Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Màster en Intel·ligència Artificial

Tesi de Màster:
Adapting Agent Platforms to Web Service Environments

Estudiant: Ignasi Gómez-Sebastià
Director: Javier Vázquez-Salceda

Data: 03 de Septembre de 2009

Contents

1 Introduction **3**
1.1 Objectives of this master thesis . . . 5
1.2 Structure of this document . . . 6

2 Requirements of an Agent platform **7**
2.1 FIPA-ABSTRACT Architecture . . . 7
2.1.1 Architecture of FIPA-ABSTRACT . . . 9
2.2 Summary . . . 20

3 The PAWS agent platform **21**
3.1 The CONTRACT Agent platform . . . 21
3.1.1 Agent architecture definition . . . 22
3.2 CONTRACT Agent components . . . 23
3.2.1 Decision Maker . . . 24
3.2.2 Contract manager . . . 25
3.2.3 Workflow manager . . . 26
3.2.4 Communication Manager . . . 26
3.2.5 Dialogue Manager . . . 26
3.2.6 Message Manager . . . 27
3.3 CONTRACT platform components . . . 28
3.3.1 Contract Storer . . . 28
3.3.2 Observer . . . 29
3.3.3 Manager . . . 29
3.3.4 Notary . . . 29
3.3.5 Context Service . . . 29
3.3.6 Domain Ontology Service . . . 29
3.4 The PAWS Agent Platform . . . 29
3.5 PAWS Agent components . . . 33
3.5.1 Agent components directly re-used from the CONTRACT platform . 33
3.5.2 Agent components generalized from the CONTRACT platform 34
3.5.3 Agent components removed from the CONTRACT platform 35
3.5.4 New agent components on the PAWS platform . . . 36
3.6 PAWS Platform components . . . 36
3.6.1 Platform components directly re-used from the CONTRACT platform 37
3.6.2 Platform components generalized from the CONTRACT platform . . 38
3.6.3 Platform components removed from the CONTRACT platform 38
3.6.4 New platform components on the PAWS platform . . . 39
3.7 PAWS platform evaluation . . . 41

3.8 Summary 42

4 A theoretical planning framework 43
4.1 Background 44
4.1.1 Planning systems 44
4.1.2 Argumentation 45
4.2 Description of the example 48
4.2.1 The scenario: activity recommendation 49
4.2.2 Workflow 50
4.2.3 Activities 50
4.2.4 Resources 50
4.3 Formal definition of the framework 51
4.4 Conflicting actions 58
4.5 Building the global plan 59
4.6 Global utilities for a global plan 63
4.7 Coordination protocol 64
4.8 Related and further work 67
4.9 Conclusion 68

5 Implementing the planning framework in PAWS 69
5.1 Protocol Modeling 69
5.1.1 Protocol Basic Concepts 71
5.1.2 High level model 71
5.1.3 INGENIAS Role model 73
5.1.4 INGENIAS Responsibilities model 74
5.1.5 INGENIAS Precedences model 74
5.1.6 INGENIAS Interaction Model 74
5.2 Protocol Implementation 75
5.2.1 PAWS protocol components: the protocol 76
5.2.2 PAWS protocol components: the behaviors 78
5.2.3 PAWS protocol components: the message types 80
5.2.4 PAWS protocol components: the roles 81
5.2.5 PAWS protocol components: the ontology 82
5.2.6 PAWS component generation procedure 82
5.3 Framework implementation 87
5.3.1 Use of agent directory 87
5.3.2 Enabling multiple participants 88
5.3.3 Adapting the ontology 90
5.3.4 Framework implementation 92
5.4 Plan definition 94
5.5 Simple test scenario 96
5.6 Use case test scenario 100
5.7 Summary 103

6 Conclusions 105
6.1 Conclusions 105
6.2 Summary of original contributions 106
6.3 Future Work 107
6.3.1 Future improvements on the PAWS platform 107
6.3.2 Future improvements on the theoretical planning framework 108

6.3.3 Applying the PAWS platform and the theoretical argumentation framework to other scenarios . . . 108

A The ALIVE Framework 111

A.1 Organizational Level . . . 112

A.2 Coordination Level . . . 112

A.3 Service Level . . . 115

Acknowledgments

Before starting this master thesis document I would like to give a brief acknowledgment to people who has made this document possible.

I would like to acknowledge my master thesis advisor Javier Vázquez-Salceda for the support provided while developing this work, for the reviews of this document and for the countless and recurrent grammatical errors pointed out during this process.

I also would like to acknowledge my colleagues Sergio Álvarez-Napagao and Roberto Confalioneri for the support provided during the analyses of the CONTRACT agent platform, and for the helping hand given while I was developing some of the components of the platform. Thanks to them I have managed to achieve a deep understanding of the CONTRACT platform. I also would like to congratulate them for the efforts done when developing their own parts of the platform.

Juan Carlos Nieves also deserves a special acknowledgment, for introducing me to the world of Argumentation Theory as well as for the support provided during the development of the theoretical planning framework presented in this master thesis.

