Faculdade de Engenharia da Universidade do Porto

From simulation to development in MAS: A JADE-based Approach

João Pedro Camacho Lopes

For Jury Evaluation

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Henrique Lopes Cardoso

June 23, 2014

From simulation to development in MAS: A JADE-based Approach

João Pedro Camacho Lopes

Mestrado Integrado em Engenharia Informática e Computação

June 23, 2014

Abstract

Multi-agent systems (MAS) present an interesting approach to the efficient development of modular systems. Frameworks exist that aid the development of this class of systems and they range from mostly general-purpose frameworks to domain-specific in an array of different domains.

Multi-agent-based simulations (MABS) are sometimes used on the course of development of a full-featured MAS – for instance, for testing purposes – for the potential gains in performance when simulating MAS. However, most platforms for MAS development are not well suited for MABS development due to scalability limitations[MTLD08]. Furthermore, an opportunity exists to partially automate the development of robust MAS from a previously tested simulation.

JADE [BCG07], a very popular MAS development framework allows the creation of seamless distributed agent systems and complies with FIPA standards for agent interaction. Unfortunately, its multi-threaded architecture falls short in delivering the necessary performance to run a local simulation with a large number of agents. Repast [Col03] is an agent-based simulation toolkit that allows creating simulations using rich GUI elements and real time agent statistics. It can easily handle large numbers of agents in a single simulation. Unlike JADE, though, Repast lacks much of the infrastructure for agent creation and interaction.

Some works [GRM+11, GHM+11] propose, as a solution for bridging the gap between MAS development and simulation, an integration of JADE and simulation features by extending this framework with a simulation layer created from scratch, or by integrating another framework, such as Repast.

This thesis proposes an integrated solution which combines an API that enables the creation of JADE-like simulations using a framework like Repast with a code conversion tool, that automatically converts that simulation into an equivalent JADE MAS. That API does so by reimplementing from scratch many JADE features, including its implementation of FIPA specifications for agent interaction and management. Since the API's architecture is very close to JADE's, conversion becomes more straightforward.

The main user for such a system would be JADE developers who need to create a simulation of their already-developed MAS. By converting their code, the developer can perform tests and simulation and later convert the simulation back to a MAS, preserving all changes. JADE developers can also create simulations from scratch using frameworks like Repast using familiar JADE-like features. Finally, this system can be interesting for Repast developers who desire to expand their knowledge of MAS development with more complex frameworks.

Validation tests demonstrate that the behaviour of the currently implemented JADE features in the API display the same behaviour as the analogous JADE ones. Furthermore, using the API it is possible to achieve very significant performance gains.

Resumo

Os sistemas multi-agente (MAS, *Multi-Agent Systems*) exprimem uma abordagem interessante no desenvolvimento de sistemas modulares e eficientes. Os MAS são compostos por elementos computacionais autónomos – chamados agentes - que são programados para *competir* ou *colaborar* de modo a, por exemplo, resolver problemas computacionais, iniciar negociação automática ou participar em jogos de computador. Existem ferramentas de software que facilitam o desenvolvimento desta classe de sistemas que podem variar entre ferramentas de âmbito geral até ferramentas focadas num domínio específico.

As simulações baseadas em agente (MABS *Multi-Agent-Based Simulations*) são frequentemente utilizadas durante o desenvolvimento de MAS completos – por exemplo, afim de testar o sistema – para beneficiar do seu desempenho mais elevado. No entanto, a maior parte das plataformas para desenvolvimento de MAS não são apropriadas para a criação de MABS [MTLD08].

O JADE [BCG07] é um exemplo de uma plataforma de desenvolvimento de MAS que permite a criação de sistemas de agents distribuídos de forma simplificada cumprindo, ainda, os standards da FIPA (*Foundation for Intelligent Physical Agents*) sobre protocolos de interação entre agentes. No entanto, a sua arquitetura em *multi-thread* não garante a performance necessária para a execução de simulações com um elevado número de agentes. O Repast [Col03] é uma plataforma de criação de simulações baseadas em agentes que permite criar simulações ricas em interfaces gráficas para visualização de dados históricos e em tempo real sobre os agentes. Consegue facilmente lidar com um grande número de agentes numa única simulação. Ao contrário do JADE, o Repast não dispõe de uma infraestrutura para criação de agentes e interação entre eles.

Alguns trabalhos estudados [GRM+11, GHM+11] propõem uma integração com o JADE de ferramentas de simulação, extendendo o JADE com ferramentas criadas de raiz ou integrado-o com outras frameworks, como o Repast.

Esta dissertação propõe uma solução integrada que combina uma API que permite a criação de simulações baseadas no JADE utilizando como base uma framework como o Repast, juntamente com uma ferramenta de conversão de código que converte automaticamente uma simulação numa applicação JADE. Esta API foi criada reimplementando de raiz várias características do JADE, includindo a sua implementação das especificações FIPA para a interação entre agentes e sua gestão. Como a arquitectura da API é muito próxima da do JADE, a conversão de código torna-se mais direta.

Os principais utilizadores esperados serão programadores de JADE que necessitem de simular o MAS que desenvolveram. Convertendo o seu código, o programador pode realizar testes e simulações, fazer as alterações desejadas à simulação e re-converter o código para uma aplicação JADE, preservando as alterações feitas. Um programador de JADE pode também criar simulações de raiz utilizando frameworks como o Repast utilizando componentes baseados em JADE que já lhe são familiares. Por outro lado, um

programador de Repast verá interesse neste sitema se desejar expandir o seu conhecimento sobre desenvolvimento de MAS utilizando frameworks mais complexas.

Testes de validação desmonstraram que o comportamento dos componentes já implementados na API mostram um comportamento identico aos componentes análogos no JADE. Além disso, utilizando a API foi possível obter ganhos significativos a nível de performance.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

ACL	Agent Communication Language
AMS	Agent Management System
AST	Abstract Syntax Tree
ATL	ATLAS Transformation Language
DF	Directory Facilitator
EPL	Eclipse Public License (open source license)
FIPA	The Foundation for Intelligent Physical Agents
FOSS	Free and Open Source Software
IDE	Integrated Development Environment
JADE	Java Agent DEvelopment Framework
JDT	[Eclipse] Java Development Tools
MABS	Multi-Agent-Based Simulation
MAS	Multi-Agent System
MTS	Message Transport Service
Repast	Recursive Porous Agent Simulation Toolkit

Chapter 1

Introduction

Multi-Agent Systems (MAS) are composed of autonomous computational elements capable of interacting with each other, called agents. The development of this class of systems comprises an interesting software paradigm but in terms of computer science history, agent-oriented programming is a relatively new subject, having gained significant traction only after the mid 1990's [Woo08]. Today, the world of MAS development is fairly fragmented and several tools and frameworks exist, each fitting the needs of a subset of all developers. More than creating newer, better frameworks, it is interesting to survey what was created an integrate some of the available works.

1.1 Context

The focus of MAS is in the interaction between agents which are typically simple in their architecture but capable of being autonomous in their decisions. When working with groups of agents, it is possible to create complex applications even if each agent is not "aware" of the greater result of its actions. One can create a parallelism between MAS and our bodies: our cells, autonomous and simple beings, exhibit a much more complex behaviour when working together. A cell in out skin, though, is not aware of what is going on in some other organ, but the body still functions as a whole[Fer99].

MAS enjoyed a rapid growth in popularity and are in widespread use nowadays. Some examples of their uses are simulating urban traffic or social environments, solving complex non-deterministic problems, electronic trading and negotiation, computer games and logistics. This list is far from exhaustive and is only meant to illustrate the versatility of MAS.

Presently, tools and frameworks for the development of all sorts of MAS are as diverse as uses exist for them. MAS development frameworks offer programmers an abstraction layer from software specification, allowing them to think more conceptually about agent-based applications. Some features found in many of these frameworks are agent architectures that simplify the creation of new agents, messaging services that provide simple

interaction between them, networking infrastructure to allow communication between agents in different hosts and registries that index agents and facilitate searching for other agents[All09].

1.2 Problem

Although their use is certainly widespread, there is no general purpose standard for MAS development, since each system has different needs. Many times, these systems are created from scratch, meaning that the developers must define all features of the system - such as its agents, their behaviour, communication and organization, using conventional programming languages and tools. Even though many frameworks offer some level of abstraction from the code, often no single framework satisfies a project's needs.

Most uses of MAS, for instance in negotiation, games or logistics, demand a small number of agents, typically with larger resource demands but without any need for global control of execution – it is acceptable for these types of systems to be based on events and for its agents to work asynchronously. In contrast, Multi-Agent-Based Simulations (MABS) are usually implemented using a large number of lightweight agents with a small resource footprint. Taking the example of traffic simulation, these applications can have many dozens or hundreds of agents representing vehicles, pedestrians and traffic lights. Furthermore, simulations usually feature global synchronization mechanisms.

MAS development frameworks generally provide the programmer with a range of features such as execution control, communication protocols or agent awareness capabilities. In spite of that, most frameworks that focus on MAS development lack synchronization mechanisms and lightweight agent infrastructure required by MABS. One of the main goals of simulations is to be able to visualize real-time, as well as historical data that allow to study emergent and evolutionary phenomena. [MTLD08]

When an application has been developed using a MAS development framework and a need later arises for the creation of simulations, porting the source code to an appropriate MABS development framework is a labour-intensive task since not only the syntax and API of the new framework is significantly different, but conceptually speaking, the adaptation may require significant changes to the application.

1.3 Motivation

Interest exists in the simulation of MAS. At any point of the development of the system, it may be valuable to test MAS in a local and controlled simulation environment and to take advantage of some features present in MABS frameworks that are not available in the destination platform of the system.The rationale for the creation of simulated agent systems is usually concerned with simulation performance. Simulations typically have a higher

performance than complex MAS frameworks. For many popular MAS frameworks, there is an opportunity to gain performance when executing tests and simulations.

As some works suggest [GHM+11, GRM+11, WPG+10], it is feasible to bridge the gap between MAS simulation and development. For instance, JADE and Repast are popular tools in widespread use and are well documented and supported by their communities so it is easier to build on top of them. The main motivation for this work is thus the potential gains in establishing this bridge by embedding FIPA-standards and other JADE specific features into a simulation tool like Repast. The development of robust MAS can be partially automated from a previously tested simulation.

1.4 Goals

The main goal of this thesis is to develop a solution for bridging the domains of simulation and MAS. In order to do that, two main sub-goals were identified.

1. **First**, the creation of an adapter or API that would allow developers to abstract from simulation frameworks' features and use familiar ones present in MAS development frameworks, thus creating "MAS-like MABS". This approach allows the resulting code to be easily ported to a full featured MAS framework.

2. **Second**, the development of a code conversion mechanism. By abstracting from the simulation tools and creating a MAS-like MABS, it becomes possible and more straightforward to engineer a tool that performs the automatic conversion of these MABS into equivalent MAS.

JADE and Repast were chosen over other frameworks for multiple reasons. Both are very popular an in widespread use. In consequence, extensive documentation and many examples and applications created by the community are available for use and study. Furthermore, their source is free and open, which enables the development of other tools based on them. Finally, both are Java frameworks which facilitates their integration.

As further discussed in Chapter 2, other tools were studied but JADE and Repast were the fittest for this thesis. Most MABS frameworks do not implement any interaction standards. Furthermore, JADE is a very rich framework and it is not the goal of this thesis to emulate all its features using Repast. With that said, the following guidelines were defined to achieve this thesis goals.

1. To replicate JADE support for FIPA standards for agent interaction and management; these standards are better explained in Chapter 2.

2. To keep the API simple and fast, support a subset of commonly used JADE features; non fundamental features for local simulations, such as JADE's networking infrastructure, should be left out.

3. Even though Repast is the featured framework for simulation development, the API should be sufficiently generic to allow support for new platforms to be added in future enhancements.

4. Programmers should not need to introduce significant changes to their original applications in order to convert code created with the API; the conversion tool should be capable of converting the code *as-is* and generate working models, which preserve the functionality of the original code.

1.5 Contents

This thesis documents the development of a tool that converts a MABS created in Repast into MAS that uses JADE. Conversely, it should allow the conversion of a JADE MAS into a Repast MABS as well. This tool is useful in the context of development of a MAS whose development started as a MABS or when the need to create a simulation arises during development. JADE and Repast were chosen for this thesis not only for their quality but also for their widespread use, available source code and documentation.

