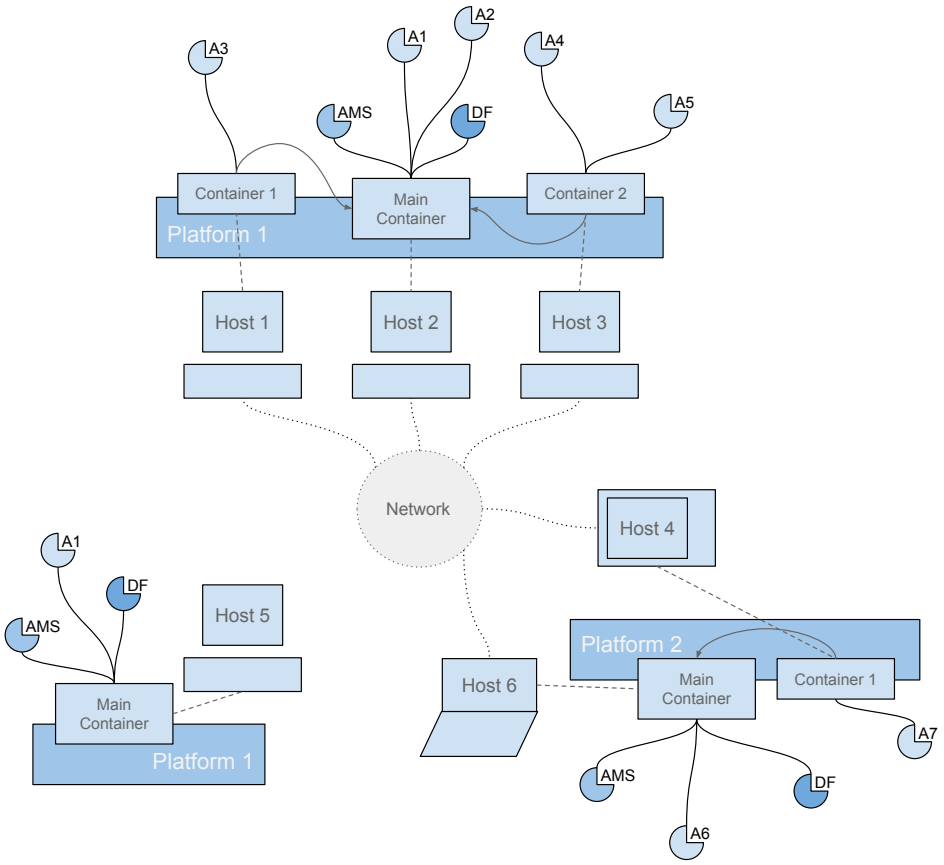

Figure 2.2: The generic architecture of a JADE application hosted in several platforms.

the JADE runtime environment and they contain several agents. A *main container* is always present in a running platform to provide the platform with management services, e.g., a *Directory Facilitator (DF)*. A newly created container can register with the main container of a platform, thus effectively joining that platform. This is done in a transparent way, even if containers are distributed on different network hosts, possibly mobile [Bergenti et al., 2014]. Several agents can be instantiated into a container, and upon creation they are provided with a so called *Agent IDentifier (AID)*, i.e., a globally unique name made of a local name plus a set of network addresses. The agent name consists in the local name of the agent followed by its platform name, and its addresses are those of the platforms the agent lives in. The AID is used in agent communications to ensure messages would reach only intended recipients. An example of JADE architecture is shown in Figure 2.2.

2.2 Notation and Background Definitions

Transition systems were introduced by Plotkin [Plotkin, 2004] to describe the structural operational semantics of programming languages. The proposed formalization uses Java statements which refer to JADE APIs. This is the reason why there is no need to specify a dedicated syntax, as, e.g., in the formalization of AgentSpeak(L) proposed by Vieira et al. [Vieira et al., 2007], that uses first-order logic and Horn clauses to define a new language which can be used to write agent programs without implementation-level details. Such an approach makes the formalization self-contained but it forces the adoption of a new programming language. On the contrary, the proposed approach relies on Java statements both for the syntax and for the semantics, thus accommodating any agent written using Java and JADE. This section is devoted to an introduction of the notations used in the formalization of JADE agents, and it provides a basic background of definitions and rules.

First, the syntax of a minimal core of Java statements is provided [Igarashi et al., 2001]. Such a fragment is composed of the syntax of FJ, and a little extension of it. Such an extension differs from FJ because void methods and two more syntax categories are introduced. The first category consists in a small set of imperative Java

$$
\begin{array}{lll}
\mathit{Cdecl} & ::= & \texttt{class}\ C\ \texttt{extends}\ D\{\ \overline{F}\ \overline{M}\ \} \\
F & ::= & C\ f; \\
M & ::= & C\ m(\overline{C\ x})\{\ \texttt{return}\ e;\ \} \mid \texttt{void}\ m(\overline{C\ x})\ \{\ \overline{s}\ \} \\
e & ::= & x \mid \mathbf{v} \mid \texttt{null} \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \mid (C)e \\
s & ::= & C\ x = e; \mid x = e; \mid \texttt{if}(b)\{\ \overline{s}_t\ \}\ \texttt{else}\{\ \overline{s}_f\ \} \mid \texttt{while}(b)\{\ \overline{s}\ \} \\
b & ::= & \texttt{true} \mid \texttt{false} \mid e_0 == e_1 \mid b_0\ \texttt{\&\&}\ b_1 \mid b_0\ \texttt{||}\ b_1 \mid \texttt{!}b
\end{array}
$$

Figure 2.3: Syntax of a minimal core of Java. C and D are metavariables that denote class names, f denotes a field name, m a method name and x a variable.

statements, namely, the declaration of a variable, the assignment, and the `if` $-$ `else` and `while` constructs. The second category introduces boolean expressions. The proposed extension is called FJ_I, because it adds imperative features to FJ.

The syntax is defined by the EBNF grammar shown in Figure 2.3. The notation $\overline{X}$ stands for the repetition of X zero or more times, and it is used to identify a sequence. An empty sequence is identified by symbol ε. The i-th element of a sequence $\overline{X}$ is indicated by X_i.

The class declaration *Cdecl* is borrowed from FJ syntax, but it contains only fields and methods. In fact, in contrast to original FJ, constructors are declared implicitly. The implicit constructor is assumed to have a number of parameters equal to the number of fields, in the order of inheritance and declaration. The implicit constructor first calls the constructor of its superclass, then it initializes all fields by means of its parameters. Fields are declared by specifying their type, which can be only a class name. Methods are divided into methods with a return type (again, a class name), and void methods which have as a body a list of statements.

Expressions e are similar to FJ expressions, with the notable exception of values $\mathbf{v}$ and `null`. A value is a Java object, i.e., a pair that consists in the class name of the object and in the values of its fields, which are also objects. It is briefly denoted by the notation $\mathbf{v} = \langle C, \overline{\mathbf{v}} \rangle$, where C is the class name and $\overline{\mathbf{v}}$ is the list of values (objects) associated with its fields. An object $\mathbf{v}$ and the sequence of its field values $\overline{\mathbf{v}}$ are indicated with the same letter.

Other two syntax categories are added, namely, the statements s and the boolean

$$fields(\texttt{Object}) = \varepsilon \qquad \text{(F-Obj)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad fields(D) = \overline{G}}{fields(C) = \overline{F}, \overline{G}} \qquad \text{(F)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad B\ m(\overline{B\ x})\,\{\,\texttt{return}\ e;\,\} \in \overline{M}}{mtype(m,C) = \overline{B} \to B} \qquad \text{(MT)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad \texttt{void}\ m(\overline{B\ x})\,\{\,\overline{s}\,\} \in \overline{M}}{mtype(m,C) = \overline{B} \to \varepsilon} \qquad \text{(MT-Void)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad m \notin \overline{M}}{mtype(m,C) = mtype(m,D)} \qquad \text{(MT-Super)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad B\ m(\overline{B\ x})\,\{\,\texttt{return}\ e;\,\} \in \overline{M}}{mbody(m,C) = \langle \overline{x}, e \rangle} \qquad \text{(MB)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad \texttt{void}\ m(\overline{B\ x})\,\{\,\overline{s}\,\} \in \overline{M}}{mbody(m,C) = \langle \overline{x}, \overline{s} \rangle} \qquad \text{(MB-Void)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\} \quad m \notin \overline{M}}{mbody(m,C) = mbody(m,D)} \qquad \text{(MB-Super)}$$

Figure 2.4: FJ fields and methods lookup.

$$C <: C \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\,\{\,\overline{F}\ \overline{M}\,\}}{C <: D}$$

Figure 2.5: Subtyping.

expressions b. Statements consist in the assignment, the `if` $-$ `else`, and the `while` constructs. A boolean expression can assume values in the set $Bool = \{\top, \bot\}$ of the Boolean domain, it can state an equality relation between two expressions e_0 and e_1 with the symbol $==$, or it can be a proposition. The symbols `&&`, `||` and `!` correspond to $\wedge$, $\vee$ and $\neg$, the logical operation on the two-element Boolean algebra on *Bool*.

As in FJ, given the syntax of such a minimal Java fragment, three auxiliary lookup functions are defined. In Figure 2.4, the rules that define such functions are shown. Starting from the class definition, a function $fields$ associates each class name with its own fields plus the inherited ones, as defined by rule (F). A function *mtype* maps to each pair of method name and class name the sequence of parameter types and the corresponding return type of the method. If such a method is void, the return type is indicated with ε. Finally, a function *mbody* returns a pair of parameters names and body statements from method name and class name, as illustrated more precisely by rules (MB), (MB-Void), and (MB-Super).

It is worth noting that class name `Object` is the name of a distinguished class that does not have a superclass, and it cannot be declared as a normal class. As a matter of fact, `Object` is a peculiar class that does not have any fields and methods. Hence, the lookup functions have special cases for it, and they return the empty sequence of fields and the empty set of methods.

A relation $<:$ is defined in Figure 2.5, in order to describe subtyping rules. As a matter of fact, a class declaration `class` C `extends` $D\{\ \overline{F}\ \overline{M}\ \}$ defines a class C of superclass D, i.e., C is a subclass of D. Each class is always subclass of itself, and the subclassing relation is transitive. As an assumption, such a relation have no cycles, e.g., if $C <: D$ then $D \not<: C$. Hence, $<:$ defines a partial order over the set of classes. One of the previous assumptions states that the `Object` class is a distinguished class that cannot be declared, nor it has fields or methods. Thus there are no D such that `Object` $<: D$, i.e., `Object` is a maximum for $<:$.

The semantics of FJ_I is given as an operational semantics which relies on a store

$$\langle x,\sigma\rangle \to_{\mathrm{FJ_I}} \sigma(x) \quad \langle \mathbf{v},\sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{v} \quad \langle \mathtt{null},\sigma\rangle \to_{\mathrm{FJ_I}} \mathtt{null} \quad \frac{\langle e_i,\sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{v_i}}{\langle (\mathtt{new}\, C(\overline{e})),\sigma\rangle \to_{\mathrm{FJ_I}} \langle C,\overline{\mathbf{v}}\rangle}$$

$$\frac{\langle e,\sigma\rangle \to_{\mathrm{FJ_I}} \langle C,\overline{\mathbf{v}}\rangle \quad fields(C) = \overline{C\ f}}{\langle e.f_i,\sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{v_i}} \qquad \frac{\langle e,\sigma\rangle \to_{\mathrm{FJ_I}} \langle C,\overline{\mathbf{v}}\rangle \quad C <: D}{\langle (D)e,\sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{v}}$$

$$\frac{\begin{array}{c}\langle e'[e/\mathtt{this},\overline{e}/\overline{x}],\sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{w} \\ mtype(C,m) = \overline{B} \to B \quad mbody(C,m) = \langle \overline{x}, e'\rangle\end{array}}{\langle e.m(\overline{e}),\sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{w}}$$

$$\frac{\begin{array}{c}\langle \overline{s}[e/\mathtt{this},\overline{e}/\overline{x}],\sigma\rangle \to_{\mathrm{jstmt}} \langle \varepsilon,\sigma'\rangle \\ mtype(C,m) = \overline{B} \to \varepsilon \quad mbody(C,m) = \langle \overline{x},\overline{s}\rangle\end{array}}{\langle e.m(\overline{e}),\sigma\rangle \to_{\mathrm{FJ_I}} \mathtt{null}}$$

Figure 2.6: Operational semantics of $\mathrm{FJ_I}$.

σ defined as follows.

$$\begin{array}{c}\sigma : Var \to Values \\ x \mapsto \mathbf{v} = \langle C,\overline{\mathbf{v}}\rangle\end{array}$$

where *Var* is the set of variables (an expression e can be a variable x, as in Figure 2.3), and *Values* is the set of values $\mathbf{v}$ (including `null`). The semantics consists in two transition systems, namely the $\mathrm{FJ_I}$ one, and the *jstmt* one. The former computes an expression e into a value, taking into account the values of the variables by means of the store σ. It does not have side effects on σ. The second transition system computes a statement s into the next statement, or into ε, which denotes termination. The assignment can change the value of a variable, thus updating the store σ, with the following notation.

$$\sigma[\mathbf{v}/x](x') = \begin{cases}\sigma(x') & \text{if } x \neq x' \\ \mathbf{v} & \text{if } x = x'\end{cases}$$

In Figure 2.6, an operational semantics of $\mathrm{FJ_I}$ with void methods is given. A configuration of $\mathrm{FJ_I}$ transition system consist of a pair $\langle e,\sigma\rangle \in Expr \times \Sigma$, where *Expr* denotes

$$\mathscr{B}[\![\texttt{true}]\!]\sigma = \top \quad \mathscr{B}[\![\texttt{false}]\!]\sigma = \bot$$

$$\mathscr{B}[\![e_0 == e_1]\!]\sigma = \begin{cases} \top & \text{if } \langle e_0, \sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{v}, \\ & \langle e_1, \sigma\rangle \to_{\mathrm{FJ_I}} \mathbf{w}, \text{ and } \mathbf{v} = \mathbf{w} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathscr{B}[\![b_0 \;\&\&\; b_1]\!]\sigma = \mathscr{B}[\![b_0]\!]\sigma \wedge \mathscr{B}[\![b_1]\!]\sigma \quad \mathscr{B}[\![b_0 \;||\; b_1]\!]\sigma = \mathscr{B}[\![b_0]\!]\sigma \vee \mathscr{B}[\![b_1]\!]\sigma$$

$$\mathscr{B}[\![\,!\,b]\!]\sigma = \neg\mathscr{B}[\![b]\!]\sigma$$

Figure 2.7: Denotational semantics of boolean expressions.

the set of all expressions and Σ the set of all stores. The set of possible configurations is called $\Gamma_{\mathrm{FJ_I}}$. The set of terminal configurations is the set of values $\Lambda_{\mathrm{FJ_I}} = \mathit{Values}$. A transition relation is defined, as usual, on $\Gamma_{\mathrm{FJ_I}} \times \Lambda_{\mathrm{FJ_I}}$ and it is indicated with $\to_{\mathrm{FJ_I}}$. Such a relation is described by the rewriting rules in the Figure 2.6.

In order to support the definition of the *jstmt* transition system, a straightforward denotational semantics for the evaluation of boolean expressions is given, and defined in Figure 2.7. A function $\mathscr{B} : \mathit{BoolExpr} \to (\Sigma \to \mathit{Bool})$ is defined from the set of all boolean expressions to the applications that maps a store with the elements of *Bool*. A definition of such a function is given by structural induction, where a store σ is fixed.

Finally, in Figure 2.8, an operational semantics of the *jstmt* transition system is given. It relies on both $\mathrm{FJ_I}$ and boolean expressions semantics. Also $\mathrm{FJ_I}$ system uses the *jstmt* one, when it has to compute the body of a void method. As a matter of fact, such a body is composed by statements, whose semantics is defined by the *jstmt* system. On the other hand, into the *jstmt* system, in the evaluation of the right-hand side of an assignment, a void method can accidentally be called. It is worth noting that this situation may cause loops, thus non-termination of programs. The proposed semantics reflects such a possibility.

$$\frac{\langle e,\sigma\rangle \to_{\text{FJ}_\text{I}} \mathbf{v}}{\langle C\,x = e;,\sigma\rangle \to_{\text{jstmt}} \langle \varepsilon, \sigma[\mathbf{v}/x]\rangle} \qquad \frac{\langle e,\sigma\rangle \to_{\text{FJ}_\text{I}} \mathbf{v}}{\langle x = e;,\sigma\rangle \to_{\text{jstmt}} \langle \varepsilon, \sigma[\mathbf{v}/x]\rangle}$$

$$\frac{\overline{s} = s_0\, s_1 \ldots s_n \quad \langle s_0,\sigma\rangle \to_{\text{jstmt}} \langle \varepsilon,\sigma'\rangle}{\langle \overline{s},\sigma\rangle \to_{\text{jstmt}} \langle s_1 \ldots s_n,\sigma'\rangle}$$

$$\frac{\mathscr{B}[\![b]\!]\sigma = \top}{\langle \texttt{if}(b)\{\,\overline{s}_t\,\}\,\texttt{else}\{\,\overline{s}_f\,\},\sigma\rangle \to_{\text{jstmt}} \langle \overline{s}_t,\sigma\rangle}$$

$$\frac{\mathscr{B}[\![b]\!]\sigma = \bot}{\langle \texttt{if}(b)\{\,\overline{s}_t\,\}\,\texttt{else}\{\,\overline{s}_f\,\},\sigma\rangle \to_{\text{jstmt}} \langle \overline{s}_f,\sigma\rangle}$$

$$\frac{\mathscr{B}[\![b]\!]\sigma = \bot}{\langle \texttt{while}(b)\{\,\overline{s}\,\},\sigma\rangle \to_{\text{jstmt}} \langle \varepsilon,\sigma\rangle}$$

$$\frac{\mathscr{B}[\![b]\!]\sigma = \top \quad \langle \overline{s},\sigma\rangle \to_{\text{jstmt}} \langle \varepsilon,\sigma''\rangle \quad \langle \texttt{while}(b)\{\,\overline{s}\,\},\sigma''\rangle \to_{\text{jstmt}} \langle \varepsilon,\sigma'\rangle}{\langle \texttt{while}(b)\{\,\overline{s}\,\},\sigma\rangle \to_{\text{jstmt}} \langle \varepsilon,\sigma'\rangle}$$

Figure 2.8: Operational semantics of jstmt.

2.3 A Formalization of JADE Entities and Events

In this section, the main features and entities of JADE are described and then formally defined. JADE APIs consist of several Java classes that help developing MASs by providing proper methods and base classes which can be directly used or extended. Only a notable subset of such classes is considered, in order to describe clearly the behaviour of a MAS. To this extent, the main classes that a developer must adopt if he/she wants to deal with JADE are selected, namely the `Agent` class, the `Behaviour` class, with some of its subclasses, and the `ACLMessage` and `MessageTemplate` classes, for agent interactions.

2.3.1 Entities

The formalization of agents consists in the definition of a new type of value, which identifies the instantiated agent in a running environment. Such a value reflects the

specific characteristic of that instantiated agent, as described previously. Values are treated as in the formalization of objects in Section 2.2.

Definition 1 (Agent)**.** *Let Agents be the set of all agents and*

$$S_A = \{\texttt{initiated}, \texttt{active}, \texttt{waiting}, \texttt{suspended}, \texttt{deleted}\}$$

the set of all agent runtime states. An agent $\mathbf{a} \in Agents$ *is a tuple*

$$\mathbf{a} = \langle C, s_a, L_b, L_e, Q \rangle$$

where $C <: \texttt{Agent}$ *is the agent class name,* $s_a \in S_A$ *is the state of the agent,* L_b *a finite list of available actions,* L_e *is a finite list of events, and* Q *is a message queue.*

The fact that each agent has a start-up and a final take-down phase is formalized by specifying the proper methods that the `Agent` class provides, namely, a void `setup` method and a void `takeDown` method. Such methods have no parameters and they are empty placeholder for user-defined code. `Agent` subclasses must redefine such methods properly.

$$mbody(\texttt{setup}, \texttt{Agent}) = mbody(\texttt{takeDown}, \texttt{Agent}) = \langle \varepsilon, \varepsilon \rangle$$
$$mtype(\texttt{setup}, \texttt{Agent}) = mtype(\texttt{takeDown}, \texttt{Agent}) = \varepsilon \to \varepsilon$$

Let us now define the second entity that JADE provides for the development of useful agents, namely the `Behaviour` class. As for the agents, behaviours formalization consists in the definition of a new type of value.

Definition 2 (Behaviour)**.** *Let Behaviours be the set of all behaviours, and*

$$S_B = \{\texttt{initiated}, \texttt{active}, \texttt{blocked}, \texttt{done}\}$$

the set of behaviour action states. A behaviour $\mathbf{b} \in Behaviour$ *is a tuple*

$$\mathbf{b} = \langle C, a, s_b \rangle$$

where $C <: \texttt{Behaviour}$ *is the name of the behaviour class,* a *is the name of the associated agent, and* $s_b \in S_B$ *is the state of the behaviour action.*

$$\texttt{OneShotBehaviour} <: \texttt{Behaviour} \qquad \texttt{CyclicBehaviour} <: \texttt{Behaviour}$$

$$\frac{B <: \texttt{Behaviour} \quad A <: \texttt{Agent}}{B \in behaviours(A)}$$

$$btype(\texttt{Behaviour}) = \varepsilon$$

$$\frac{B <: \texttt{OneShotBehaviour}}{btype(B) = \texttt{one} - \texttt{shot}} \qquad \frac{B <: \texttt{CyclicBehaviour}}{btype(B) = \texttt{cyclic}}$$

Figure 2.9: Behaviours lookup.

The `Behaviour` base class provides a field `myAgent`, which is used to access the associated agent of the behaviour, and a method `action`, which is empty but it can be overridden by the developer in order to implement the actual action.

$$\texttt{Agent myAgent} \in fields(\texttt{Behaviour})$$
$$mbody(\texttt{action}, \texttt{Behaviour}) = \langle \varepsilon, \varepsilon \rangle$$
$$mtype(\texttt{action}, \texttt{Behaviour}) = \varepsilon \to \varepsilon$$

Two lookup functions are defined in Figure 2.9, namely the *behaviours* and the *btype* functions. From an agent class name, *behaviours* computes the set of all available behaviours. As an assumption, each subtype of `Behaviour` is available for every agent. The *btype* function maps its action type, namely $\texttt{one} - \texttt{shot}$ or `cyclic`, to each behaviour class name.

In a MAS, instantiated agents can exchange messages with each other. A message is another entity of the JADE system: it has a sender, a list of recipients, a content and a performative, which may be `inform`, `request`, $\texttt{not} - \texttt{understood}$, and so on, as in FIPA specifications. The message entity of the JADE system is defined as follows.

Definition 3 (Message)**.** *A* message $\mathbf{m} \in$ *Messages is a tuple:*

$$\mathbf{m} = \langle a, R, c, p \rangle$$

$$
\begin{array}{rcl}
p & ::= & \texttt{accept}-\texttt{proposal} \mid \texttt{agree} \mid \texttt{cancel} \mid \texttt{cfp} \mid \texttt{confirm} \\
& \mid & \texttt{disconfirm} \mid \texttt{failure} \mid \texttt{inform} \mid \texttt{inform}-\texttt{if} \mid \texttt{inform}-\texttt{ref} \\
& \mid & \texttt{not}-\texttt{understood} \mid \texttt{propagate} \mid \texttt{propose} \mid \texttt{proxy} \mid \texttt{query}-\texttt{if} \\
& \mid & \texttt{query}-\texttt{ref} \mid \texttt{refuse} \mid \texttt{reject}-\texttt{proposal} \mid \texttt{request} \\
& \mid & \texttt{request}-\texttt{when} \mid \texttt{request}-\texttt{whenever} \mid \texttt{subscribe} \mid \texttt{unknown} \\
t & ::= & x \mid \bar{x} \mid \mathbf{v} \mid p \mid t_0 \wedge t_1 \mid t_0 \vee t_1 \mid \neg t
\end{array}
$$

Figure 2.10: Performatives and message templates syntaxes.

where a is the name of the sender agent, R is a list of recipients, c is an object that acts as the content of the message, and p is a performative.

Finally, *message templates* are entities that play a key role in agent messaging. The simplest form of message template can be understood as a particular message which can have undefined values. Such template describes the kind of messages the agent expects from other agents. JADE provides the method `receive(t)`, where `t` is a template, to allow an agent to pick the first message in its message queue which matches the message template. JADE allows combining simple message templates into complex message templates using connectives. Therefore, a template is a sort of logical combination of message templates. In Figure 2.10, the complete list of performatives and the syntax used for message templates is shown. In detail, a message template can be a variable, denoting, for example, the desired sender agent, or it can be a sequence of variables, as recipients of the message. It can be also a value, i.e., an object, or a performative. Combination of templates are made by using logical connectives.

2.3.2 Events

During the life cycle of a MAS, a sequence of events is formed and each agent is fed with a sub-sequence of such events. The sub-sequence of events of each agent are partitioned into external and internal ones, because they can change the state of the system—these are the external events—and/or the state of agents and respective behaviours—these are the internal events. Note that in JADE each agent is single-

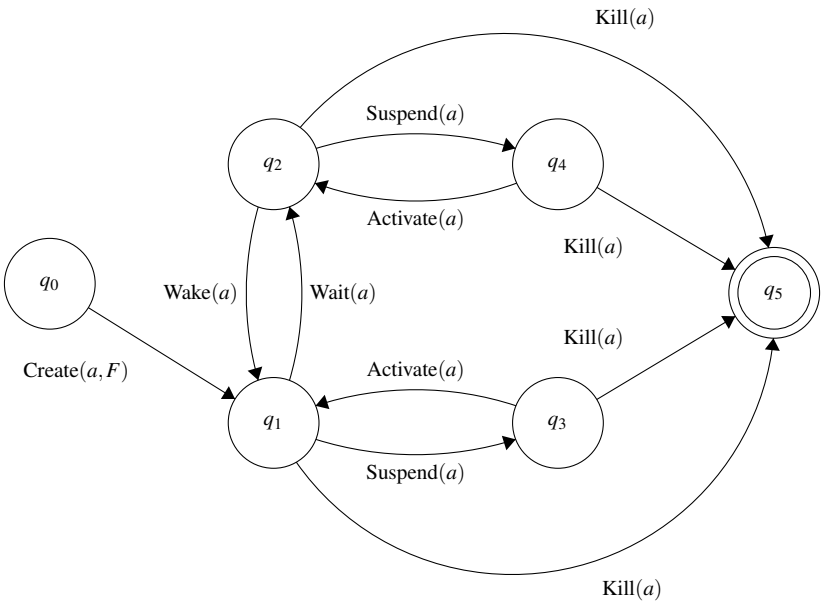

Figure 2.11: Agent life cycle events and their relationships. The life cycle state constants are $q_0 =$ `initiated`, i.e., the initial state, $q_1 =$ `active`, $q_2 =$ `waiting`, $q_3 =$ `suspended from active`, $q_4 =$ `suspended from waiting` and the final state $q_5 =$ `deleted`.

threaded and it manages only one event at a time. Events are formalized as predicates and the set of notable events that JADE manages, called *Events*, contains the following elements:

- Create(a, C) denotes the creation of the agent a of class C, and it is and external event;
- Kill(a) denotes the destruction of the agent a, and it is an external event;
- Wait(a) denotes the state change of the agent a to `waiting`, and it is an internal event;
- Wake(a) denotes the state change of the agent a from `waiting` to `active`, and it is an internal event;
- Suspend(a) denotes the state change of the agent a to `suspended`, and it is an internal event;
- Activate(a) denotes the state change of the agent a from `suspended` to the *previous* state, and it is an internal event;
- Create(a, b, C) denotes the creation of the behaviour b of class C, associated with the agent a, and it is an external event;
- Block(b) denotes the state change from `active` to `blocked` for the behaviour b, and it is an internal event;
- Restart(b) denotes the state change from `blocked` to `active` for the behaviour b, and it is an internal event;
- Done(b) denotes the state change from `active` to `done` for the behaviour b, and it is an internal event;
- Message(m) denotes that the message m was sent, and it is an external event;
- Start(MAS) denotes the creation of the MAS, and it is an external event; and

- End(MAS) denotes the termination of the MAS, together with all related agents and behaviours, and it is an external event.

The events in *Events* that denote changes in the state of an agent are related to each other as shown by the *Finite State Machine (FSM)* in Figure 2.11. Formally, such a FSM can be defined as a quintuple

$$\langle q_0, Q_A, L_A, \delta_A, F_A \rangle$$

where $q_0 =$ `initiated` is the initial state, $Q_A = \{q_0, q_1, \ldots, q_5\}$ is the finite set of all states, L_A is the finite set of agent internal events, plus the Create(a,C) event, $\delta_A : Q_A \times L_A \rightarrow Q_A$ is the transition function defined informally by the figure, and $F_A = \{q_5\}$ contains only the final state $q_5 =$ `deleted`.