Last but not least I would like to mention the support provided by Rocio Díaz during the development of this work. Without her moral boosts, this work would not had been possible.

Ignasi Gómez-Sebastiá. September 2009

Chapter 1

Introduction

Service-oriented architectures (SOA) are becoming a main-stream approach for building distributed systems, as they offer heterogeneous, open, scalable and distributed solutions. The idea is encapsulating key (and very concrete) functionality in services that are remotely accessible, able to discover each other, and able to communicate and combine in order to offer more complex (and generic) functionalities. Thus, computing is not anymore the result of an action in a single computer, but the result of a network of computers interacting with each other (i.e. computation as interaction). Standards such as the OASIS Web Service Reference Architecture [22] define how distributed services should be defined and composed, focusing on interoperability issues. However, little attention has been paid to supporting the engineering of collections of services.

The engineering of service-based applications represents a new challenge if the field of software engineering. Networked applications based on the notion of independent software systems that can be dynamically deployed, modified and composed (resulting in more complex systems) open the door to new types of software systems that have nothing to do with the classic notion of software. Thus, the need for profound changes in the way in which software systems are designed, deployed and maintained arises. These changes imply a shift from a top-down design to design methodologies that allow integrating new functionalities into existing running systems formed by active, distributed and independent processes. In other words, shifting from seeing software systems as static and monolithic (in the sense of homogeneous) pieces of software to seeing software as a compound of independent, heterogeneous, distributed, and changing small pieces of software that are inter-connected together (and thus, interdependent) in order to form more complex systems. Methods to ensure the robustness of individual software applications do not map effectively to service-based applications.

A new point of view, taking into account not only the properties of individual applications but also the objectives and dynamics of the system as a whole arises. In order to tackle this issue, a wide range of research has emerged in the recent years, including among others, multi-agent systems. This field tries to apply a wide range of social phenomena often seen in human societies to computational systems, including:

- coordination patterns that are often used among humans to solve common problems amongst individuals (e.g. coordination to sell goods in an auction). Interaction patterns between limited sets of actors trying to achieve a particular goal can be modeled. These models provide formal and sound foundation for the interactions that occur. They also provide the basis for choreography and orchestration of service-oriented sys-

tems, and detail the interactions required for the execution of a particular workflow.

- characterizing the nature of autonomous actors in a environment, modeling potential rational behavior. For robust deployments it is critical to see services not as objects which are invoked at will and owned by the invoker, but as potentially owned by third parties (e.g. in a call to Google-calendar service) with its own terms, conditions of service, and possible behavior.

What's more, not only results from the research in multi-agent systems can be applied to SOA, but both fields also show some common points. Even though multi-agent systems have not been developed as part of the SOA, parallelisms can be found between both approaches. For instance, some agent applications use agents to distribute computation by offering services to other agents in the domain. The SOA community has already identified some potential in integrating agent research in SOA. For instance, in [23] they have identified some significant short-comings in WS-Agreement which could benefit from some solutions already existing in agent technologies:

- WS-Agreement messages are limited to *offer* and *agree* types. What's more, the WS-Agreement specification is only used at the last stage of the interaction, where the parties close their interaction as a contract specified as a WS-Agreement. Just two messages cannot be enough for modeling the complex negotiations required on some domains. Introducing additional semantics to the communication between services can fill this gap. Such semantics are already available on the multi-agent systems world (e.g. in the form of speech acts).

- Even if a wider variety of messages would be available, WS-Agreement would still suffer from the lack of an interaction protocol specified between the parties. Without specifications on how to build interaction protocols, the interactions that can be modeled are rather limited and simple. For instance, without protocols it is not possible to model an iterative-contract net interaction.

- In general, WS-Agreement is a specification with vague and unclear semantics. The WS-Agreement focuses only on defining a high-level template for agreements and offers. A language able to express the elements in the service description terms and guarantee terms is required. Such languages are already available on the multi-agent systems world in the form of ontologies.

As summary, two clear ideas can be extracted from this analysis:

- Service-oriented architectures are becoming the standard for building distributed systems. As distributed systems are likely to become the future of software systems, service-oriented architectures will be the standard for building software systems in the future. Regarding this, service composition is key for service-oriented architectures to be functional, as the capabilities provided by each isolated service are not enough for solving complex problems.

- Actual service-oriented architectures standards are trying to tackle some issues that have been addressed long ago by researches in multi-agent systems. Integrating agent research in service-oriented architectures is an approach for tackling this issue.

Thus integration between service-oriented architectures and multi-agent systems is a cutting edge interesting issue.