Chapter 2 starts by surveying tools whose goal is to produce MABS. Three frameworks were selected for a more detailed study because their approach is the most relevant to the goals of this thesis. They propose solutions based on enhancing JADE to enable simulations capabilities in it. The rest of the chapter is dedicated to comparing JADE and Repast's features and to include an introduction to some concepts regarding FIPA specifications.

Chapter 3 provides a conceptual definition of SAJaS and MasSim2Dev, the developed tools, including an overview of their features and usage scenarios. This chapter also describes how FIPA Specifications are present in the API.

Chapter 4 gives a detailed description of the software architecture, including a description of how agents execute internally, in the API. This chapter is concluded with a discussion on the perspectives for extending the tools, referring to the discussion of future work in the conclusions.

Chapter 6 presents scenarios used to validate the system. They were subjected to code conversion to verify that the correct execution of the code had been preserved. Their performance was also subject to analysis.

This thesis is concluded with a description of suggested future work and some final notes and conclusions.

Chapter 2

Background

Studying ways to bridge the gap between the fields of MAS and MABS has been the subject of several works and resulted in the development of different tools, each with its own purpose. Some works were in search of increased performance for MABS; others tried to cater specific needs such as providing an appropriate learning environment. This chapter describes some works that are relevant to this thesis and makes a comparison between some of them.

2.1 Related Work

Several frameworks exist that offer support to the development of MAS or MABS. Some are domain specific, meaning that their purpose was well defined in their conception. MASeRaTi[ADK+14], MATSim[BMR+08] and SUMO[KEBB12] are some examples of MABS frameworks for traffic and transports simulation.

Other works like Repast[Col03], NetLogo[TW04], GALATEA[DU00] and Plasma [WPG+10] are considered general-purpose. This list comprises only tools that are free and open source and is not meant to be exhaustive.

Some works propose approaches that are very similar to the solution proposed in this thesis, namely the bridging of the domains of MAS and Simulation. MISIA, JRep and Plasma were built on top of JADE to create a simulation environment based on it. These three works were studied with more detailed due to these similarities.

2.1.1 MISIA

MISIA is a middleware whose goal is to enhance the simulation of intelligent agents and to allow the visualization and analysis of agent's behaviour. It was developed by the Bioinformatic, Intelligent Systems and Educational Technology Research Group (BISITE) from Universidad de Salamanca[1]. It is no longer an active project; as a research experiment, work on this middleware evolved into other more specific tools.

[1] http://bisite.usal.es/

MISIA's approach, as suggested by Figure 2.1, is to use a middle layer that acts as the bridge between two other layers that interact with JADE and Repast. By extending the agents in Repast and JADE, communicating through a coordinator and synchronizing their state, these agents work as a single one.

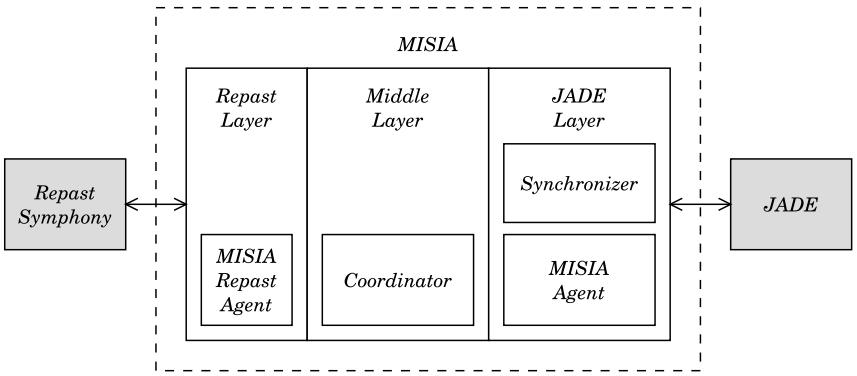


Figure 2.1: High-level representation of MISIA's architecture (adapted from [GRM+11])

One of the challenges identified by the authors when re-implementing the FIPA interaction protocols was synchronizing them with the Repast tick-based simulation model. Given JADE's event-driven architecture, MISIA proposes the use of a coordinator agent that informs the JADE-Agent when a tick has passed. It also proposes its own implementation of the interaction protocols supported by JADE, making them tick-friendly.

2.1.2 JRep

JRep is another platform for integrating JADE and Repast Symphony in the same framework, by means of a middleware. To demonstrate its use, the authors present an example of a smart airport and how it can be simulated using JREP.

JRep's approach is not as complex as MISIA's. By having the Repast Simphony agent encapsulate a JADE agent representation, synchronization is immediate and is assured without requiring an external coordinator. The two agent representations take care of synchronizing any state changes. Figure 2.2 represents the basic structure of JRep.

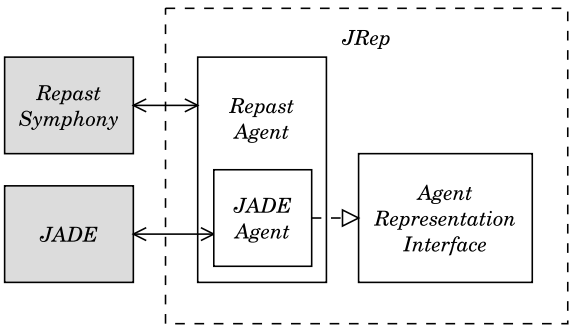


Figure 2.2: High-level representation of JRep's architecture (adapted from [GHM+11])

Each agent takes care of interfacing their respective frameworks. The interaction between agents in JRep is performed using FIPA ACL and the protocol implementations are those provided by the JADE platform. Similarly to MISIA, an Agent Representation Interface is used to introduce the concept of schedule in the JADE agent.

2.1.3 PlaSMA

Unlike the two previous frameworks, the PlaSMA system is based solely on the JADE platform. The distributed simulation is synchronized by entities called "Controllers" who communicate with the "Top Controller", keeping the pace of the simulation and handling agent lifecycle management as well. Figure 2.3 illustrates this architecture. PlaSMA, unlike MISIA and JRep, is still an active project.

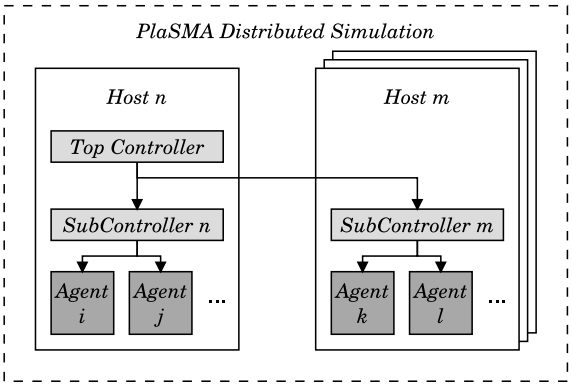


Figure 2.3: High-level representation of PlaSMA's architecture (adapted from [WPG+10])

JADE is a very rich platform but, for many simulation scenarios, the overhead introduced by it has a significant impact on simulation performance [MTLD08].

Even though both MISIA and JRep attempt to integrate the features from both JADE and Repast, as far as Repast simulations are concerned JADE's multi-threaded infrastructure affects their performance very significantly. The main advantage of our approach is, therefore, the possibility of using Repast with JADE features, namely FIPA specifications including interaction protocols, without the need to interface with JADE.

2.2 JADE and Repast

To achieve the goals of this thesis, two frameworks were chosen: JADE and Repast. Both are very popular and in widespread use and have been the foundation of the creation of other tools. They are also free and open source and extensively documented.

JADE is a framework for development of FIPA-compliant fully featured MAS. It aims at simplifying the creation of distributed agent applications by seamlessly hiding all complexity regarding its distributed architecture, including the tasks of agent discovery and the handling of messages.

Repast is a toolkit that provides an environment for the creation of MABS using POJO[2]. It makes it fairly simple to collect agent data and generate displays for it, including charts, grids and others.

Table 2.1: Summary of JADE and Repast features.

	JADE	**Repast**
Communication	FIPA ACL	Method calls Shared resources
Distribution	Yes	No
Simulation Tools	No	Yes
Scalability	Limited	High
Ontologies	Yes	No
Open Source	Yes	Yes
Agent Execution	Behaviour-based Multi-threaded Event-driven Assync	Schedule-based Single-threaded Tick-driven Sync

In JADE, as table 2.1 illustrates, agents execute in separate threads and while this architecture facilitates the platform's distribution, JADE's agent are heavy in terms of resources. Experiments with JADE show that the platform's scalability is limited in number of agents and that the global system performance drops quickly for large number of agents [MTLD08] [GRM+11]. This further strengthens the idea that using JADE or a JADE-Repast hybrid, as describe in the related work, is not the best course of action is performance is an important issue.

In Repast, agent execution is scheduled manually. An agent class can contain annotations that indicate which methods should be called and when (for instance every tick or every 100 ticks). This approach is very flexible, allowing to schedule any method but more complex structures are non-existent in Repast.

In contrast, JADE agent actions can be executed in their setup and takedown, but most are encapsulated in objects called Behaviours. JADE has many different kinds of behaviours that function in different ways, such as running one single task once or running them cyclically. Other behaviours implement FIPA interaction protocols, which agents can use to interact with other agents. These behaviours have states that, as Figure 2.4 explains, are triggered by JADE when certain events occur, such as starting and terminating.

[2]Plain Old Java Objects

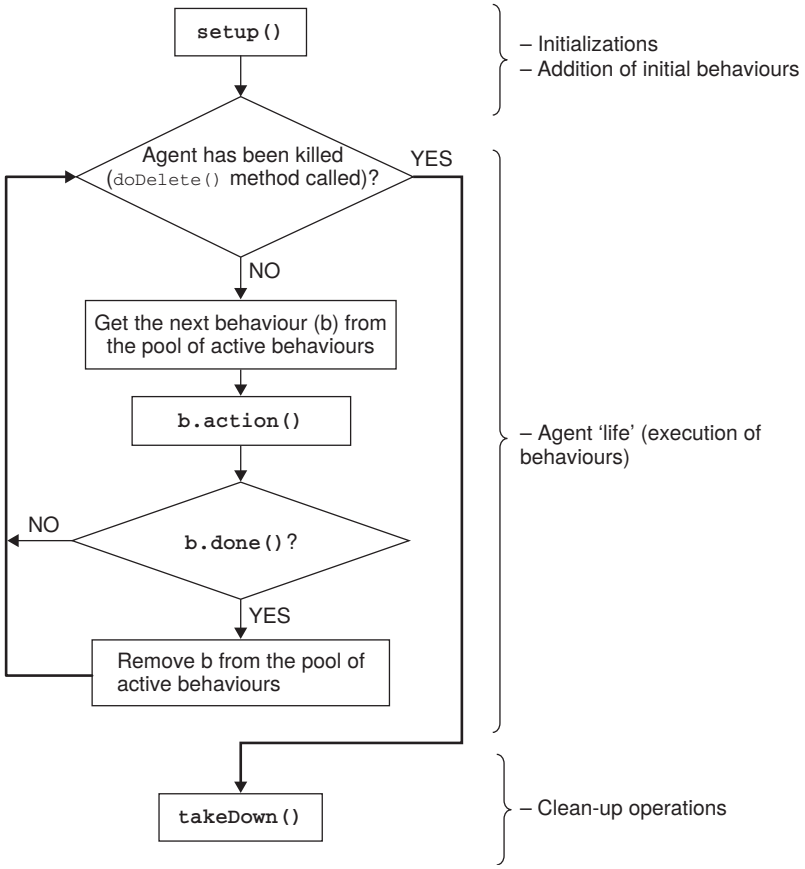


Figure 2.4: JADE Behaviour execution states and events (adapted from [BCG07])

2.3 FIPA Specifications in JADE

The Foundation for Intelligent, Physical Agents (FIPA) was created to specify certain aspects of MAS[BCG07]. FIPA Specifications include standards for agent interaction and agent management (among others). JADE is a FIPA-compliant MAS development framework, viz. the standards proposed by FIPA are part of its architecture. This section describes some of these concepts which are most relevant to this thesis.