An agent state change is activated by a specific event in E. Note that for an agent a, the states $q_3 =$ `suspended from active` and $q_4 =$ `suspended from waiting` correspond to the same state $s_a =$ `suspended` $\in S_A$ of JADE. To properly merge them into one the previous state of the agent has to be memorized, so that a correct reaction to an Activate(a) event could be provided. A FSM does not have explicit memory, so a list of events occurred in the agent life cycle has to be maintained. Such a list is called L_e and it appears in Definition 1 as a specific agent feature.

Figure 2.12 shows the FSM which represents the state changes of a behaviour. As for agent state changes, a FSM is defined as a quintuple

$$\langle q_0, Q_B, L_B, \delta_B, F_B \rangle$$

where $q_0 =$ `initiated` is the initial state, $Q_B = S_B = \{q_0, q_1, q_2, q_3\}$ is the finite set of all states, L_B is the finite set of behaviour internal events, plus the Create(a,b,C) event, $\delta_B : Q_B \times L_B \rightarrow Q_B$ is the transition function, and $F_B = \{q_3\}$ is the set of final states, containing only the $q_3 =$ `done` state. No additional memory is required to model the life cycle of a behaviour because no *return to previous state* is demanded.

2.4 Syntax and Semantics of JADE

This section describes an extension of the FJ_I and *jstmt* syntax and semantics with notable features selected from the wide range of JADE APIs. A list of few method

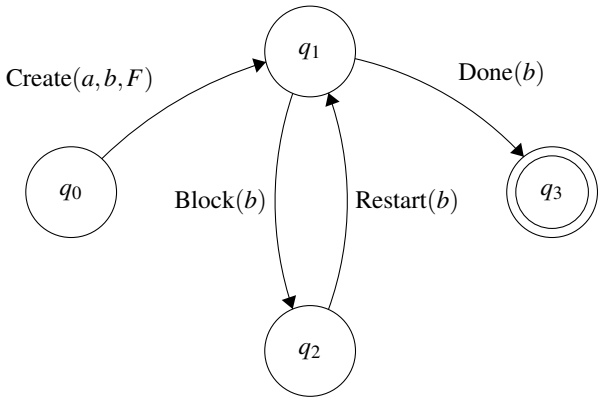

Figure 2.12: Behaviour life cycle events and their relationships: $q_0 =$ `initiated`, $q_1 =$ `active`, $q_2 =$ `blocked` and $q_3 =$ `done`.

$$
\begin{array}{lcl}
com & ::= & e.\texttt{addBehaviour}(e) \mid e.\texttt{removeBehaviour}(e) \mid e.\texttt{send}(e) \\
& \mid & x = e.\texttt{receive}() \mid x = e.\texttt{receive}(e) \mid e.\texttt{addReceiver}(e) \\
& \mid & e.\texttt{setContent}(e) \\
ev & ::= & e.\texttt{doWait}() \mid e.\texttt{doSuspend}() \mid e.\texttt{doWake}() \mid e.\texttt{doActivate}() \\
& \mid & e.\texttt{doDelete}() \mid e.\texttt{block}() \mid e.\texttt{restart}()
\end{array}
$$

Figure 2.13: JADE commands and events.

calls are added to the basic syntax of FJ_I. Such calls are fundamental in writing JADE agents and behaviours, because they actually provide them with an agent oriented semantics, over the Java one.

In Figure 2.13, a syntax of a JADE fragment is given. Two syntax categories are defined, namely commands, and event commands. As a matter of fact, some statements and method calls of JADE APIs are used to manage the agent behaviours list, or the agent communication, and they are called *commands*. Apart from such commands, other method calls are devoted to managing agents and behaviours life cycles, by triggering internal events, and they are called *event commands*.

It is worth noting that the proposed extension uses FJ_I expressions and variables, and defines two more assignments. Method calls and assignments already exist in FJ_I syntax but they are computed by FJ_I and *jstmt* semantics into objects (values), and they have side effects only on the objects memorized in a store σ. Thus, in particular, within FJ_I syntax and semantics, the resulting values and effects of JADE APIs method calls are not distinguishable from all other method calls. A purpose of the proposed formalization is to decouple the object-oriented meta-model from the agent-oriented one, so the identification of JADE features is necessary. After such an identification, a semantics that takes into account the modifications in agent, behaviour, and message values, can be defined. To this extent, three more stores are defined, namely an agent store, a behaviour store and a message store.

Definition 4 (JADE Stores)**.** *An* agent store α *is an application from the set of variables to the set of agent values.*

$$\begin{aligned} \alpha &: Var \to Agents \\ x &\mapsto \mathbf{a} = \langle C, s_a, L_b, L_e, Q \rangle \end{aligned}$$

Similarly, a behaviour store β *and a* message store μ *map variables into the set of*

behaviour and message values, respectively.

$$\begin{aligned} \beta &: Var \to Behaviours \\ & x \mapsto \mathbf{b} = \langle C, a, s_b \rangle \\ \mu &: Var \to Messages \\ & x \mapsto \mathbf{m} = \langle a, R, c, p \rangle \end{aligned}$$

Given a store, an entity, which is a tuple, can be accessed. Sometimes it is needed only to know an element of such tuple, e.g., the internal state of a behaviour. For this reason, the *natural projection* is defined over each store, indicated with π_i where i is the position of the element to access. As an example, to extrapolate the agent state from an agent entity the composition is used

$$\begin{aligned} \pi_2 \circ \alpha &: Var \to S_A \\ & x \mapsto s_a \end{aligned}$$

As a general convention, $\pi_i \circ \alpha$ is called α_i. Such notations are the same for each store.

With the usual Plotkin notation [Plotkin, 2004], an agent, behaviour or message store, can be updated as in Section 2.2. If the intention is to replace only an element, the projection notation is allowed in combination with the update notation. For example, given a variable $a \in Var$ that denotes an agent, changing the state of such an agent into a new state $s_a \in S_A$ is done as follows.

$$\alpha_2[s_a/a](x) = \begin{cases} s_a & \text{if } x = a \\ \alpha_2(x) & \text{otherwise.} \end{cases}$$

When modifying an element of the agent entity using the above notation, the other elements of such an entity remain unchanged.

2.4.1 Semantics of Commands with Events

A coherent semantics to manage the events and the state changes that occur in the life cycle of an agent is needed. To obtain this, a transition system is defined as follows

$$\langle \Gamma_{\text{com}}, L_{\text{com}}, \to_{\text{com}}, \Delta_{\text{com}} \rangle$$

$$\frac{}{\langle \alpha, \beta \rangle \xrightarrow{\varpi}_{\text{ev}} \langle \alpha, \beta \rangle} \quad \text{(No-Event)}$$

$$\frac{\alpha(a) = \langle C, s_a, L_b, L_e, Q \rangle \quad L'_e = [L_e | E_a] \quad s'_a = \delta_A(s_a, E_a) \in S_A}{\langle \alpha, \beta \rangle \xrightarrow{E_a}_{\text{ev}} \langle \alpha[\langle C, s'_a, L_b, L'_e, Q \rangle / a], \beta \rangle} \quad \text{(A-State)}$$

$$\frac{\alpha(a) = \langle C, s_a, L_b, L_e, Q \rangle \quad L'_e = [L_e | E_a] \quad \delta_A(s_a, E_a) \notin S_A}{\langle \alpha, \beta \rangle \xrightarrow{E_a}_{\text{ev}} \langle \alpha[\langle C, \texttt{suspended}, L_b, L'_e, Q \rangle / a], \beta \rangle} \quad \text{(A-Susp)}$$

$$\frac{\begin{array}{c} E_a = \text{Activate}(a) \quad \alpha(a) = \langle C, s_a, L_b, L_e, Q \rangle \quad L'_e = [L_e | E_a] \\ L_e = [\ldots | \text{Wait}(a), \overline{E}, \text{Suspend}(a)] \quad \forall i \in \{0, \ldots, n\}\ E_i \neq \text{Wake}(a) \end{array}}{\langle \alpha, \beta \rangle \xrightarrow{E_a}_{\text{ev}} \langle \alpha[\langle C, \texttt{waiting}, L_b, L'_e, Q \rangle / a], \beta \rangle} \quad \text{(A-Act1)}$$

$$\frac{\begin{array}{c} E_a = \text{Activate}(a) \quad \alpha(a) = \langle C, s_a, L_b, L_e, Q \rangle \quad L'_e = [L_e | E_a] \\ L_e = [\ldots | \text{Wake}(a), \overline{E}, \text{Suspend}(a)] \quad \forall i \in \{0, \ldots, n\}\ E_i \neq \text{Wait}(a) \end{array}}{\langle \alpha, \beta \rangle \xrightarrow{E_a}_{\text{ev}} \langle \alpha[\langle C, \texttt{active}, L_b, L'_e, Q \rangle / a], \beta \rangle} \quad \text{(A-Act2)}$$

$$\frac{\begin{array}{c} E_a = \text{Activate}(a) \quad \alpha(a) = \langle C, s_a, L_b, L_e, Q \rangle \quad L'_e = [L_e | E_a] \\ L_e = [\text{Create}(a, C), \overline{E}, \text{Suspend}(a)] \quad \forall i \in \{0, \ldots, n\}\ E_i \neq \text{Wait}(a) \end{array}}{\langle \alpha, \beta \rangle \xrightarrow{E_a}_{\text{ev}} \langle \alpha[\langle C, \texttt{active}, L_b, L'_e, Q \rangle / a], \beta \rangle} \quad \text{(A-Act3)}$$

$$\frac{\beta(b) = \langle C, a, s_b \rangle \quad s'_b = \delta_B(s_b, E_b)}{\langle \alpha, \beta \rangle \xrightarrow{E_b}_{\text{ev}} \langle \alpha, \beta_2[s'_b / b] \rangle} \quad \text{(B-State)}$$

Figure 2.14: Event system.

$$\frac{\langle s,\sigma\rangle \to_{\text{jstmt}} \langle s',\sigma'\rangle \quad \langle \alpha,\beta\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha',\beta'\rangle}{\langle s,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E}_{\text{com}} \langle s',\sigma',\alpha',\beta',\mu\rangle} \quad \text{(Jstmt)}$$

$$\frac{\langle c,\sigma,\alpha,\beta,\mu\rangle \to_{\text{stmt}} \langle \varepsilon,\sigma',\alpha'',\beta'',\mu'\rangle \quad \langle \alpha'',\beta''\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha',\beta'\rangle}{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon,\sigma',\alpha',\beta',\mu'\rangle} \quad \text{(Com)}$$

$$\frac{\overline{s} = s_0\, s_1 \ldots s_n \quad \langle s_0,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon,\sigma',\alpha',\beta',\mu'\rangle}{\langle \overline{s},\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E}_{\text{com}} \langle s_1 \ldots s_n,\sigma',\alpha',\beta',\mu'\rangle} \quad \text{(Seq)}$$

$$\frac{\langle a,\sigma\rangle \to_{\text{FJ}_\text{I}} \langle C,\overline{\mathbf{v}}\rangle \quad C <: \texttt{Agent} \quad \alpha \xrightarrow{\text{Wait}(a)}_{\text{ev}} \alpha'' \quad \langle \alpha'',\beta\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha',\beta'\rangle}{\langle a.\texttt{doWait}(),\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon,\sigma,\alpha',\beta',\mu\rangle} \quad \text{(E-Wait)}$$

$$\frac{\langle b,\sigma\rangle \to_{\text{FJ}_\text{I}} \langle C,\overline{\mathbf{v}}\rangle \quad C <: \texttt{Behaviour} \quad \beta \xrightarrow{\text{Block}(b)}_{\text{ev}} \beta'' \quad \langle \alpha,\beta''\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha',\beta'\rangle}{\langle b.\texttt{block}(),\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon,\sigma,\alpha',\beta',\mu\rangle} \quad \text{(E-Block)}$$

Figure 2.15: Semantics of commands with events.

where Γ_{com} is the set of configurations, L_{com} is a set of labels, and $\rightarrow_{\text{com}} \subseteq \Gamma_{\text{com}} \times L_{\text{com}} \times (\Gamma_{\text{com}} \cup \Delta_{\text{com}})$ is a transition function, where Δ_{com} is a set of terminal configurations.

A configuration consists in a tuple where the first element is a command or an event command from Figure 2.13, or else a statement from Figure 2.6. Other elements of such a configuration are stores, one for each kind of entity. In fact, σ is used to memorize and update Java objects, α, β and μ to maintain information about agents, behaviours, and messages, respectively. A correct form for a configuration according to the above definition is $\langle c, \sigma, \alpha, \beta, \mu \rangle$, but for the sake of clarity and when no ambiguity can arise, all five elements of configurations are not enumerated in the description of transition rules. Terminal configurations are configurations with no commands. The symbol ε is used in place of c for such configurations.

During the execution of a command or a statement, an event may cause a state change in one of the entities stored. For this reason, the set of labels L_{com} is defined as the set of all internal events. Thus, a transition is made by a configuration, an event, and another configuration or terminal configuration. There are also some execution steps where no event occurs. These steps are indicated with a special symbol ϖ as label. In order to correctly update agents and behaviours after an event occurrence, a transition relation is defined over JADE stores, and it is called $\rightarrow_{\text{ev}}$. When needed, such a transition relation is used by the system $\rightarrow_{\text{com}}$.

Figure 2.14 shows the rules of the event transition system. In detail, when no event occurs, agent and behaviour stores remains unchanged, as stated by the rule (No-Event). Rules (A-State) and (B-State) are grounded on the two transition functions δ_A and δ_B, and they update properly the agent and behaviour states according to the returning value of such functions. For clarity, an agent event is indicated with the name E_a and a behaviour event with E_b. Moreover, the agent list of events is always updated with the last event E_a occurred. The only exceptions consist in the occurrence of the events Suspend(a) and Activate(a). In order to remain coherent with JADE, there is no distinction between the agent states q_3 and q_4 of Figure 2.11. So, as in rule (A-Susp), the agent is provided with the state `suspended`. If the agent is re-activated after a suspension, its list of events has to be inspected to find out what

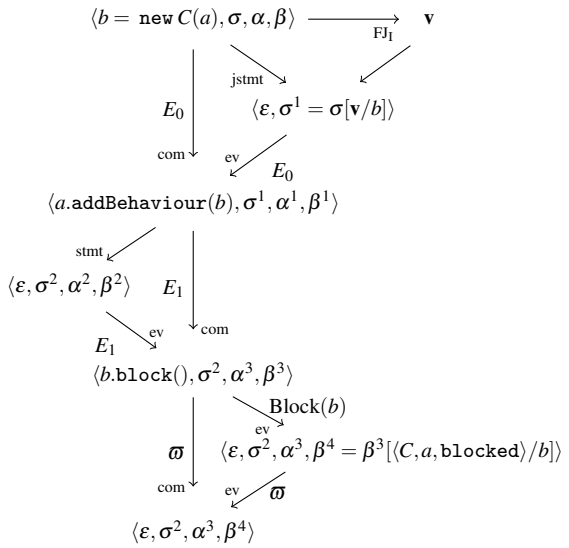

Figure 2.16: Example of computation of the command transition system and subsystems.

was its previous state. Such a mechanism is described by rules (A-Act1), (A-Act2), and (A-Act3). An agent can be in its `waiting` state only if a Wait(a) occurs, and it remains there until it is waken. Meanwhile, a sequence of suspension and re-activation may happens. This is showed in detail by the rule (A-Act1). Similarly, rules (A-Act2), and (A-Act3) describe the situation in case the previous state was `active`. A better solution for the problem of re-activation, in terms of simplicity of the proposed transition rules, could be to maintain two agent states (the actual and the previous) across the computation. But that solution does not provide meaningful information about the life cycle of such an agent, and adds an unnecessary field to the definition of agent values.

In Figure 2.15 the $\rightarrow_{\text{com}}$ transition relation is defined. A generic statement that can be computed by the *jstmt* system is indicated with s. This is expressed by the (Jstmt) rule, which delegates to the *jstmt* system the execution of s and to the event system the effects caused by the label E.

The extended syntax in Figure 2.13 provides also commands that produce effects

on JADE stores α, β and μ. For this reason, a system that computes correctly such commands is necessary. Such a system has a transition relation, called $\rightarrow_{\text{stmt}}$, that is defined and explain in detail later, in Section 2.4.2 and 2.4.3, when the agent life cycle and the execution of its actions are studied in deep. Rule (Com) delegates to such a system the semantics of a generic command, indicated with c, and, *after* that, it propagates the event E.

A sequence of Java statements and JADE commands is denoted by $\bar{s}$ in rule (Seq). Statements are processed in their order, and the event E occurs when computing s_0. Then, other events can take place in the next steps of computation, starting from the execution of s_1.

A particular attention should be dedicated to the last category of the extended syntax, namely, the statements *ev* that trigger events, called event commands. In Figure 2.15 there are two example of transitions that explain the semantics of such commands. The first one, (E-Wait), triggers the event $\text{Wait}(a)$ of the agent a that calls the command `doWait()`. Such an event occurs *before* the event E indicated by the label. Moreover, the expression a has to be checked to ensure that it is actually an agent, by computing its value with the FJ_I transition system. Such a check is necessary in order to update correctly the proper store. In the next, that check is not indicated in the premise of a rule, but it is left implicit. To recognize the entities involved, from now on all expressions that compute into an agent are indicated with a, and all expressions that compute into a behaviour with b.

Similarly, the event command `block()` triggers the event $\text{Block}(b)$ of the behaviour b, before the event E. Rule (E-Block) describe this situation. Other rules are defined for event commands, that trigger agent and behaviour events. But they are not reported here because they work exactly as the two previous rules.

In Figure 2.16, an example of computation of the command transition system and sub-systems is shown. Suppose that there is a sequence of three commands, namely, $b = \texttt{new}\, C(a)$; $a.\texttt{addBehaviour}(b)$; $b.\texttt{block}()$. The variable a is a variable that denotes an agent, and b is a behaviour of class C. While the first command is executed, the event E_0 occurs, and while the second command is executed, the event E_1 occurs. The semantics of the first command is given by the sub-system *jstmt*, which updates

$$\frac{E = \text{Create}(a,C)}{\langle \varepsilon, \alpha \rangle \xrightarrow{E} \langle a.\texttt{setup}(), \alpha[\langle C, \texttt{active}, [], [E], [] \rangle / a] \rangle} \quad \text{(A-Create)}$$

$$\frac{\alpha_1(a) = C \quad mbody(\texttt{setup}, C) = \langle \varepsilon, \bar{s} \rangle \quad \langle \bar{s}, \sigma, \alpha, \beta, \mu \rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu' \rangle}{\langle a.\texttt{setup}(), \sigma, \alpha, \beta, \mu \rangle \xrightarrow{E} \langle \varepsilon, \sigma', \alpha', \beta', \mu' \rangle} \quad \text{(A-Setup)}$$

$$\frac{E = \text{Kill}(a) \quad \alpha \xrightarrow{E}_{\text{ev}} \alpha'}{\langle \varepsilon, \alpha \rangle \xrightarrow{E} \langle a.\texttt{takeDown}(), \alpha' \rangle} \quad \text{(A-Kill)}$$

$$\frac{\alpha_1(a) = C \quad mbody(\texttt{takeDown}, C) = \langle \varepsilon, \bar{s} \rangle \quad \langle \bar{s}, \sigma, \alpha, \beta, \mu \rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu' \rangle}{\langle a.\texttt{takeDown}(), \sigma, \alpha, \beta, \mu \rangle \xrightarrow{E} \langle \varepsilon, \sigma', \alpha'[\varpi/a], \beta', \mu' \rangle} \quad \text{(A-TakeDown)}$$

Figure 2.17: Agent life cycle.

the underlying Java store σ with the new value **v**. As usual, **v** is an object, which is the evaluation of the expression `new` $C(a)$ obtained by the FJ system. The actual creation of a location for the behaviour inside the store β is managed by the system com, but the rule for such an update is given in the next sections. For now, α^1 and β^1 denotes the stores that result from the correct managing of the event E_0 and the behaviour creation. Then, the second command is an agent-oriented one, provided by JADE APIs. The semantics of such a command is given by the system stmt, which updates correctly all the stores. After this, the event E_1 updates agents and behaviours. Finally, the third command triggers an event, namely, the event Block(b) of the behaviour b. Thus, b state is changed into the `blocked` one by the event system. After that, the event system have to manage the next event, but there is no one, so all stores remain unchanged and the computation terminates.

$$\frac{\alpha_1(a) = C \quad \beta_1(b) \in behaviours(C) \quad L'_b = [\alpha_3(a)|b]}{\langle a.\texttt{addBehaviour}(b), \alpha, \beta\rangle \rightarrow_{\text{stmt}} \langle \varepsilon, \alpha_3[L'_b/a], \beta\rangle} \quad \text{(A-AddB)}$$

$$\frac{\alpha_3(a) = [b_0, \ldots, b_i, b, b_{i+1}, \ldots, b_n] \quad L'_b = [b_0, \ldots, b_i, b_{i+1}, \ldots, b_n]}{\langle a.\texttt{removeBehaviour}(b), \alpha\rangle \rightarrow_{\text{stmt}} \langle \varepsilon, \alpha_3[L'_b/a]\rangle} \quad \text{(A-RmvB)}$$

Figure 2.18: Adding and removing behaviours.

2.4.2 Agent Life Cycle

As for commands and events, an operational semantics is defined by means of a transition system, in order to describe how the states of entities, namely, agents, behaviours, and messages, change in a computation of a JADE MAS. The proposed transition system is defined as a structure $\langle \Gamma, L, \rightarrow, \Lambda\rangle$ where Γ is a finite set of configurations, L is a finite set of labels, $\rightarrow: \Gamma \times L \times (\Gamma \cup \Lambda)$ is the transition relation, and Λ is a set of terminal configurations.

As in command transition system, a configuration is in the form $\langle C, \sigma, \alpha, \beta, \mu\rangle$, where C is a command (or statement). Its syntax is the FJ extended one, as defined in Section 2.2, plus the syntax of commands and event commands defined in Figure 2.13. There are also terminal configurations, i.e., configurations where the command C is replaced by a symbol `E` which indicates the end of the computation. The symbol ε is used, but not for terminal configurations. In fact, this transition system allows configurations with no commands. Labels in L are internal or external events.

Such a transition system is a super-system of all others defined in this chapter. It manages agent life cycles, i.e., initialization of agents and final clean-ups, and behaviour actions. Agent life cycle starts when an agent a of class C is created. When such an event happens, the initialization phase starts. Formally, when the event Create(a, C) occurs, the system make a step from a configuration with no commands to a setup configuration, where the agent is stored into a new variable a and the `setup` method of the agent is called. Rule (A-Create) shows this transition.

The initialization phase consists in the execution of the method `setup` body, as in rule (A-Setup). The sub-system of command is used in order to manage a the

sequence of statements that compose the body. As a matter of fact, setup body may contain both Java statements and JADE commands or event commands. Moreover, the label E is an event that is propagated by the system of commands as shown in previous sections.

When an event Kill(a) occurs, the final phase of the agent a starts. The agent is immediately updated by the system of events, i.e., its state became `deleted`. Then, its `takeDown` method is called. The execution of `takeDown` body uses the commands sub-system, as illustrated by the rules (A-Kill) and (A-TakeDown).

During the two phases of initialization and take-down, usually the agent manages its list of behaviour, adding and removing tasks. Two commands of JADE APIs are used, namely, the $a.\texttt{addBehaviour}(b)$ and $a.\texttt{removeBehaviour}(b)$. Their semantics is defined by the transition relation $\rightarrow_{\text{stmt}}$, which is shown in rules (A-AddB) and (A-RmvB). In particular, adding a behaviour implies checking if such a behaviour is available for the agent class C, by means of the lookup function $behaviours(C)$. If so, the new behaviour is added to the bottom of the agent behaviour list L_b. Removing a behaviour b is done by simply by removing it from the agent list L_b, no matter what its position is.

2.4.3 Behaviour Actions

Once the agent finishes its initialization, it is ready to perform actions. A behaviour from its list is selected, and the actual execution of the action starts. A selector for the behaviours of an agent is defined, called *behaviour scheduler*, indicated by $\mathscr{S}$. This function returns the current behaviour to execute, from the list L_b of an agent a, as follows.

$$\mathscr{S} : Agents \rightarrow Behaviours$$
$$a \mapsto b \in \alpha_3(a)$$

Rules that describe in detail the execution of a behaviour action are listed in Figure 2.19. Creating a new behaviour b of class C is a command $b = \texttt{new}\ C(\overline{e})$ that can be launched, e.g., by an agent in its setup. The syntax is the usual Java syntax, thus

$$\frac{\begin{array}{c}\langle b = \texttt{new}\, C(\overline{e}), \sigma\rangle \rightarrow_{\text{jstmt}} \langle \varepsilon, \sigma'\rangle \quad C <: \texttt{Behaviour} \\ \beta[\langle C, e_0, \texttt{initiated}\rangle/b] \xrightarrow{\text{Create}(a,b,C)}_{\text{ev}} \beta'' \quad \langle \alpha, \beta''\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha', \beta'\rangle\end{array}}{\langle b = \texttt{new}\, C(\overline{e}), \sigma, \alpha, \beta, \mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu\rangle} \quad \text{(B-Create)}$$

$$\frac{\mathscr{S}(a) = b \quad \beta'_3(b) = \texttt{active} \quad \alpha'_2(a) = \texttt{active} \quad \langle \alpha, \beta\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha', \beta'\rangle}{\langle \varepsilon, \alpha, \beta, \mu\rangle \xrightarrow{E} \langle b.\texttt{action}(), \alpha', \beta', \mu\rangle} \quad \text{(B-Selected)}$$

$$\frac{\begin{array}{c}\langle \overline{s};\ a.\texttt{removeBehaviour}(b), \sigma, \alpha, \beta, \mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu'\rangle \\ \beta(b) = \langle C, a, s_b\rangle \quad \mathit{mbody}(\texttt{action}, C) = \langle \varepsilon, \overline{s}\rangle \quad \mathit{btype}(C) = \texttt{one-shot}\end{array}}{\langle b.\texttt{action}(), \sigma, \alpha, \beta, \mu\rangle \xrightarrow{E} \langle \varepsilon, \sigma', \alpha', \beta'_2[\texttt{done}/b], \mu'\rangle} \quad \text{(B-OneShot)}$$

$$\frac{\begin{array}{c}\langle \overline{s}, \sigma, \alpha, \beta, \mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu'\rangle \\ \beta(b) = \langle C, a, s_b\rangle \quad \mathit{mbody}(\texttt{action}, C) = \langle \varepsilon, \overline{s}\rangle \quad \mathit{btype}(C) = \texttt{cyclic}\end{array}}{\langle b.\texttt{action}(), \sigma, \alpha, \beta, \mu\rangle \xrightarrow{E} \langle \varepsilon, \sigma', \alpha', \beta', \mu'\rangle} \quad \text{(B-Cyclic)}$$

Figure 2.19: Behaviour actions.

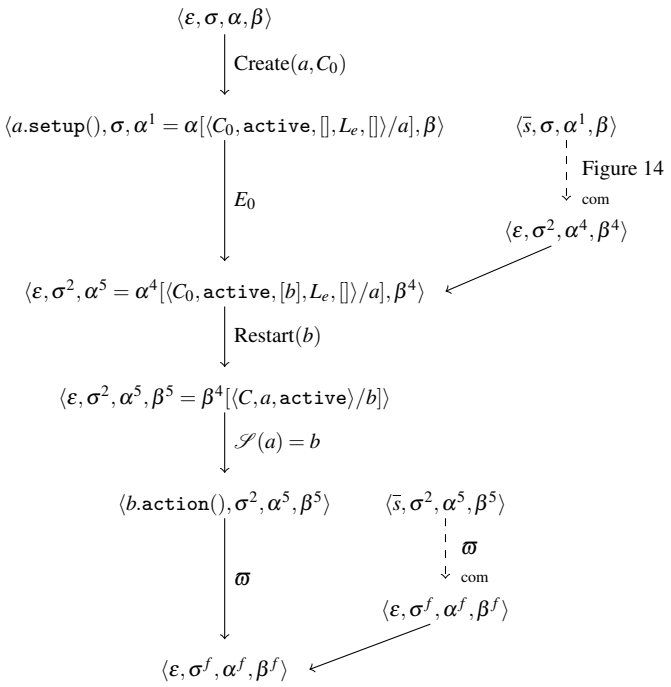

Figure 2.20: Agent life cycle example.

the store σ is updated by the *jstmt* system. Meanwhile, a location b into the store β is created, with the initial value $\mathbf{b} = \langle C, e_0, \texttt{initiated} \rangle$, where C is the behaviour class, e_0 is the first parameter of the constructor, that represents the agent, and `initiated` is its initial state, according to Figure 2.12. Moreover, such a command triggers the event Create(e_0, b, C) of the behaviour, and the store is coherently updated by the system of events. Then, if there is another event E during the transition, such an event is considered *after* the creation of the behaviour. Rule (B-Create) define this transition. When both a behaviour b and an agent a are active, i.e., they are in the `active` state, and $b \in L_b = \alpha_3(a)$, the behaviour scheduler of the agent a is able to choose such a behaviour b for the next execution. If it is the case, the action of the behaviour b starts. As in Rule (B-Selected), this means the actual call of the method `action` of the behaviour b.