1.1 Objectives of this master thesis

This master thesis tries to address the above-mentioned issues by **creating an agent platform suitable for encapsulating web-services into agents**, providing them with typical agent capabilities (such as learning or complex communication and reasoning mechanisms). The objective of this point is to create a generic, modular agent platform that is able to run lightweight agents. The agents should be able to easily invoke web-services, effectively encapsulating them. They also should be able to easily coordinate for composing the invoked services in order to perform complex tasks. Thus, the platform must provide facilities to allow the agents perform these service invocations. The key points of the platform should be:

- Simplicity: For providing basic agent capabilities to web-services heavy and complex reasoning mechanisms are not required. A simple reasoner (implemented for instance as a Java class) should be enough. What's more, the platform is not to provide many specific and advanced components either. It should be enough with the basic components that every agent platform provides.
- Modularity: This need is generated by the first need. If the platform is to be generic and simple (i.e. just the basic components are to be provided) it should be easy for users of the platform to code new components and attach them to the platform. This would allow users to adapt the platform to the needs of the domain where they are going to apply it. It is desirable that components can be attached not only to the platform, but also to the agents.
- Usability: Agents should be easy to code, deploy and maintain. The main effort of the users of the platform should be in coding the encapsulated services, not the agents in the platform.
- Service-oriented: Services should be easy to invoke to be effectively encapsulated. Service discovery, adding the feature of finding substitutive services for unavailable ones, is a desirable feature. What's more, agent coordination should be easy to implement, and coordination constraints should be specified in terms of web-services, when possible.

The requirement of coordination mechanisms in the platform arises a second objective for the thesis: **integrate into the agent platform a planning framework that allows agents to coordinate deciding how to enact sets of actions in a collaborative way**. This planning framework is to be defined formally, implemented and integrated in the agent platform. The planning framework should be generic, rather than oriented to service composition. That is, the framework should be usable to coordinate any set of actions given a group of intelligent actors enacting them and a set of constraints to be met when between the actions. As there is a good number of planning frameworks already defined, implemented and running, the framework created for this objective should provide a new contribution to the planning world. The contribution chosen has been trying to create a planner that, based on argumentation theory, is able to provide a conflict-free global plan, given:

- A group of local plans formed by sets of alternative chains of actions. This is meant to model the set of actions to be coordinated.
- The definition of the resources consumed by the execution of each of the actions in the local plans. This is meant to model the restrictions that the global plan should meet.

1.2 Structure of this document

The rest of this document is structured as follows:

- Chapter 2: Analysis of the FIPA-ABSTRACT standard. This standard defines the components agent platforms should implement, and the minimal functionalities they should provide. It is important analyzing this standard because the developed agent platform should be compliant with it. At the same time, the standard provides a complete overview on agent platforms.

- Chapter 3: Development of the PAWS (Platform for Agentified Web Services) agent platform. This chapter presents in depth how the platform has been developed. The chapter starts by explaining an existing web-service oriented agent platform which is domain-bound, the CONTRACT agent platform. The chapter goes on by describing how the CONTRACT platform has been modified in order to fit more general needs, and how the PAWS platform has been developed based on these modifications. During the process, the architecture and the components of the PAWS platform (along with their functionalities) are described.

- Chapter 4: Formal definition of the planning framework. This chapter introduces the planning framework to be integrated in the agent platform from a theoretical point of view. The chapter also provides an example (inspired on a real scenario) of an action coordination problem that is used to help the reader understand the formal concepts introduced in the chapter. The example is also used as a test-case when analyzing the implementation of the framework in the next chapter.

- Chapter 5: Implementation of the formal framework and integration in PAWS. This chapter explains the process followed to model, implement and integrate the theoretical planning framework in the PAWS platform. The chapter is meant to explain not only how the framework has been implemented, but also how the PAWS agents have been designed and coded in order to integrate the planning framework, and how the planning module resulting from the implementation of the theoretical planning framework has been integrated in PAWS. This chapter could be used as reference for future PAWS users for understanding how to integrate new modules on PAWS and how to design and adapt PAWS agents to the needs of their domain. The chapter also introduces a graphical tool for defining plans, and a couple of test-cases applied to the planning framework.

- Chapter 6: Conclusions and further work: This chapter explains the conclusions extracted from the development of the work presented in this master thesis, the contributions of the work, and how could it be applied to some existing scenarios. This chapter also gives an overview about how the work could be continued, how could it be improved and how new lines of work can arise from it.

- Appendix A: The ALIVE framework: This appendix gives an overview of the ALIVE framework which is being developed in collaboration with several universities and enterprises within the frame of the FP7 project ALIVE (ICT-215890) funded by the European Commission.

Chapter 2

Requirements of an Agent platform

This chapter analyzes the FIPA-ABSTRACT agent platform architecture. This will fulfill three main goals. First of all, analyzing an standard like FIPA-ABSTRACTS will show the properties an agent platform should meet. Second, it gives a quick overview on agent platforms, introducing some concepts (e.g. *Directory Facilitator*, *performative*, etc.) that are used later in this document. Last but least, it provides an overview of an abstract agent platform, describing components that are common on all platforms. Understanding this abstract platform also helps understanding our agent platform that will be introduced in *Chapter 3*

An agent platform is a set of software entities (e.g. Java processes) that run the agents and services in the platform. Agents are autonomous software entities with intelligent capabilities (e.g. learning, communication and cooperation, etc.). Agents get support from services in tasks such as obtaining an unique identifier, sending messages to other agents, storing data in persistent bases or finding out which other agents are on the platform.