2.3.1 Agent Management

The Directory Service (DF) is a component that provides a yellow page service and is part of the FIPA Agent Management Specification. It allows one agent to perform searches about agents rendering specific services. Only agents that are registered in the DF will be indexed and agents can register and deregister themselves at any time.

When searching the DF, agents can use templates that filter the search results. A DF Agent Description represents this template and contains the fields listed in Table 3.1.

The (Agent Management Service) AMS is a mandatory component in FIPA-compliant agent platforms whose purpose is to manage the agent platform, namely the creating and

Table 2.2: FIPA ACL Message Parameters

Parameter	Category of Parameters
performative	Type of communicative acts
sender	Participant in communication
receiver	
reply-to	
content	Content of message
language	Description of Content
encoding	
ontology	
protocol	Control of conversation
conversation-id	
reply-with	
in-reply-to	
reply-by	

deletion of agents. Registration of each agent in the AMS is required for agents to interact, since it is from the AMS that agents obtain their own AID, needed identify the agent in communication.

2.3.2 Messaging

The MTS is a service for transportation of ACL messages between agents. It is responsible for resolving agent addresses, in order to be able to deliver those messages. The MTS may request information from the AMS to perform this address resolution.

The ACL Message is the envelope that contains the details for communication. The Agent Comunication Language (ACL) stipulates what fields a message should contain. Table 2.2 was adapted from the FIPA ACL Message structure specification and contains the list of fields in a message. Not all of them are mandatory. FIPA specifies the `performative` as the only mandatory field, although the `sender`, `receiver` and `content` are expected to be present.

2.3.3 Interaction Protocols

The most relevant FIPA interaction protocols to this thesis that are available in JADE as will be explained further ahead in Chapter 3, are the FIPA Request, and the FIPA Contract Net. JADE supports a few other protocols, namely FIPA Propose, Iterated FIPA Request and Query and FIPA Subscribe.

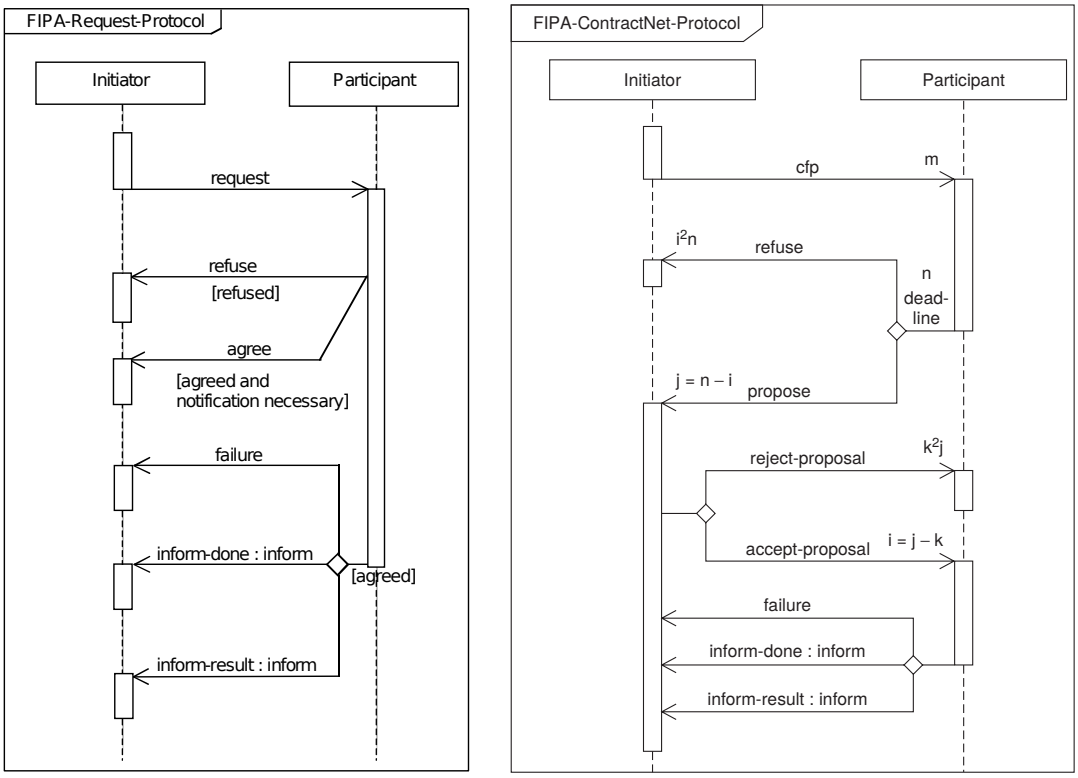
In JADE, the AchieveRE protocol encompasses the multiple "request-like" behaviours such as FIPA-Request. It is a simple protocol with three moments of interaction, as Figure 2.5a shows: a request, a response of acceptance or refusal and a facultative result

Table 2.3: Interaction protocols supported in JADE

Protocol(s)	Initiator class	Initiator class
FIPA request FIPA Query	AchieveREInitiator	AchieveREResponder
FIPA Contract Net	ContractNetInitiator	ContractNetResponder SSContractNetResponder

notification. JADE allows the use of other interaction protocols with the AchieveRE: FIPA-query, FIPA-Request-When, FIPA-recruiting and FIPA-brokering. Interaction using this protocol can be 1:1 or 1:N.

The Contract Net protocol starts with a Call for Proposals (CFP) sent to one or more agents, which can reply with a proposal or with a refusal to propose. The initiator can then accept or reject the proposals. As in FIPA-Request, the final result notification is facultative. The ContractNetResponder class from JADE resets itself after terminating the protocol and stays waiting for new CFPs. JADE provides an alternative responder class called SSContractNetResponder that terminates after a single session (SS stands for single session).



(a) FIPA-Request protocol (b) FIPA-Contract-Net protocol

Figure 2.5: Sequence diagrams for the protocols Contract Net and Request.

In JADE, agent activity is programmed through the notion of behaviours. For interaction protocols, two complementing behaviours are used for each side of the interaction, and JADE's API supports the most important protocols with built-in initiator and responder behaviours.

FIPA Interaction Protocols typify communication interactions among agents by specifying two roles: initiator (the agent starting the interaction) and responder (a participant in the interaction). Each protocol defines precisely which messages are sent by each role ad in which sequence.

In order to create an application using these protocols, programmers only need to extend these behaviours and implement the message handlers. All the complexity regarding the interaction and networking infrastructure is hidden and taken care of by JADE, allowing the programmer to focus on the implementation of agent behaviour.

2.4 Summary

Although the specific problem of reimplementing JADE features in Repast using a pure Java approach has not been approached before in the available literature, it is clear that some related work is useful in the definition of the solution proposed in this thesis.

JREP and MISIA show that interoperability between the two frameworks is possible and they helped understand the limitations of each framework. They both attempted to complement Repast's lack of communication protocols by creating an interface with JADE's implementation of FIPA interaction protocols. The usefulness of these works is limited, though, since the source code is not readily available and neither project is still being developed and supported.

Other works related to the integration of features from Repast and JADE are available and these were selected as the most relevant. While the goal of this thesis is not to use both frameworks simultaneously, these works give a valuable insight into the shortcomings of both frameworks as well as providing some interesting comparisons between their features. Open source projects exist that contemplate the use of FIPA ACL as a library, but none was found to be actively maintained or properly documented. Therefore, this thesis contemplates the creation of a Java API that brings JADE-like features to simulation tools, including FIPA standards.

Chapter 3

Closening MAS Simulation and Development through a JADE-based API

The modelling of MAS can be accomplished through the use of a wide range of available platforms, as explained before. In certain cases, in order to benefit from different features, modelling the same system in multiple development environments may be necessary.

As explained in Introduction, the two mains goals of this thesis are the creation of an adapter to provide abstraction from MABS frameworks, allowing the creation of JADE-based simulations, and the creating of a mechanism that uses this adapter to convert MABS into MAS. To accomplish these goals, an integrated system was designed, which comprises two main components :

1. The **Simple API for JADE-based Simulations (SAJaS)**, which provides a set of features present in JADE; those features were reimplemented from scratch in an attempt to simplify their internal complexity, preserving JADE-like external execution;

2. The **MAS Simulation to Development code convertion tool(MasSim2Dev)** in the form of an Eclipse plug-in that is capable of mapping JADE and SAJaS features and convert applications developed using one into applications based on the other, automatically. The plugin has no Repast or JADE dependencies in the code, buts contains a dictionary file that is specific to JADE and Repast.

This chapter makes a high level overview of both components and how they work together. After an overview in the next section, Section 3.2 explains how FIPA standards for agent interaction and management are implemented in SAJaS. Section 3.3 contains a description of the expected scenarios where this system could be used.

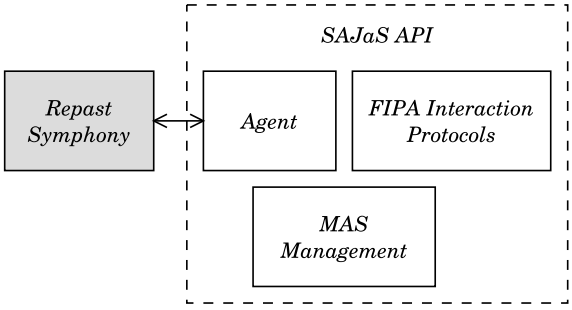


Figure 3.1: Basic structure of SAJaS

3.1 Overview

SAJaS is an API meant to be used with simulation frameworks to enable JADE-based features in them, including agent interaction protocols and agent management services. The API also uses JADE's concept of behaviours which encapsulate most of agents' actions.

SAJaS was initially created to be used with Repast Simphony. However, it was developed in a way that allows its integration with other simulation tools. The interface between Repast and SAJaS is made in a single point, the Scheduler, which is a Repast-specific structure. If developers desire to add support for other simulation frameworks, different implementations of the scheduler can be created.

Another important aspect kept in mind when developing SAJaS was the importance of keeping a close semblance to JADE's own API not only to make the code conversion more straightforward, but to allow proficient JADE developers to create SAJaS-based simulations using a familiar JADE-based API. MasSim2Devwas developed as an Eclipse plugin. It provides programmers with two possible actions: convert code from JADE to SAJaS and in the opposite direction.

SAJaS does not yet implement all JADE features. For the purposes of this thesis, a set of the most common features were selected which allows for the simulation of scenarios with some complexity, including negotiation between agents and the creation of custom behaviours. Features in JADE applications which are not available in SAJaS will typically be shown as errors when converted; MasSim2Devsimply ignores them and they should be re-implemented manually as necessary. Chapter 6 shows examples of scenarios created with SAJaS and JADE that were successfully converted while preserving functionality.

Figure 3.1 tries to compare SAJaS with JRep and MISIA, described in Chapter 2. The API's general structure is simpler; it does not intend to maintain an active connection to a JADE platform, eliminating the need for synchronization. Instead, our goal is to replicate in our API the main features of JADE, allowing for a straightforward and dependency free feature mapping between SAJaS and JADE.

Table 3.1: Information contained in the DF Agent Descriptions and Service Description structures. M: Mandatory field. S: Mandatory when agent registers.

DFAgentDescription	**ServiceDescription**
name ^{S} : AID	service name ^{M} : String
services : ServiceDescriptions	type ^{M} : String
protocols : Strings	
ontologies : Strings	
languages : Strings	

3.2 FIPA Specifications

Part of the rationale for the creation of SAJaS was to enable agent interaction standards in otherwise standard-less simulation frameworks, namely Repast, the one featured in this thesis. This section explains the FIPA standards for agent interaction and management implemented in SAJaS.

As mentioned before, SAJaS follows JADE architecture very closely, including how FIPA standards are implemented. The implemented features can be divided in the following categories:

1. **Agent Management**, which includes the directory facilitator (DF) service, the structures used by it and the Agent Management Service (AMS),

2. **Messaging**, including the ACL Message, the Message Template and the Message Transport Service (MTS), and

3. **Interaction Protocols** that allow agents to exchange ACL Messages in a standardized manner.

3.2.1 Agent Management

The AMS functions as a directory of all agents in a MAS and every agent is automatically registered in the AMS upon its initialization. BY the time they are registered, agents are also assigned an Agent Identifier (AID) used throughout the application. Only the AMS and the agent itself know who the AID belongs to.