A distinguishing features of JADE behaviours is their type, which refers to their permanence in the list of available actions of an agent. Behaviour can be *one-shot* or *cyclic*, by sub-classing the `OneShotBehaviour` or the `CyclicBehaviour` classes, respectively. The lookup function *btype* is used to access such a type, as defined in Figure 2.9. The action of a one-shot behaviour is performed only once, because such a behaviour is removed from the list of the agent immediately after the first execution of its action. Conversely, the action of a cyclic behaviour can be performed repeatedly, since such a behaviour is not removed from the list of the agent after the execution of its action. Rules (B-OneShot) and (B-Cyclic) formalize this aspect.

As an example, in Figure 2.20, the life cycle of an agent is shown. At the first step, an agent a of class C_0 is created. Thus it starts its initialization by calling the method `setup` redefined in the body of C_0. Suppose that its setup body contains the statements illustrated in Figure 2.16. Hence, the sub-system of commands executes such statements as in the first example, creating a behaviour b of class C and adding it to the behaviour list of the agent. Then, it is blocked by the command b.`block`(). Then, an event Restart(b) re-activate such a behaviour, so it is ready to be chosen by the behaviour scheduler. In fact, the agent behaviours list contains only an element, and the agent is active. In this very simple example, the scheduler selects the only behaviour it can select, b. This fact triggers the call of the `action` method of b, redefined in class C. Statements that compose the body of the action are computed by the sub-system of commands, and, finally, the stores are updated into their final values.

2.5 Syntax and Semantics of Message Passing

Although there are no constraints on the definition of the actions of behaviours, actions are often used for message passing, i.e., to send and receive messages. Such communication-related tasks are always implemented using patterns that are well-known to JADE programmers. In Figure 2.21 rules that concern message creations, sending, and receptions are shown. First, if an agent intends to send a message, it allocates a specific behaviour to do it. The code of such a behaviour prepares the

$$\frac{\langle m \;=\; \texttt{new ACLMessage}(p), \sigma\rangle \rightarrow_{\text{jstmt}} \langle \varepsilon, \sigma'\rangle \quad \langle \alpha, \beta\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha', \beta'\rangle}{\langle m \;=\; \texttt{new ACLMessage}(p), \sigma, \alpha, \beta, \mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu[\langle \varepsilon, [], \varepsilon, p\rangle / m]\rangle} \quad \text{(M-Create)}$$

$$\frac{R' = [\mu_2(m)|r]}{\langle m.\texttt{addReceiver}(r), \mu\rangle \rightarrow_{\text{stmt}} \langle \varepsilon, \mu_2[R'/m]\rangle} \quad \text{(M-AddReceiver)}$$

$$\frac{}{\langle m.\texttt{setContent}(c), \mu\rangle \rightarrow_{\text{stmt}} \langle \varepsilon, \mu_3[c/m]\rangle} \quad \text{(M-SetContent)}$$

$$\frac{\langle \varepsilon, \alpha, \beta, \mu_1[a/m]\rangle \xrightarrow{\text{Message}(m)} \langle \varepsilon, \alpha'', \beta'', \mu'\rangle \quad \langle \alpha'', \beta''\rangle \xrightarrow{E}_{\text{ev}} \langle \alpha', \beta'\rangle}{\langle a.\texttt{send}(m), \alpha, \beta, \mu\rangle \xrightarrow{E}_{\text{com}} \langle \varepsilon, \alpha', \beta', \mu'\rangle} \quad \text{(M-Send)}$$

$$\frac{\begin{array}{c}\mu_2(m) = R \quad r \in R \quad \alpha(r) = \langle C, s_a, L_b, L_e, Q\rangle \\ Q' = [Q|m] \quad b \in L_b \quad (s_a = \texttt{waiting} \vee s_a = \texttt{active}) \quad E = \text{Message}(m)\end{array}}{\langle \varepsilon, \alpha, \beta, \mu\rangle \xrightarrow{E} \langle \varepsilon, \alpha[\langle C, \texttt{active}, L_b, L_e, Q'\rangle / r], \beta_3[\texttt{active}/b], \mu\rangle} \quad \text{(M-Event)}$$

$$\frac{\alpha_5(a) = [m|Q'] \quad \sigma(m) = \mathbf{v}}{\langle x = a.\texttt{receive}(), \sigma, \alpha, \mu\rangle \rightarrow_{\text{stmt}} \langle \varepsilon, \sigma[\mathbf{v}/x], \alpha_5[Q'/a], \mu[\mu(m)/x]\rangle} \quad \text{(M-Receive)}$$

$$\frac{\alpha_5(a) = []}{\langle x = a.\texttt{receive}(), \sigma, \alpha, \mu\rangle \rightarrow_{\text{stmt}} \langle \varepsilon, \sigma[\texttt{null}/x], \alpha, \mu[\varpi/x]\rangle} \quad \text{(M-EmptyQueue)}$$

Figure 2.21: Messaging.

message to be sent, and then it delivers the message using the method `send` of its agent, provided by JADE APIs. The preparation of a message consists in its creation, by instantiating an object of class `ACLMessage`, which constructor takes a performative as a parameter. Thus, the command $m = \texttt{new ACLMessage}(p)$ is used to create a new location m into the messages store μ. The value stored is initially empty, except for the performative p, as shown in rule (M-Create).

Other features of the messages can be set after its creation, by using some methods of the class `ACLMessage`. The extended syntax of commands in Figure 2.13 in-

$$\mathscr{M} : M \times T \to (\Sigma \to \mathit{Bool})$$

$$\mathscr{M}[\![m,x]\!]\mu = \begin{cases} \top & \text{if } \mu_1(m) = x \vee \mu_2(m) = [x] \\ \bot & \text{otherwise} \end{cases} \qquad \mathscr{M}[\![m,\overline{x}]\!]\mu = \begin{cases} \top & \text{if } \mu_2(m) = [\overline{x}] \\ \bot & \text{otherwise} \end{cases}$$

$$\mathscr{M}[\![m,\mathbf{v}]\!]\mu = \begin{cases} \top & \text{if } \mu_3(m) = \mathbf{v} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathscr{M}[\![m,t_1 \wedge t_2]\!]\mu = \mathscr{M}[\![m,t_1]\!]\mu \wedge \mathscr{M}[\![m,t_2]\!]\mu$$

$$\mathscr{M}[\![m,t_1 \vee t_2]\!]\mu = \mathscr{M}[\![m,t_1]\!]\mu \vee \mathscr{M}[\![m,t_2]\!]\mu$$

$$\mathscr{M}[\![m,\neg t]\!]\mu = \neg\mathscr{M}[\![m,t]\!]\mu$$

$$\frac{\alpha_5(a) = [m_0,\ldots,m_k,m,m_{k+1},\ldots,m_n] \quad Q' = [m_0,\ldots,m_k,m_{k+1},\ldots,m_n] \quad \mathscr{M}[\![m,t]\!]\mu = \top \quad \mathscr{M}[\![m_i,t]\!]\mu = \bot,\ i = 0,\ldots,k}{\langle x = a.\texttt{receive}(t),\alpha,\mu\rangle \to_{\text{stmt}} \langle \varepsilon, \alpha_5[Q'/a], \mu[\mu(m)/x]\rangle} \quad \text{(M-Template)}$$

Figure 2.22: Match function for message templates.

cludes the calls at such methods, namely, m.`addReceiver`(r), and m.`setContent`(c). The former adds an agent r as a recipient of the message, and the latter set the content c of a message. Such a content can be any object. Those calls are handled by the system *stmt*, because they only update the store of messages. Rules (M-AddReceiver) and (M-SetContent) define their semantics.

Finally, the prepared message can be sent. To this extent, the class `Agent` provides a method `send`. The call of such a method is managed by the system of commands. First, the sender of the message is updated with the reference of the agent who calls the *send* instruction. This is shown in rule (M-Send). Moreover, delivering a message triggers an external event, Message(m), which purpose is explained in the next rule (M-Event).

Then, if an agent wants to react at the reception of a message, it has to add a behaviour devoted to wait for incoming messages. Such a behaviour waits until it perceives an event and it has to figure out if that event is actually a message. If this is the case, it reads the incoming message and, eventually, it performs an action in response. For this reason, when the event Message(m) occurs, all recipients in m receivers list re-activate all their behaviours, as shown in the conclusion of the rule (M-Event). All recipients add m at the end of their message queue Q. Moreover, if one of the recipient agents is in the `waiting` state, it changes immediately to the `active` one.

Once the behaviours are active, those who wait for messages has to read it. Reading a message is done by calling the method `receive` of the agent, which returns the first message in the agent queue, as in (M-Receive). If the queue is empty, it returns the value `null`, as in (M-EmptyQueue). JADE users must be familiar with such a semantics. In fact, the pattern used for message reception follows a list of fixed steps, namely, *(i)* the reception of the message by means of the instruction $m = a$.`receive`$()$, *(ii)* an `if` statement that controls if the resulting value of m is not `null`, and *(iii)* if it is the case, the action, else, *(iv)* the blocking of the current behaviour.

Another option is to specify a message template, in order to read only certain types of messages. The command $x = a$.`receive`(t), where t is a template, is used to filter messages in the agent a message queue. The syntax of a template was shown

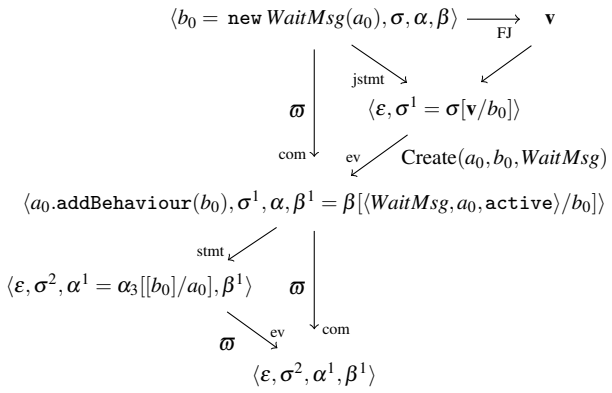

Figure 2.23: Set-up of the agent a_0.

previously in Figure 2.10. A template is a condition over message feaures. If one of the incoming messages matches the template, the variable x is assigned to the value of such a message.

To formally define the matching of a message by a message template, a semantic function $\mathscr{M}$ is introduced over variables, templates and stores. Such function takes values in the set $Bool = \{\top, \bot\}$ of the Boolean domain and with the same symbols used in the syntax (i.e., $\wedge$, $\vee$ and $\neg$) are indicated the logical operation on $Bool$. In Figure 2.22, a denotational semantics for such a matching is defined, and the semantics of the command $x = a.\texttt{receive}(t)$ is given as a transition (M-Template) in the stmt system.

2.6 An Example of Agent Set-Up and Behaviour Actions

As an example, a simple interaction between two agents is considered. The first agent, a_0, starts and waits for an incoming `INFORM` message. If such a message is received, a_0 do its action. The reception and the action are managed by a behaviour of the agent, called b_0. The second agent, a_1, starts and then it activates a behaviour b_1, which sends a *'ping'* message to a_0, with performative `INFORM`. In the following, the JADE code for agent a_0 set-up is shown.

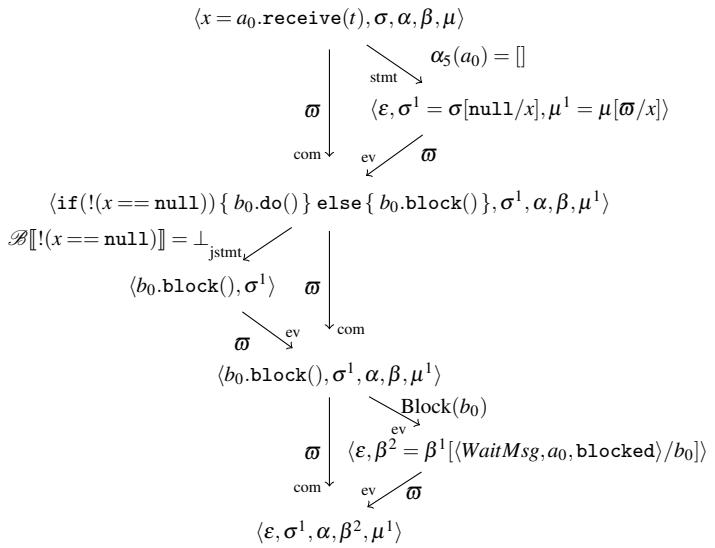

Figure 2.24: Receiving a message with an empty queue.

```
1  // Agent a0
2  void setup(){
3      WaitMsg b0 = new WaitMsg(this);
4      this.addBehaviour(b0);
5  }
```

In Figure 2.23, the semantics of such a code is illustrated. As an assumption, no event occur during the initialization of a_0, for simplicity. First, the new behaviour b_0 of class `WaitMsg` is created, and turns its state into `active`. Then, a_0 adds such a behaviour to its behaviour list. Suppose that the agent a_1 initializes in a similar manner: it creates a new behaviour b_1 of class `SendMsg` and adds it to the behaviour list. Thus, JADE code and its semantics are analogs to that shown for agent a_0.

Classes `WaitMsg` and `SendMsg` are coded in JADE as subclasses of the behaviour classes `CyclicBehaviour` and `OneShotBehaviour`, respectively. The implementation of their `action` methods is shown below.

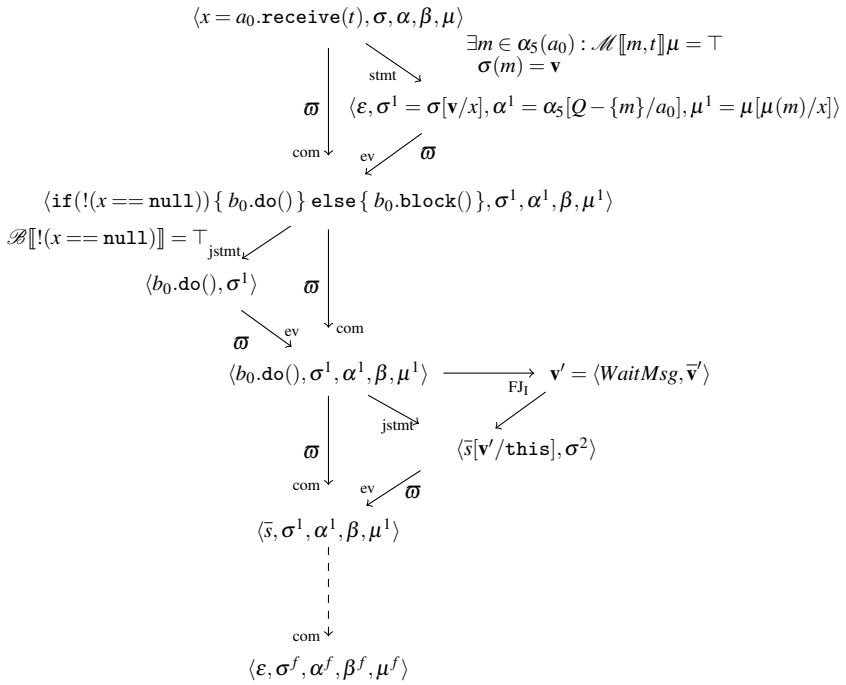

Figure 2.25: Receiving the INFORM message and doing the action.

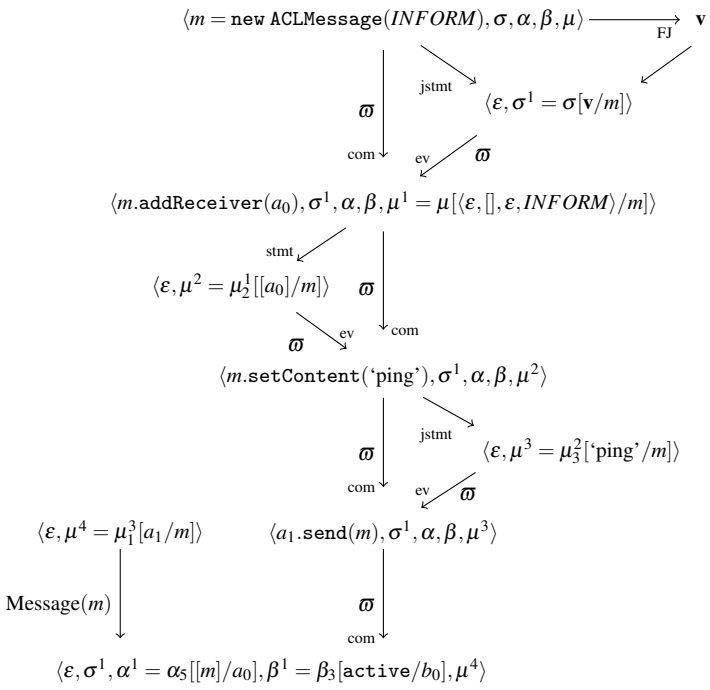

Figure 2.26: Sending a 'ping' message to a_0.

```
1  // Behaviour WaitMsg
2  void action(){
3      // Creating a template t = INFORM
4      Object x = this.myAgent.receive(t);
5      if (x != null) {
6          this.do();
7      } else {
8          this.block();
9      }
10 }
11
12 void do() {
13     // ...
14 }
```

For a `WaitMsg` behaviour, the action consists in the creation of a template t, which is the expected performative, the reception of a message x, and the processing of such a message, if it exists. In fact, if the agent message queue is empty, the value of message x is set to `null`, and the behaviour is blocked.

```
1  // Behaviour SendMsg
2  void action(){
3      // Performative p = INFORM
4      ACLMessage m = new ACLMessage(p);
5      // a0 is the AID of the receiver agent
6      m.addReceiver(a0);
7      m.setContent('ping');
8      this.myAgent.send(m);
9  }
```

For a `SendMsg` behaviour, the action concerns only the creation and sending of a message. First, a new `ACLMessage` is created, with a given performative. Then, the AID of a_0 is added to the list of recipients and the content of the message is set as a raw string 'ping'. Finally, the method `send` of the owner agent is used, by means of the internal field `myAgent`.

In Figures 2.24, 2.25, and 2.26, three computations of the given actions are shown. No internal event occur during such computations.

The first figure concerns the execution of behaviour b_0, when the agent a_0 message queue is empty. Rule (M-EmptyQueue) states that both message and Java stores update x with the `null` value. Then, from the semantics of FJ_I and jstmt, given in Figure 2.8, the system com enters the `else` block of the `if` construct. As shown in a previous example, the command `block` of behaviours triggers an event Block(b_0), which changes the behaviour state from `active` to `blocked`.

The second figure shows the actual reception of an `INFORM` message. This is done by the system stmt, with the rule (M-Template), which controls the matching of the template t for all messages in a_0 message queue. Then the queue is updated with a new queue when the matching message is removed. In this case, the value of x became that of the message extracted from the queue, and the system enters the `if` block. Such a block contains a call of a void method `do`, which is managed by the systems FJ_I and jstmt.

The third figure reports the execution of the agent a_1 action. It creates a new `ACLMessage`, which is an object stored in σ by the jstmt system, and, in particular, it is a message as defined in Definition 3, so it is stored in μ by the system of JADE commands. Then, using rules (M-AddReceiver) and (M-SetContent), the fields of the message are updated. Finally, the call of the method `send` of the agent triggers an event Message(m), which updates the message queue of the recipient agent a_0 and re-activate all blocked behaviours, as in rules (M-Send) and (M-Event).

2.7 Interaction Protocols

In this section the *interaction protocol transition system* is defined. In JADE, interaction protocol roles are modeled as *FSMs*. FSMs are commonly formalized as 5-tuples consisting of a set of states, here called S, an initial state $q_0 \in S$, a set of labels L, a transition function $\delta : S \times L \to S$ and a set of final states $F \in S$. Thus, the formalization

$$
\begin{array}{lcl}
ip & ::= & \texttt{REQUEST} \mid \texttt{CONTRACT\_NET} \mid \texttt{BROKERING} \mid \texttt{DUTCH\_AUCTION} \\
 & \mid & \texttt{ENGLISH\_AUCTION} \mid \texttt{ITERATED\_CONTRACT\_NET} \mid \texttt{PROPOSE} \mid \texttt{QUERY} \\
 & \mid & \texttt{RECRUITING} \mid \texttt{REQUEST\_WHEN} \mid \texttt{SUBSCRIBE} \mid \texttt{ITERATED\_REQUEST} \\
role & ::= & \texttt{Initiator} \mid \texttt{Responder}
\end{array}
$$

Figure 2.27: JADE interaction protocols and roles syntaxes.

of a *FSMBehaviour* is an ordinary FSM, as follows.

$$\text{FSM} = \langle q_0, S, L, \delta, F \rangle \quad \text{with } F \subset S, L \subseteq p;$$
$$\delta : S \times L \to S;$$
$$\mathbf{b} = \langle C, a, s_b, s \rangle \quad \text{with } s \in S.$$

It is worth noting that an additional state is added to the behaviour entity, to take into account FSM state changes. But such a standard representation is not enough to capture the semantics of JADE protocols. There are some issues to consider when deal with an interaction protocol in JADE. First of all, for each state of the FSM the agent can do an action: this means that there is a block of code to perform when the agent reach a state. On the second hand, it is desirable to take account of all state changes and updates that may be a result of an action within the behaviour entity that models the FSM.

The *interaction protocols transition system* is defined as a labeled transition system

$$\langle c_0, \Gamma_\delta, P, \delta, \to_\delta, \Lambda_\delta \rangle$$

where Γ_δ is a set of configurations of the form $\langle c, \beta \rangle$, with $c \in s \cup com \cup ev := Com$, $c_0 \in \Gamma_\delta$ is the initial configuration, $P \subseteq p$ is a set of performatives, used as labels, $\delta : S \times P \to S$ is the FSM transition fucntion, Λ_δ is a set that contains terminal configurations $\langle \varepsilon, \beta \rangle$ and $\to_\delta$ is the interaction protocol transition function, defined as follows:

$$\to_\delta \subseteq \Gamma_\delta \times P \times \Gamma_\delta.$$

Moreover, two additional transition functions are defined:

$$\gamma : S \times B \to Com;$$
$$\varphi : Com \to Com \cup Com^{*}.$$

Such functions allow to control which statements to execute, by matching a command with a state of the FSM and then by associating that command with the block of code needed to perform the requested action. The γ transition function returns such a command, given the internal state of the current behaviour in B. The φ transition function acts as a bridge to the commands transition system, by specifying the statement or the block of code to execute, associated with the γ command.

2.7.1 Rule Schema for Protocols

In order to define the semantics of each FIPA interaction protocol using the above described transition system, the approach of *operational semantics* [Plotkin, 2004] is used. Such method is about giving a semantics to programming languages, but it can be used to define the natural semantics over an arbitrary, labeled or unlabeled, transition system.

Given a protocol $q \in ip$ and an agent role $r \in role$ in such a protocol, defined in Figure 2.27, in JADE API there is only one FSM behaviour that describes the role r in protocol q. Usually, the name of the protocol and its role are specified in the class name `B` of such behaviour. Then, assuming that $b \in B$ is the FSM behaviour associated with the agent a, with $\beta_1(b) = C$, the rule to apply depends on the chosen protocol and agent role, and also on the current internal state of b, i.e. $\beta_4(b) = s$. Therefore, the expected semantics has to take account of the values q, r and s.

The idea is that from an initial configuration $c_0 = \langle c, \beta \rangle$, the system switches to another configuration $c_1 \in \Gamma_\delta$, through the performative l of the first message: $c_0 \xrightarrow{l}_\delta c_1$. That is, tuple $\langle c_0, l, c_1 \rangle$ belongs to the relation $\to_\delta$. A notation $\vdash_b$ is added to remind what is the behaviour name (protocol and role) at each step: $\vdash_b c_0 \xrightarrow{l}_\delta c_1$.

Some conditions have to be satisfied to guarantee the transition to be applied. For example, for FIPA *Request* protocol, the first incoming message for the participant

must be a request. These conditions are specific for each step of the chosen protocol, and have to be tested before allowing a transition of the main system. This leads to another notation, that helps to synthesize in a single rule schema every rule of the interaction protocols: $B_q^{(r,s)}$ denotes a formula which contains a condition that depends on the protocol q, role r and current state s. When $B_q^{(r,s)}$ is true, the rule is applicable.

At each step, as mentioned before, the agent has to perform an action. The command associated to each state of the FSM behaviour is obtained as the value of the γ transition function: $\gamma(s,b) = c$. Then, the associated action consists of the block of code $\varphi(c)$, that is assumed to be executable by the subsystem of commands. It is taken as an assumption that such a transition system performs the statements contained in the obtained block and updates the stores correctly. If it terminates, a new list of updated stores is given. The generic computation is denoted as follows.

$$\langle \varphi(C), \sigma, \alpha, \beta, \mu \rangle \rightarrow^*_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu' \rangle$$

where $\rightarrow^*_{\text{com}}$ indicates the Kleene closure of the command transition function.

In summary, the rule schema can be written in the usual operational semantics notation, as follows.

$$\frac{B_q^{(r,s)} \quad \langle \varphi(c), \sigma, \alpha, \beta, \mu \rangle \rightarrow^*_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta', \mu' \rangle \quad \beta'_4(b) = s}{\vdash_b \langle c, \beta \rangle \xrightarrow{l}_{\delta} \langle c', \beta'_4[\delta(s,l)/b] \rangle} \quad \text{(ProtocolRuleSchema)}$$

where $c = \gamma(s,b)$, and $c' = \gamma(\delta(s,l),b)$.

2.8 Achieve Rational Effect Protocols

FIPA provides many interaction protocols that are quite similar to the *FIPA Request* protocol. For this reason, the JADE implementations of such protocol roles are collected together using the classes `AchieveREInitiator`, for the initiator, and `AchieveREResponder` for the participant. In general, such classes are used to implement all protocols where the first message is sent by the initiator in order to achieve a *Rational Effect* (*RE*), then there is a response and an eventual result notification by the participant, here called responder. Such protocols are for example *FIPA Query*, *FIPA*

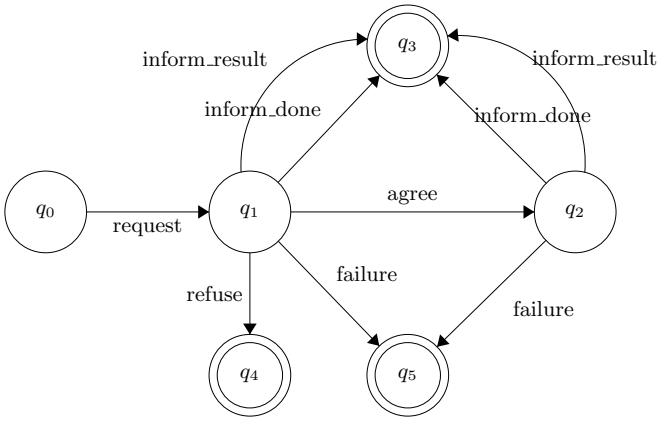

Figure 2.28: Achieve Rational Effect Protocol Initiatior FSM: $q_0 = \texttt{send} \in S$ is the initial state, $q_1 = \texttt{receive reply} \in S$, $q_2 = \texttt{agreed} \in S$, and $q_3 = \texttt{success} \in S$, $q_4 = \texttt{refused} \in S$, $q_5 = \texttt{failed} \in S_I$ are final states. The query protocol FSM is analogous, with the only differences in the labels request and inform–done, which change respectively in query–if/ref and inform–t/f.

Request-when, *FIPA Brokering*, etc. Even if FIPA does not specify such a protocol, it is inherently part of FIPA specifications because the semantics of FIPA communications, as described in FIPA specifications, is based on *agents performing communicative acts to achieve rational effects*.

2.8.1 Achieve Rational Effect Initiator

The initiator class for above described protocols is a behaviour which consists of a finite number of internal states, a transition function over such states, and a collection of class methods to handle the action to perform while reaching a particular state. Hence, it is a FSM behaviour, that can be formalized as discussed in the previous section.

$$\text{AchieveREInitiator} = \langle c_0, \Gamma_\delta, P', \delta, \rightarrow_\delta, \Lambda_\delta \rangle$$

with:

$$P' \subseteq p; \quad c_0 = \langle \texttt{send(m)}, \beta \rangle \in \Gamma_\delta;$$
$$\beta(b) = \langle \texttt{AchieveREInitiator}, a, s_b, \texttt{send} \rangle.$$

According to FIPA specifications of request and query protocols, an *achieve rational effect initiator* is a behaviour $b \in B$ which implements the FSM in Figure 2.28.

It is worth noting that in JADE, both the performative `inform_result` and `inform_done` are joint together into the performative `inform`, but this fact does not change substantially the structure of the FSM. The transition function δ is thus defined by the Figure 2.28 above, as usual. For example, $\delta(q_0, \text{request}) = q_1$. Note that there is no performative that moves the behaviour state from `success`, `refused` or `failed` to another. This is obvious since these states denote final states, but in the proposed formalization, the transition function has to reach a terminal configuration in order to recognize the achievement of the aforementioned rational effect. This is formalized by adding an ε-transition to a fake final state $q_6 = \texttt{final}$, i.e., a transition with no label. Then, indicating with $\varepsilon \in P'$ the 'empty' performative: $\delta(q_3, \varepsilon) = \delta(q_4, \varepsilon) = \delta(q_5, \varepsilon) = q_6$.