The *Foundation for Intelligent Physical Agents* (FIPA from now on) is an international organization aimed to developing open specifications([13]) for agent systems. Such specifications focus on enabling interoperability between agents and agent-based applications. By developing these specifications FIPA wants to promote the industry of Intelligent Agents.

The FIPA specifications define agent infrastructure, including agent communication language (that is, how messages are transferred and represented) as well as some optional properties (such as how to encrypt them). Agent services and support for management ontologies are also specified. Specifications also include agent applications by defining several application domains (e.g. network management).

2.1 FIPA-ABSTRACT Architecture

FIPA-ABSTRACT, the last version of the specifications, aims to reach interoperability via *architectural abstractions*, that is, common characteristics that can be found in different agent technologies. Such abstractions should be formally related to every valid implementation, because, if every concrete agent architecture follows this specification, it will be able to inter-operate with other architectures. This property can be ensured because of the fact that following these *architectural abstractions* will make every agent architecture present a common abstract design.

Even though FIPA specifications aim at defining an inter-operable agent architecture, revisions of the specifications lack backwards compatibility. Architectures built using FIPA-97 and FIPA-98 specifications have limited interoperability with FIPA-Abstract. However, architectures built using FIPA-2000 are fully inter-operable with FIPA-Abstract.

2.1.0.1 Aims of FIPA-ABSTRACT

FIPA-ABSTRACT focuses on providing inter-operable message exchange between agents in different architectures. Such architectures might be using different implementations of the following abstractions and yet be interoperable:

- Set of services available to agents and services in the platform, with special mention to 'Service Discovery' and 'Directory Service' services.
- Message transport abstraction
- Agent communication language abstraction
- Content language abstraction

However, FIPA-ABSTRACT does not cover the following abstract areas:

- Agent lifecycle and management
- Agent mobility and Identity
- Agent Domains
- Conversational policy

The FIPA-ABSTRACT architecture defines two basic elements: *opaque typed elements* and *associations*. *Opaque typed elements* can be understood by specific implementations of a service while being meaningless (that is, opaque) to other components, or even other implementations of the same component. Thus, in order to add a new service, or a new implementation for an existing service, all that is to be done is to define a new *opaque typed element* and associate it to the incoming service. *Associations* between an agent and a service allow the agent to invoke the service via handler structures. This concept resembles factory design pattern [12].

2.1.0.2 Why to analyze FIPA-ABSTRACT

Analyzing and understanding these specifications is key for our work, because an agent platform that complies with the specifications will be able to freely operate with other platforms (as long as these platforms also comply with the specifications), regardless implementation details used in the platforms. This is a very desirable property, taking into account interoperability is a key concept in open, distributed and heterogeneous scenarios, that is, scenarios where agent-based technologies fit perfectly.

It must be noticed that FIPA specifications in general, and FIPA-ABSTRACT in particular, describe an abstract architecture that cannot be directly implemented, it just provides the basic guides on how to build an agent system. In this scope, a *realization* of an element denotes the definition of an abstract element in terms of a concrete architecture implementation. Designers must bear in mind that FIPA specifications describe the minimum required elements of an agent architecture, they do not prohibit the introduction of new elements.

2.1.1 Architecture of FIPA-ABSTRACT

The FIPA-ABSTRACT Architecture defines the architectural elements (and their relationships) required in order to allow agents to locate each other and communicate with each other.

2.1.1.1 Basic concepts

Agents communicate interchanging *messages* encoded in *Agent Communication Language* (ACL). These messages represent *speech acts* [27], and include the concept of *performative*. *Services* denote a set of services(e.g. web-services) providing support functionalities for agents. Such functionalities include *agent directory*, *message transport* and *service directory*. *Services* can be agents or pieces of software that are accessed via method invocation procedures. However, it must be noticed that an agent providing a service is more restricted in its behavior than a general purpose agent, due to having to preserve the semantics of the service.

2.1.1.2 Directory facilities

Directory facilities are services that allow agents to register their descriptions, as well as to query existing descriptions in order to find agents with which they can interact. The main *directory facility* defined in FIPA-ABSTRACT is the *agent directory*, also known as *Directory Facilitator*.

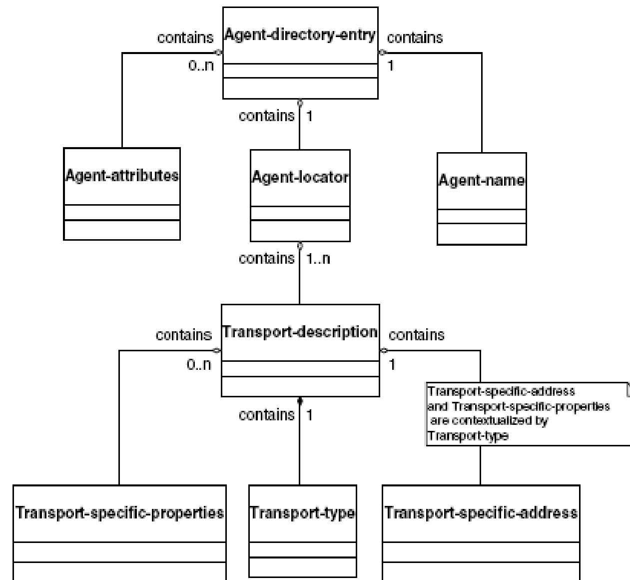

Figure 2.1: Agent-directory UML relationships elements

When an agent wants to register itself in the *agent directory* it binds itself to one or more *transports* that will allow other agents to reach him. This operation can be performed via the *message transport service*. Once transport information is available, an *agent directory entry* can be built and the agent can publish it on the *agent directory*. Using this procedure an