The DF is a facultative directory where agents can register themselves as service providers. The DF Agent Description is a structure used to describe one agent when it registers itself in the DF or to describe a group of agents when performing a search. The DF Agent Description can contain one or more Service Descriptions. The fields of these structures are listed in Table 3.1.

3.2.2 Messaging

The implementation of the ACL Message is essentially identical to JADE's. SAJaS also implements the Message Template, which is used to filter incoming messages. A series of template factories exist in the API that allow one to create message templates.

3.2.3 Interaction Protocols

JADE has the support for many interaction protocols. The most common ones were selected to be included in SAJaS. The implemented protocols were the "request-like" Achieve Rational Effect (AchieveRE) protocol, the Contract Net protocol and the Single Session Contract Net - including the Responder Dispatcher behaviour, which uses it.

The AchieveRE encompasses multiple FIPA protocols, namely Request, Query, Request-When, Recruiting and Brokering protocols, as defined in JADE's documentation.

3.3 Usage Scenarios

This section is meant to describe both the scenarios where this system is expected to be useful, as well as the actual possible use cases. Figure 3.2 illustrates these scenarios.

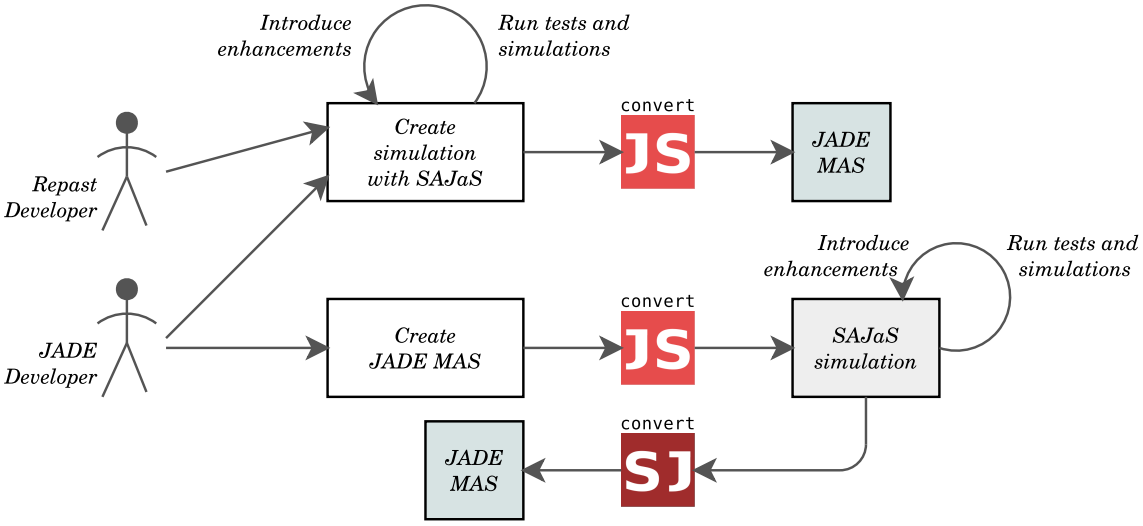


Figure 3.2: Two possible work flows for SAJaS users

One possible scenario is when a JADE developer created a JADE MAS and desires to perform some tests and simulations in a local and controlled environment. The developer can use this tool to convert the MAS into a MABS. Eventually, the application can be converted back if changes were made while in the simulation format.

A second possible scenario could be that of a developer that intends to create a MABS with the goal of later creating a MAS out of it. This could be a Repast developer who

desires to create more complex agent simulations or a JADE developer that wants to create Repast simulations using familiar JADE-like tools.

A third scenario is when a developer simply wants to create a complex agent-based, FIPA-compliant simulation. In this case, there is no need for a code conversion tool, but SAJaS can be used as a standalone library.

Chapter 4

Software Architecture

In this chapter, the composition of the system is described with more detail. One of the challenges when developing SAJaS was deciding how to implement assynchronous JADE-based features in a tick-based environment. This is described in Section 4.1. Following it is the definition of the architecture of the system in the form of conceptual diagrams. Closing this chapter is a discussion on the extension perspectives of the system.

4.1 Agent Execution in SAJaS

As described in Section 2.2, JADE execution can be concurrent and parallel, since JADE supports distributed and multi-threaded agent systems. Execution in Repast, on the other hand, is not concurrent. Repast uses a time-share type of execution, granting each agent the right to perform its tasks until they finish them, in sequence, but in no particular order.

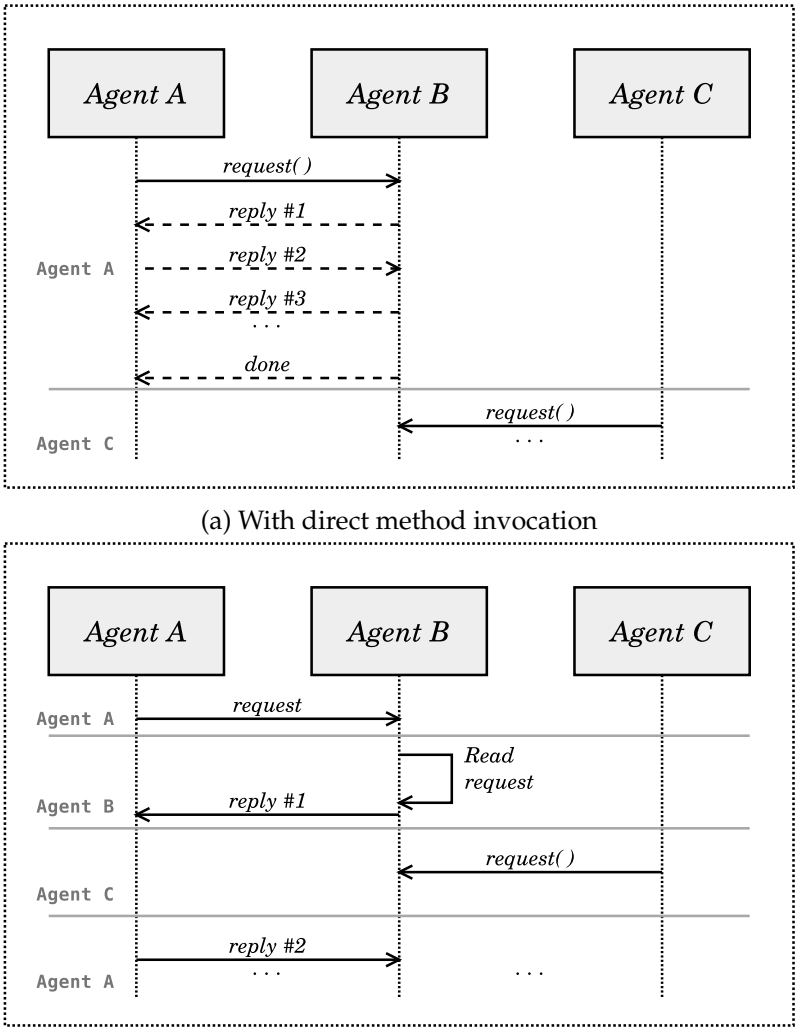
4.1.1 Asynch-like execution in Repast

Except when executed during their setup or takedown, agents' actions in JADE are encapsulated in Behaviours. In JADE, multiple agents can be executing their behaviours simultaneously. In Repast, however, all scheduled objects run consecutively with variable execution sequence. This schedule is one of the components in SAJaS that is specific to its Repast interface. How behaviours are scheduled can be defined by the programmer.

Even though a local application can take advantage of direct method invocation, when the simulation platform is single-threaded - as Repast is - there is a risk of the simulation stagnating if, for instance, two agents engage in a very long "conversation".

Figure 4.1a represents a scenario where two agents engage in a conversation that involves multiple multiple replies from both sides. With direct method invocation, the response is instantaneous but other agents don't get any time to execute in-between.

In Figure 4.1b on the other and, each agent is allowed to execute one task - send one message, in this case. Messages stay waiting until the agent reads and processes it. This way, Agent C didn't have to wait for the other two to finish.



(a) With direct method invocation

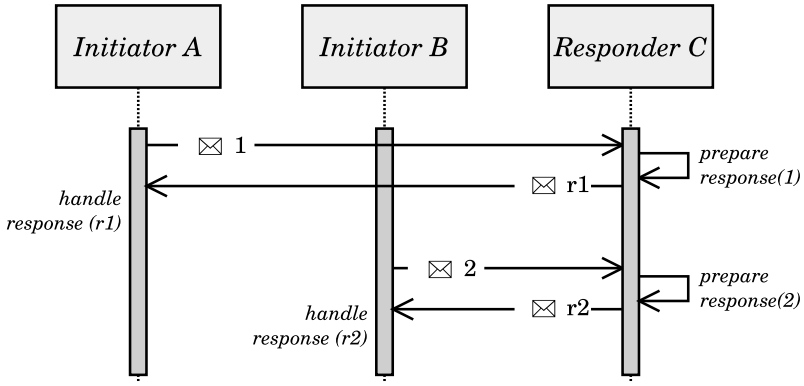
(b) Asynchronously

Figure 4.1: Smaller time "allowances" favour fair time sharing

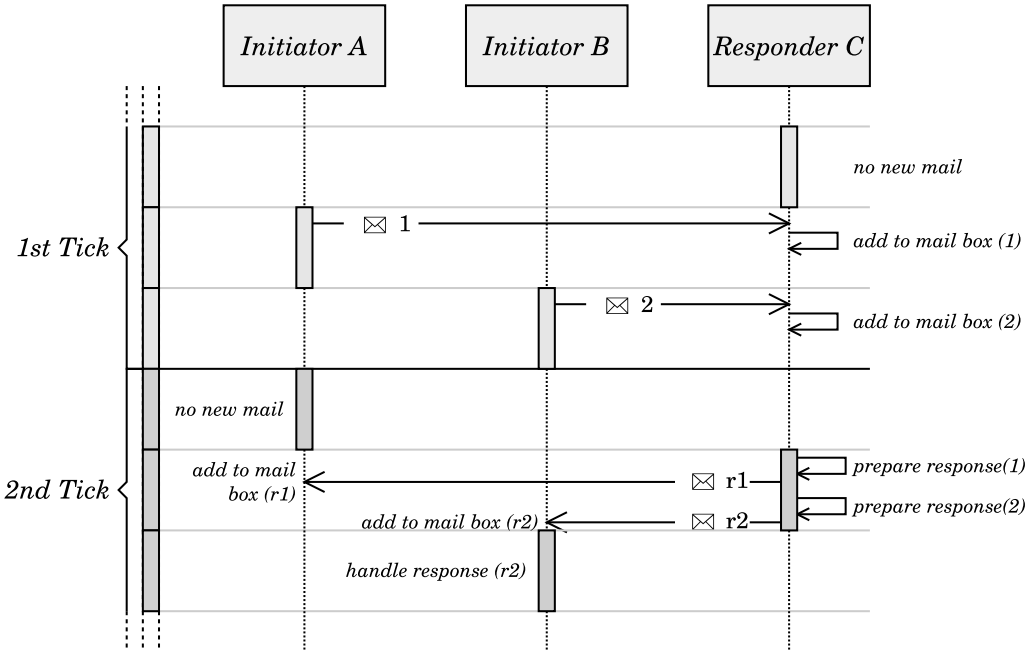
In [MTLD08], authors concluded that increasing the granularity of the system, it is possible to improve its overall performance, even if individual tasks are delayed. The granularity of an agent task is explained as the communication-to-computation, or "a measure of the amount of computation an agent executes before entering the communication phase of one simulation time step".

In SAJaS, as in JADE, agent interaction occurs using the messaging service. Therefore, asynchronous execution is appropriate for the kind of applications developed for JADE and SAJaS. Simulations in Repast, though, usually depend on the synchrony of the environment. The use of ACLMessages, which wait in the message queue until processed, allows to maintain a synchronous execution, while simulating an asynchronous one.

To better demonstrate the differences between agent execution in both frameworks, Figure 4.2 represents a scenario where two agents send a message to a third one who then replies. In SAJaS (Fig. 4.2b), messages are delivered to agent C's message queue,



(a) In JADE, running in parallel



(b) In SAJaS.