If b is an achieve rational effect initiator, then:

$$\gamma(q_0, b) = \texttt{send}(m) \qquad \gamma(q_1, b) = m = \texttt{receive}(t)$$
$$\gamma(q_2, b) = \texttt{handleAgree}(m) \quad \gamma(q_3, b) = \texttt{handleInform}(m)$$
$$\gamma(q_4, b) = \texttt{handleRefuse}(m) \quad \gamma(q_5, b) = \texttt{handleFailure}(m)$$
$$\gamma(q_6, b) = \varepsilon$$

where m, t denote a message and a message template, respectively, and ε is a notation that describes the absence of a command to execute.

In the following the rules for *Request* and *Query* protocols are described, which are conditions for (ProtocolRuleSchema). In case of $r = \texttt{Initiator}$ and $q = \texttt{REQUEST}$,

the rule conditions are as follows:

$$B_q^{(r,q_0)} : \mathscr{M}[\![m, \texttt{request}]\!]\mu = \top$$
$$B_q^{(r,q_1)} : t = \texttt{agree} \vee \texttt{refuse} \vee \texttt{inform} \vee \texttt{failure}$$
$$B_q^{(r,q_2)} : \mathscr{M}[\![m, \texttt{agree}]\!]\mu = \top$$
$$B_q^{(r,q_3)} : \mathscr{M}[\![m, \texttt{inform}]\!]\mu = \top$$
$$B_q^{(r,q_4)} : \mathscr{M}[\![m, \texttt{refuse}]\!]\mu = \top$$
$$B_q^{(r,q_5)} : \mathscr{M}[\![m, \texttt{failure}]\!]\mu = \top$$

where m,t are the message and the template specified by γ. These conditions define six different rules based on the rule schema in Section 2.7.1.

Another achieve rational effect protocol is $q = \texttt{QUERY}$, whose six rules are defined below.

$$B_q^{(r,q_0)} : \mathscr{M}[\![m, (\texttt{query}-\texttt{if} \vee \texttt{query}-\texttt{ref})]\!]\mu = \top$$
$$B_q^{(r,q_1)} : t = (\texttt{agree} \vee \texttt{refuse} \vee \texttt{inform} \vee \texttt{failure})$$
$$B_q^{(r,q_2)} : \mathscr{M}[\![m, \texttt{agree}]\!]\mu = \top$$
$$B_q^{(r,q_3)} : \mathscr{M}[\![m, \texttt{inform}]\!]\mu = \top$$
$$B_q^{(r,q_4)} : \mathscr{M}[\![m, \texttt{refuse}]\!]\mu = \top$$
$$B_q^{(r,q_5)} : \mathscr{M}[\![m, \texttt{failure}]\!]\mu = \top$$

where, as above, m,t are the message and the template specified by γ.

2.8.2 Achieve Rational Effect Responder

An *achieve rational effect responder* is a behaviour $b \in B$ which implements the FSM in Figure 2.29. In the proposed formalization, it is defined as follows.

$$\text{AchieveREResponder} = \langle c_0, \Gamma_\delta, P', \delta, \rightarrow_\delta, \Lambda_\delta \rangle$$

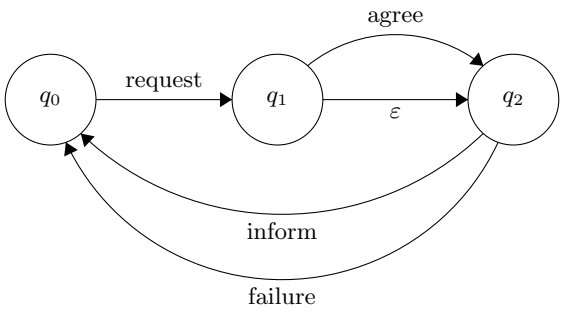

Figure 2.29: Achieve Rational Effect protocol responder FSM: $q_0 = \texttt{receive} \in S$ is the initial state, $q_1 = \texttt{send reply} \in S$ and $q_2 = \texttt{agreed} \in S$. The query protocol FSM is analogous, with the only differences in the label request which change in query–if/ref.

with:

$$P' \subseteq p; \quad c_0 = \langle m = \texttt{receive}(t), \beta \rangle \in \Gamma_\delta;$$
$$\beta(b) = \langle \texttt{AchieveREResponder}, a, s_b, \texttt{receive} \rangle.$$

Note that there is no final states: this means that a responder waits indefinitely for new requests or queries. In JADE, such a behaviour can be stopped by killing the associated agent with an external event.

If b is an achieve rational effect responder, then the γ transition function is defined as follows:

$$\gamma(q_0, b) = m = \texttt{receive}(t) \quad \gamma(q_1, b) = m = \texttt{handleRequest}(req)$$
$$\gamma(q_2, b) = m = \texttt{prepareResultNotification}(req, resp)$$

where $m, req, resp$ denote messages and t denotes a template.

Then, the rules for both *Request* and *Query* protocols are specified, which are conditions for (ProtocolRuleSchema). In case of $r = \texttt{Responder}$ and $q = \texttt{REQUEST}$,

the rule conditions are the following.

$$
\begin{aligned}
B_q^{(r,q_0)} &: t = \texttt{request} \\
B_q^{(r,q_1)} &: \mathscr{M}[\![req, \texttt{request}]\!]\mu = \top \\
&\wedge \mathscr{M}[\![resp, (\texttt{agree} \vee \texttt{refuse})]\!]\mu = \top \\
B_q^{(r,q_2)} &: \mathscr{M}[\![req, \texttt{request}]\!]\mu = \top \\
&\wedge \mathscr{M}[\![resp, (\texttt{agree} \vee \texttt{refuse})]\!]\mu = \top \\
&\wedge \mathscr{M}[\![m, (\texttt{inform} \vee \texttt{failure})]\!]\mu = \top
\end{aligned}
$$

where $m, req, resp$, and t are the messages and the template of the γ function. The above conditions define three rules for the responder role in request protocol.

When $r = \texttt{Responder}$ and $q = \texttt{QUERY}$, the conditions change in the following.

$$
\begin{aligned}
B_q^{(r,q_0)} &: t = \texttt{query} - \texttt{if} \vee \texttt{query} - \texttt{ref} \\
B_q^{(r,q_1)} &: \mathscr{M}[\![req, (\texttt{query} - \texttt{if} \vee \texttt{query} - \texttt{ref})]\!]\mu = \top \\
&\wedge \mathscr{M}[\![resp, (\texttt{agree} \vee \texttt{refuse})]\!]\mu = \top \\
B_q^{(r,q_2)} &: \mathscr{M}[\![req, (\texttt{query} - \texttt{if} \vee \texttt{query} - \texttt{ref})]\!]\mu = \top \\
&\wedge \mathscr{M}[\![resp, (\texttt{agree} \vee \texttt{refuse})]\!]\mu = \top \\
&\wedge \mathscr{M}[\![m, (\texttt{inform} \vee \texttt{failure})]\!]\mu = \top
\end{aligned}
$$

where, as usual, $m, req, resp$ and t are the messages and the template defined by γ.

2.9 Contract Net Interaction Protocol

Another example of JADE implementation of interaction protocol is based on *FIPA Contract Net* interaction protocol. The implementation consists in two classes called `ContractNetInitiator` and `ContractNetResponder`, both extension of the class `FSMBehaviour`.

An intuitive explanation of the protocol is that a number of agents are involved in a *call of proposal* (*cfp*). In fact, the initiator first sends a cfp to $m \in \mathbb{N}$ other agents, the participants. Only n of the m participants reply to the initiator, and they can refuse

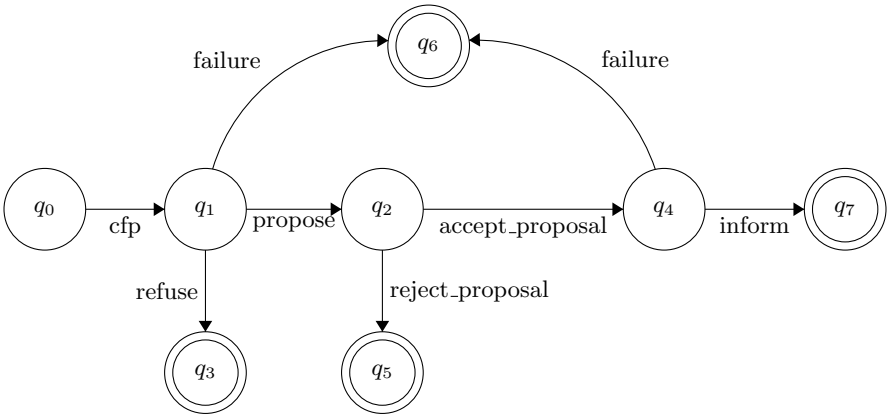

Figure 2.30: Contract Net protocol initiator FSM: $q_0 = \texttt{send cfp} \in S$ is the initial state, $q_1 = \texttt{receive reply} \in S$, $q_2 = \texttt{proposed} \in S$, $q_4 = \texttt{accepted} \in S$, and $q_3 = \texttt{refused} \in S$, $q_5 = \texttt{rejected} \in S$, $q_6 = \texttt{failure} \in S$, $q_7 = \texttt{success} \in S$ are final states.

the call or respond with a proposal. Suppose that $i \leq n$ is the number of the refusing participants and $j = n - i$ the number of the proposals. Then the initiator examines all the j replies and choose which to accept. Indicating with $k \leq j$ the number of refused proposals and with $l = j - k$ the number of those accepted, the l participants start to perform the action specified in their proposal. Finally, they send to the initiator the result of their action, which may be a failure or a success.

2.9.1 Contract Net Initiator

The formalization of the Contract Net initiator implementation is given as in the previous section, using the interaction protocol transition system.

$$\text{ContractNetInitiator} = \langle c_0, \Gamma_\delta, P', \delta, \rightarrow_\delta, \Lambda_\delta \rangle$$

with:

$$P' \subseteq p; \quad c_0 = \langle \texttt{send}(m), \beta \rangle \in \Gamma_\delta;$$
$$\beta(b) = \langle \texttt{ContractNetInitiator}, a, s_b, \texttt{send cfp} \rangle.$$

The δ transition function is defined as in Figure 2.30. Similarly to the Section 2.8.1 initiator, there are some final states that are indistinguishable from others (all states belong to S), then it is necessary to add a fake final state $q_8 = \texttt{final}$ and some ε-transitions to the function δ, namely $\delta(q_3,\varepsilon) = \delta(q_5,\varepsilon) = \delta(q_6,\varepsilon) = \delta(q_7,\varepsilon) = q_8$.

If $b \in B$ is a contract net initiator, then the γ transition function definition is given below.

$$\gamma(q_0,b) = \texttt{send}(m) \quad \gamma(q_1,b) = m = \texttt{receive}(t)$$
$$\gamma(q_2,b) = \texttt{handlePropose}(\mathit{propose},\mathit{acceptances})$$
$$\gamma(q_3,b) = \texttt{handleRefuse}(m) \quad \gamma(q_4,b) = \gamma(q_1,b)$$
$$\gamma(q_5,b) = \varepsilon \quad \gamma(q_6,b) = \texttt{handleFailure}(m)$$
$$\gamma(q_7,b) = \texttt{handleInform}(m) \quad \gamma(q_8,b) = \varepsilon$$

where $m, \mathit{propose}$ denote messages, $\mathit{acceptances}$ denotes a list of messages (the accepted proposal messages) and t denotes a template.

The eight rules which specialize the (ProtocolRuleSchema) are shown in the following.

$$B_q^{(r,q_0)} : \mathscr{M}[\![m, \texttt{cfp}]\!]\mu = \top$$
$$B_q^{(r,q_1)} : t = (\texttt{propose} \vee \texttt{refuse} \vee \texttt{failure})$$
$$B_q^{(r,q_2)} : \mathscr{M}[\![\mathit{propose}, \texttt{propose}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![\mathit{acceptance}, (\texttt{accept} - \texttt{proposal} \vee \texttt{reject} - \texttt{proposal})]\!]\mu = \top$$

$$B_q^{(r,q_3)} : \mathscr{M}[\![m, \texttt{refuse}]\!]\mu = \top$$
$$B_q^{(r,q_4)} : t = \texttt{inform} \vee \texttt{failure}$$
$$B_q^{(r,q_6)} : \mathscr{M}[\![m, \texttt{failure}]\!]\mu = \top$$
$$B_q^{(r,q_7)} : \mathscr{M}[\![m, \texttt{inform}]\!]\mu = \top$$

where $m, t, \mathit{propose}, \mathit{acceptance}$ are defined by γ.

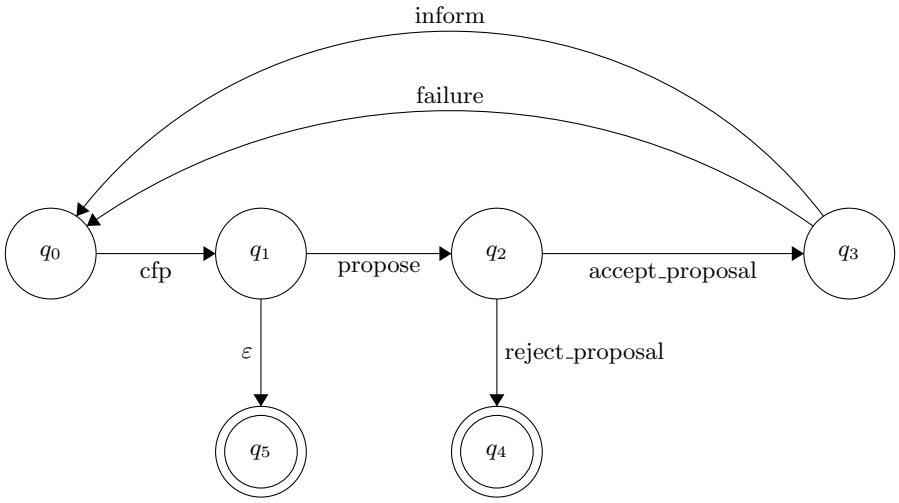

Figure 2.31: Contract Net protocol responder FSM: $q_0 = \texttt{receive cfp} \in S$ is the initial state, $q_1 = \texttt{send reply} \in S$, $q_2 = \texttt{receive next} \in S$, $q_3 = \texttt{accepted} \in S$, and $q_4 = \texttt{refused} \in S$ is a final states.

2.9.2 Contract Net Responder

The Contract Net responder formalization is the interaction protocol transition system shown in the following.

$$\text{ContractNetResponder} = \langle c_0, \Gamma_\delta, P', \delta, \rightarrow_\delta, \Lambda_\delta \rangle$$

with:

$$P' \subseteq p; \quad c_0 = \langle m = \texttt{receive}(t), \beta \rangle \in \Gamma_\delta;$$
$$\beta(b) = \langle \texttt{ContractNetResponder}, a, s_b, \texttt{receive cfp} \rangle.$$

As usual, the FSM transition function δ is defined in Figure 2.31. It is worth noting that it has two final states, but it also may not terminate its execution. The final states are handled by adding a transition $\delta(q_4, \varepsilon) = \delta(q_5, \varepsilon) = q_6$ where $q_6 = \texttt{final}$ is a final state.

If $b \in B$ is a contract net responder, then the γ transition function is defined as follows.

$$\gamma(q_0,b) = m = \texttt{receive}(t) \quad \gamma(q_1,b) = m = \texttt{handleCfp}(c)$$
$$\gamma(q_2,b) = \gamma(q_0,b)$$
$$\gamma(q_3,b) = m = \texttt{handleAcceptProposal}(c,p,a)$$
$$\gamma(q_4,b) = \texttt{handleRejectProposal}(c,p,r)$$
$$\gamma(q_5,b) = \gamma(q_6,b) = \varepsilon$$

where m,c,p,r,a are messages and t a template.

Five interaction protocols rules are specified by putting the following conditions in (ProtocolRuleSchema).

$$B_q^{(r,q_0)} : t = \texttt{cfp}$$
$$B_q^{(r,q_1)} : \mathscr{M}[\![c,\texttt{cfp}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![m,(\texttt{propose} \vee \texttt{refuse} \vee \texttt{failure})]\!]\mu = \top$$
$$B_q^{(r,q_2)} : t = \texttt{accept} - \texttt{proposal} \vee \texttt{reject} - \texttt{proposal}$$
$$B_q^{(r,q_3)} : \mathscr{M}[\![c,\texttt{cfp}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![p,\texttt{propose}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![a,\texttt{accept} - \texttt{proposal}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![m,(\texttt{inform} \vee \texttt{failure})]\!]\mu = \top$$
$$B_q^{(r,q_4)} : \mathscr{M}[\![c,\texttt{cfp}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![p,\texttt{propose}]\!]\mu = \top$$
$$\wedge\, \mathscr{M}[\![r,\texttt{reject} - \texttt{proposal}]\!]\mu = \top$$

where m,c,p,r,a,t are the same of the γ function definition.

2.10 System Start-Up and Termination

As defined in Section 2.3.2, there are two external events which denotes the start-up and termination phase of the whole MAS, namely Start(MAS), and End(MAS).

A computation of a MAS consists in a sequence of configurations and events, agent and behaviour creations, managed by the main JADE transition system, as detailed in previous Section 2.4. A JADE 'program' is written as a list of classes, and the transition system does nothing until the MAS starts. In the start-up phase, objects, agents, behaviours and messages stores are initialized as empty stores. When an event End(MAS) occur, the MAS terminates. This aspect is formalized in rule (MAS-End). The final configuration consists in a tuple where the symbol `E` is used in place of a command to indicate termination. If there is a command to be computed when the event End(MAS) occurs, such a command is actually executed. Stores are updated according to the system com, and no event occurs in this phase.

$$\frac{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{\varpi}_{\text{com}} \langle \varepsilon,\sigma',\alpha',\beta',\mu'\rangle}{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{\text{End(MAS)}} \langle \mathtt{E},\sigma',\alpha',\beta',\mu'\rangle} \qquad \text{(MAS-End)}$$

During the execution of the main system, some events may occur at the same time. This happens, for example, when two agents are active on the same MAS, and they can be subject of internal events. Moreover, events may occur during the selection of a behaviour for the next action of the agent. Two events E_0 and E_1 which occur at the same time are indicated with the notation $E_0||E_1$.

$$\frac{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E_0} \langle c',\sigma',\alpha',\beta',\mu'\rangle}{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E_0||E_1} \langle c',\sigma',\alpha',\beta',\mu'\rangle} \qquad \text{(E-parallel1)}$$

$$\frac{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E_1} \langle c',\sigma',\alpha',\beta',\mu'\rangle}{\langle c,\sigma,\alpha,\beta,\mu\rangle \xrightarrow{E_0||E_1} \langle c',\sigma',\alpha',\beta',\mu'\rangle} \qquad \text{(E-parallel2)}$$

The above rules (E-parallel1) and (E-parallel2) describe how parallel events can be processed. Such rules are symmetric, and they introduce non-determinism in the computation of a MAS. Moreover, events can be also parallel to a behaviour selection, e.g. $S(a) = b$, which occurs when both the agent a and the behaviour b are `active`. In such a case, the action selection and the event processing are executed simultaneously, following rules similar to those for parallel events. How the system resolves parallel event occurrences or behaviour selection, i.e., what event is processed as first,

is unknown. Rules for parallelism reflects such a uncertainty. This is not surprising, because a JADE MAS is, in fact, a distributed system. Sometimes, parallel events can have side effects. For example, they can update one or more stores. How stores are modified depends on what event occurs first, so it is undecidable. If events update simultaneously different stores, how they are updated is not important. Otherwise, if two or more events modify the same store, and, in particular, the same location, e.g., of an agent a, the final value of such an agent depends on the events order of execution. As stated before, such an order is uncertain, so the final value of a can not be predetermined. This non-determinism can not be avoided when deal with distributed system such as JADE.

2.11 A Complete Example

A complete example of a MAS computation is shown in Figure 2.32. The example refers to the agents a_0 and a_1 of Section 2.6, showing in detail their life cycles.

First, the agent a_0 of class C_0 is created, as a consequence of the Create(a_0, C_0) event, and immediately calls its `setup` method, according to rule (A-Create). The description of agent set-up was shown in Figure 2.23.

Then, because the behaviour b_0 was activated during a_0 set-up, and both a_0 and b_0 are in the `active` state, the behaviour scheduler is able to choose b_0 as current action for a_0. Meanwhile, the event Create(a_1, C_1) occurs, causing the creation of a second agent of class C_1. Two rules, namely (B-Selected) and (A-Create), are applicable at the current configuration. The two described transition are parallels, so how stores are updated is unknown. But they affect different locations, so the order of execution is not important. In Figure 2.32, such transitions are illustrated. Action of b_0 follows the steps shown in Figure 2.24, blocking the behaviour b_0, because a_0 message queue is still empty. At the same time, the agent a_1 performs its setup, activating the behaviour b_1.

At the end of b_0 action and a_1 set-up, the behaviour scheduler choses b_1 as the only possible action for the agent a_1, and no action for a_0. This is the only possibility, due to the `blocked` state of b_0. Action of b_1 is then performed, as shown in

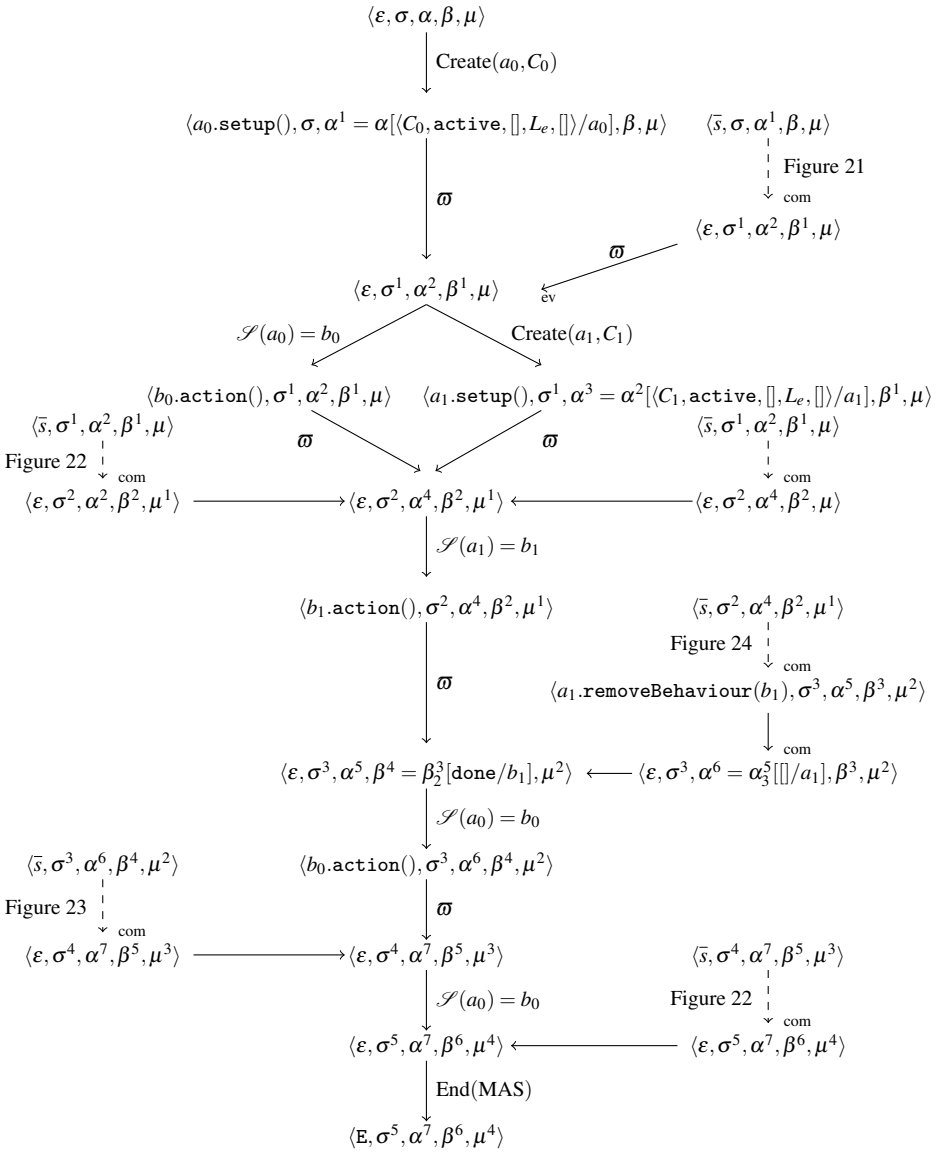

Figure 2.32: A complete example of a JADE MAS.

Figure 2.26, and a message m is sent, adding it to a_0 message queue. Because b_1 was declared as one-shot behaviour, its action is followed by a $a_1.\texttt{removeBehaviour}(b_1)$ command, as in rule (B-OneShot), which causes the emptying of the agent a_1 behaviours list. Then, the state of the behaviour b_1 is updated to `done`.

Moreover, the sending of the message m reactivates all blocked behaviour, due to the event Message(m), as stated by the rules (M-Send) and (M-Event). In this case, only the behaviour b_0 is reactivated. This means that the behaviour scheduler is able to choose it for the next action of a_0. Now, its action follows the steps reported in Figure 2.25, extracting the message from the queue of a_0 and processing it. At the end of such an action, behaviour b_0 and agent a_0 are still both in their `active` states, so the scheduler selects again b_0 for the next action. The last action found an empty message queue, and it blocks the behaviour b_0. Finally, an event End(MAS) occurs, causing the system to proceed into a terminal configuration.

2.12 Purposes and Future Directions of the Formalization

In this chapter, the JADE platform was analyzed and its main feature were formalized using LTS. The formalization focuses on a set of selected methods of JADE APIs, thus highlighting the core features that the platform provides. The formalization has the purpose of delineating a precise direction in the development of JADEL as agent-based language built on top of the JADE platform. As a matter of fact, JADEL is meant to be a *Domain-Specific Language (DSL)* that translates directly into JADE code. Hence, the JADEL semantics relies heavily on the JADE one. In the next Chapter, JADEL syntax categories are defined, and rules that specify the formal semantics of JADEL are shown in the same fashion of the JADE formalization.

Chapter 3

The JADEL Programming Language

Third Law. A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

– Isaac Asimov

In this chapter, the main design decisions and the technical details behind the development of the *JADE Language (JADEL)* are discussed. In particular, JADEL must be a *Domain-Specific Language (DSL)* designed to provide the developer with agent-oriented abstractions and to reduce the complexity of using the *Java Agent DEvelopment Framework (JADE)* by means of a lighter and dedicated syntax. The effort in developing a DSL for JADE agents aims at facilitating code readability and dropping many implementation details, in favor of a lighter syntax that focuses on agent-oriented abstractions only, thus improving its compliance with the *Agent-Oriented Programming (AOP)* paradigm.

The choice of describing JADEL as a DSL for agents and *Multi-Agent Systems (MASs)*, and the fact that the current implementation of its tools include an optimizing compiler and a rich text editor for the Eclipse *IDE (Integrated Development Environ-*

ment), emphasize the fact that JADEL targets real-world applications in the scope of *Model-Driven Engineering (MDE)*.

A description of the initial version of JADEL can be found in [Bergenti, 2014], where its main abstractions are described and the overall motivations of the work are explained. The paper [Bergenti, 2014] also outlines the translation of a JADEL source code to a semantically equivalent Java source code that uses JADE. The language described in this dissertation is a new version of JADEL, first presented in [Bergenti et al., 2017a]. This version enhances the original work by gathering the results of preliminary works [Bergenti et al., 2016a,d,c] in terms of syntax improvements, new features, and a better integration with Java.

3.1 An Agent-Oriented Domain-Specific Language

JADEL is meant to be a DSL for AOP capable of finding widespread applicability in the scope of real-world *Model-Driven Development (MDD)*. This is the reason why JADEL is designed around the features of *Xtext*[1], a framework which provides effective support for the development of DSLs [Eysholdt and Behrens, 2010; Bettini, 2013]. The use of Xtext facilitates the design of a DSL because it eases the main steps involved in such a task, e.g., the creation of the grammar and the implementation of the compiler. This section first introduces Xtext and summarizes some its main features, then it shows how such features are used in the design of JADEL.

3.1.1 Xtext and Xtend

Xtext is a software framework intended to support language developers, and it is meant to help the definition of DSLs that adhere to the following criteria [Eysholdt and Behrens, 2010; Bettini, 2013]:

1. Programming in a DSL must be efficient for its users, because they probably have to spend much time in doing it;

[1] `www.eclipse.org/Xtext`

2. The gap between user skills and language abstractions must be reduced as much as possible, facilitating the understanding of language concepts by programmers;

3. The interaction among sources written from different developers must be transparent, thus reducing the chance of misunderstanding and faulty code;

4. Tools to read, browse, search, and compare source codes are needed;

5. The code must be easy to maintain in order to remain for long time in production without requiring too much effort by developers to fix it when a change happens.