agent can make itself, along with its capabilities, public to other agents in the environment. At this point, other agents can query *agent directory*, retrieving information about the agent that has registered (possibly along with information from other agents) and if they consider it convenient, contact it. *Agent-directory-entries* consist in: a) an *agent-name* b) an *agent-locator* c) a set of zero or more *agent-attributes*. Agents have not the obligation of publishing themselves on directories, they can exchange *agent-directory-entries* via other means. For instance agents involved in a negotiation protocol can exchange their *agent-directory-entries* without publishing it on *agent-directories*.

Agent-directories provide the following functionalities for agents:

- Register: By providing an *agent-directory-entry* an agent can publish itself in one or more *agent-directories*. The operation might fail if the agent has no rights to register in the selected *agent-directory* or if the provided *agent-directory-entry* is not valid. The operation will also fail if the *agent-directory* already contains an *agent-directory-entry* with the same *agent-name* [1] as the entry to be registered.

- Modify: By providing a new *agent-directory-entry* an agent can modify an existing *agent-directory-entry* in a given *agent-directory*. The directory will search for an entry with the same *agent-name* as the entry provided. If it is not found, the operation returns an error. The directory will also make sure the provided entry is valid, and the agent has permissions to modify the entry in the directory. Once these conditions are checked, the directory proceeds to update the entry value. For each key-value pair in the provided entry, if the pair is found in the existing entry, the new value is set. If the pair exists and the value in the provided entry is null, the pair is deleted from the existing entry. If the pair is not found, the pair is added to the existing entry.

- Deregister: By providing an *agent-directory-entry* an agent can delete an existing *agent-directory-entry* in a given *agent-directory*. The directory will search for an entry with the same *agent-name* as the entry provided. If it is not found, operation returns an error. The directory will also make sure the agent has permissions to delete the entry in the directory. Once these conditions are checked, the directory proceeds to delete the entry value. Notice that, in the provided entry, only the *agent-name* value is significant, other values are not checked at all.

- Search: By providing an *agent-directory-entry* an agent can search a given *agent-directory* for entries that match searching criteria. The directory will make sure the provided entry is valid, and the agent has permissions to search for entries in the directory. Once these conditions are checked, the directory proceeds to search for entries matching the searching criteria. The provided *agent-directory-entry* is treated as a search pattern, thus, a given entry existing in the *agent-directory* is returned if all key-value pairs in the provided entry are equal to the pairs in the existing entry. FIPA-ABSTRACT recommends implementing, at least, matching criteria of 'same value' (i.e., value of the pair in the provided entry is equal to the value of the pair in the existing entry) and 'any value' (i.e., value of the pair in the existing entry does not matter for matching condition).

FIPA contemplates the possibility of an *agent directory* registering into other *agent directories* to form *federations*. A search for a service in an *agent directory* that has another directories registered is performed, first of all locally (that is, in the directory where search operation is invoked) and then extended to other directories among the same *federation*.

[1] FIPA-ABSTRACT contemplates the possibility of adding more constraints

This arises the need of providing additional parameters to the search operation (such as searching in other directories only if local search returns no results, or searching only until a limited number of results have been obtained, or until a limited number of directories have been queried) in order to avoid high cost (or even infinite) searches.

Another *directory facility* defined by FIPA-ABSTRACT is the *service directory*. A *Service directory* can be seen as analogous to an *agent directory* with the difference that both agents and services can register in it or query it. Many FIPA-ABSTRACT implementations model the *service directory* as a simple local table of fixed size, whereas using more complex and distributed technologies (such as LDAP) for the *agent directory*. A *service-root* is provided to every starting-up agent. This *service-root* is no more than a list of basic services (*agent-directory*, *message-transport*, etc.) that connect the agent with the environment.

A *directory facility* which is analogous to *agent directory* is the *Agent Discovery Service*, or *ADS*. The *ADS* provides the same functionalities as the *agent directory*, but is specially suited for ad hoc networks, where network nodes are not very stable (that is, they join or leave very frequently). In such cases, the agents use the *agent directory* for discovering (and be discovered by) agents that are on the same *Agent Platform*, whereas the *ADS* is used for discovering (and be discovered by) agents residing on remote devices of the ad hoc network. The only functional difference between the *ADS* and the *agent directory* is that *ADS* provides subscription and de-subscription procedures. Subscription allows agents to pass an *agent-directory-entry* template with the generic characteristics of the agents they want to be notified about. When agent whose *agent-directory-entry* matches the template registers on the *ADS*, subscribed agents will be notified. The same happens when the agent de-registers from the *ADS*. A de-subscription procedure is available to allow agents to stop receiving notifications. Subscription and de-subscription procedures are specially fit on the *ADS* due to the high volatility of the agents registered on this component, that is, they are expected to register and de-register more often than the agents on the *agent directory*.