Figure 4.2: Communication and agent interaction

and processed only in C's turn. In JADE (Fig. 4.2a), messages can arrive concurrently. Their arrival triggers an event and they are processed right away in the receiving agent's thread. In this case, agent C handles the messages as they arrive and issues the respective replies.

While the diagram above represents the agents as scheduled objects, their behaviours are the ones actually being scheduled and one agent typically initiates multiple behaviours. It is worth noting that the order by which Repast executes each scheduled behaviour is not predictable. To remove the influence that a fixed execution order can have in the outcome of a simulation, the schedule is randomized every tick. As a result, it is not guaranteed that all the behaviours of a single agent are executed consecutively.

This is the expected execution when working with Repast as well as with JADE (given its multi-threaded nature) and it is up to the programmer to ensure that the application does not rely on the order of execution.

4.1.2 Messaging issues

An issue arose while testing multiple concurrent sessions of agent interaction, as in the case of the second experiment in Chapter 6. The initial implementation of the interaction protocols in SAJaS would receive up to one message from the agent's mail queue in each tick. If the agent would receive N CFPs, then it would take at least N ticks to process them all. While this didn't raise any problem with a single session scenario, with multiple sessions, contract net initiators would start to initiate protocols faster than the responders would terminate them. For this reason, SAJaS was changed so that when possible, all matching ACLMessages should be processed in a single tick.

4.2 SAJaS Architecture

From the point of view of the MAS programmer, working with this API feels the same as working with JADE, although limited to the presently available features. However, SAJaS has a much simpler internal architecture, which attempts to implement only the fundamental components needed for everything else to work. The most evident feature that was not ported from JADE was the network layer that enables the creation of distributed MAS.

Section 4.2.1 provides a broad overview of the whole system, followed by a deeper analysis of the implementation of the behaviours and protocols in Section 4.2.2. Section 4.2.3 explains SAJaS' interfacing points, i.e. the way the programmer can effectively use the API.

4.2.1 Conceptual Model

The diagram in Figure 4.3 represents the internal architecture of SAJaS. The module captioned "My Implementation" represents an hypothetical implementation of a MABS that uses SAJaS. The "Repast Simphony" package's purpose is to represent the dependence of one of SAJaS classes to Repast libraries.

Within the API, three different categories of classes are represented.

1. First, inside the "Repast" package are the classes that are specific to the Repast implementation of SAJaS. Repast contains an interface called "ContextBuilder" which contains the build method, the first one called when the simulation is initiated.

Second, classes whose border is dotted are internal classes that are not meant to be used by programmers; some are not present in JADE and therefore their use cannot

be ported by the conversion tool. Finally, the rest rest of the classes are the ones that programmers are expected to use and are directly supported by JADE and by the code conversion tool.

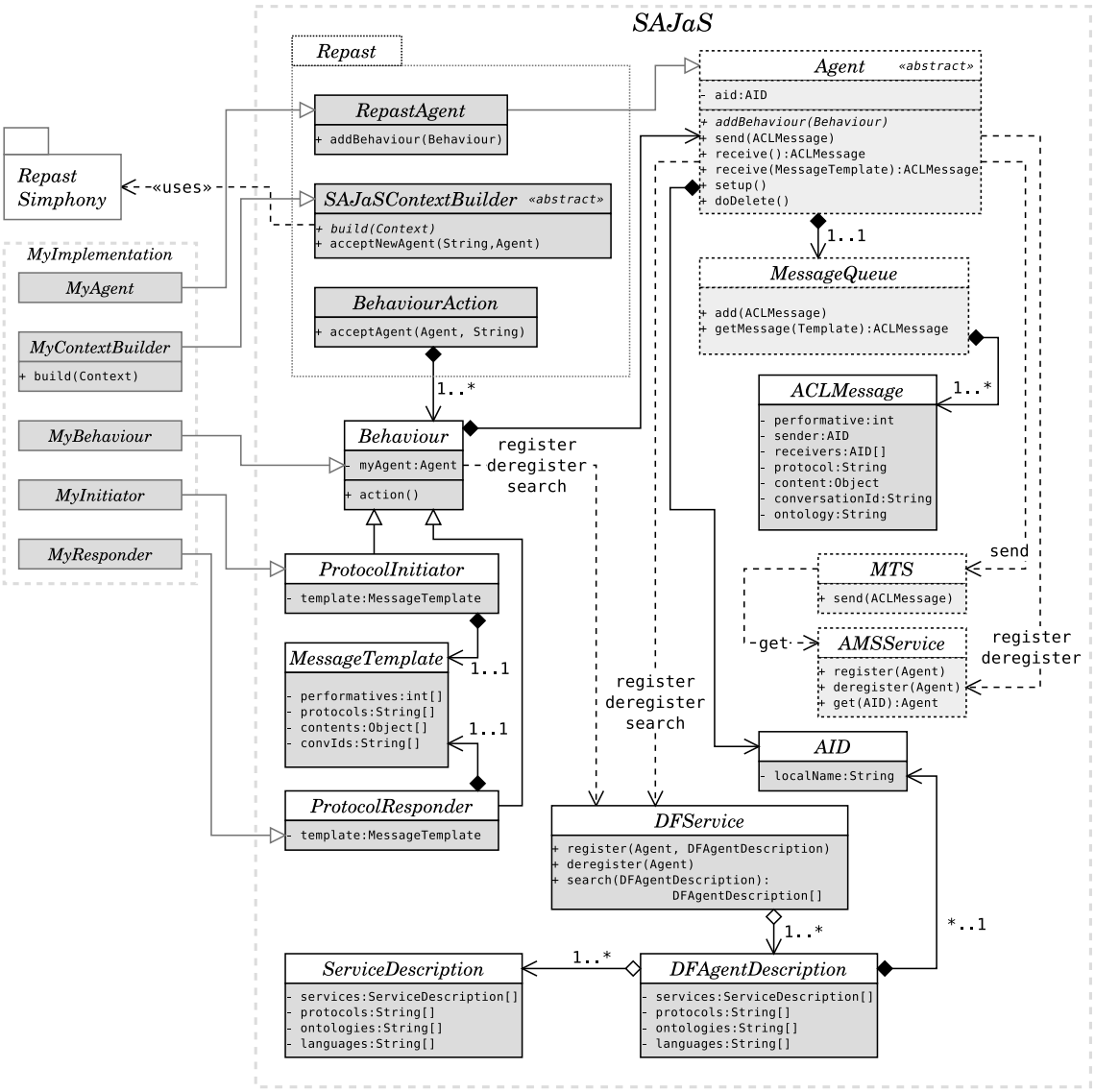


Figure 4.3: Architecture of SAJaS. The classes with doted border are internal.

While tasks can be performed during agent setup and takedown, in runtime they are typically executed inside Behaviours. In the diagram, the Behaviour, the Protocol Initiator and the Protocol Responder are actually representations of a family of classes. For simplicity of this diagram, their complete representation is isolated in Figure 4.4.

An agent in SAJaS contains a MessageQueue which holds its queue of ACLMessages that are yet to be processed. The behaviours access to the agent who owns them and can call `receive()` on it to obtain the next message that matches the a template.

The DFService, as described before, provides the yellow page service. Agents can register or deregister themselves from the DF and perform searches. The DF contains a

set of DFAgentDescriptions that describe agents and the services they perform. To quickly fetch agents when a search query arrives, the DF contains a mapping of the agents that are registered with each service, protocol, ontology and language. That means that, internally, the Directory Facilitator actually contains multiple sub-directories.

Agent interaction is always established through the use of ACLMessages. At no point to agent have access to other agents. Agents use an AID to identify themselves and other Agents. Besides the Agent itself, only the AMSService knows the address corresponding to an AID.

Behaviours that implement interaction protocols contain a MessageTemplate. This template is not fixed, but changes depending on the state of the protocol. The template is used to retrieve one message at a time from the agent's MessageQueue. One template can match multiple ACLMessages in the queue, but it always returns one single message.

4.2.2 Behaviours and Protocols

Figure 4.4 represents the Behaviours with more detailed, as implemented by SAJaS. The Behaviour superclass contains methods that re triggered by certain events.

1. `action` is called once per tick
2. `onStart` is called immediately before the first call to `action`
3. `onEnd` is called right after the behaviour ended;
4. `done` is called every tick to determine if the behaviour has ended and if true is returned, it triggers `onEnd`.

The methods `action` and `done` are abstract in Behaviour, so other behaviours have to implement it. Methods `onStart` and `onEnd`, though, are implemented but do nothing in Behaviour.

The five "responder" and "initiator" classes implement the two protocols currently available in SAJaS. The FSMBehaviour, which they extend, contains a dynamic list of states and transitions. These can be registered and unregistered at runtime, typically before initiating the behaviour or in its setup. Each state is itself a behaviour.

Each protocol can be represented as a different state machine. For instance, in a Contract Net, the initiator's state starts as "sending CFP", then "waiting for replies", "waiting for result notifications" and finally "finished". Protocols in SAJaS were initially implemented using Java `enums`. An `enum`[1] is an immutable variable type that can only take a predefined set of constant values.

Even though it is important to preserve JADE-like execution, simplifying non-essential internal features that are invisible to the programmer provides increased performance.

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

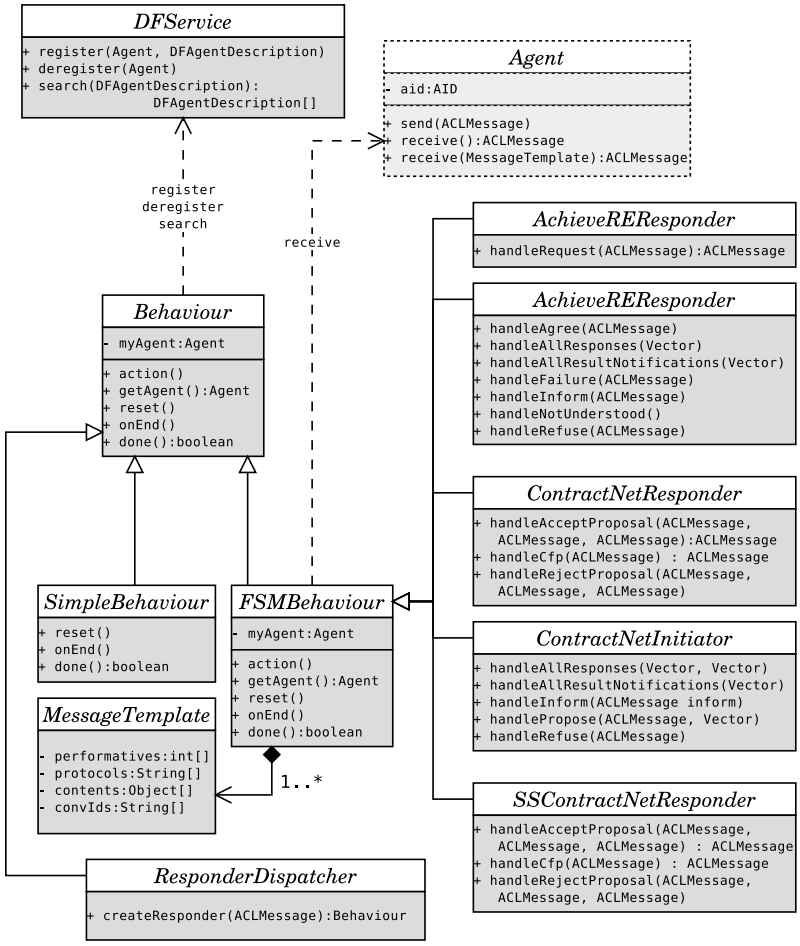


Figure 4.4: Details of the Behaviours and Protocols in SAJaS

Tests were performed to decide whether these behaviours should use `enum`s or the dynamic FSM. Three possible solutions were tested:

1. **First**, a JADE-like approach where each state is a Behaviour in a dynamic FSM;
2. **Second**, an hybrid approach, using a single state in a dynamic FSM, where that state is an `enum` containing the actual FSM; this approach is more faithful to JADE's than using olely `enum`s;
3. **Third**, a pure `enum`-based approach.

Using the contract net scenario describe further ahead in Chapter 6, the performance of these approaches was compared. A pure `enum`-based FSM was shown to be significantly faster than a dynamic FSM. For this reason, the `enum`-based approach was kept and overrides the dynamic FSM from the superclass. The FSMBehaviour was also kept to allow the creation of custom behaviours by programmers using SAJaS.