The features of Xtext cover many aspects of language design and implementation. First, Xtext provides a DSL to express *Extended Backus-Naur Form (EBNF)* grammars, from which a parser can be easily obtained with the use of the well-known parser generator *ANTLR (ANother Tool for Language Recognition)*[2]. Moreover, Xtext provides a base grammar, called *Xbase* grammar [Efftinge et al., 2012], which is highly extensible and it is used to implement the basic features of the *Xtend* language[3], such as expressions, and type references. Xtend is a dialect of Java, and this means that its syntax and semantics rely on those of Java, but specific syntactic facilities are provided to make it lighter and simpler. An important feature of Xtend is its complete interoperability with Java source code, due to the fact that there is an explicit one-to-one mapping between an Xtend source code and a semantically equivalent Java source code. In particular, the use of Xtext and Xtend to create the grammar of JADEL turned out to be appropriate for the intended scope: domain-specific agent-oriented elements are easily added to a solid base of rules that defines Xtend core features, ensuring a tight integration with Java, and therefore a tight integration with JADE. As a matter of fact, JADEL relies on Xtend expressions rather than introducing a new syntax. This choice has the advantage of grounding JADEL on a solid grammar for expressions whose primary goal is to support the construction

[2] `www.antlr.org`

[3] `www.eclipse.org/xtend`

of procedural languages. Moreover, it greatly simplifies the construction of tools like compilers.

Second, Xtext framework comes together with the latest versions of the Eclipse IDE, thus providing a complete integration with such an important tool. In fact, Xtext offers editing support for Eclipse by allowing the customization of the outline view, of the behaviour of content proposals, of quick fixes, and, above all, by providing a language editor with DSL-based syntax highlighting. JADEL was meant to be implemented with Xtext also because of this last feature, which is necessary for JADE programmers who are used to work with professional tools.

Finally, since Xtext uses the *Eclipse Modeling Framework (EMF)*, the managed abstract syntax trees are all EMF models. This fact ensures easy integration with Eclipse modeling tools, and provides a solid support for MDD. Actually Xtext offers specific APIs to generate Java code from generated EMF models. The use of such APIs ensures that all JADEL entities are easily mapped into Java classes, methods, and fields, while it guarantees that JADEL specific constructs and expressions are readily compiled into Java snippets.

3.2 JADEL Core Abstractions

As discussed in the previous chapter, JADE provides a number of abstractions, and related Java classes, for the construction of MASs. JADEL selects only a few primary abstractions among them in order to offer to the developer a clear view of agents and MASs. More precisely, only four main abstractions that JADE implements were chosen:

1. *Agents*;
2. *Behaviours*;
3. *Communication ontologies*; and
4. *Interaction Protocols*.

The selection of such abstractions is a design choice for JADEL, even though the selected abstractions do not represent a comprehensive list of what JADE offers. The following considerations are the basis for such a selection.

1. JADEL surely need an agent abstraction which allows the description of JADE agent classes because agents are fundamental atoms in all JADE systems;

2. The behaviour becomes the second abstraction, because JADE uses behaviours to handle actions and tasks of the agent; and

3. JADE provides the *FIPA (Foundation for Intelligent Physical Agents)*[4] compliancy by providing support to agent communication and, in particular, to interaction protocols. For this reason, communication ontologies and interaction protocols become JADEL abstractions.

Agents must have features to permit the programmer to correctly manage the three phases of the life cycle of an agent: initialization, control loop and take-down. During the initialization phase the agent usually activates its behaviours, and it executes their actions in the control loop until the take-down phase is possibly entered. Finally, actions must handle events and communications, with the aid of ontologies and specific interaction patterns provided by interaction protocols. The four chosen abstractions are presented in details in the rest of this section.

3.2.1 Agents

Agents in JADEL are the main entities. They use ontologies and behaviours, and they take roles in interaction protocols. As a matter of fact, the declaration of an agent uses some JADEL expressions especially defined to ease common tasks, such as behaviours activation and deactivation, messages creation and sending, and so on. The definition of such domain-specific expressions relies on the Xbase grammar, which uses Xtend expressions. In Figure 3.1, the JADEL grammar that extends Xtend expression is shown, using the EBNF language, where

[4] `www.fipa.org`

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{xexpr} \mid \mathit{actb} \mid \mathit{dactb} \mid \mathit{extr} \mid \mathit{send} \mid \mathit{creat} \mid \mathit{role} \\
\mathit{actb} & ::= & \texttt{activate behaviour } \overline{x\ \texttt{as}}^{?}\ b(\overline{\mathit{expr}}^{*}) \\
\mathit{dactb} & ::= & \texttt{deactivate behaviour } \overline{x\ \texttt{as}}^{?}\ b(\overline{\mathit{expr}}^{*}) \\
\mathit{role} & ::= & \texttt{take role } \overline{x\ \texttt{as}}^{?}\ r(\overline{\mathit{expr}}^{*}) \\
\mathit{extr} & ::= & \texttt{extract } x\ \texttt{as } t\ \overline{\texttt{from } y}^{?} \\
\mathit{creat} & ::= & \texttt{create } \mathit{msg} \\
\mathit{send} & ::= & \texttt{send } \mathit{msg} \\
\mathit{msg} & ::= & \texttt{message } \overline{m}^{?}\ \{\ \overline{\mathit{msgexpr}}^{*}\ \} \\
\mathit{msgexpr} & ::= & \mathit{pexpr} \mid \mathit{oexpr} \mid \mathit{cexpr} \mid \mathit{rexpr} \\
\mathit{oexpr} & ::= & \texttt{ontology is } o \\
\mathit{cexpr} & ::= & \texttt{content is } x \\
\mathit{rexpr} & ::= & \texttt{receivers are } l \\
\mathit{pexpr} & ::= & \texttt{performative is INFORM} \mid \texttt{REQUEST} \mid \ldots
\end{array}
$$

Figure 3.1: JADEL grammar for extended Xtend expressions. *xexpr* refers to standard Xtend expressions, metavariables x, y denote JADEL variables, b denotes a behaviour, t denotes the name of a type, m denotes a message, o denotes an ontology, and l denotes a list of recipient of a message.

1. $\overline{X}^{*}$ stands for the repetition of X zero or more times;
2. $\overline{X}^{+}$ stands for the repetition of X one or more times; and
3. $\overline{X}^{?}$ stands for optional, namely the occurrence of zero or one X.

A syntax category is defined for each new expression, while Xtend expressions are denoted as *xexpr* and not further detailed. As previously mentioned, an expression can be an *xexpr* or it can be one of the following:

1. A behaviour activation, where a variable name x could be specified for the behaviour and b is actually the class name of the behaviour;
2. A behaviour deactivation;

$$
\begin{array}{lcl}
adecl & ::= & \texttt{agent}\ a\ (\overline{t\ x}^{*})\ \overline{onto}^{?}\ \overline{\texttt{extends}\ a_{base}}^{?} \\
 & & \{\ \overline{field}^{*}\ \overline{method}^{*}\ \overline{aevent}^{+}\ \} \\
onto & ::= & \texttt{uses ontology}\ o \\
aevent & ::= & \texttt{on}\ event\ \{\ \overline{expr}^{*}\ \} \\
event & ::= & \texttt{create} \mid \texttt{destroy} \\
field & ::= & (\texttt{var} \mid \texttt{val})\ \vec{t}^{?}\ f\ \overline{= expr}^{?} \\
method & ::= & t\ m(\overline{t\ mpar}^{*})\ \{\ \overline{expr}^{*}\ \}
\end{array}
$$

Figure 3.2: JADEL grammar for agents. Metavariables a, a_{base} denote agents, o denotes an ontology name, t denotes a type name, x denotes a variable, f denotes the name of a field, m denotes the name of a method, $mpar$ denotes the name of a parameter, while $expr$ are extended expression defined in Figure 3.1.

3. A content extraction, where x is the name of the variable for containing the object of type t, extracted from the message y;

4. A message sending expression, where *msg* is a specific construct for defining a message in-line;

5. A message creation expression, that also uses the *msg* construct; and

6. A role activation, that operates exactly as a behaviour activation but r is the class name of a role in an interaction protocol.

The syntax category *msg* defines a first construct, useful for defining a message as a data structure. Such a data structure is composed of an ontology, a content, a recipients list, and a performative. Such components are defined using the *oexpr*, *cexpr*, *rexpr* and *pexpr* syntaxes, respectively.

Given the syntax of expressions, JADEL provides a grammar for agents, shown in Figure 3.2. JADEL aims at making the declarations of agents clearer than semantically equivalent declarations made with JADE by means of a lighter syntax, and at highlighting clearly the connections of the agent with other entities. As a matter of fact, an ontology could be specified in the agent definition by means of the

$$\frac{\texttt{agent}\ A_1(\overline{t\ x}^*)\ \texttt{extends}\ A_2\{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\}}{A_1 <: A_2 <: \texttt{Agent}} \quad \text{(A-JADE)}$$

$$\frac{\texttt{agent}\ A_1(\overline{t\ x}^*)\ \texttt{extends}\ A_2\{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad fields(A_2) = \overline{G}^*}{fields(A_1) = \overline{t\ x}^*\ \overline{F}^*\ \overline{G}^*} \quad \text{(A-Fields)}$$

$$\frac{\begin{array}{c}\texttt{agent}\ A_1(\overline{t\ x}^*)\ \texttt{uses ontology}\ O_1\ \texttt{extends}\ A_2\{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \\ ontologies(A_2) = \overline{O}^*\end{array}}{ontologies(A_1) = O_1\ \overline{O}^*} \quad \text{(A-Onto)}$$

$$\frac{\texttt{agent}\ A\ \ldots\ \{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad \texttt{on create}\ \{\overline{expr}^*\} \in \overline{aevent}^+}{mtype(\texttt{setup}, A) = \varepsilon \to \varepsilon \quad mbody(\texttt{setup}, A) = \langle \varepsilon, \overline{expr}^* \rangle} \quad \text{(A-Setup)}$$

$$\frac{\texttt{agent}\ A\ \ldots\ \{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad \texttt{on destroy}\ \{\overline{expr}^*\} \in \overline{aevent}^+}{mtype(\texttt{takeDown}, A) = \varepsilon \to \varepsilon \quad mbody(\texttt{takeDown}, A) = \langle \varepsilon, \overline{expr}^* \rangle} \quad \text{(A-TakeDown)}$$

Figure 3.3: Rules that specify the operational semantics of JADEL agents.

syntax category *onto*. The declaration of an agent is allowed to extend the declaration of another agent, with the usual semantics of inheritance. Two event handlers are provided to support initialization and take-down phases, namely the `on create` and `on destroy` constructs. Moreover, fields, methods and event handlers rely on the *expr* syntax category. As a result, behaviours and roles can be activated in such initialization and take-down phases, but they can also be activated inside agent methods, since activation is an expression which was part of extended expressions.

The semantics of JADEL is formally defined by means of operational rules and auxiliary lookup functions [Bergenti et al., 2017c]. The three lookup functions from the operational semantics of *Featherweight Java (FJ)* [Igarashi et al., 2001], namely $fields$, *mtype* and *mbody*, are used to connect the semantics of the agent-oriented features of JADEL with the semantics of the host language, *Xtend*, because it is a syntactic dialect of Java. Actually, the agent abstraction that JADEL provides is mapped

into a Java class that derives from `Agent`, as stated by the rule (A-JADE). Such a class is provided by JADE in its API and, obviously, it has fields and methods. In detail, in the operational semantics of FJ, the $fields$ lookup function associates each class name with its own fields plus inherited fields. For JADEL agents, $fields$ works exactly as in FJ, as shown by rule (A-Fields) in Figure 3.3, but it adds to the fields also the agent parameters. Such parameters are then filled with the arguments passed at agent instantiation. There are other differences between FJ classes and JADEL agents. For example, the two agent event handlers `on create` and `on destroy` implicitly provide two methods, as shown by the rules (A-Setup) and (A-TakeDown) in Figure 3.3, which are not part of FJ.

Methods are identified by means of the two functions *mtype* and *mbody*. The first function takes the name of the method and the name of the class, and returns a mapping between the parameter types and the return type of the method. When the return type is `void`, or there are no parameters, ε is conventionally used. The second function, *mbody*, also takes the name of the method and the name of the class, and it returns a pair, whose first element is a list of parameters, and whose second element is the actual body of the method. The definition of *mtype* and *mbody* for JADEL agents is the same that of FJ. An additional auxiliary lookup function is defined for ontologies. The function *ontologies* takes an agent and it returns a list of ontologies, as in rule (A-Onto), when an ontology is specified by the declaration `uses ontology`.

3.2.2 Behaviours

On the basis of the syntax for extended expressions shown in Figure 3.1, JADEL provides the specific syntax for behaviours shown in Figure 3.4. Two types of behaviours are allowed, namely `cyclic` and `oneshot`. Notably, the semantics of such types of behaviour is same with JADE cyclic and one-shot behaviours, respectively. A behaviour can be specific to a group of agents, i.e., it can take advantage of the common characteristics of such agents in the definition of its action, by means of the domain-specific syntax `for` a, where a denotes the name of a declared agent. Also, a behaviour can refer to a specific communication ontology, with the aid of the *onto* declaration. The body of a behaviour contains a set of fields, a set of methods and a

$$
\begin{array}{lcl}
\mathit{bdecl} & ::= & \mathit{btype}\ \texttt{behaviour}\ b\ (\overline{t\ x}^{*})\ \overline{\texttt{for}\ a}^{?}\ \overline{\mathit{onto}}^{?} \\
 & & \overline{\texttt{extends}\ b_{base}}^{?} \\
 & & \{\ \overline{\mathit{field}}^{*}\ \overline{\mathit{method}}^{*}\ \overline{\mathit{bevent}}^{+}\ \} \\
\mathit{btype} & ::= & \texttt{cyclic} \mid \texttt{oneshot} \\
\mathit{bevent} & ::= & \texttt{on message}\ \overline{m}^{?}\ \overline{\texttt{when}\ \mathit{wexpr}}^{?}\ \texttt{do}\ \{\ \overline{\mathit{expr}}^{*}\} \\
 & \mid & \texttt{do}\ \{\ \overline{\mathit{expr}}^{*}\} \\
\mathit{wexpr} & ::= & \mathit{wexpr}\ \texttt{or}\ \mathit{wexpr} \mid \mathit{wexpr}\ \texttt{and}\ \mathit{wexpr} \mid \\
 & \mid & \texttt{not}\ \mathit{wexpr} \mid \mathit{pexpr} \mid \mathit{oexpr} \mid \mathit{cexpr}
\end{array}
$$

Figure 3.4: JADEL grammar for behaviours. Metavariables b, b_{base} denote behaviours, t denotes a type, x denotes a variable, a denotes an agent type, o denotes an ontology, m denotes a message, while *expr*, *cexpr*, *pexpr*, and *oexpr* are extended expressions in shown Figure 3.1.

non-empty set of event handlers. In fact, at least one event handler must be present in order to define the *action* of the behaviour. Behaviours can extend other behaviours, with the usual semantics of sub-classing, and all event handlers of the base behaviour are added to the resulting derived behaviour.

Event handlers offer a way to perform an action upon the activation of a behaviour by means of the keyword `do`, or they can manage incoming message by using the construct `on when do`. The name of the message to be processed can be specified after the `on message` keywords, and a message template can be constructed by combining the *when expressions* defined by the syntax category *wexpr*.

The operational semantics of JADEL behaviours is shown in Figure 3.5, in Figure 3.6 and in Figure 3.7. The behaviour abstraction of JADEL is mapped into a Java class that derives from `OneShotBehaviour` or `CyclicBehaviour`, depending on the specified type of behaviour, as shown in rules (B-OneShot) and (B-Cyclic). Behaviour fields, instead, are obtained not only by the user declared fields, but also by behaviour parameters, and there is an implicitly declared field `theAgent`, which identifies the agent that is currently using a behaviour, as shown in rules (B-Fields) and (B-FieldsA) of the Figure 3.5. An additional auxiliary lookup function is defined

$$\frac{\texttt{oneshot behaviour } B_1(\overline{t\ x}^*) \texttt{ extends } B_2\{\overline{F}^*\ \overline{M}^*\ \overline{bevent}^+\}}{B_1 <: B_2 <: \texttt{OneShotBehaviour}} \quad \text{(B-OneShot)}$$

$$\frac{\texttt{cyclic behaviour } B_1(\overline{t\ x}^*) \texttt{ extends } B_2\{\overline{F}^*\ \overline{M}^*\ \overline{bevent}^+\}}{B_1 <: B_2 <: \texttt{CyclicBehaviour}} \quad \text{(B-Cyclic)}$$

$$\frac{BT \texttt{ behaviour } B_1(\overline{t\ x}^*) \texttt{ extends } B_2\{\overline{F}^*\ \overline{M}^*\ \overline{bevent}^+\} \quad fields(B_2) = \overline{G}^*}{fields(B_1) = \overline{t\ x}^*\ \overline{F}^*\ \overline{G}^*} \quad \text{(B-Fields)}$$

$$\frac{BT \texttt{ behaviour } B_1(\overline{t\ x}^*) \texttt{ for}A \texttt{ extends } B_2\{\overline{F}^*\ \overline{M}^*\ \overline{bevent}^+\} \quad fields(B_2) = \overline{G}^*}{fields(B_1) = \overline{t\ x}^*\ \overline{F}^*\ \overline{G}^*\ a}$$

where a is the field A `theAgent` $= (A)$ `myAgent`; (B-FieldsA)

$$\frac{BT \texttt{ behaviour } B_1 \texttt{ extends } B_2\{\ldots \overline{bevent}^+\} \quad events(B_2) = \overline{bevent_2}^+}{events(B_1) = \overline{bevent}^+\ \overline{bevent_2}^+} \quad \text{(B-Events)}$$

Figure 3.5: Rules that specify the operational semantics of JADEL behaviours.

for events. The function *events* maps the name of a behaviour with its list of declared events. It is worth noting that the list of events is not limited to the event handlers that are specified in the behaviour, but it also contains inherited events, as shown in rule (B-Events). The management of events also requires the definition of such inherited events, even if they are not JADEL abstractions, at least explicitly. In fact, in JADEL, event handlers are translated into inner classes of the host behaviour, and they are composed of specific fields and methods, which collectively define the actual action of the behaviour.

In Figure 3.6, rules (E-Inner0) and (E-Inner) define the *innerclasses* lookup function, which takes the name of a behaviour and a list of events, and it returns a pair whose first element is the definition of the current inner class plus the already defined inner classes, and whose second element is the number of the processed events. Finally, the `action` method of the behaviour runs in sequence all behaviour event

$$\frac{\overline{bevent}^{*} = \emptyset}{innerclasses(B, \overline{bevent}^{*}) = \langle \varepsilon, 0 \rangle} \quad \text{(E-Inner0)}$$

$$\frac{bevent_n \in events(B) \quad innerclasses(B, \overline{bevent}^{*}) = \langle E, n \rangle}{innerclasses(B, bevent_n\, \overline{bevent}^{*}) = \langle \texttt{private class Event}n\{\overline{F}^{*}\ \overline{M}^{*}\}\ E, n+1 \rangle}$$
$$\text{where } \overline{F}^{*} \text{ and } \overline{M}^{*} \text{ depend on } bevent_n \quad \text{(E-Inner)}$$

$$\frac{events(B) = bevent_0 \ldots bevent_N}{mbody(\texttt{action}, B) = \langle \varepsilon, \texttt{super.action(); Event0.run(); ... Event}N\texttt{.run();} \rangle}$$
$$mtype(\texttt{action}, B) = \varepsilon \to \varepsilon \quad \text{(E-Action)}$$

Figure 3.6: Rules that specify the operational semantics of behaviour events.

handlers, which would typically check their condition and return immediately. It is worth noting that if there is more than one event handler with the same receiving conditions (i.e., `when` expression), and a message matching the conditions arrives, the order in which behaviours are triggered is not fixed. In fact, the `receive` method searches for such a message in the agent message queue, and extract it from the queue if it is found. Then, it processes the message, and it returns in the list of behaviours of the agent when finished. Meanwhile, if another behaviour tries the same `receive`, it does not find the message and it blocks. A blocked behaviour is reactivated at the reception of a new message. If such a new message matches again the conditions, there is no way to decide which behaviour should process the message. They could be again in the agent list, but they are managed by the agent internal scheduler and the developer has no control over it. Thus, it is important to state correctly the `when` conditions, and JADEL simple syntax helps in avoiding such mistakes.

Each inner class `Event`*n* has a list of fields and methods, which are identified by looking at the definition of the event handler. As shown in rule (E-Fields), if the event handler is in the form of the `on when do` construct, two fields are implicitly defined, namely, an `ACLMessage` field and a `MessageTemplate` field. Three meth-

$$\frac{bevent_n : \texttt{on message}\ m\ \texttt{when}\{wexpr\}\texttt{do}\{\overline{expr}^*\}}{fields(\texttt{Event}n) = \texttt{ACLMessage}\ m;\ \texttt{MessageTemplate}\ mt = wexpr;} \quad \text{(E-Fields)}$$

$$\frac{bevent_n : \texttt{on message}\ m\ \texttt{when}\{wexpr\}\texttt{do}\{\overline{expr}^*\}}{\begin{array}{c} mbody(\texttt{receive}, \texttt{Event}n) = \langle \varepsilon, m = \texttt{theAgent.receive(mt);} \rangle \\ mtype(\texttt{receive}, \texttt{Event}n) = \varepsilon \rightarrow \varepsilon \end{array}} \quad \text{(E-Receive)}$$

$$\frac{bevent_n : \texttt{on message}\ m\ \texttt{when}\{wexpr\}\texttt{do}\{\overline{expr}^*\}}{\begin{array}{c} mbody(\texttt{doBody}, \texttt{Event}n) = \langle \varepsilon, \overline{expr}^* \rangle \\ mtype(\texttt{doBody}, \texttt{Event}n) = \varepsilon \rightarrow \varepsilon \end{array}} \quad \text{(E-Do)}$$

$$\frac{bevent_n : \texttt{on message}\ m\ \texttt{when}\{wexpr\}\texttt{do}\{\overline{expr}^*\}}{\begin{array}{c} mbody(\texttt{run}, \texttt{Event}n) = \langle \varepsilon, \overline{runexpr}^* \rangle \\ mtype(\texttt{run}, \texttt{Event}n) = \varepsilon \rightarrow \varepsilon \end{array}} \quad \text{(E-Run)}$$

Figure 3.7: Rules for defining fields and methods of behaviour events.

ods of Event*n* are also defined, namely the `receive`, `doBody` and `run`. Those are all `void` methods without parameters. Rule (E-Receive) shows the definition of the `receive` method. The `doBody` method contains the Java translation of the expressions contained in the `do` block, as stated by the rule (E-Do). The `run` method contains the usual pattern for message reception, as documented in virtually all teaching material on JADE (see, e.g., [Bellifemine et al., 2007]), and it is reported in the following listing.

```
1 m = receive();
2 if (m != null) {
3     doBody();
4 } else {
5     block();
6 }
```

Such a sequence of Java statements is denoted by $\overline{runexpr}^*$ and used in the rule (E-Run), in Figure 3.7.

$$
\begin{array}{lcl}
\mathit{odecl} & ::= & \texttt{ontology}\ o\ \overline{\texttt{extends}\ o_{base}}^{?}\ \{\ \overline{\mathit{propdecl}}^{*}\ \overline{\mathit{cdecl}}^{*}\ \overline{\mathit{pdecl}}^{*}\ \} \\
\mathit{propdecl} & ::= & \texttt{proposition}\ \mathit{prop} \\
\mathit{cdecl} & ::= & \texttt{concept}\ c\ (\ \overline{\mathit{cpar}}^{*}\ )\ \overline{\texttt{extends}\ c_{base}}^{?} \\
\mathit{pdecl} & ::= & \texttt{predicate}\ p\ (\ \overline{\mathit{cpar}}^{*}\ )\ \overline{\texttt{extends}\ p_{base}}^{?} \\
\mathit{cpar} & ::= & \overline{\texttt{many}}^{?}\ c\ x \mid \overline{\texttt{many}}^{?}\ \mathit{cbasic}\ x \\
\mathit{cbasic} & ::= & \texttt{aid} \mid \texttt{bool} \mid \texttt{byte\_sequence} \mid \texttt{content\_element\_list} \\
& \mid & \texttt{date} \mid \texttt{float} \mid \texttt{integer} \mid \texttt{string}
\end{array}
$$

Figure 3.8: JADEL grammar for ontologies. Metavariables o, o_{base} denote ontologies, *prop* denotes a proposition, c, c_{base} denote concepts, p, p_{base} denote predicates and x denotes a generic variable.

3.2.3 Communication Ontologies

The third JADEL abstraction is the ontology. A communication ontology consists of a set of propositions, a set of predicates, and a set of concepts, which can be basic or composite. Propositions are first-order logics well-formed formulas. Predicates are first-order logics predicates with an arity, and their arguments are terms formed using concepts. Basic (or atomic) concepts are simple terms that are provided by JADE. They can be composed to create other (composite) concepts, which can be used to express complex terms. Composite concepts can be seen as function symbols in first-order logics. As a matter of fact, they are terms with arguments, and such arguments are terms themselves. Predicates are used to state relations among concepts, while concepts are used to to describe entities of the domain. Both concepts and predicates can be derived from other base concepts and predicates, respectively. Also an entire ontology can be derived from another ontology, and this means that the resulting set of propositions, set of concepts, and set of predicates is the union of the respective sets of the two ontologies.

JADE programmers tend to agree that the implementation of communication ontologies is an error-prone task because of the large amount of implementation details and repetitive idioms in ontology class definitions that shift the focus on technical parts rather than on the semantics of involved ontology elements. Therefore, a con-

$$
\begin{array}{lcl}
\textit{rdecl} & ::= & \texttt{role}\ r\ (\overline{t\ x}^{*})\ \overline{\texttt{for}\ a}^{?}\ \texttt{as}\ \textit{protocol} : \textit{rname} \\
 & & \{\ \overline{\textit{field}}^{*}\ \overline{\textit{method}}^{*}\ \overline{\textit{revent}}^{+}\ \} \\
\textit{revent} & ::= & \texttt{on}\ P\ m\ \{\ \overline{\textit{expr}}^{*}\ \} \\
\textit{protocol} & ::= & \texttt{FIPA\_REQUEST} \mid \texttt{FIPA\_QUERY} \mid \texttt{FIPA\_REQUEST\_WHEN} \\
 & \mid & \texttt{FIPA\_CONTRACT\_NET} \mid \texttt{FIPA\_PROPOSE} \mid \ldots \\
\textit{rname} & ::= & \texttt{Initiator} \mid \texttt{Responder}
\end{array}
$$

Figure 3.9: JADEL grammar for roles in FIPA interaction protocols. Metavariables t, x, a, P and m denote type references, variables, agents, performatives and messages, respectively.

cise syntax is needed to permit the automation of many repetitive tasks, e.g., the registration of schemas, and to drop all details that do not directly concern with the ideas behind the creation of an ontology.

The syntax that JADEL adopts for communication ontologies is shown in Figure 3.8. The defined syntax follows precisely the intended meaning of propositions, predicates and concepts in JADE, and the keyword `extends` explicits derivation among concepts and predicates, as well as derivation among ontologies. Notably, the semantics of the declaration of a communication ontology can be obtained by the simple translation into the corresponding JADE classes, `Ontology` for the ontology itself, `Predicate` for propositions and predicates, and `Concept` for composite concepts. Such a semantics is not detailed here because it would require an in-depth description of the whole JADEL compiler, which is briefly outlined in [Bergenti, 2014].

3.2.4 Message Passing and Interaction Protocols

Extended expressions introduced in Figure 3.1, besides providing constructs to activate and deactivate behaviours, provide specific features for the creation of messages, the sending of messages and the extraction of the content of messages. JADEL models messages as data structures made of a number of fixed properties: the performative, the list of recipients, the ontology and the content. The performative denotes

$$\frac{expr : \texttt{activate behaviour}\ x\ \texttt{as}\ b(\overline{xexpr}^*) \quad expr \in C <: \texttt{Agent}}{b\ x = \texttt{new}\ b(\overline{xexpr}^*);\ \texttt{this.addBehaviour}(x);} \quad \text{(Expr-Act)}$$

$$\frac{expr : \texttt{activate behaviour}\ x\ \texttt{as}\ b(\overline{xexpr}^*) \quad expr \in C <: \texttt{Behaviour}}{b\ x = \texttt{new}\ b(\overline{xexpr}^*);\ \texttt{theAgent.addBehaviour}(x);} \quad \text{(Expr-ActB)}$$

$$\frac{expr : \texttt{performative is}\ P \quad expr \in wexpr}{\texttt{MessageTemplate.matchPerformative}(P);} \quad \text{(Expr-Perf)}$$

$$\frac{w_1\ \texttt{or}\ w_2 \in wexpr}{\texttt{MessageTemplate.or}(w_1, w_2);} \qquad \frac{w_1\ \texttt{and}\ w_2 \in wexpr}{\texttt{MessageTemplate.and}(w_1, w_2);}$$

$$\frac{\texttt{not}\ w \in wexpr}{\texttt{MessageTemplate.not}(w);} \quad \text{(Expr-MT)}$$

Figure 3.10: Rules that specify the operational semantics of relevant expressions.

the type of the message, according to FIPA standards, and exactly one performative must be specified in order to create a new message. The list of recipients specifies the AIDs of all agents that are intended to receive the message. Ontologies are identified by their names, and they must be declared as described previously in this section. Finally, the content of the message can be either a proposition, a concept, a predicate, a string of characters or a sequence of bytes.