The last *directory facility* is the *Agent Management System* which is similar to the *agent directory*. The *Agent Management System* keeps a list of all agents in the system (either they are registered on the *agent directory* or not) along with their names and *transports* they can be reached by. The *Agent Management System* does not contain any additional information about services offered by the agents. Thus, *Agent Management System* is usually known as *White pages service* whereas *agent directory* is usually known as *yellow pages service*.

2.1.1.3 Agent Messages

An Agent message is typically written in the Agent Communication Language (ACL) and, basically, consists in a *sender*, any number of *receivers*, the type of *communicative act* represented (as denoted on *Section 2.1.1.4*) and some *content*. Along with the *content* the ontology [2] that is to be used to interpret the content must be provided. The complete list of components in the messages is as follows:

- performative: denotes the type of *communicative act* of the message. Should be a FIPA standard *communicative act* as denoted on *Section 2.1.1.4*
- sender: identifies sender of the message. FIPA-ABSTRACT allows omitting this field, in the case an anonymous message is sent. Will be, typically, an agent name.
- receiver: identifies the intended recipient of the message. FIPA-ABSTRACT allows a set of values in this parameter for multi-cast messages. FIPA-ABSTRACT allows

[2] An ontology is a vocabulary for representing knowledge about entities and the relationships between them

omitting this field, in the case a broadcast message to all agents is to be sent. Will be, typically, an agent name.

- reply-to: indicates that the next messages to be sent in this conversation are to be directed to the agent specified in this parameter (via agent name) and not to the sender of the message.
- content: domain-specific part of the communication.
- language: used to express the content. In order to be able to use FIPA performatives, the language must be able to represent propositions, actions and terms. Languages such as KIF and SL comply with this condition.
- encoding: used to specify the encoding of the content. If ommited, the encoding will be specified in the envelope of the message.
- ontology: specifies the ontology used to give meaning to the content. Can be omitted if the receiver of the message can have no misundersting about the ontology being used.
- protocol: interaction protocol that the sender of the message expects to be employed during the conversation. Can be set to null, but FIPA recommends using protocols in all cases in order to reduce the ambiguity of the interaction.
- conversation-id: identifies the set of both past and incoming messages belonging to the same conversation. If the *protocol* parameter is used, initiator of the protocol is obliged to specify a valid value for this parameter. This value will remain unchanged in all the messages sent in the scope of the protocol.
- reply-with: sets a value to be used in the parameter *in-reply-to* by responding agent.
- in-reply-to: denotes the message as a reply of a former one. Typically, if an agent receives a message with the *reply-with* parameter set to value 'this is a reply', it will respond with the *in-reply-to* parameter set to value 'this is a reply'.
- reply-by: sets a deadline after which the sender of the message does not want to receive a reply for the message.

When a message is interchanged between agents, it is encoded into a *payload* (using an appropriate *encoding-representation*) and included in an *envelope*. The *encoding-service* is the component responsible of transforming messages into *payloads* for sending them, and transforming *payloads* to messages upon reception. The envelope in which the *payload* is included defines *transport-descriptions* (how to send the message, via what transport, to which address, etc.) of both sender and receiver. A set of transport descriptions, specifying how to reach a given agent via different transport protocols, is held in an *agent-directory* service. Envelope can also include optional fields such as encoding-representation or security data. Such optional fields allow, for instance, the inclusion of a public key in the envelope, then by encrypting pay-load and message it can be assured that message has no meaning for entities without the private key. Once the message is into the envelope, the *message-transport-service* takes care of sending it using the selected transport protocol.

It must be noted that :

- While the *envelope* can add additional data to the message, the payload cannot, it only encodes the message in a suitable representation.

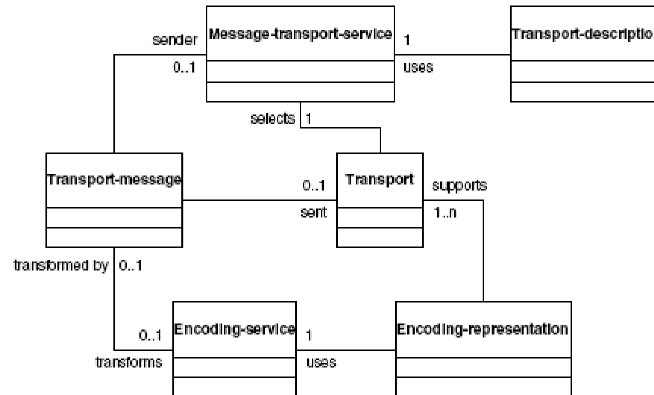

Figure 2.2: Transport UML relationships element

- An agent's name is preserved through the agent's lifetime, however transports used to communicate with it can change. New transports can be added or removed, or existing ones can change.