4.2.3 Using SAJaS

To use SAJaS, it is important to understand how it should be interfaced. As explained earlier, the classes that are represented with a dotted border in Figure 4.3 are internal and not supposed to be used by programmers because there is no JADE support for them. For programmers with experience in JADE, using SAJaS is very straightforward, although not all features all present.

Creating agents in SAJaS should be done using the appropriate implementation aimed at the simulation framework – RepastAgent, in this case. The Agent class in SAJaS is abstract and implementing it directly would require the reimplementation of abstract methods by each agent. Specifically, the `addBehaviour` method schedules a Behaviour and this scheduler is part of the simulation tool.

Creating Behaviours is also done using one of the already implemented subclasses. Each protocol behaviour, for instance, contains a series of message handlers that should be implemented when needed.

The messaging service is accessed through the `receive` and `send` methods in the Agent class. The DF Service is also accessible directly through the DFService class.

For MABS based on SAJaS and Repast, a program launcher must implement the Repast interface `ContextBuilder`. The `SAJaSContextBUilder` class implements it in order to provide an extra abstraction layer from repast. To create a Repast launcher, the programmer should extend `SAJaSContextBUilder` and implement the `setup` method. The ContextBuilder does not exist in JADE and therefore, conversion doesn't yet create a runnable JADE application. However, all that needs to be done is to call the `setup` method from within the JADE main class.

4.3 Extension Perspectives

SAJaS was developed keeping in mind the possibility to create extensions to it's features in the future. While this thesis was initially focused on providing Repast with support for FIPA standards for agent interaction (and all the infrastructure that supports it), the API evolved to incorporate many more JADE-based features. It is purposely generic and to allow support for other simulation development tools to be added will few changes to the API itself. To support another framework, a scheduling mechanism should be crated that is appropriate for that framework. That scheduler should run call the BehaviourAction one per "tick".

Chapter 5

Code Conversion Tool

There are multiple ways to tackle the problem of code transformations. The brute force approach would be to parse the source code, create an abstract syntax tree (AST), which represents all code constructions in a program, perform certain transformations in the tree, and then generate back the code from the new AST. Fortunately, there are free and open source projects that developers can use to do exactly this with significantly reduced effort.

From the available tools, three of the most relevant ones were selected, i.e. tools for Java code transformation that are open source, well documented and still supported. They were Spoon, ATL and JDT. Before proceeding to the the overview of the developed code conversion tool in Section 5.1, Section 5.1

5.1 Background

5.1.1 Spoon - Program Analysis and Transformation in Java

Spoon is a tool created to take advantage of the new features introduced on the release of Java 5, namely annotations and generics. It is a Java transformation tool that uses annotation processing, compile-time reflection and templates in a pure-Java environment to enable programmers to create their own platforms for code transformations. These transformations occur at compile time and annotations are used as parameters for compilation. The programmer uses plain Java code to define the transformations that should occur, for instance, adding code snippets to the beginning of a method in a class. Finally, Spoon can be seamlessly integrated in an IDE such as Eclipse [PN+06].

5.1.2 ATL - ATLAS Transformation Language

ATL - *ATLAS Transformation Language*, is both the name of a language and its enclosing plug-in and allows the creation of model transformations. ATL does not focus on applying transformations to the source code.

As explained [JK06], "a source model Ma is transformed into a target model Mb according to a transformation definition mma2mmb.atl written in the ATL language", which is also a model itself. The three must conform to their respective metamodels MMa, MMb and ATL which then conform to the metametamodel MOF.

5.1.3 Eclipse Java Development Tools (JDT)

The Java Development Tools[1] are are a group of tools that provide Eclipse the necessary means to become a full-featured Java IDE. These tools make up what we perceive as the "Eclipse IDE" and programmers can use these libraries in their Java projects and perform tasks that require a certain introspection of the code itself.

JDT was chosen for the development of the code generation tool. Some of its most interesting features are the automatic cloning of projects, the handling of classes, imports, methods and fields as objects and the possibility of doing complex manipulation tasks without parsing the code. It does, however, allow the use of a high level AST for a more direct manipulation of the source code.

JDT can be broken down into five main components:

APT - Annotation Processing Tool
: This component can be used to parse annotations in the code. Annotations are a Java construction introduced in Java 5 and can be used to add meta information about classes, objects and methods. These annotations can then be parsed by frameworks that use the java object that enclose them.

Core - Java IDE headless infrastructure
: The core for the Java IDE, it allows to transverse the Java element tree and find package fragments, compilation units, binary classes, types, methods and fields. This is the most extensively used component by the code transformation tool.

Debug - Debug support for Java
: This tool makes the debug facilities of Eclipse possible. It provides functionalities such as execution control and contextual expression evaluation.

Text and UI - Java editing support and Java IDE user interface
: These components make development in Eclipse possible by providing a GUI for code editing, enriched with syntactic errors highlighting, among other interesting aids.

5.2 Specification

JDT was the tool chosen to develop the code conversion tool. It is accessible to plugin developers from within Eclipse. Using JDT, it is possible to handle Java concepts as

[1] https://www.eclipse.org/jdt/

objects, including classes, methods, fields, and import declarations. The plugin acts as a translator that, as Figure 5.1 suggests, changes the MABS dependencies from one platform to the equivalent in the other platform.

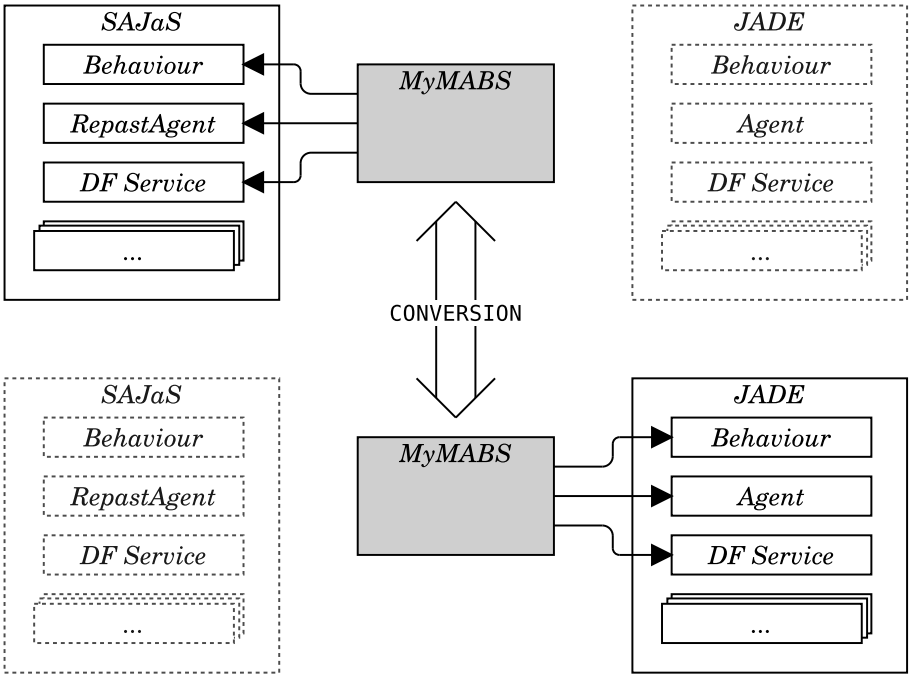


Figure 5.1: Representation of the conversion of code.

After conversion, no dependency to the previous platform exists in the generated project. Naturally, this is not true is JADE features that are not yet available in SAJaS are used in the MAS. The same happens when using internal features from SAJaS that are not present in JADE.

5.3 Execution

Internally, plugin execution can be described in three events: activation of the plugin, search for classes in the dictionary, and applying of dictionary rules.

When this plugin is installed, two icons should appear in the toolbar of the IDE, as shown in Figure 5.2. Clicking one of the buttons, triggers the activation of the respective action. To run the plugin, a project should be selected from the Project Explorer and the button corresponding to the appropriate task should be pressed. The button SJ converts a SAJaS application into a JADE MAS and the button JS converts a JADE MAS into a SAJaS application.

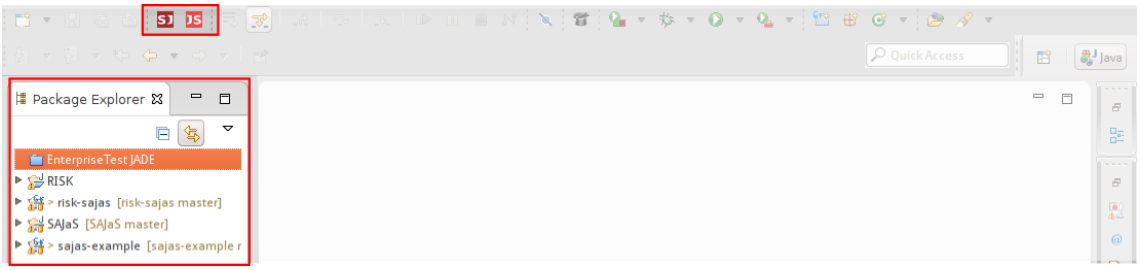


Figure 5.2: Screen capture of Eclipse, showing the two plugin buttons and the Project Explorer view.

When the plugin is activated, it triggers a series of actions.

1. Clone the selected project;
2. Change all references to SAJaS classes in class imports;
3. Inject needed libraries in the new project and add them to the build path;
4. Fix hierarchy (e.g classes that extended `RepastAgent` must now extend `Agent`).

The following pseudo-code snippet allows a better understanding of the conversion algorithm.

```
main:
  dictionary <= loadDicitonary()
  project <= getSelectedProject()
  newProject <= project.clone()
  newProject.open()

  foreach packg in newProject
    foreach compilationUnit in packg
      if compilationUnit.type = JavaClass
        class <= compilationUnit
        applyTransformations(class)
      end-if
    end-for
  end-for
end.
```

The main algorithm of the plugin which traverses the project

```
applyTransformations(class):
  imports <= class.imports
  foreach import in imports
    if dictionary.containsEntry(import)
      // Applies transformations here
      entry <= dictionary.getEntry(import)
      class.imports[import] <= entry[import].value
```

```
        if entry[import].isSuperclass
          setSuperClass(class, entry[imp].value)
        end-if
      end-if
    end-for
  return.
```

The algorithm to apply the transformations

When the hierarchy needs to be fixed, which is represented in the previous pseudo-code as a call to `setSuperClass(..)`, it means that the Java type name of the superclass is not the same for JADE and SAJaS. Currently this occurs solely for the Agent class. Throughout SAJaS' code, all references to the agent use the abstract type `core.Agent`. However, agent types in simulations based on SAJaS+Repast always extend the `RepastAgent` type.

To perform the mapping of the class imports between the frameworks, a dictionary file exists within the plugin. It allows for quick upgrades to the tool in the future without having to edit the actual code. The dictionary contains annotations that inform how to deal with the hierarchy fixing problem. It also contains information about extra dependencies, such as JADE and Repast libraries. Table 5.1 contains the information available in the current version of the dictionary about the imports and superclasses.

5.4 Extension Perspectives

MasSim2Devis closely related to SAJaS, so updates to SAJaS will demand changes to the plugin as well. To facilitate its update, MasSim2Devrelies on a dictionary file that contains a mapping of the classes between SAJaS and JADE. The dictionary also supports some annotations that allow more that just a direct translation between two classes. Other extensions to the hardware would require deeper changes to the code.

Table 5.1: Contents of the dictionary file inside the plugin. The classes marked with a › symbol are superclasses.

SAJaS **up.fe.liacc.sajas**	**JADE** **jade**
core.AID	
core.behaviours.SimpleBehaviour	
domain.AMSService	
domain.DFService	
domain.FIPAException	
domain.FIPANames	
lang.acl.ACLMessage	
lang.acl.MessageTemplate	
lang.acl.UnreadableException	
proto.AchieveREInitiator	
roto.AchieveREResponder	
proto.ContractNetInitiator	
proto.ContractNetResponder	
core.behaviours.FSMBehaviour	
proto.SSContractNetResponder	
proto.SSResponderDispatcher	
proto.ProposeInitiator	
proto.ProposeResponder	
wrapper.AgentController	
domain.FIPAAgentManagement.DFAgentDescription	
domain.FIPAAgentManagement.ServiceDescription	
domain.FIPAAgentManagement.SearchConstraints	
core.behaviours.Behaviour	
›core.RepastAgent core.Agent	›core.Agent

Chapter 6

Validation

The main goal of this thesis was to develop a solution for bridging the gap between the simulation and MAS domains that, while maintaining a familiar JADE-like environment, would offer significant performance improvements to MABS development. To validate this approach and the developed tools, a set of experiments were designed, in an attempt to cover and test all the available features.