In JADEL, a specific construct `on when do` is used to handle the reception of a message. Such construct, shown in Figure 3.4, captures the event corresponding to an incoming message, and which can express conditions on it by means of a message template defined in the `when` block. Message templates refer to the properties of a message, and they can be combined using logic connectives. The closing block `do` contains the intended action, i.e., the code which is executed when the behaviour is chosen by the behaviour scheduler of the agent.

Agents can send and receive messages to and from other agents by means of ordinary behaviours using the constructs mentioned above, but they can also take a role in a FIPA interaction protocol. The adoption of interaction protocols enhance

interoperability among agents, and FIPA explicitly suggests the use of interaction protocols. A role in JADEL is identified by its name, the protocol it refers to, and the actual role of the agent in such a protocol, namely initiator or responder. It consists of a set of event handlers that filter incoming messages using their performatives, as in FIPA specifications. The grammar that JADEL adopts for roles is shown in Figure 3.9. In order to separate ordinary behaviours from roles in interaction protocols, JADEL provides the specific expression `take role`, which can be used to create and activate a behaviour in the scope of an interaction protocol. Such an expression is part of the grammar of extended expressions, as shown in Figure 3.1.

In Figure 3.10, the semantics of some interesting expressions is shown. Expressions are directly translated into Java code that uses the API of JADE. For example, the `activate behaviour` construct declares a new object x of type b, and it adds the object to the list of behaviours of the agent by means of `addBehaviour`, which is a method of class `Agent`. Rules (Expr-Act) and (Expr-ActB) show the activation of a behaviour in two cases, i.e., the activation inside an agent and inside another behaviour. Rules (Expr-Perf) and (Expr-MT) show the translation of a `when` expression into a `MessageTemplate`.

3.3 JADEL Programmer's Guide

In this section, a detailed description of how a MAS can be build using JADEL is provided. The guide is divided in four main steps, namely the creation of the ontology, the design and implementations of behaviours, the declaration of agents, and the introduction of interaction protocols. Each step shows how deal with JADEL abstractions and helps the reader in writing the source code, by means of a well-known example. As a matter of fact, in order to show the actual use of JADEL, this section describes the JADEL implementation of one of the examples available in the JADE distribution, namely the book trading example, which is one of the examples that are most often used to learn JADE APIs. The example is useful to illustrate the specific JADEL syntax.

3.3.1 Problem Statement

Suppose that an agent would like to buy a book. It would search for sellers and it would ask each of them for the price of the desired book. Then, it would choose the seller with the best offer, and it would send a purchase order to it. In this problem, an agent must act as a *buyer*, searching for other agents that own and sell some books, which are called the *seller* agents.

Communication is managed with an ontology, and with a number of specific behaviours. Following the design of the original JADE example, five different behaviours are used. Three of such behaviours are provided to implement the tasks of the *buyer* agent:

1. One behaviour to send a call for proposals to all sellers, specifying what item the agent wants to buy;
2. One behaviour to wait for all proposals; and
3. One behaviour to accept the best proposal and effectively place the order.

Seller agents wait for requests and propose their prices. If a proposal is accepted, the seller agent actually sells the item to the buyer. Hence, two behaviours managing those two steps are needed:

1. One behaviour to wait for requests and propose a price when a request arrives; and
2. One behaviour to process all orders and inform the buyer agent the result of the operation.

The ontology must handle all the types of message content involved, adding to the communication a semantics that permits buyer and seller agents to understand each other.

3.3.2 First Step: the Ontology Definition

The exemplified ontology permits a seller agent to communicate with its customers. In particular, the seller agent is an agent that can sell CDs and books. CDs and books

are items with a serial number and a price. A book is characterized by its title. A CD is characterized by its title and by a list of tracks. Tracks have a title and a duration, but they are not items. Only items can be sold or bought by an agent.

Declaring a JADEL ontology simply means using the keyword `ontology` followed by the ontology name:

```
1 ontology MusicShopOntology {
```

In this particular ontology, the generic `Item` is a concept composed of two atomic concepts, namely the `id`, which is an `integer` and represents the serial number of the item, and the `price`, also an `integer` in this example, for simplicity. Such a concept is coded in JADEL as follows.

```
2     concept Item(integer id, integer price)
```

Also a `CD` and a `Book` are concepts, which extends the `Item` definition. CDs are composed of `Tracks`, that are composite concepts themselves. A `Track` is declared as a concept with two atomic concepts that describe its `title` and `duration`. The fact that a CD could have one or more tracks is modeled by composing the `CD` concept with the `Track` concept, using the keyword `many`.

```
3     concept Track(string title, integer duration)
4     concept CD(string title, many Track tracks) extends Item
5     concept Book(string title) extends Item
```

The relations between agents and the items that they own or that they want to buy are modeled as `Owns` and `Sells` predicates, respectively.

```
6     predicate Owns(aid owner, Item item)
7     predicate Sells(aid buyer, Item item)
```

The following listing summarizes all the previous considerations and shows the complete implementation of the ontology.

```
1 ontology MusicShopOntology {
2     concept Item(integer id, integer price)
3     concept Track(string title, integer duration)
```

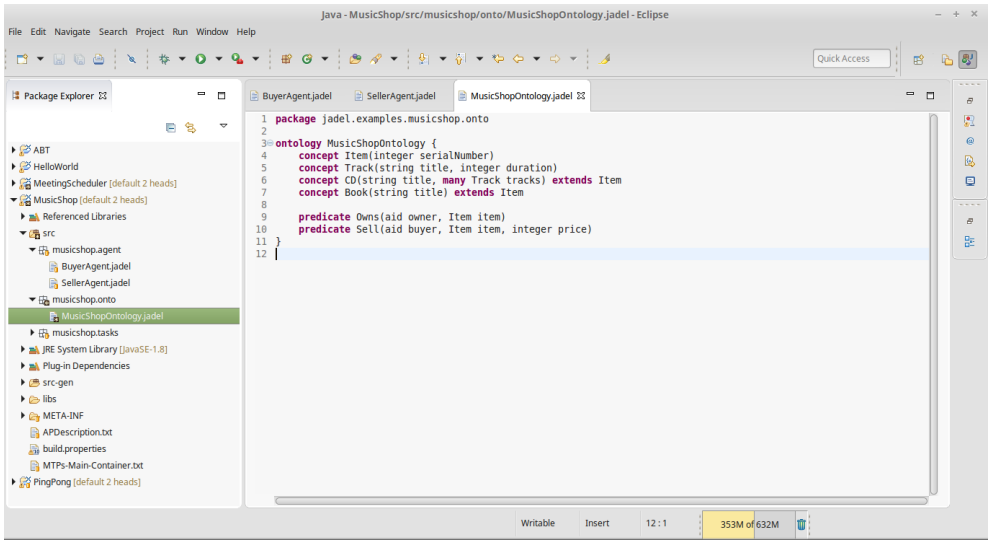

Figure 3.11: Music Shop Ontology in JADEL

```
4       concept CD(string title, many Track tracks) extends Item
5       concept Book(string title) extends Item
6
7       predicate Owns(aid owner, Item item)
8       predicate Sells(aid buyer, Item item)
9   }
```

In Figure 3.11, a screen-shot of the JADEL editor shows the discussed ontology `MusicShopOntology`, with the syntax highlight and the integration with Eclipse.

3.3.3 Second Step: Designing and Implementing Behaviours

As a preliminary example, assume that an agent owns a single CD, and that is waiting for another agent that require such a CD. The following listing defines a behaviour that uses the previously declared `MusicShopOntology` for managing communication, and that waits for a request message.

```
1   cyclic behaviour Sell(CD ownedCD)
```

```
2     uses ontology MusicShopOntology {
```

In order to wait for messages, the behaviour must be a `cyclic` behaviour, and the construct `on when do` must be used to filter incoming messages.

```
3     on message msg
4     when {
5         performative is REQUEST and
6         ontology is MusicShopOntology
7     }
```

A message `msg` is waited, and the stated conditions in the `when` clause require its performative to be `REQUEST` and its ontology to be `MusicShopOntology`.

```
8     do {
9         extract msgContent as String
10
11        if (ownedCD.name == msgContent) {
12            send message {
13                performative is INFORM
14                ontology is MusicShopOntology
15                receivers are #[msg.sender]
16                content is Sells(msg.sender, ownedCD)
17            }
18        }
19    }
20 }
```

When the conditions are met, the content of the message is extracted as a string of characters and, after a check, a response message is sent. The response message consists in the predicate `Sells`, where the buyer AID is filled with the sender of the received message.

As another example, consider a simple behaviour that also uses the ontology `MusicShopOntology`, as exemplified in the original JADE tutorial about ontologies. Such a behaviour permits to agents which activate it to send a request message for an item to buy. Hence, it must be one-shot and its action must be triggered when the

behaviour is first activated. All the arguments of the message are passed as parameters to the behaviour, as they are available to the agent.

```
1  oneshot behaviour Buy(Item requestedItem, List<AID> sellers)
2      uses ontology MusicShopOntology {
```

The JADEL source code of the action is shown in the following. Since there are no conditions for triggering such an action, only the keyword `do` is used. The construct `send message` is used to define the structure of the message and to sending it to a list of sellers, which is a parameter of the behaviour.

```
3      do {
4          send message {
5              performative is REQUEST
6              receivers are sellers
7              content is requestedItem
8          }
9      }
10 }
```

Clearly, the two behaviour `Sell` and `Buy` shown above are not suitable for obtaining a precise solution of the proposed problem. As a matter of fact, there must be more than a seller agent, and prior to the selling of the item there is the proposal processing by the buyer, which search for the best price. To this extent, a correct way to send a request to the seller agent is to use the `CFP` performative, and keep waiting for proposals. A behaviour to do this is shown in the following. It is a one-shot behaviour, i.e., it is removed from the list of active behaviours of the agent when its action is completely performed. The only task of such a behaviour is to send the call of proposals to all seller agents, which are found in the multi-agent system using the directory service of the platform. Two JADEL constructs are used to send call for proposals, namely, the `do` construct to perform an auto-triggered action, and the `send message` expression. The use of the `for` keyword in the behaviour declaration specifies the agent class associated with the behaviour. The behaviour is meant to work with instances of the `BuyerAgent` class only, so that it can take advantage of properties and methods of such a class. Just like JADE, JADEL provides all behaviours with the

field `theAgent`, which is a reference to the agent usable by the behaviour. Such a field ensures that behaviours can have a direct access to the agent that uses them, and it is normally used to let behaviours manipulate the internal state of their agents. Suppose that a buyer agent knows the list of available sellers and that it saves the item it wants to buy. Furthermore, assume that the agent stores the list of available sellers in a field `sellers` and the item in a field `item`. Those fields can be easily accessed within the behaviour by means of the expressions `theAgent.sellers` and `theAgent.item`, respectively.

```
1  oneshot behaviour SendRequests for BuyerAgent {
2      do {
3          send message {
4              performative is CFP
5              receivers are theAgent.sellers
6              content is theAgent.item
7              ontology is MusicShopOntology
8          }
9      }
10 }
```

Once all call for proposals are sent, the buyer agent needs only to wait for actual proposals to decide where to buy the desired item. The `WaitResponses` cyclic behaviour provided for the `BuyerAgent` class is used to wait for proposals. The act of waiting for a proposal is implemented by means of the JADEL construct `on when do`. The fields of the behaviour are defined with the keyword `var`, which means that they are mutable fields. The keyword `val`, instead, declares an immutable field. This is coherent with Xtend syntax, which is used also in the body of the `do` block. `WaitResponses` processes incoming messages with performative `PROPOSE`.

```
1  cyclic behaviour WaitResponses for BuyerAgent {
2      var int messageCounter = 0
3      var AID bestSeller
4      var int bestPrice
5
6      on message msg
```

```
7       when { performative is PROPOSE }
```

The action of such a behaviour extract the content of the message as an integer value, `price`, that is compared with the current best price. The behaviour maintains a counter to track the number of sellers that have already replied to the call for proposals. When such a counter matches the total number of sellers, a new behaviour `SendOrder` is activated. The selection of the best price is made by extracting the content of the received message. The AID of the corresponding seller agent is obtained by inspecting the sender field of the message, with the expression `msg.sender`.

```
8       do {
9           extract msgcontent as String
10          var price = Integer.parseInt(msgcontent)
11          messageCounter++
12
13          if (bestSeller == null || bestPrice > price) {
14              bestSeller = msg.sender
15              bestPrice = price
16          }
17
18          if (messageCounter == theAgent.sellers.size) {
19              messageCounter = 0
20
21              activate behaviour
22                      SendOrder(theAgent, bestSeller)
23          }
24      }
25  }
```

The complete listing of the `WaitResponses` behaviour is also shown in Figure 3.12.

Finally, the last behaviour of the buyer agent can be defined. Similarly to the `SendRequests` behaviour, the only purpose of the behaviour `SendOrder` is to send a message.

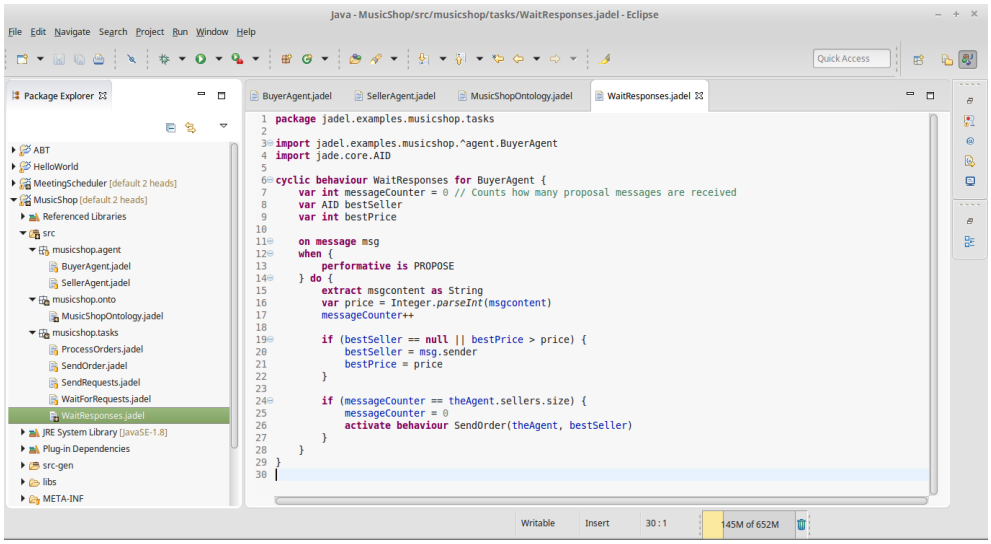

Figure 3.12: Example of cyclic behaviour

```
1 oneshot behaviour SendOrder(AID seller) for BuyerAgent {
```

`SendOrder` is activated by the `WaitResponses` behaviour, that pass the AID of the best seller to it as an argument. The message to be sent is an acceptance message to the seller agent that offered the best proposal. Thus, `ACCEPT_PPROPOSAL` is its performative and its recipient is the seller agent chosen previously. The content of the message is, again, the item that the agent wants to buy, accessed by the notation `theAgent.item`. For creating the recipients list, the construct `#[]`, that denotes a generic list, is borrowed from Xtend syntax.

```
2     do {
3         send message {
4             performative is ACCEPT_PROPOSAL
5             receivers are #[seller]
6             content is theAgent.item
7             ontology is MusicShopOntology
8         }
9     }
```

```
10 }
```

The two behaviours that are needed for the seller agent are explained in the following. The first behaviour contains an event handler to receive call for proposals from buyer agents. Such a behaviour extracts the message content as a generic item and it searches the internal catalogue of the agent for the requested item. In fact, each seller agent owns a list of items that it can sell. Then, if the desired item is found, the seller agent sends a proposal to the message sender, i.e., the buyer agent. On the contrary, it notifies the buyer agent with a failure message.

```
1  cyclic behaviour WaitForRequests for SellerAgent {
2      on message msg
3      when {
4          performative is CFP and
5          ontology is MusicShopOntology and
6          content is Item
7      } do {
8          extract item as Item
9
10         var int price = theAgent.catalogue.get(item)
11
12         if (price != null) {
13             send message {
14                 performative is PROPOSE
15                 ontology is MusicShopOntology
16                 receivers are #[msg.sender]
17                 content is price
18             }
19         } else {
20             send message {
21                 performative is REFUSE
22                 receivers are #[msg.sender]
23                 content is "not␣available"
24             }
25         }
```

```
26      }
27  }
```

Also the second behaviour waits for an incoming message, but it processes only acceptance messages. Such a behaviour extracts the content of the message as an item and it removes such an item from the catalogue. Finally, it notifies the buyer agent that the transaction is positively closed. A check is added to manage the case of multiple orders for the same item. Such a situation is possible when more than one buyer request for the same item and, in addition, two or more of them concurrently receive the proposal of the seller, and the same seller agent becomes the best seller for all such buyers. In such cases, the order message which arrives first is positively closed, while the others fail.

```
1  cyclic behaviour ProcessOrders for SellerAgent {
2      on message msg
3      when {
4          performative is ACCEPT_PROPOSAL and
5          ontology is MusicShopOntology
6      } do {
7          extract item as Item
8
9          var int price = theAgent.catalogue.remove(item)
10
11         if (price != null) {
12             send message {
13                 performative is INFORM
14                 receivers are #[msg.sender]
15                 content is Sells(msg.sender, item)
16             }
17         } else {
18             send message {
19                 performative is FAILURE
20                 receivers are #[msg.sender]
21                 content is "not␣available"
22             }
```

```
23                  }
24              }
25          }
```

3.3.4 Third Step: the Agent Declaration

As in the presentation of behaviour, a preliminary example of agent is shown. The `Customer` agent activates the `Buy` behaviour, passing to it its arguments, namely, a `Book`, which is a concept in the `MusicShopOntology`, and a list of AID. For doing this, the construct `on create` is used, in order to manage initialization. The `on destroy` simply logs a goodbye message.

```
1  agent Customer uses ontology MusicShopOntology {
2      var AID sellerAID = new AID("seller_agent")
3      var Book book = new Book()
4
5      on create {
6          book.title = "Title␣of␣the␣book"
7
8          activate behaviour Buy(this, book, #[sellerAID])
9      }
10
11     on destroy {
12         log("Goodbye")
13     }
14 }
```

In detail, two fields are declared to store the AID of the seller agent and the desired item. In the initialization phase, the agent sets the title of the book it wants to buy and then it adds the `Buy` behaviour to its list of active behaviours. During the life cycle of the agent, the action of such a behaviour is performed as soon as the behaviour is actually selected by the internal scheduler of the agent. In fact, as soon as the initialization phase is complete, the internal scheduler of the agent selects the active behaviour, and the request to buy the book is immediately sent to the seller. In this example, the agent does nothing until it is killed and, during its take-down phase, it simply writes a log message.

Finally, the JADEL source code of seller agents and of buyer agents is shown in Figure 3.13 and Figure 3.14, respectively. Seller agents declare a catalogue field,

```
1 agent SellerAgent(List<Item> catalogue) uses ontology
    MusicShopOntology {
2     on create {
3         activate behaviour WaitForRequests(this)
4         activate behaviour ProcessOrders(this)
5     }
6 }
```

Figure 3.13: Source code for the seller agent.

```
1 agent BuyerAgent(Item item, AID... sellers) uses ontology
    MusicShopOntology {
2     on create {
3         activate behaviour SendRequests(this)
4         activate behaviour WaitResponses(this)
5     }
6 }
```

Figure 3.14: Source code for the buyer agent.

while buyer agents declare an item field and a list of AID. Each field declaration automatically generates two public methods, a getter and a setter, for such a field. This is the reason why the agent fields are reachable within a behaviour associated with that agent. In the initialization phase of the agent, the command line arguments passed to seller agents are used to populate their catalogues. Behaviours are simply activated inside the agent set-up, and how actions are performed is decided by the internal scheduler of the agent.

The proposed implementation of the book trading example follows precisely the structure of the corresponding JADE example available from the JADE official Web site, but the implementation details are very different. As a matter of fact, JADE implementation does not make use of ontologies, and it provides seller agents with a GUI. Moreover, buyer agents share a single behaviour class, and the action method of such a class is modeled as a *Finite State Machine (FSM)*. Such an implementation

is very similar to an instance of the FIPA contract net interaction protocol, but it does not implement it explicitly.

Notably, the proposed JADEL implementation presents some weaknesses. For example, it assumes that all seller agents would eventually respond, and failures of seller agents are not handled by buyer agents. Taking into account all such cases would grow the example over the desired complexity, and the example is intentionally kept simple to become a reference to learn the basis of JADEL programming.

3.3.5 Optional Step: the Use of Interaction Protocols

In order to improve the example in terms of the used features of JADEL, interaction protocols and roles can be used for seller and buyer agents. In Figure 3.15, a sketch of the role for buyer agents is shown. It follows the specification of the `FIPA contract net` initiator. Some event handlers are defined, and the body of the reception of a `PROPOSE` message is similar to the previous implementation. Managing the sending and the reception of messages is not needed, as they are handled by the underlying JADE runtime.

In Figure 3.16, the necessary changes are shown in the code of `BuyerAgent` declaration. The seller agent can be viewed as a responder in a contract net interaction protocol. Implementations of seller agent declaration and its role are omitted for the sake of brevity, but they are similar to the buyer ones.

```
1  role Buyer(ACLMessage cfpMsg) for BuyerAgent
2  as FIPA_CONTRACT_NET:Initiator {
3      var bestPrice
4      var bestSeller
5
6      on PROPOSE msg {
7          extract price as Integer
8
9          if (bestSeller == null || bestPrice > price) {
10             bestSeller = msg.sender
11             bestPrice = price
12         }
13
14         if (cfpMsg.replyByDate < System.currentTimeMillis())
               {
15             message reply {
16                 performative is ACCEPT_PROPOSAL
17                 receivers are #[bestSeller]
18                 content is theAgent.item
19                 ontology is MusicShopOntology
20             }
21         }
22     }
23
24     on INFORM msg {
25         // ...handles INFORM messages...
26     }
27     on REFUSE msg {
28         // ...handles REFUSE messages...
29     }
30     on FAILURE msg {
31         // ...handles FAILURE messages...
32     }
33 }
```

Figure 3.15: Initiator role for a buyer agent using the contract net protocol.

```
1  agent BuyerAgent(Item item, AID... sellers)
2  uses ontology MusicShopOntology {
3      on create {
4          create message cfpMsg {
5              performative is CFP
6              ontology is MusicShopOntology
7              receivers are sellers
8              content is item
9          }
10
11         cfpMsg.replyByDate = System.currentTimeMillis() +
               TIMEOUT
12
13         take role Buyer(this, cfpMsg)
14     }
15 }
```

Figure 3.16: New implementation of the buyer agent with interaction protocols.

Chapter 4

JADEL Examples and Benchmarks

I've seen things you people wouldn't believe.

– Roy Batty

This chapter shows some other examples of JADEL source code, and an evaluation of the JADEL capabilities by means of some metrics, such as the *Lines of Code (LOCs)* one. The first example is the implementation of a well-known agent-oriented algorithm, namely the *Asynchronous BackTracking (ABT)* [Yokoo et al., 1998]. Then, a comparison between JADEL and actor-oriented languages and patterns is provided, by implementing some relevant benchmarks found in the Savina suite [Imam and Sarkar, 2014b]. As a last example, a particular concurrency problem is also tested. For all of the examples, a summary of the main results in terms of readability, agent-oriented features rate, and amount of code concludes the language assessment.

4.1 Implementation of the ABT Algorithm

The ABT algorithm is a well-known algorithm to solve *Distributed Constraint Satisfaction Problems* (*DCSPs*) [Yokoo et al., 1998]. DCSPs are distributed variants of constraint satisfaction problems and, as such, a DCSP consists in a finite set of variables and a finite set of constraints over such variables. As in [Yokoo et al., 1998], variables are denoted as $x_1, x_2, \ldots, x_n$. Each variable x_i takes values in a domain, called D_i. Constraints are subsets of $D_1 \times \cdots \times D_n$, and a DCSP is *solved* if and only if a value is assigned to each variable, and each assignment satisfies all constraints. In a DCSP constraints and variables are distributed among agents. Such agents manage a number of variables and they know the constraints over managed variables. Commonly, each agent is associated with just one variable, and it finds an *assignment* of its variable, i.e., a pair (x_i, d) where $d \in D_i$, that satisfies involved constraints. The interactions in the *Multi-Agent System (MAS)* allows each agent to obtain the assignments of other agents, and to check if constraints are really satisfied. Informally, a DCSP is solved if each agent finds a local solution that is consistent with the local solutions of other agents. In [Yokoo and Hirayama, 2000], a survey of the main algorithms for solving DCSPs is given. In particular, pseudocode and examples are shown for the ABT, the asynchronous weak-commitment search, the distributed breakout, and the distributed consistency algorithms.

The ABT algorithm solves DCSPs under three assumptions: each agent owns exactly one variable, all constraints are in the form of binary predicates, and each agent knows only the constraints that involve its variable. Because it is not necessarily true that all agents in a MAS know each other, they can communicate only if there is a connection between the sender and the receiver of a message. For each agent, the agents who are directly connected with it are called its *neighbors*. In ABT, each agent maintains an *agent view*, which is the agent local view of its neighbors assignments. Communication is addressed by using two types of messages, *OK* and *NoGood*, which work as tools to exchange knowledge on assignments and constraints. More precisely, OK messages are used to communicate the current value of the sender agent variable, and NoGood messages provide the recipient with a new constraint.

Agents are associated with a priority order, which can be, e.g., the alphabetical order of their names. OK messages flow from top to bottom of the priority list of agents, and NoGood messages, instead, go up from lowest priority agents to highest ones. The core of the algorithm is the *check agent view* procedure, which controls if the current known assignments are consistent with the agent value. If not, procedure *backtrack* is used to send NoGood constraints to neighbors. The rest of the algorithm is given in terms of event handling constructs which react at other agents messages.

The ABT algorithm was originally described using a pseudocode [Yokoo and Hirayama, 2000]. The proposed implementation in JADEL follows precisely the original pseudocode. The presentation of the JADEL source code is structured into the presentation of the ontology, of the agents, of support procedures and of event handlers, as follows.

Ontology

From ABT pseudocode, messages are divided into different categories, but there is no specification or definition of an ontology. JADEL takes advantages from a light syntax for defining communication means which describes how agents could interoperate in a given application. The ontology for ABT algorithm includes propositions, concepts, and predicates, as shown in the JADEL code below.

```
1 ontology ABTOntology {
2     concept Assignment(aid index, integer value)
3     predicate OK(Assignment assignment)
4     predicate NoGood(many Assignment assignmentList)
5     proposition NoSolution
6     proposition Neighbor
7     predicate Solution(many Assignment assignmentList)
8 }
```

The assignment is a central concept in ABT algorithm. Its implementation consists in the definition of an ontology term which is composed of an agent identifier, i.e., x_i, and the value of its variable, i.e. d_i, called `index` and `value`, respectively. The two predicates used in the main part of the algorithm, namely, the `OK` and the `NoGood`

predicates, are defined on the basis of the definition of the `Assigment`. In fact, an OK message is the current assignment of the agent, while the NoGood message is a sequence of forbidden assignments. Also a predicate `Solution` is defined, which is used to communicate to other agents the solution of the problem, when found. `NoSolution` and `Neighbor` are simply propositions, that agents can exchange to indicate the algorithm termination with no solutions, and the neighbor request, respectively.

Agents

ABT pseudocode describes event handlers and main procedures, but it does not illustrate how agents should be written. In JADEL, an agent must be defined. Such an agent is called `ABTAgent`. It consists of some properties, among which there are the agent view and the set of neighbors. The initialization of an `ABTAgent` is done by filling the set of neighbors with the identifiers of connected agents, and by setting the priority of the each agent. Moreover, `ABTAgent` provides two important methods, namely, `checkConstraints` and `assignVariable`. The first checks if all constraints are satisfied by current assignments in agent view, while the second selects a value which is consistent with agent view and assigns it to the variable owned by the agent. Both methods return `true` if the operation was successful and `false` if it is was not.

Procedures

The core procedure of the ABT algorithm is the *check agent view* procedure, which controls if the current value $my\_value \in D_i$ of the agent x_i is *consistent* with its agent view. A value $d \in D_i$ is called consistent with the agent view if for each value in agent view, all constraints that involve such value and d are satisfied. If this is not the case, the agent has to search for another value. At the end, if none of the values in D_i satisfies the constraints, another procedure is called, namely, the *backtrack* procedure. Otherwise, an OK message is sent to the agent neighbors, which contains the new assignment. The pseudocode of the check agent view procedure is shown in Figure 1.