2.1.1.4 Communicative Acts

FIPA defines standard *communicative acts* (also known as *performatives*) for helping to reach interoperability by providing standard composite *communicative acts* that can be used in ACL messages. Two agents developed in different contexts will be able to interact more easily if they use the same *communicative acts*, and give them the same meaning, in the messages they exchange. FIPA defines the following standard *communicative acts*, although it encourages developers to define their own composite acts and make them available to the community:

- Propose: to send a proposal to perform a certain action given certain preconditions. This act includes both the action that is proposed and the preconditions that must be met before performing the action. Can be sent as a reply to a previous *proposal* in a negotiation protocol.

- Call for proposals (cfp): a sender requests proposals to perform a given action. The act includes the action requested and the preconditions that must be met before the action is performed. Such preconditions include, at least, one value-free parameter. Replies to the request are *communicative acts* (typically *propose*) that have a value set on this parameter. As the general purpose of the *cfp* is to start a negotiation process, the act can also include a *protocol* parameter, specifying the conversational structure to be followed during the negotiation. Notice that *cfp* can be started by agents that do not intend the action to be performed, but just want to know the availability or disposition of other agents to perform it.

- Request: to ask the receiver to perform a given action. The act includes the action as parameter. Notice this action can be another *communicate act*, such as requesting to inform about a given believe.

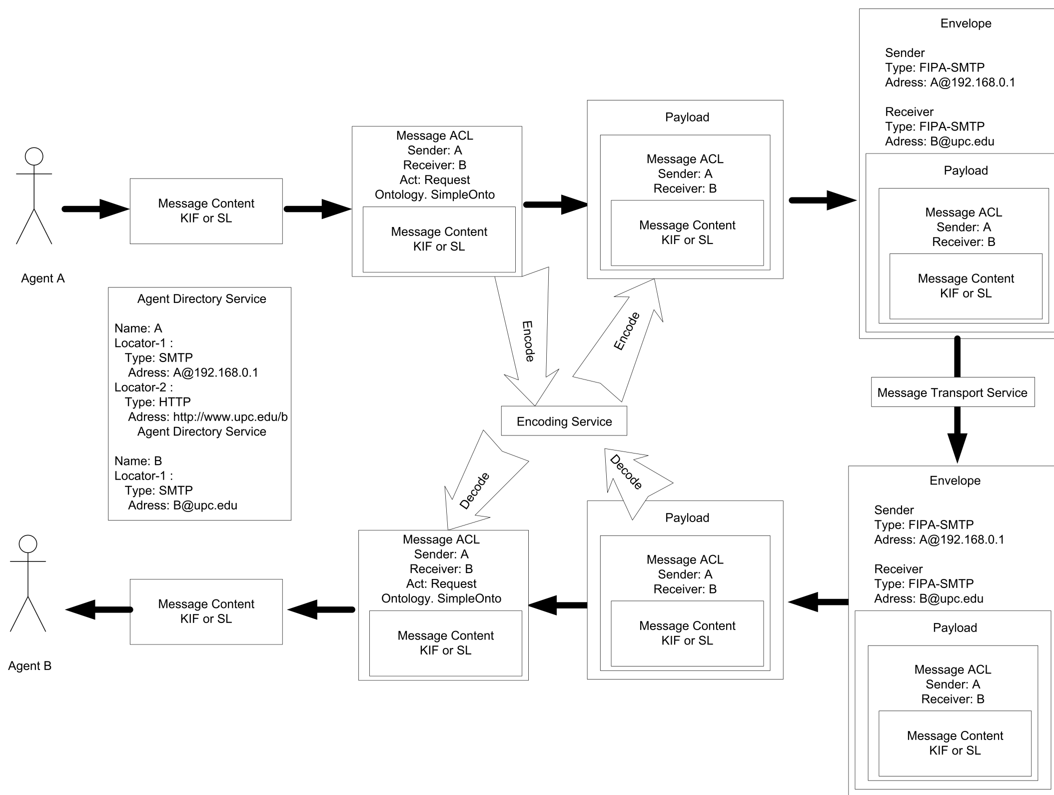

Figure 2.3: Message Interchange Flow Diagram

- Request when: to ask the receiver to perform a given action when some conditions are met. Very similar to the *request* act, this act includes also a parameter for specifying the conditions.

- Request whenever: to ask the receiver to perform a given action each time some conditions are met. Very similar to the *request* act, this act includes also a parameter for specifying the conditions.

- Accept Proposal: a sender accepts a previously received proposal (via *propose* act). The message includes the action that the sender intends to do at some point in the future, and some preconditions that must be met before the sender performs the action.

- Agree: a sender accepts a previously received request (via *request* act). The message includes the action that the sender intends to do at some point in the future, and some preconditions that must be met before the sender performs the action. The message can also include some qualifiers regarding how the action will be performed. For instance, when agreeing to perform a requested task, the sender can specify that it will perform such task with low priority.