The first example consists in a simple contract net between one buyer and multiple sellers. In the second example, multiple contract nets run concurrently and some of the buyers include available computational trust about sellers. The third example is a board game called Risk developed prior to this thesis. The goal of this last example was to test SAJaS in a "real" example of an already developed MAS, one that had not been developed specifically for this thesis. All tests were performed in a laptop using an Intel i7 CPU (8 logical cores) at 2.20 GHz and 8GB of RAM.

6.1 Simple Contract Net

In this scenario, one agent sends a call for proposals (CFP) to multiple agents which then reply with their individual proposals. This simple scenario was created to test the core of SAJaS and the plugin. The goal was not only to create a working simulation model based on SAJaS, but also to demonstrate that the execution of a simulation in SAJaS and the equivalent JADE application generated from it are identical and that performance in SAJaS is higher. For that purpose, this scenario was developed as a simulation in SAJaS and Repast and then converted to a JADE MAS.

6.1.1 Experimental Setup

The diagram in Figure 6.1 illustrates the contract net created for this test. An agent (the buyer) intends to purchase a certain quantity of three kinds of goods: rice, flour and oats. Besides the quantities of each product it needs, the buyer also stipulates a maximum price

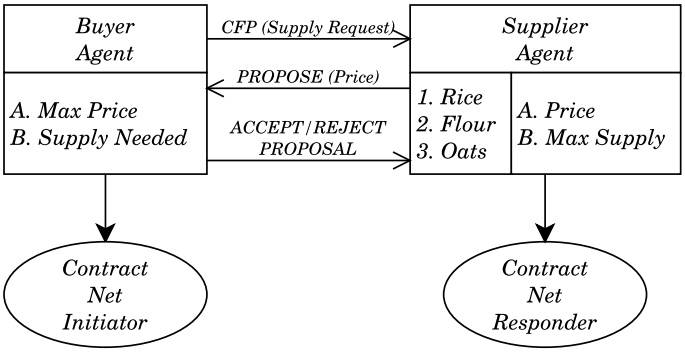


Figure 6.1: Representation of the contract net scenario.

for the whole deal. The buyer will issue a call for proposals (CFP) containing a request for supplies to all agents that announce themselves as suppliers in the DF.

Supplier agents have a maximum supply capacity and a price for each product. After receiving a CFP, the supplier replies with a PROPOSAL containing a price for each product if the demanded supply is within the seller's capacity. Otherwise, a REFUSE message will be sent to the buyer. Finally, the buyer agent compares all valid proposals, chooses the cheapest offer for each of the three products and replies with an ACCEPT PROPOSAL to the best offers, and REJECT PROPOSAL to all others.

To ensure the proper comparison of results, a fixed data set with values for prices and demand was used in both frameworks. This experiment focused on two simple metrics to evaluate the result: time and outcome.

Multiple executions were performed with varying numbers of suppliers (as sugested by Figure 6.2); for each case, the simulation was executed 10 times. The execution time was measured from the begining of the protocol, until all suppliers were notified.

In JADE, the simulation was tested in two different setups: first, with all agents running in a single container; second, with the supplier agents in one container and the buyer in a separate one (but in the same host). In this configuration, all communication between agents happens across containers. The second validation metric was the actual result of the protocol, i.e who were the supplier agents chosen by the buyer and their price proposals.

6.1.2 Results

After 10 executions, the average performance of the experiment was calculated for each number of agents and is represented in Figure 6.2. The performance of the simulation based on SAJaS was significantly better, excelling when the number of agents is high. JADE was able to perform better when using two distinct containers.

Regarding the outcome of the protocol, the same values were obtained in both implementations for each number of agents, confirming that the execution is identical in both implementations.

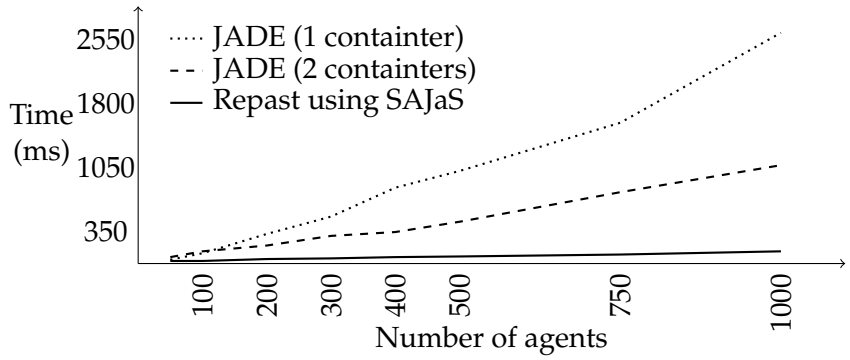


Figure 6.2: Average execution time of each framework in the different experiments.

6.2 Multiple Contract Net using Trust

This scenario is similar to the previous one, but attempts to perform a better coverage of the features present in SAJaS, namely the use of the AchieveRE Protocol and the Responder Dispatcher which allows for multiple contract nets to be handled without deadlocks occuring. However. rather than comparing the exact value obtained from the simulation as in the previous scenario, the goal is to compare the overall behaviour of all agents.

6.2.1 Experimental Setup

This simulation is composed by multiple buyers and multiple sellers running simultaneously. After the sellers register themselves in the DF, each buyer will perform a search for sellers of the particular good it needs to purchase. Then, the buyer sends this list of agents to the CTAgent (CT standing for computational trust). The CTAgent will calculate a trust value for each seller based on its past contracts with buyers and return the top 5 sellers.

With this information, the buyer sends a CFP only to the 5 top agents and accept the best proposal from them. Some sellers will occasionally violate the contract after acceptance. The buyer will then inform the CTAgent if the contract was fulfilled or violated. The compilation of this composes the trust of the seller.

Some buyers are programmed to ignore trust and rely solely on the proposal. The idea is that informed buyers eventually avoid contacting sellers programmed to violate contracts more often. The goal of this experiment is to model this scenario in SAJaS, convert it to JADE and verify that the obtained results are very similar.

After one contract is concluded, Buyers start a new one, performing a new DF search for the next product they want to buy, request information from the CTAgent and issuing a new CFP.

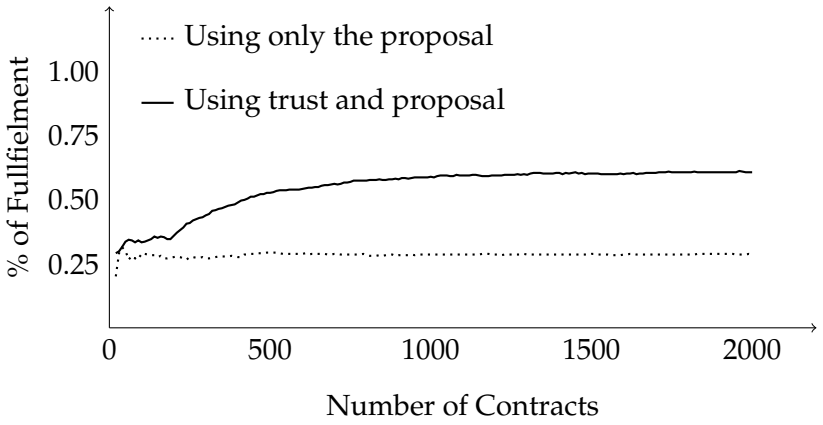


Figure 6.3: Average result of 5 executions of the Multiple Contract Net scenario in JADE during 2000 contracts

6.2.2 Results

The experiment was executed 5 times in JADE and 5 times in SAJaS. The scenario is composed of 20 buyers using computational trust, 20 not using trust and 80 sellers. A total of 2000 contracts were recorded to create the following charts in Figures 6.3 and 6.4. As shown, buyers who made use of computational trust had more successful contracts. The fluctuations early in the simulation are due to the initial lack of trust information.

As expected and as shown in the charts, the same outcome was observed both in JADE and SAJaS. With this experiment, it was possible to test the Request protocol - when requesting computational trust, the Contract Net - when purchasing goods, the Responder Dispatcher - to handle CFPs from multiple buyers concurrently, the messaging system and the DF service.

In terms of performance, the outcome of this experiment, as ilustrated by the chart in Figure 6.5, may not seem as impressive when compared with the first experience. However, the algorithm employed to calculate trust is very time consuming. In a setup without the use of trust by any of the buyers, the performance difference between JADE and SAJaS is comparable to the first scenario.

6.3 Risk Multiplayer Game

The previous examples covered most SAJaS features. However, it is important to test the API and the conversion tool with a real working example – a MAS that was not originally developed with the API in mind. For this experiment, an agent-based board game create prior to the development of SAJaS was selected.

RISK is a multiplayer strategy board game played in turns. It is a game currently owned by Hasbro and the full rules manual is available online [1]. The game used for this

[1] http://www.hasbro.com/risk/

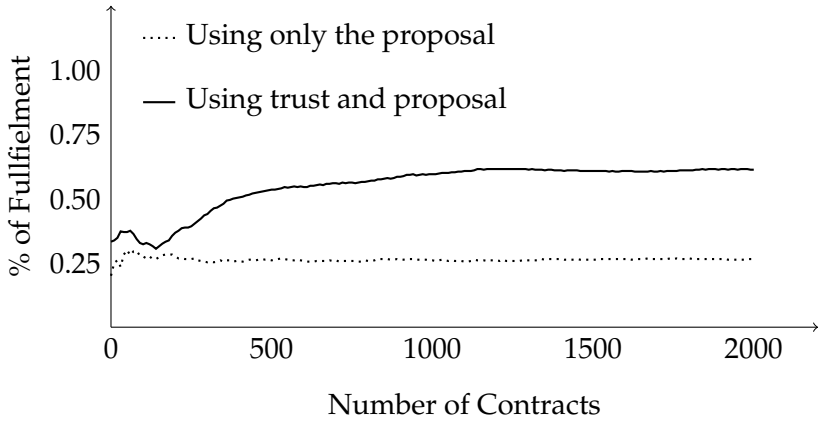


Figure 6.4: Number of contracts executed in JADE and SAJaS. Final number of contracts is 2000 for both frameworks.

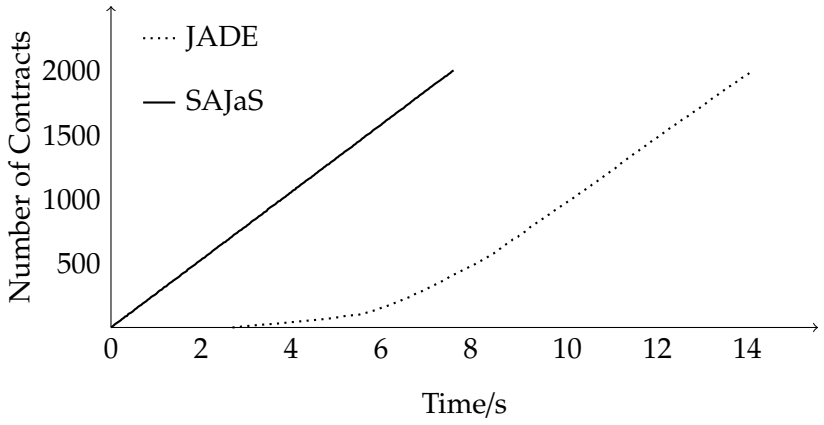


Figure 6.5: Average result after 5 executions of the Multiple Contract Net using Trust scenario in JADE

experiment was developed with JADE.

6.3.1 Game definition and architecture

RISK was implemented as a multi-agent-based game where each agent represents one player[GC14]. A total of five players participate in the game. An extra agent exists as the game manager, responsible for synchronizing the board state with everyone else and for validating moves.