Algorithm 1 Procedure check agent view.

```
1: Procedure check agent view
2: while agent_view and my_value are inconsistent do
3:   if no value in D_i is consistent with agent_view then
4:     backtrack
5:   else
6:     select d ∈ D_i where agent_view and d are consistent
7:     my_value ← d
8:     send(OK, (x_i, d)) to neighbors
9:   end if
10: end while
```

In the JADEL implementation of the ABT algorithm, the *check agent view* procedure becomes a one-shot behaviour. In fact, its action has to be performed only once, when the behaviour activates, as follows.

```
1 oneshot behaviour CheckAgentView for ABTAgent {
```

The keyword `for` denotes which agents are allowed to activate such a behaviour. In this case, such agents are instances of the `ABTAgent` class. Inside the behaviour, methods and public fields of the agent can be called by using the field `theAgent`, which is implicitly initialized with an instance of the agent specified. If no agent is specified with the `for` keyword, `theAgent` refers to a generic agent. The behaviour `CheckAgentView` does not need to wait for messages, or events, so the keyword `do` is used, as follows.

```
2 do {
3     if(!theAgent.checkConstraints()) {
4         if(!theAgent.assignVariable()) {
5             activate behaviour Backtrack(theAgent)
6         } else {
7             activate behaviour SendOK(theAgent)
8         }
9     }
```

Algorithm 2 Procedure backtrack.

```
1: Procedure backtrack
2: generate a nogood V
3: while V is an empty nogood do
4:     broadcast to other agents that there is no solution
5:     terminate this algorithm
6: end while
7: select (x_j, d_j) where x_j has the lowest priority in a nogood
8: send(nogood, (x_i, V)) to x_j
9: remove (x_j, d_j) from agent_view
10: check agent view
```

```
10 }
```

The procedure *backtrack* is meant to locally correct inconsistencies. First, a new constraint NoGood has to be generated. Generating a NoGood is done by checking all assignments that are present into the agent agent view. If one of these is removed, and then the agent succeeds in choosing a new value for its variable, it means that such an assignment is wrong. Hence, that assignment is added to the NoGood constraint. After this phase, the new generated NoGood can be empty. If no assignment appears within that new constraint, then there is no solution for the DCSP. Otherwise, a NoGood message has to be sent to the lowest priority agent, and then its assignment has to be removed from agent view. Then, a final check of the agent view is done. JADEL implementation of such a procedure is another one-shot behaviour, whose code follows precisely the original pseudocode of the algorithm.

```
1 oneshot behaviour Backtrack for ABTAgent {
```

As a matter of fact, a NoGood constraint is created, by copying the agent agent view and removing the agent AID.

```
2     do {
3         var V = new HashMap<AID, Integer>(theAgent.agentview)
4         var sortedVariablesList = V.keySet.sort
```

```
5           V.remove(theAgent.AID)
```

Then, removing one by one the assignment in the agent view, the behaviour checks if the agent is able to select a variable.

```
6           for(v : sortedVariablesList) {
7               var removed = V.remove(v)
8               if(theAgent.assignVariable(V)) {
9                   V.put(v, removed)
10              }
11          }
```

Finally, the behaviour checks the newly created NoGood. If it is empty, a behaviour `SendNoSolution` activates. Otherwise, the NoGood is sent and the agent view is checked.

```
12          if(V.isEmpty){
13              activate behaviour SendNoSolution(theAgent)
14          } else {
15              activate behaviour SendNoGood(theAgent, V)
16              theAgent.agentview.remove(V.keySet.max)
17              activate behaviour CheckAgentView(theAgent)
18          }
19      }
20  }
```

Event handlers

Others procedures specified in the original ABT pseudocode concern the reception of messages. When the agent receives an OK message, it has to update its agent view with that new information, then it must check if the new assignment is consistent with others in agent view as in Figure 3. The reception of a message requires a cyclic behaviour, which waits cyclically for an event and checks if such an event is a message.

```
1   cyclic behaviour ReceiveOK for ABTAgent {
```

Algorithm 3 Reception of an OK message.

1: **while** received (OK, (x_j, d_j)) **do**
2: revise agent view
3: **check agent view**
4: **end while**

To ensure that such a message is the correct one, namely, an OK message, some conditions have to be specified. JADEL provides the construct `on-when-do` to handle this situation. The clause `on` identifies the type of event and eventually gives to it a name. If the event is a message, the clause `when` contains an expression that filters incoming messages, as follows.

```
2  on message msg
3  when {
4      ontology is ABTOntology and
5      performative is INFORM and
6      content is OK
7  }
```

Conditions in `when` clause can be connected by logical connectives `and`, `or`, and they can be preceded by a `not`. They refer to the fields of the message, namely, `ontology`, `performative`, and `content`. Fields that are not relevant can be omitted, and multiple choices can be specified. For example a behaviour can accept `REQUEST` or `QUERY` messages with `performative is REQUEST or performative is QUERY`. The clause `do` is mandatory and contains the code of the action.

```
8  do {
9      extract receivedOK as OK
10     val a = receivedOK.assignment
11
12     theAgent.agentview.replace(a.index, a.value)
13     activate behaviour CheckAgentView(theAgent)
14 }
```

The content of the message is obtained by means of the JADEL expression `extract`,

Algorithm 4 NoGood message reception.

```
1: while received (nogood, (x_j, V)) do
2:    record V as a new constraint
3:    while V contains an agent x_k that is not its neighbor do
4:       request x_k to add x_i as a neighbor
5:       add x_k to its neighbor
6:    end while
7:    old_value ← current_value
8:    check agent view
9:    while old_value = current_value do
10:      send(OK, (x_i, current_value)) to x_j
11:   end while
12: end while
```

which manages all the needed implementation details and gives a name and a type to the content. Once the content of type `OK` of the message is obtained, its assignment is used to revise the agent view. Then, the behaviour `CheckAgentView` is activated.

Finally, the pseudocode of the procedure that manages the reception of a NoGood message is a cyclic behaviour for `ABTAgent`. In JADEL, such a procedure is a cyclic behaviour for `ABTAgent`.

```
1 cyclic behaviour ReceiveNoGood for ABTAgent {
```

Checking if the event is a message, and then, if the message is actually a NoGood message, is done similarly to the OK reception, by using the clauses `on` and `when`, as shown in the following code.

```
2 on message msg
3 when {
4     ontology is ABTOntology and
5     performative is INFORM and
6     content is NoGood
7 }
```

Inside the `do` body, the message content is extracted as a NoGood and it is recorded as a new constraint. As an assumption, the agent holds a set of constraints within the field `constraint` which is accessed by the agent instance `theAgent`.

```
8  do {
9      extract receivedNoGood as NoGood
10     val newConstraints = receivedNoGood.assignmentList
11     theAgent.constraints.putAll(newConstraints)
```

Then, if some constraints involve an agent which is not in the agent neighborhood, a request is sent to such an agent, in order to create a new link.

```
12 for (x : newConstraints.keySet) {
13     if (!theAgent.neighbors.contains(x)) {
14         activate behaviour SendRequest(theAgent, x)
15
16         theAgent.neighbors.add(x)
17     }
18 }
```

Finally, the agent view must be checked, and if the previous value of the agent variable x_i remains unchanged, an OK message is sent.

```
19 var oldValue = theAgent.agentview.get(theAgent.AID)
20
21 activate behaviour CheckAgentView(theAgent)
22
23 if (oldValue == theAgent.agentview.get(theAgent.AID)) {
24     activate behaviour SendOK(theAgent)
25 }
```

4.2 Implementation of Savina Benchmarks

Other evaluations on JADEL are made by comparing it using the Savina benchmarks [Imam and Sarkar, 2014b]. Savina is a benchmark suite to test actor libraries

performances[1]. For each benchmark, Savina provides an implementation by using the actor features of Akka [Wyatt, 2013], Functional-Java[2], GPars[3], Habanero-Java library [Imam and Sarkar, 2014a], Jetlang[4], Jumi[5], Lift[6], Scala [Haller and Odersky, 2009], and Scalaz[7]. The benchmarks that Savina provides are divided into *classic micro-benchmarks*, *concurrency benchmarks* and *parallelism benchmarks*.

Micro-benchmarks are simple benchmarks which test specific features of an actor library. For example, the classic `PingPong` benchmark measures the message passing overhead, while the `Counting` benchmark tests message delivery overhead. Concurrent benchmarks focus on classic concurrency problems, such as the dining philosophers, and they represent more realistic tests than micro-benchmarks. Finally, parallelism benchmarks exploit pipeline parallelism, phased computations, divide-and-conquer style parallelism, master-worker parallelism, and graph and tree navigation. In [Imam and Sarkar, 2014b], the scope and the characteristics of each benchmark are discussed, and some experimental results are shown. It is worth noting that Savina is a suite that helps testing actor-oriented solutions, and it does not consider agent-oriented features. Nevertheless, Savina benchmarks are also suitable to analyze some features of agent programming languages, such as concurrency and message passing. For this reason, in this dissertation a few benchmarks are taken from those proposed by Savina, and re-implemented in JADEL. Savina does not yet contains inter-languages comparisons. As a matter of fact, sources are written in Java and Scala, and all benchmarks shows almost the same code: the differences among them are due to the various actor implementations. Additional language comparisons could be useful to evaluate the elegance, the readability and the simplicity of a given solution, beside its performances. Only a few Savina micro benchmarks are considered here, namely, `PingPong`, `ThreadRing`, `Counting`, `Big`, and `Chameneos` benchmarks.

[1] The source code of the thirty benchmarks can be found at `github.com/shamsimam/savina`.
[2] `www.functionaljava.org`
[3] `www.gpars.org`
[4] `github.com/jetlang`
[5] `jumi.fi/actors.html`
[6] `liftweb.net/api/26/api/#net.liftweb.actor.LiftActor`
[7] `github.com/scalaz`

It is worth noting that Savina benchmarks are thought for actor-based systems, and thus are heavily based on message passing. JADEL ontologies help in managing such task effectively. In the JADEL source code below, the simple ontology used for implementing the `PingPong` example is shown. The commented parts are the identifiers of the message objects which Savina implementation defines and uses for messages.

```
1 ontology PingPongOntology {
2     proposition Start // PingPongConfig.StartMessage
3     proposition Ping  // PingPongConfig.SendPingMessage
4     proposition Pong  // PingPongConfig.SendPongMessage
5     proposition Stop  // PingPongConfig.StopMessage
6 }
```

The `PingPong` classic example consists in the definition of two agents which uses such an ontology, exchanging *N* `Ping` and `Pong` messages, alternatively. In the following listing, the source code of the ping agent, i.e., the initiator agent, is shown.

```
1 agent PingAgent(AID pongAgent)
2 uses ontology PingPongOntology {
3     on create {
4         activate behaviour
5                 WaitForStartOrPong(this, PingPongConfig.N)
6     }
7 }
```

Then, the pong agent, i.e., the responder agent has the following source code in JADEL.

```
1 agent PongAgent(AID pingAgent)
2 uses ontology PingPongOntology {
3     on create {
4         activate behaviour WaitForPingOrStop(this, 0)
5
6         activate behaviour
7                 SendInformMsg(this, #[pingAgent], new Start)
```

```
8       }
9   }
```

Similarly, `ThreadRing` agents are defined. In this benchmark, N agents exchange R `Ping` messages, and they are limited to communicate only with the next agent in the ring. As for the `PingPong` benchmark, an ontology which follows precisely the Savina structure of message object is defined. In this example, message content are predicates rather than propositions, because they need to carry information between involved agents.

```
1  ontology ThreadRingOntology {
2      predicate Ping(integer left)
3              // ThreadRingConfig.PingMessage
4      predicate Data(aid next)
5              // ThreadRingConfig.DataMessage
6      predicate Exit(integer left)
7              // ThreadRingConfig.ExitMessage
8  }
```

The JADEL source code for agent definition is listed below.

```
1  agent ThreadRingAgent(AID nextAgent, int id)
2  extends JadelBaseAgent
3  uses ontology ThreadRingOntology {
4      on create {
5          activate behaviour WaitForMsg(this)
6
7          if (id == ThreadRingConfig.N - 1) {
8              activate behaviour
9                      SendInformMsg(this, #[nextAgent],
10                     new Ping(ThreadRingConfig.R))
11         }
12     }
13 }
```

As an example of `cyclic` behaviour, the following code shows the reception of an increment message in the `Counting` example. In this example, a `Producer` agent

sends *N* increment messages to a `Counter` one, which counts the number of arrived messages. When the counter agent received all messages, it must inform the other agent of the resulting value of its count. The behaviour `WaitForMsg` is a `cyclic` behaviour, as in ABT event handlers, because it must wait for a message and repeat its action each time a message arrives. For this scope, the construct `on when do` is used, as follows.

```
1  cyclic behaviour WaitForMsg for Counter {
2      on message msg
3      when {
4          content is Increment
5      } do {
6          theAgent.count = theAgent.count + 1
7
8          if (theAgent.count >= CountingConfig.N) {
9              activate behaviour
10                     SendInformMsg(theAgent,
11                     #[theAgent.producerAgent],
12                     new ResultingValue(theAgent.count))
13
14             activate behaviour Delete(theAgent)
15         }
16     }
17 }
```

Other micro benchmarks are implemented in the same fashion, with

1. A definition for each different kind of agent involved, which activates needed behaviours in the start up phase of its life cycle by means of the `on create` handler;

2. The definition of a number of `cyclic` behaviour whose purpose is to intercept messages and process the correct ones; and

3. The definition of an ontology which terms are equivalent to the Savina ones.

Hence, the methodology used in [Bergenti et al., 2017b] for implementing the ABT algorithm is the common way to creating agents and MASs with JADEL, whether the example is a very simple one (e.g., the `PingPong`), or a more complex algorithm as ABT.

4.3 The Santa Claus Coordination Problem Example

JADEL was tested on the Santa Claus problem, as described in [Iotti et al., 2018], which is a classic coordination problem first introduced in [Trono, 1994]. In its simplest form, the problem is expressed as follows. Santa Claus disposes of nine reindeer and ten elves. He sleeps awaiting a group formed of all of his reindeer or three of his elves. When three elves are ready and awaken Santa Claus, he must work with them at discussing toys R&D. Similarly, when the group of reindeer is ready and awakes Santa Claus, they work together to deliver toys. Santa Claus gives priority to the group of reindeer, when both groups are ready at the same time. This simple problem is a classic exercise that exploit concurrency and parallelism, and a number of different solutions are available. The difficulty resides in the expression of the various groups and of the communication patterns. Moreover, the problem could be extended by incrementing the number of Santa Claus, elves and reindeer, thus opening a wide variety of other problems: for example, reindeer and elves must help one Santa at a time, and Santas cannot wait too long for a group, releasing reindeer and elves when their group is not ready in time. This requires a correct sentencing of messages, and the addition of timing constraints. Thus, it is important that the given solution provide scalable constructs and re-usable expressions, in order to deal with more complex problems.

In JADEL, the Santa Claus problem is addressed by the individuation of three types of agents, namely the `Santa`, `Reindeer`, and `Elf` agents. These three definitions are sufficient for the underlying JADE platform to distinguish among agent roles in the problem. Such agents can exchange messages. The core of the agent communication is the ontology.

```
1 ontology SantaOnto {
```

```
2       predicate ElfMessage(aid id)
3       predicate ReindeerMessage(aid id)
4       proposition OK
5   }
```

The ontology `SantaOnto` defines two similar predicates, which relate an AID to the type of sender agent, elf or reindeer. The AID of an agent in JADE is unique for each agent in the platform and consists of its name (chosen when the agent starts) and its address (the name of the machine where the agent lives). A reindeer or an elf must send to Santa this type of message when it is ready. The proposition `OK` is used by Santa Claus when one of the groups is ready, to inform the group members the work is about to start.

In the proposed solution, the ready message is sent by the behaviour below.

```
1   oneshot behaviour Ready(AID[] santas, Predicate pred) {
2       do {
3           var Random rand = new Random()
4           var santa = santas.get(rand.nextInt(santas.length))
5
6           send message {
7               performative is INFORM
8               content is pred
9               receivers are #[santa]
10              ontology is SantaOnto
11          }
12      }
13  }
```

The behaviour is declared as `oneshot` behaviour. Such kind of behaviours contain an action, defined inside the curly brackets of the construct `do`. When an agent activates a behaviour, it adds such a behaviour into its personal list of tasks, and, during its life-cycle, the agent tries to execute all tasks. Clearly, if a behaviour must wait for a message, and no messages are present in the agent mailbox, that behaviour cannot run its action. But `oneshot` actions can be performed without any trigger, and, after their executions, they are removed from the agent list. In this case, the behaviour

`Ready` has two parameters, namely, a list of AID and a predicate. It randomly select a Santa Claus from its list of Santa's identifiers and send to him an inform message. This behaviour can be used by both a reindeer and an elf, because they can specify the predicate to be sent.

A `Santa` agent must receive such a message and reply correctly. The following behaviour shows the reception of a `ReindeerMessage`.

```
1  cyclic behaviour WaitForMessages
2  for Santa {
3      on message m
4      when {
5          content is ReindeerMessage
6      } do {
7          extract reindeer as ReindeerMessage
8          theAgent.reindeer.add(reindeer.id)
9
10         if (theAgent.reindeer.size == 9) {
11             send message {
12                 performative is INFORM
13                 receivers are
14                 theAgent.reindeer.toList
15                 content is new OK
16                 ontology is SantaOnto
17             }
18
19             activate behaviour
20             DeliverToys(theAgent)
21
22             theAgent.reindeer.clear
23         }
24     }
25 }
```

Handling a message reception requires a `cyclic` behaviour, i.e., a behaviour which remains in the agent list waiting for messages, also after the execution of its action.

With the keyword `for`, the behaviour specifies a type of agent. It is useful, for example, when the behaviour have to access some fields of the agent, or referring to it in any way. The special field `theAgent` is used inside the action for accessing the specified agent. The construct `on when do` handles the message. The keyword `on` specifies the type of event, a message, the keyword `when` contains a boolean condition over such a message, to filter incoming messages, and the keyword `do` contains the actual action. The expression `extract` creates a reference for the content of the message, which can be now accessed. When the agent receives nine of those messages, it sends an `OK` message to all the reindeer involved, and then it activates a behaviour to deliver toys.

An almost identical construct `on when do` can be added after the previously shown one, in the same behaviour, to manage `ElfMessage`s. The combination of these constructs ensures the priority of the reindeer with respect to the elves. As a matter of fact, the JADE translation of the behaviour creates two different event handlers, but they are activated in sequence, as follows.

```
1  public void action() {
2      super.action();
3      _event0.run();
4      _event1.run();
5  }
```

When a reindeer or an elf receives an `OK` message, the following behaviour activates.

```
1  cyclic behaviour WaitForOK {
2      on message msg
3      when {
4          content is OK
5      } do {
6          if (theAgent instanceof Reindeer) {
7              activate behaviour DeliverToys(theAgent)
8          } else {
9              activate behaviour DiscussToysRD(theAgent)
10         }
```

```
11      }
12  }
```

Finally, `Reindeer`, `Elf`, and `Santa` agent types can be defined. For reindeer and elves, the declaration are very similar. They take as parameters the list of Santa available, and they uses the `SantaOnto` ontology for sending messages. In their initialization phase, they activate the two behaviours above, filling the `Ready` parameters with the correct predicate.

```
1  agent Reindeer(AID... santas)
2  uses ontology SantaOnto {
3      on create {
4          activate behaviour
5          WaitForOK(this)
6
7          activate behaviour
8          Ready(this, santas, new ReindeerMessage(this.AID))
9      }
10 }
```

The agent `Santa` is even simpler in its definition. As a matter of fact, it only has to sleep waiting for groups of reindeer or elves.

```
1  agent Santa uses ontology SantaOnto {
2      var Set<AID> reindeers = newHashSet
3      var Set<AID> elves = newHashSet
4
5      on create {
6          activate behaviour
7          WaitForMessages(this)
8      }
9  }
```

4.4 Experimental Results

Methods to evaluate DSLs can be found in, e.g., [Challenger et al., 2016a], which focuses on MASs. Other surveys, such as [Mernik et al., 2005] and [Oliveira et al., 2009], highlight the main advantages of the use of DSLs.

The comparison between the ABT pseudocode and its JADEL implementation is done by defining some metrics, which help to get an idea of JADEL advantages and disadvantages. Then, JADEL code is compared with an equivalent JADE code, measuring the amount of code written, and the percentage of agent-oriented features of such a code. Nevertheless, comparing a pseudocode with an actual implementation is a difficult task, due to the informal nature of the pseudocode, and the implicit technical details that it hides. Moreover, pseudocodes from different authors may look different, depending on their syntax choices and their purposes. There are not standard methods for evaluating the closeness of a source code to a pseudocode, and its actual effectiveness in expressing the described algorithm. Hence, the evaluation is limited to the use case of JADEL shown previously: the ABT example.

The first consideration that is made in evaluating the JADEL implementation of ABT is that ABT pseudocode is presented by means of procedures and event handlers, with the aid of the keywords `when` and `if`. As a second consideration, the notation used inside the ABT pseudocode is the same of the DCSP formalization. As a matter of fact, there are *agentview* and *neighbors* sets, and assignments are denoted as (x_i, d_i), where x_i is the variable associated with the i-th agent, and $d_i \in D_i$. A message is identified according to its type and its content, i.e., $(OK, (x_i, d_i))$ for an OK message, or $(nogood, (x_i, V))$ for a NoGood. Such characteristics of ABT pseudocode allow to talk about *similarity* between it and the JADEL source code. In fact, in JADEL, both procedures and event handlers are represented as behaviours of the agent. In particular, procedures are one-shot behaviours that define an auto-triggering actions, while event handlers are cyclic behaviours, each of them waits for the given event and then performs its action. Hence, each behaviour can be associated with a procedure or an event handler, and analyze each of them separately. Moreover, calls to procedures in ABT pseudocode translate into the activation of the corresponding

behaviour in JADEL. Also, the sending of a message is done by activating a specific JADEL behaviour. Hence, each *send* instruction in ABT pseudocode is associated to that activation. The DCSP notation is used also in JADEL, by means of the two maps, *theAgent.agentview* and *theAgent.neighbors*, and by defining some ontology terms. As a matter of fact, terms *OK* and *NoGood* are predicates in a JADEL ontology, and they contain an assignment, and a list of assignments, respectively. Each assignment consists in a *index* and a *value*, i.e., x_i and d_i, respectively. The domain D_i of a variable is defined once in the start-up phase of the agent and it is never modified during the execution of its actions. ABT pseudocode notations are associated with the respective JADEL notation described above. Finally, the reception of a message is done by using the construct `on when do`, which is the corresponding of ABT pseudocode construct **when** *received*(...) **do**. In summary, *(i)* ABT procedures are associated with JADEL oneshot behaviours, *(ii)* ABT event handlers are associated with JADEL cyclic behaviours, *(iii)* procedure calls and *send* instructions are associated with the correct behaviour activation, *(iv)* references to agentview or agent neighborhood are associated with the respective JADEL agent fields, and *(v)* receptions of messages are associated with JADEL constructs and expressions that concern reception and content extraction from a message.

In the following, it is said that a line of ABT pseudocode *corresponds* to a line (or, a set of lines) of JADEL implementation, if it falls in one of the previous cases. Then, for each line of ABT pseudocode, the number of the corresponding *Lines Of Code* (*LOC*) of JADEL implementation is counted. The absolute value of the difference between ABT lines and corresponding JADEL LOC is used as a first, rough, distance. For example, in the reception of an *OK* message, the first line of the pseudocode corresponds to the `on when do` constructs to capture the correct event, and filter other messages that are not complied with the expected structure, as follows.

```
1  on message msg
2  when {
3      ontology is ABTOnto and
4      performative is INFORM and
5      content is OK
```

```
6 }
```

Moreover, the `extract` expression is used to obtain the message content.

```
1 extract receivedOK as OK
```

Hence, in this case there are six LOCs instead of one line of the pseudocode. Thus, the distance is of five LOCs. Such a distance gives an idea of the amount of code which is necessary to translate pseudocode into JADEL, in case of ABT example. A summary is shown in Table 4.1, where a count of nested blocks also presented. ABT pseudocode and JADEL implementation do not differ significantly in terms of nested blocks, and JADEL code often requires one more level (the `do` block), but its structure is usually very similar to ABT pseudocode.

Then, it is desirable to obtain a quantification of the *complexity* of the source code. In fact, JADEL code sounds similar to ABT pseudocode also because of its structure. The depth of each block of code is used as a measure.

The count of nested blocks makes more sense when JADEL code is compared to the equivalent JADE one. Such an equivalent implementation is obtained directly from the available JADEL compiler [Bergenti, 2014], which translates JADEL code into Java and uses JADE APIs. In fact, JADEL entities translate into classes which can extend JADE `Agent`, `CyclicBehaviour`, `OneShotBehaviour`, and `Ontology` base classes, while JADEL event handlers translate into the correct methods of JADE APIs, in order to obtain the desired result. JADE code is automatically generated from the JADEL one, and this means that the final code may introduce some redundancy or overhead. For this reason, a JADE code that implements ABT algorithm directly was also wrote. Nevertheless, this alternative implementation is as complex as JADE generated code, because of some implementation details that JADE requires.

A comparison between JADEL and JADE implementation is made in terms of amount of code, i.e., by counting the number of non-comment and non-blank LOCs of each entity, namely, the `ABTAgent`, the `ABTOntology`, and all the behaviours. Results are shown in Table 4.2. In order to emphasize the advantage in using JADEL instead of JADE, the percentage of lines which contains agent-oriented features over the total number of LOCs is also shown in Table 4.2. Agent-oriented features are each

Table 4.1: The number of LOCs of the JADEL implementation against that of the ABT pseudocode. Δ_l is the number of LOCs of the JADEL implementation minus the number of LOCs of the pseudocode, for each event handler. Δ_d is the count of nested blocks in the JADEL implementation minus the count of nested blocks in the pseudocode, for each event handler.

Event handler	Δ_l [LOCs]	Δ_d [levels]
Check agentview	2	1
Backtrack	6	2
Received OK	6	0
Received nogood	7	1

Table 4.2: Number of LOCs of the JADEL implementation of the ABT pseudocode against the number of LOCs of the corresponding JADE implementation, designed to exactly match the JADEL implementation.

Classes	JADEL [LOCs]	JADE [LOCs]
ABTAgent	57	149
ABTOntology	7	113
Behaviours	138	380

reference to the agent world. For example, keywords `agent`, `behaviour`, `ontology` are agent-oriented features, but are also special expressions. In JADE, agent-oriented features are simply the calls to the API. Table 4.2 shows that the JADEL implementation is far lighter than the JADE one, and that it is more dense in terms of agent-oriented features. Such measures can be viewed as an indication of simplicity of JADEL code with respect to JADE.

The comparison between the chosen Savina benchmarks and their JADEL implementation is done by using the metrics of LOCs, with some restrictions. As in the ABT case, JADEL code is also compared with an equivalent JADE code, in terms of amount of code written. The main problem here is the different structure of a JADEL implementation, developed by using the JADEL approach, and the structure

Table 4.3: Number of LOCs, for Scala, JADEL and JADE implementation of selected examples from Savina benchmark suite.

Benchmark	JADEL	Scala	JADE (generated source)
Base	55	68	197
PingPong	62	73	295
ThreadRing	46	53	239
Counting	72	40	308
Big	90	76	407
Chameneos	121	112	574
Philosopher	108	64	503

of a benchmark in Savina. Savina and JADEL projects are analyzed in terms of file written, utilities, and base classes, and only relevant parts of the benchmarks are evaluated (for example, configuration files are not counted). For a deeper evaluation, JADEL ontologies and Savina message objects are treated separately.

Savina benchmarks are structured as follows. There is a Java-written file of configurations, where parameters are initialized and managed (e.g. the number of pings N for the `PingPong` example), and objects for message passing are implemented. There is also a Scala source code that contains the implementations of actors, a class which implements the benchmark, i.e., a class that manages the iteration and the cleanup phases of each test, and an entry point for the benchmark. Similarly, JADEL benchmarks are structured as follows. The configuration file is the same as Savina. There is a Java file that implements the benchmark and the entry point. The most important, there is a JADEL file which contains the agents that are used in such a benchmark, their behaviours and an ontology for agent communications. It is worth noting that ontology predicates, concepts and propositions completely substitute that objects in the configuration file which are used in Savina for message passing. So, the JADEL implementation uses only a part of such a file, for getting parameter values, and does not take advantage of message objects.

Then, all benchmarks share a common base of methods and utilities. In Savina,

Table 4.4: Number of LOCs, for Scala, JADEL and JADE implementation of ontologies and messages.

Benchmark	JADEL Ontology	Scala Message Objects	JADE Ontology
Base	4	0	37
PingPong	6	29	57
ThreadRing	5	30	94
Counting	5	3	62
Big	5	21	47
Chameneos	7	32	141
Philosopher	7	6	97

for each considered actor framework, an actor base class is implemented. In JADEL, a base agent with two behaviours is implemented. Finally, Scala, JADEL and JADE LOCs are calculated using the following rules:

1. Blank lines or comments are not counted;
2. Prints for debugging are not counted;
3. Regarding Savina benchmarks, actors implementations are counted and also the definition and implementation of `Message` objects;
4. Regarding JADEL, the agent, behaviour and ontology implementations are counted; and
5. Regarding JADE, all generated files are counted.

Each measurement of the Savina suite is done by considering the Scala actor implementation of the benchmark. In Table 4.3, LOCs of some examples are shown, namely, the `PingPong`, `ThreadRing`, `Counting`, `Big`, `Chameneos` and `Philosopher` benchmarks. Table 4.4 emphasize the fact that JADEL syntax for ontologies is very light and the number of LOCs for ontologies remains little for each example, as opposed to Savina message objects or JADE ontologies.

Table 4.5: Number of LOCs and percentage of Agent-Oriented (AO) features over the total number of LOCs, for JADEL and JADE implementation of the Santa Claus example.

Classes	JADEL [LOCs]	JADEL [AO]	JADE [LOCs]	JADE [AO]
Agents	19	57.89	54	33.34
SantaOnto	5	80.00	64	17.19
Behaviours	68	61.76	237	29.54

Regarding the Santa Claus example, as a first notable problem, reindeer and elves codes easily overlap, because JADEL does not have a pattern-matching engine at that level. As a matter of fact, agent types translates into Java classes, and, when an agent starts, it can be recognized only by its AID. Thus, two different predicates are used in order to distinguish a reindeer from an elf, duplicating the code for managing messages.

Another problem can be the lack of transparency of event handling in behaviours. The priority is stated by the order of the `on when do` constructs, which translates into a JADE action that maintains such an order. This is ultimately a technical implementation details, that is not clear.

Despite these problems, the JADEL implementation presents some important advantages. As a matter of fact, the target domain of the language is the JADE domain, and its purpose is to ease the construction of JADE MASs. In terms of LOCs, the scope has been achieved, as illustrated in Table 4.5. The table also show another measure, namely the rate of agent-oriented features per LOCs. The gain in using JADEL instead of JADE is evident from this type of analysis, in terms of simplicity, ease-of-use, and development time. Moreover, comparing JADE and JADEL performance shows a negligible overhead, because JADEL heavily relies on JADE architecture, and JADEL agents run on the JADE platform. As another result, the proposed implementation of the Santa Claus problem is easily scalable for large number of reindeer, elves and Santas. In fact, each agent is a thread, and behaviours are tasks that can be executed in parallel. The order of the `on when do` constructs in the `Santa` agent

behaviour ensures the correct order of priority, and the developer does not have to take account of these technical issues when building its own MAS.

In summary, JADEL was meant for providing a lighter syntax for JADE features, and thus simplifying the use of the framework. The effort made in selecting appropriate expressions and cutting tedious and repetitive implementation details was successful, especially for the development of ontologies. Hence, the language is suitable for developing agent-based systems, and implementing agent-oriented algorithms. In this last example its adaptability to other types of problems, in particular, coordination problems, is evaluated. Such problems are studied mainly for actor-based technologies, where concurrency and synchronization are central issues. Nevertheless, agent frameworks could add intelligent behaviours to the simpler actor model, and agents can behave like actors in certain situations, when requested. JADEL shows a good adaptability, and confirms the advantage compared to the JADE approach.

Conclusions

This thesis presented the *JADE Language (JADEL)*, a novel agent-oriented *Domain-Specific Language (DSL)*. A fragment of the variety of architectures and different targets of *Agent-Oriented Programming (AOP)* was shown in the first chapter, in order to make an almost complete view of the context that JADEL aims at joining. First, *Agent-Oriented Software Engineering (AOSE)* was defined, and the major existing approaches to AOSE and agent-oriented *Model-Driven Development (MDD)* were described. Some of them concern meta-models, others provide tools for modeling agents and *Multi-Agent Systems (MASs)*, but most of them are methodologies that discipline the developments from early design phases to the actual implementation. Then, some of the most popular agent-oriented frameworks and *Agent Platforms (APs)* were illustrated, pointing out the importance of AOP in real-life applications. Finally, *Agent Programming Languages (APLs)* were presented as a class of programming languages, which spaces from abstract to general-purpose ones, and that are very different for purpose and usage. The high number of different agent-oriented proposals, from agent-oriented methodologies to platforms, frameworks, and languages, confirms the relevance of AOP as a mature programming paradigm.

The second chapter is devoted to a formalization of the platform taken as basis for JADEL: the *Java Agent DEvelopment Framework (JADE)*. One of the major goals of the proposed formalization was to decouple JADE agents and MASs from implementation details and from the Java meta-model. Therefore, the main abstractions that compose a JADE MAS were identified and precisely defined, for describing them formally. Another major goal of the proposed formalization is to clarify the intended se-

mantics of JADE APIs by means of clear and consistent operational-semantics rules which closely follow the execution model of JADE agents. This helped addressing the key aspects of the life cycle of agents and it contributes to avoid possible misunderstandings about its semantics. The formalization focused, as expected, only the main entities of the MAS, thus offering a complete view of the whole system in the clear terms of well-defined stores. Examples of computations were obtained from simple JADE agents and behaviours, and explain how the proposed transition system operates on the life cycle of those agents and of their behaviours. A complete analysis of a complex agent with several behaviours, some of them dynamically added and/or removed, leads to a significant growth of the involved transition systems. As a future development, the implementation of automatic tools capable of reasoning on agents and MASs will be addressed, to target real-world applications of the proposed formalization. Nevertheless, the proposed formalization brings two important results. The first is that the main transition system strictly follows the intended semantics of JADE APIs, and this fact permits to focus only notable elements of JADE agents and behaviours, decoupling their respective semantics from the semantics of Java statements. The second is the capability of the transition system to separately and concisely describe the whole MAS, using stores which allow the access to the currently instantiated agents and behaviours.

In the third chapter, the details of the JADEL programming language were explained, together with the main motivations that lead to its core decisions. In particular, the use of Xtext and Xtend and the consequent integration with the Eclipse IDE provide a natural approach to MDD for developing real-world complex applications and ensuring the interoperability with Java. Then, JADEL abstractions were presented, and their syntax was illustrated. The definition of JADEL semantics is quite straightforward, given the basis of JADE semantics and thanks to the light syntax of JADEL. Moreover, a step-by-step programmer's guide for JADEL is provided. Given a well-known JADE example, the JADEL implementation was obtained. Starting from the definition of the ontology, to behaviours implementation, to agent declarations, and, finally, interaction protocols, the example exploited all JADEL features and expressions. Future developments of JADEL regard the actual release of

the language as an open-source project, and the production of appropriate guides and documentation for current JADE users.

Finally, the fourth chapter presented further examples of JADEL implementation of diverse problems and algorithms. A final discussion of such examples and benchmarks was provided. A quantitative assessment of JADEL was obtained by using the metrics of *Lines of Code (LOCs)* and the percentage of *Agent-Oriented (AO)* features in the code. Those indicators are useful in a first approximation to enlighten some JADEL advantages, namely, its lighter syntax and the conciseness in the definition of ontologies. Nevertheless, other aspects are difficult to be precisely evaluated, due to their qualitative nature. For instance, some constructs and expressions are meant to reduce the complexity of the framework and to improve readability rather than to reduce the amount of code written. As illustrative examples, the `on when do` and the `extract` are specifically designed to clarify message reception, while the corresponding JADE patterns are repetitive and full of implementation details. JADEL was shown to be sufficiently expressive to fully reproduce the chosen examples and the best results with respect to JADE are obtained in the ontology implementation. Also roles and behaviours are shorter and clearer, due to the frequent use of domain-specific constructs, especially for message passing. As previously said, numbers of LOCs and the percentage of AO-LOCs are not sufficient to measure the actual advantage in the use of JADEL. For this reason, future works will address the problems of language evaluation and testing. Such an evaluation could be done by scheduling the development of an application by a team of several users. Users could have different experiences in the use of JADE, from beginners to experts. Moreover, making an application that deals with a real-world problem is a good strategy for testing.

Glossary

3APL An Abstract Agent Programming Language 22

AALAADIN An agent-oriented meta-model, which consists of three main concepts: agents, groups, and roles 14

ABT Asynchronous BackTracking 119

ACL Agent Communication Language 5

ADELFE Atelier de Développement de Logiciels à Fonctionnalité Emergente 16

AGENT0 The first agent-oriented programming language 21

AgentSpeak(L) A BDI-based abstract agent-oriented programming language 22

AI Artificial Intelligence 3

AID Agent IDentifier 32

ANTLR ANother Tool for Language Recognition 87

AOP Agent-Oriented Programming 1, 13, 85, 147

AOSE Agent-Oriented Software Engineering 14, 27, 147

AP Agent Platform 4, 17, 27, 147

API Application Program Interface 5, 17, 27

APL Agent Programming Language 4, 13, 20, 27, 147

AUML Agent UML 15

BDI Belief, Desire, Intention 15, 18

CArtAgO Common Artifacts for Agents Open framework 18

CLAIM A Computational Language for Autonomous Intelligent and Mobile Agents 23

DAI Distributed Artificial Intelligence 4
DF Directory Facilitator 32
DSL Domain-Specific Language . . . . . iii, 1, 83, 85, 147
DSML Domain Specific Modeling Language 8

EBNF Extended Backus-Naur Form 87
EMF Eclipse Modeling Framework 88

FIPA Foundation for Intelligent Physical Agents 5, 19, 89
FJ Featherweight Java 11, 92
FSM Finite State Machine 44, 67, 114

Gaia An agent-oriented methodology, which views a MAS as a computational organization consisting of various interacting roles 15
GOAL Goal-Oriented Agent Language 24
GPL General Purpose Language 7

IA Intelligent Agent 3, 18
IAF INGENIAS Agent Framework 18
IDE Integrated Development Environment 2, 85
IDK INGENIAS Development Kit 17
INGEME INGENIAS Meta-Editor 18
INGENIAS Engineering for Software Agents 16
IP Interaction Protocol 5, 15, 88

JaCa An agent model based on the combination of Jason BDI agents and CArtAgO environmental artifacts 18
JaCaMo Jason, CArtAgO, Moise 18
JACK An agent-oriented BDI-based platform thought for commercial use 18
JADE Java Agent DEvelopment Framework . . iii, 1, 19, 27, 85, 147

JADEL JADE Language . . . . . iii, 1, 27, 85, 147
Jadex An agent-oriented software framework that implements a BDI-based reasoning engine 19
JAL JACK Agent Language 23
Janus An agent-oriented platform which allows the construction of holonic agents and MASs 19
Jason An agent-oriented BDI platform which provides an interpreter for the AgentSpeak(L) language 20
JDE JACK Development Environment 18
JPL JACK Plan Language 18

KLAIM Kernel Language for Agents Interaction and Mobility . . . 23

LOC Line Of Code 2, 119, 149
LTS Labeled Transition System 9, 27

MAS Multi-Agent System . . . . . iii, 4, 13, 27, 85, 120, 147
MDD Model-Driven Development 1, 14, 86, 147
MDE Model-Driven Engineering 8, 14, 86
METATEM A concurrent agent-oriented programming language based on temporal logic 22
Moise An agent-oriented organizational methodology, which introduces normative specifications beside roles, groups, and missions 18

OOP Object-Oriented Programming 4
OWL Web Ontology Language 24

PASSI Process for Agent Societies Specification and Implementation 16
PIM4Agents Platform-Independent meta-model for Multi-Agent Systems 17
PIMM Platform-Independent Meta-Model 17
PLACA PLAnning Communicating Agents 22
PROFETA Python RObotic Framework for dEsigning sTrAtegies . . . 25

Prometheus An agent-oriented methodology, which consists of three phases: system specification, architectural design, and detailed design 15

RA Rational Agent 3

SA Software Agent 4

SARL A general-purpose agent-oriented programming language that permits the development of holonic MASs 24

SEA_L Semantic web-Enabled Agent Language 24

SEA_ML Semantic web-Enabled Agent Modeling Language 24

SOS Structural Operational Semantics 9

Tropos An agent-oriented methodology based on the notion of agents with mental capabilities 15

UML Unified Modeling Language 2, 14

VigilAgent An agent-oriented model-driven technique, which uses model-to-model and model-to-text transformation to ease interoperability between Prometheus and INGENIAS 17

WADE Workflows and Agents Development Environment 19

Xbase The base grammar of Xtext 87

XML Extensible Markup Language 19

Xtend A dialect of Java used in Xtext 87

Xtext A software framework for implementing DSLs 86

Bibliography

Alves-Foss, J. (1999). *Formal syntax and semantics of Java*. Number 1523. Springer Science & Business Media.

Bădică, C., Budimac, Z., Burkhard, H.-D., and Ivanovic, M. (2011). Software agents: Languages, tools, platforms. *Computer Science and Information Systems*, 8(2):255–298.

Baldoni, M., Baroglio, C., Calvanese, D., Micalizio, R., and Montali, M. (2016). Towards data-and norm-aware multiagent systems. In *International Workshop on Engineering Multi-Agent Systems*, pages 22–38. Springer.

Baldoni, M., Baroglio, C., and Capuzzimati, F. (2014a). A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technology (TOIT)*, 14(4):23.

Baldoni, M., Baroglio, C., and Capuzzimati, F. (2014b). Typing multi-agent systems via commitments. In *International Workshop on Engineering Multi-Agent Systems*, pages 388–405. Springer.

Bauer, B., Müller, J. P., and Odell, J. (2001). Agent UML: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):207–230.

Bellifemine, F., Bergenti, F., Caire, G., and Poggi, A. (2005). JADE – A Java agent development framework. In Bordini, R. H., Dastani, M., Dix, J., and El Fal-

lah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 125–147. Springer.

Bellifemine, F., Caire, G., and Greenwood, D. (2007). *Developing multi-agent systems with JADE*. Wiley Series in Agent Technology. John Wiley & Sons.

Bellifemine, F., Poggi, A., and Rimassa, G. (2001). JADE: A FIPA 2000 compliant agent development environment. In *Procs. of the* 5^{th} *Int'l Conference on Autonomous Agents*, pages 216–217. ACM Press.

Bergenti, F. (2014). An introduction to the JADEL programming language. In *Procs. IEEE* 26^{th} *Int'l Conf. Tools with Artificial Intelligence (ICTAI)*, pages 974–978. IEEE Press.

Bergenti, F., Caire, G., and Gotta, D. (2012). Interactive workflows with WADE. In *Procs.* 21^{st} *IEEE Int'l Conf. Collaboration Technologies and Infrastructures (WETICE 2012)*, pages 10–15. IEEE.

Bergenti, F., Caire, G., and Gotta, D. (2014). Agents on the move: JADE for Android devices. In *Procs. Workshop "From Objects to Agents"*, volume 1260 of *CEUR Workshop Proceedings*.

Bergenti, F., Caire, G., and Gotta, D. (2015a). Large-scale network and service management with WANTS. In *Industrial Agents: Emerging Applications of Software Agents in Industry*, pages 231–246. Elsevier.

Bergenti, F., Gleizes, M.-P., and Zambonelli, F. (2004). *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Springer.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016a). A case study of the JADEL programming language. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2016)*, volume 1664 of *CEUR Workshop Proceedings*, pages 85–90.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016b). Interaction protocols in the JADEL programming language. In *Procs. 6th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016c). Interaction protocols in the JADEL programming language. In *Procs. 6th Int'l Workshop Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*, pages 11–20. ACM Press.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2017a). Agent-oriented model-driven development for JADE with the JADEL programming language. *Computer Languages, Systems & Structures*.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2017b). A comparison between asynchronous backtracking pseudocode and its JADEL implementation. In *Procs. of the 9th Int'l Conference on Agents and Artificial Intelligence (ICAART)*, volume 2 of *ScitePress*, pages 250–258.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2017c). Overview of an operational semantics for the JADEL programming language. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2017)*, volume 1382 of *CEUR Workshop Proceedings*, pages 55–60.

Bergenti, F., Iotti, E., and Poggi, A. (2015b). Outline of a formalization of JADE multi-agents system. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2015)*, volume 1382 of *CEUR Workshop Proceedings*, pages 123–128.

Bergenti, F., Iotti, E., and Poggi, A. (2015c). An outline of the use of transition systems to formalize JADE agents and multi-agent systems. *Intelligenza Artificiale*, 9(2):149–161.

Bergenti, F., Iotti, E., and Poggi, A. (2016d). Core features of an agent-oriented domain-specific language for JADE agents. In *Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection*, pages 213–224. Springer International Publishing.

Bernon, C., Camps, V., Gleizes, M.-P., and Picard, G. (2005). Engineering adaptive multi-agent systems: The ADELFE methodology. *Agent-Oriented Methodologies*, pages 172–202.

Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761.

Bordini, R. H., Braubach, L., Dastani, M., Seghrouchni, A. E. F., Gomez-Sanz, J. J., Leite, J., O'Hare, G., Pokahr, A., and Ricci, A. (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1).

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons.

Braubach, L., Pokahr, A., and Lamersdorf, W. (2005). Jadex: A BDI-agent system combining middleware and reasoning. In Unland, R., Calisti, M., and Klusch, M., editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Caire, G., Gotta, D., and Banzi, M. (2008). WADE: A software platform to develop mission critical applications exploiting agents and workflows. In *Procs. 7^{th} Int'l Joint Conf. Autonomous Agents and Multiagent Systems*, pages 29–36.

Cenciarelli, P., Knapp, A., Reus, B., and Wirsing, M. (1999). An event-based structural operational semantics of multi-threaded Java. In *Formal Syntax and Semantics of Java*, pages 157–200. Springer.

Challenger, M., Demirkol, S., Getir, S., Mernik, M., Kardas, G., and Kosar, T. (2014). On the use of a domain-specific modeling language in the development of multi-agent systems. *Engineering Applications of Artificial Intelligence*, 28:111–141.

Challenger, M., Kardas, G., and Tekinerdogan, B. (2016a). A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, 24(3):755–795.

Challenger, M., Mernik, M., Kardas, G., and Kosar, T. (2016b). Declarative specifications for the development of multi-agent systems. *Computer Standards & Interfaces*, 43:91–115.

Cossentino, M. (2005). From requirements to code with the PASSI methodology. *Agent-Oriented Methodologies*, 3690:79–106.

Cossentino, M., Gaud, N., Galland, S., Hilaire, V., and Koukam, A. (2007). A holonic metamodel for agent-oriented analysis and design. *HoloMAS*, 7:237–246.

Damiani, F., Giannini, P., Ricci, A., and Viroli, M. (2012). Standard type soundness for agents and artifacts. *Scientific Annals of Computer Science*, 22(2):267–326.

De Nicola, R., Ferrari, G. L., and Pugliese, R. (1998). KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on software engineering*, 24(5):315–330.

Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G., and Mernik, M. (2012). SEA_L: A domain-specific language for Semantic Web enabled multi-agent systems. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 1373–1380.

Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G., and Mernik, M. (2013). A DSL for the development of software agents working within a Semantic Web environment. *Comput. Sci. Inf. Syst.*, 10(4):1525–1556.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., and Hanus, M. (2012). Xbase: Implementing domain-specific languages

for Java. In *Procs. 11th Int'l Conf. Generative Programming and Component Engineering (GPCE 2012)*, pages 112–121. ACM Press.

El Fallah-Seghrouchni, A. and Suna, A. (2003). Claim: A computational language for autonomous, intelligent and mobile agents. In *Procs. Int'l Workshop Programming Multi-Agent Systems (ProMAS 2003)*, pages 90–110. Springer.

Eysholdt, M. and Behrens, H. (2010). Xtext: Implement your language faster than the quick and dirty way. In *Procs. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications companion (OOPSLA 2010)*, pages 307–309. ACM.

Felleisen, M. (1991). On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75.

Ferber, J. and Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *Procs. 3rd Int'l Conf. Multi Agent Systems (ICMAS'98)*, pages 128–135. IEEE Press.

Fichera, L., Messina, F., Pappalardo, G., and Santoro, C. (2017). A python framework for programming autonomous robots using a declarative approach. *Science of Computer Programming*, 139:36–55.

Fisher, M. (1994). A survey of concurrent MetateM – The language and its applications. In *Temporal Logic*, pages 480–505. Springer.

Fisher, M. (1996). Temporal semantics for concurrent METATEM. *Journal of Symbolic Computation*, 22(5-6):627–648.

Fortino, G., Rango, F., Russo, W., and Santoro, C. (2015). Translation of statechart agents into a BDI framework for MAS engineering. *Engineering Applications of Artificial Intelligence*, 41:287–297.

Fuentes-Fernández, R., García-Magariño, I., Gómez-Rodríguez, A. M., and González-Moreno, J. C. (2010). A technique for defining agent-oriented en-

gineering processes with tool support. *Engineering Applications of Artificial Intelligence*, 23(3):432–444.

Galland, S., Gaud, N., Rodriguez, S., and Hilaire, V. (2010). Janus: Another yet general-purpose multiagent platform. In *Seventh AOSE Technical Forum, Paris*.

Gascueña, J. M., Navarro, E., and Fernández-Caballero, A. (2012). Model-driven engineering techniques for the development of multi-agent systems. *Engineering Applications of Artificial Intelligence*, 25(1):159–173.

Gascueña, J. M., Navarro, E., Fernández-Caballero, A., and Martínez-Tomás, R. (2014). Model-to-model and model-to-text: Looking for the automation of VigilAgent. *Expert Systems*, 31(3):199–212.

Getir, S., Challenger, M., and Kardas, G. (2014). The formal semantics of a domain-specific modeling language for Semantic Web enabled multi-agent systems. *International Journal of Cooperative Information Systems*, 23(3):1–53.

Gómez-Sanz, J. J., Fernández, C. R., and Arroyo, J. (2010). Model-driven development and simulations with the INGENIAS agent framework. *Simulation Modelling Practice and Theory*, 18(10):1468–1482.

Hahn, C., Madrigal-Mora, C., and Fischer, K. (2009). A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266.

Haller, P. and Odersky, M. (2009). Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220.

Hindriks, K. V. (2009). Programming rational agents in Goal. *Multi-Agent Programming:*, pages 119–157.

Hindriks, K. V., De Boer, F. S., Van Der Hoek, W., and Meyer, J.-J. (1999a). An operational semantics for the single agent core of AGENT0. Technical report.

Hindriks, K. V., De Boer, F. S., Van Der Hoek, W., and Meyer, J.-J. C. (1998). Formal semantics for an abstract agent programming language. In *Intelligent Agents IV Agent Theories, Architectures, and Languages*, pages 215–229. Springer.

Hindriks, K. V., De Boer, F. S., Van der Hoek, W., and Meyer, J.-J. C. (1999b). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401.

Hindriks, K. V., De Boer, F. S., Van Der Hoek, W., and Meyer, J.-J. C. (2000). Agent programming with declarative goals. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 228–243. Springer.

Hindriks, K. V. and Dix, J. (2014). Goal: A multi-agent programming language applied to an exploration game. In *Agent-Oriented Software Engineering*, pages 235–258. Springer.

Howden, N., Rönnquist, R., Hodgson, A., and Lucas, A. (2001). JACK intelligent agents-summary of an agent infrastructure. In *Procs. of the* 5^{th} *Int'l conference on Autonomous Agents*.

Hübner, J. F., Boissier, O., Kitio, R., and Ricci, A. (2010). Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous agents and multi-agent systems*, 20(3):369–400.

Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450.

Imam, S. M. and Sarkar, V. (2014a). Habanero-Java library: A Java 8 framework for multicore programming. In *Procs. of the 2014 Int'l Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools (PPPJ'14)*, pages 75–86. ACM.

Imam, S. M. and Sarkar, V. (2014b). Savina – An actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Procs. of the* 4^{th} *Int'l Workshop on Pro-*

gramming based on Actors, Agents & Decentralized Control (AGERE!), pages 67–80. ACM.

Iotti, E., Bergenti, F., and Poggi, A. (2018). An illustrative example of the JADEL programming language. In *Procs. of the* 10^{th} *Int'l Conference on Agents and Artificial Intelligence (ICAART)*, volume 1 of *ScitePress*, pages 282–289.

Joy, B., Steele Jr, G. L., Gosling, J., and Bracha, G. (1998). The Java language specification.

Kardas, G. (2013). Model-driven development of multiagent systems: A survey and evaluation. *The Knowledge Engineering Review*, 28(04):479–503.

Kravari, K. and Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344.

Moraitis, P. and Spanoudakis, N. (2006). The Gaia2Jade process for multi-agent systems development. *Applied Artificial Intelligence*, 20(2-4):251–273.

Nikraz, M., Caire, G., and Bahri, P. A. (2006). A methodology for the development of multi-agent systems using the JADE platform. *International Journal of Computer Systems Science & Engineering*, 21(2):99–116.

Oliveira, N., Pereira, M. J., Henriques, P., and Cruz, D. (2009). Domain specific languages: A theoretical survey. In *INFORUM'09 Simpósio de Informática*. Faculdade de Ciências da Universidade de Lisboa.

Pavón, J., Gómez-Sanz, J., and Fuentes, R. (2006). Model driven development of multi-agent systems. In *Procs. European Conf. Model Driven Architecture-Foundations and Applications*, pages 284–298. Springer.

Pavón, J., Gómez-Sanz, J. J., and Fuentes, R. (2005). The INGENIAS methodology and tools. In *Agent-Oriented Methodologies*, pages 236–276. IGI Global.

Plotkin, G. D. (2004). A Structural approach to Operational Semantics. *J. Log. Algebr. Program.*, 60-61:17–139.

Rao, A. S. (1996). AgentSpeak (L): BDI agents speak out in a logical computable language. In *MAAMAW 1996: Agents Breaking Away*, pages 42–55. Springer.

Ricci, A. and Santi, A. (2013). Typing multi-agent programs in simpAL. *Programming Multi-Agent Systems*, pages 138–157.

Ricci, A., Viroli, M., and Omicini, A. (2006). CArtAgO: A framework for prototyping artifact-based environments in MAS. *E4MAS*, 6:67–86.

Rodriguez, S., Gaud, N., and Galland, S. (2014). SARL: A general-purpose agent-oriented programming language. In *Procs. of the IEEE/WIC/ACM Int'l Joint Conferences of Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 3, pages 103–110. IEEE Press.

Russell, S. J. and Norvig, P. (2002). Artificial intelligence: a modern approach.

Shoham, Y. (1991). AGENT-0: A simple agent language and its interpreter. In *Procs. of the 9^{th} National Conference on Artificial Intelligence (AAAI)*, volume 91, pages 704–709.

Shoham, Y. (1993). Agent-Oriented Programming. *Artificial intelligence*, 60(1):51–92.

Shoham, Y. (1997). An overview of agent-oriented programming. In Bradshaw, J., editor, *Software agents*, volume 4, pages 271–290. MIT Press.

Thomas, S. R. (1993). PLACA, an agent oriented programming language.

Trono, J. A. (1994). A new exercise in concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265.

Vieira, R., Moreira, Á. F., Wooldridge, M., and Bordini, R. H. (2007). On the formal semantics of speech-act based communication in an agent-oriented programming language. *J. Artificial Intelligence Research*, pages 221–267.

Winikoff, M. (2005). JACK intelligent agents: An industrial strength platform. In *Multi-Agent Programming*, pages 175–193. Springer.

Winikoff, M. and Padgham, L. (2004). The Prometheus methodology. In *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, pages 217–234. Springer.

Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. MIT press.

Wooldridge, M. (1997). A knowledge-theoretic semantics for concurrent MetateM. *Intelligent Agents III Agent Theories, Architectures, and Languages*, pages 357–374.

Wooldridge, M. (1998). Verifiable semantics for agent communication languages. In *Procs. Int'l Conference on Multi-Agent Systems*, pages 349–356. IEEE.

Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The GAIA methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312.

Wyatt, D. (2013). *Akka concurrency*. Artima Incorporation.

Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685.

Yokoo, M. and Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The GAIA methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370.

Acknowledgments

To my first supporter and the most important person of my life, Luca Zarotti, my boyfriend: I do not know what I would do without you. Thank you very much!

I am especially grateful to him, who has encouraged me for all these years, convincing me that I am good enough for pursue this objective. And, also, for his constant moral and emotional support.

I am also very grateful to my tutor, Agostino Poggi, and to my co-tutors and co-authors Federico Bergenti and Stefania Monica: thank you for giving me the opportunity to work with you, and thank you for your precision and patience in correcting my mistakes.

I am grateful to all my family, my parents Claudio and Patrizia, and my siblings Alessio and Teresa. And all the cats, of course. We shared relaxing and funny moments that help me in recharging my energies: so, thank you all.

I would like also to thank my fellow doctoral students for their support, advice, and, of course, for their friendship. Many thanks!

A special mention goes to Riccardo Monica, my office mate: I really appreciate our interesting talks and funny chats, thank you.

Last but not least, I would like to thank all the friends, my fellow adventurers of many roleplaying campaigns, that have supported me during last three years. In particular, to my closest friends Luigi Corvacchiola and (again) Stefania Monica: “So Long, and Thanks for All the Fish!”