The game board is a world map containing 52 territories that loosely resemble real countries and regions of the world. The territories are grouped in 5 continents that, when fully conquered, award the player bonuses. The game manager produces the GUI shown in Figure 6.6. Each territory can host an unlimited number of soldiers, which players can move to adjacent territories. Each player also receives more soldiers each round. The goal of the game is to engage in battles against enemy adjacent territories. Territories with more soldiers have higher chances of defeating the enemy when attacking or defending a territory. The winner is the player that is able to conquer all territories.

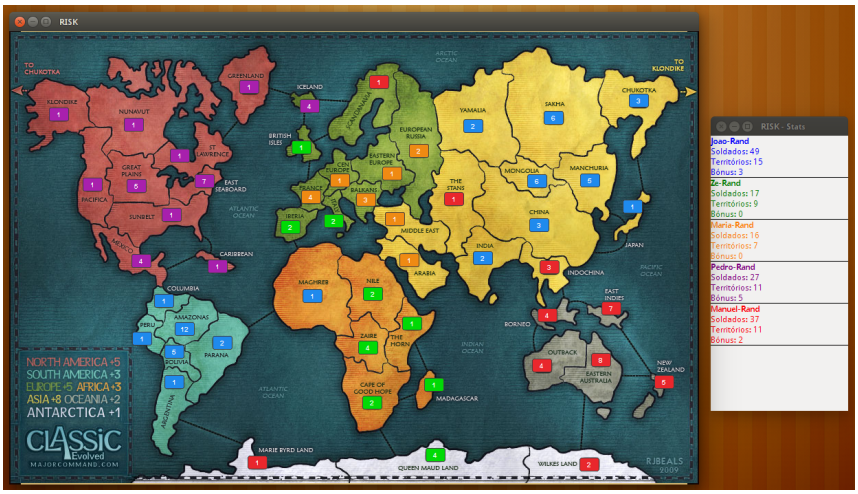


Figure 6.6: Risk game board

6.3.2 Experimental Setup

Player agents have different behavioural architectures and are classified as aggressive, defensive, opportunistic or random. Communication occurs between the players and the game agent mainly using the FIPA REQUEST protocol. The game also heavily relies on custom Finite State Machine Behaviours (FSMBehaviour) to control game progress.

To evaluate the performance of the game, logging features were introduced to the original source code of the application. The original JADE MAS was converted using the developed plugin and two metrics were selected for comparison of the two applications. The first metric was the overall performance of the applications, obtained in the form

of total number of game rounds on the course of ten seconds. The second metric is the overall result of the game, measured as the total number of soldiers each player has over the course of the game.

Two game setups were also used when comparing both metrics. For the first setup, five random agents were put to play against one another. This typically results in a game that never ends, since these players do not follow any kind of strategy. For the second setup, one deliberative player agent and four random agents would play together. In this setup, the strategy of the deliberative player typically assures its victory.

6.3.3 Results

As a first test, RISK was executed with 5 Random player agents. Random players do not follow any plan and the game has usually little progress over time and no one ever wins. Eventually, random players start hoarding soldiers and eventually their army is so large that the game stagnates.

The game was run for 120 rounds and the total elapsed time was used to produce the chart in Figure 6.7. The chart represents the average values over 5 repetitions of the same experiment. It is possible to verify that when the same scenario was run in SAJaS, there was a significant increase in performance.

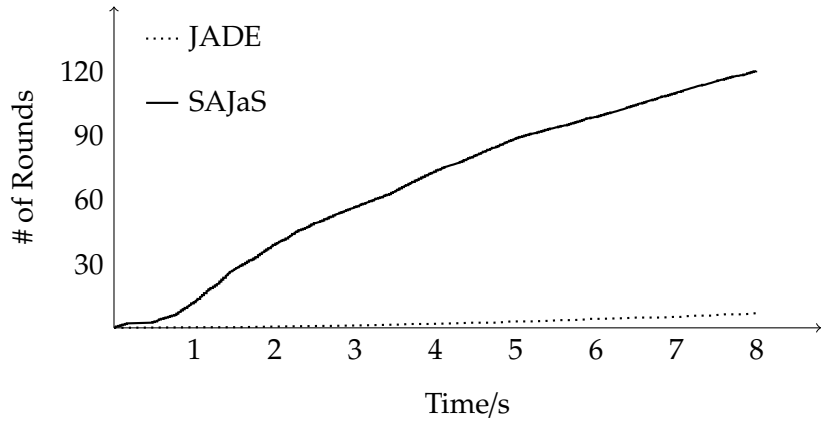


Figure 6.7: Performance of a Risk match with 5 random agents. (DRAFT)

In order to demonstrate the proper execution of the game, another scenario was created, containing 4 Random players and 1 Deliberative player. The Deliberative player follows a strategy that allows it to choose the best way to dispose its soldier on its territories, decide the best time and place to attack and which territories to fortify. This way it has a clear advantage over an agents that attacks and moves soldiers randomly. The chart in Figure 6.8 shows the average outcome, in total number of soldiers, throughout a whole game until the deliberative wins. In the chart, the number of agents for the Random agent is actually the average of all 4 Random agents.

Figure 6.8: Average outcome of a Risk match with a deliberative agent.

6.4 Summary

These three experiments were designed to validate the results of this thesis.Technically, the scenarios described in this chapter covered all currently available features in SAJaS. The Achieve RE protocol was used in the Enterprise scenario to communicate with the CT Agent. and it was widely used in Risk for all communications. The Contract Net protocol was covered in the first two experiments. The Enterprise scenario also allowed multiple concurrent contracts to take place by using the Responder Dispatcher.

All scenarios made use of the DF service, the AMS, the MTS and of containers like the ACL Message, the Message Template, the DFAgentDescription and the Service Description. The creation of custom Simple Behaviours and Finite State Machine (FSM) Behaviours was covered by Risk. It was possible to demonstrate that bringing JADE and Repast together is not only feasible using the developed tools, but also provides increased performance when compared with JADE MAS.

Chapter 7

Conclusions

MAS are used in many different domains and research based on simulation has seen an increase in the use of agent-based approaches in their development. As a result, many frameworks for the development of MABS have been created. With a review of the available literature, it is possible to conclude that there is interest in bridging the domains of MAS and MABS. While MABS are typically not used in production, MAS are usually not the most appropriate to perform tests and simulations, more so when large number of agents are in use.

Some works were studied whose motivation was to bridge these domains. MISIA, JRep and PlaSMA proposed approaches that extended the MAS development framework JADE with simulation development features. This thesis proposed an alternative approach in which JADE-based features were included in the MAS framework Repast without relying in JADE libraries – those features were re-implemented for this purpose.

7.1 Main Contributions

To bridge the gap between MAS development and simulation, the Simple API for JADE-based Simulations (SAJaS) was created. JADE developers are able to create Repast-based simulations using familiar JADE features. The MAS Simulation to Development code convertion tool(MasSim2Dev) was also developed to enable the conversion of simulations created with SAJaSinto JADE MAS.

The main issues detected during the development of SAJaSwere related to to the different nature of the execution of Repast and JADE. JADE's agents run in multiple threads and global agent synchronization does not exist. Agent interaction happens asynchronously. The kind of simulation frameworks targeted by this API, like Repast, relies on synchronized events that occur consecutively in "ticks". By using ACLMessages in Repast and a messaging system that relies on messages being kept in a mail queue, asynchronous agent interaction is possible despite Repast synchronism. Messages are processed when the simulation grants execution the the agent who owns them.

Negotiation scenarios were created to validate both SAJaSand MasSim2Dev. The goals were to demonstrate that it was possible to create agent-based scenarios with some complexity using SAJaS's implemented features and that after its conversion using MasSim2Devwas not only feasible, but the resulted MAS displayed identical behaviour when executed. Furthermore, the SAJaSversion is significantly faster. The two negotiation scenarios showed that a Repast simulation based on SAJaSwas many times faster than its JADE equivalent.

7.2 Future Work

Although the development and validation of SAJaS proved its concept and is already capable of being used to developed some kinds of MABS, there is still room for development and future enhancements and not only to the API and the plugin. Following are some interesting ideas of future work.

7.2.1 Increasing SAJaS support of JADE features

The most immediate extension to this project is to extend the rage of supported JADE features present in SAJaS. While it is important to maintain the simplicity of SAJaS's internals to avoid negatively affecting simulations performance, to allow the conversion of even more complex scenarios. The core of SAJaS allows to easily extend the API with more interaction protocols and other types of behaviours.

7.2.2 Extending native support to other simulation tools

While developing SAJaS, a conscious effort was made to keep the API very generic, defining a clear separation within SAJaS between Repast-specific elements and everything else. This modularity allows future extensions without changing the API and opens doors to future integration with other simulation tools.

7.2.3 Enhancing the plugin with more options and features

Possible enhancements to the plugin could include providing support for user configurations. From the point of view of the user of the plugin, the code conversion tool provides no flexibility beyond choosing the direction of conversion (from SAJaS to JADE or the reverse). For instance, the plugin could allow the selection of the name and location of the newly generated project, enable the manual selection of the classes to be converted (presently, the whole project is always cloned and transformed) and provide automatic creation of "stub launchers" that would allow to quickly test is the generated project executed correctly.

Other configuration options could affect specific internal traits of the generated source code. When a protocol behaviour is expecting messages, one by one all messages that match its template are handled in that tick. The rationale for this design choice was described in Section 4.1.2. However, the developer may desire to allow only a single message to be processed in each tick.

7.2.4 Enable support for charts and displays in SAJaS

One interesting feature in Repast is to be able to create real time visualizations of agent data. This is possible in part because agents in Repast are executed locally, so access to this data is facilitated. In could be interesting to include a information display tools - like charts and grids - that could be ported between frameworks. A "Display Agent" would handle all visualizations and agents would provide him with the information to display. This agent could be fully compatible with JADE and be converted with the plugin.

References

[ADK+14] T Ahlbrecht, J Dix, M Köster, P Kraus, and Jörg P Müller. A scalable runtime platform for multiagent-based simulation. Technical report, Technical Report IfI-14-02, TU Clausthal, 2014.

[All09] R Allan. Survey of agent based modelling and simulation tools. *BT Technology Journal*, 2009.

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007.

[BMR+08] M Balmer, K Meister, M Rieser, K Nagel, Kay W Axhausen, Kay W Axhausen, and Kay W Axhausen. *Agent-based simulation of travel demand: Structure and computational performance of MATSim-T*. ETH, Eidgenössische Technische Hochschule Zürich, IVT Institut für Verkehrsplanung und Transportsysteme, 2008.

[Col03] N Collier. Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36, 2003.

[DU00] J Dávila and M Uzcátegui. Galatea: A multi-agent simulation platform. In *Proceedings of the International Conference on Modeling, Simulation and Neural Networks*, 2000.

[Fer99] Jacques Ferber. Multi-agent systems: an introduction to distributed artificial intelligence. 1999.

[GC14] J S. V. Gonçalves and P Costa. Multi-Agent-based Risk. May 2014.

[GHM+11] J Gormer, G Homoceanu, C Mumme, M Huhn, and J Muller. Jrep: Extending repast simphony for jade agent behavior components. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 02*, pages 149–154. IEEE Computer Society, 2011.

[GRM+11] E García, S Rodríguez, B Martín, C Zato, and B Pérez. Misia: Middleware infrastructure to simulate intelligent agents. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 107–116. Springer Berlin Heidelberg, 2011.

[JK06] F Jouault and I Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[KEBB12] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5:128–138, December 2012.

[MTLD08] D Mengistu, P Troger, L Lundberg, and P Davidsson. Scalability in distributed multi-agent based simulations: The jade case. In *Future Generation Communication and Networking Symposia, 2008. FGCNS'08. Second International Conference on*, volume 5, pages 93–99. IEEE, 2008.

[PN+06] R Pawlak, C Noguera, et al. Spoon: Program analysis and transformation in java. 2006.

[TW04] S Tisue and U Wilensky. Netlogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, pages 16–21, 2004.

[Woo08] Michael Wooldridge. *An introduction to multiagent systems*. Wiley. com, 2008.

[WPG+10] T Warden, R Porzel, Jan D Gehrke, O Herzog, H Langer, and R Malaka. Towards ontology-based multiagent simulations: The plasma approach. In *24th European conference on modelling and simulation (ECMS 2010). European council for modelling and simulation*, pages 50–56, 2010.