- Cancel: a sender informs the receiver that he does not hold any more the intention to perform a previously requested, and agreed, action. The act includes the request to be canceled. Notice that the receiver is free to ignore the cancellation. Also notice

that, in order to send this message, sender must believe that the action to be canceled is either ongoing or planned, but not fully executed yet.

- Failure: a sender tells another agent that an action has been attempted to perform, and the attempt has failed. The act includes the reason of the failure. The receiver is encouraged to believe the action has not been done yet, and that the sender believes (or believed at the time of attempting it) the action is feasible. However, if this reason of the failure is the constant *true*, it means the sender believes there is little any agent can do to re-attempt the action with success.

- Refuse: to refuse performing a given action. Act includes the action as parameter. Optionally, an explanation for the refusal can be provided. Receiver of this message is encouraged to believe that the action has not been performed, and that it is not feasible according to the sender's beliefs.

- Reject proposal: the sender rejects a proposal to perform some action. Used in the scope of a negotiation process. The act includes both the rejected action and the act representing the proposal being rejected. Can also include the reason of the rejection.

- Query if: the sender asks another agent if a given proposition is true or not, according to its beliefs. The act includes the proposition as parameter. The sender has not knowledge of the truth value of the proposition and believes the receiver does have this knowledge. Receiver will either reply with *inform* or *refuse* acts.

- Query ref: to query the receiver about the object that corresponds to a descriptor, typically a name. Very similar to a *query if* act, it is used when the proposition provided is an expression matching an object with its identifier.

- Subscribe: the sender requests the receiver to inform it about the value of a variable, and inform again whenever this value changes. The act includes an expression describing the variable as a parameter. Note this act is a persistent version of *query ref* act.

- Confirm: to inform the receiver that a given proposition is true. The act includes the proposition as parameter. The sender must believe the proposition to be true and intend the receiver to believe it to be true (although it is up to the receiver to update its mental state or not upon message reception). The sender must also believe the receiver to have uncertainty about the proposition, otherwise, other *communicative acts* (such as *inform*) must be used.

- Disconfirm: to inform the receiver that a given proposition is false. The act includes the proposition as parameter. The sender must believe the proposition to be false and intend the receiver to believe it to be false (although it is up to the receiver to update its mental state or not upon message reception). The sender must also believe the receiver to have uncertainty about the proposition or to believe it is true.

- Inform: to inform the receiver that a given proposition is true. The act includes the proposition as parameter. The sender must believe the proposition to be true and intend the receiver to believe it to be true (although it is up to the receiver to update its mental state or not upon message reception). The sender must also believe the receiver to have no knowledge of the truth of the proposition, otherwise, other *communicative acts* (such as *confirm*) must be used.

- Inform if: to request the receiver if a given proposition is believed to be true or not. The act includes the proposition as parameter. The receiver replies with an *inform* act stating the truth value of the proposition. If the receiver holds no truth value for the proposition (or does not want to share the value with the sender) a *refuse* act is sent as reply instead.

- Inform ref: to inform the receiver of the object that correspond to a descriptor, typically a name. Very similar to *inform* act, it is used when the proposition provided is an expression matching an object with its identifier.

- not understood: the sender informs the receiver that it did not understood an action performed by receiver. Typically, this action is the sending of a message. The act includes the action that has not been understood (for instance, a *communicative act*) and can optionally include the reason why it has not been understood.

- Propagate: to request the receiver to interpret an embedded message, and send it to other agents. The act includes the set of agents that must receive the propagated message and the ACL message to propagate. Optionally, conditions for propagation (for instance, a timeout or a precondition that must be met before the message is propagated) can also be included.

- Proxy: similar to the *propagate*. The difference is that this act will not include a set of explicit agent names as a parameter, but a set of abstract agent descriptions. The receiver of this act will identify agents that meet the conditions specified in the set of agent descriptions and send the embedded message to them.

2.1.1.5 Agent

An *Agent* is a computational process that implements the functionality of an application autonomously and communicating with other agents. An agent can be instantiated via Java components, COM objects, C++ programs, etc. It can run on a process in a physical computer or on some interpreter (such as a Java Virtual Machine). Agents communicate using the *Agent Communication Language*.

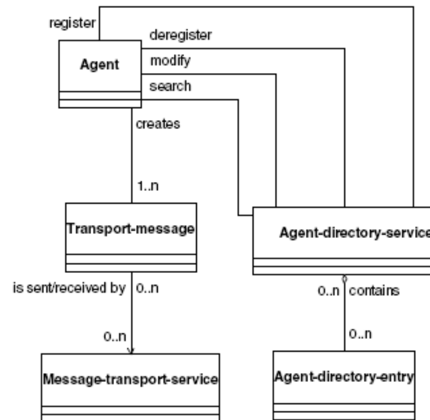

Figure 2.4: Agent UML relationships element

Agents have an unique *agent-name*. Means to ensure this name is not altered, forged or duplicated must be provided. An *agent-name* should not denote any properties about the agent, FIPA-ABSTRACT provides the concept of *nickname* for this purpose. The agent must also have one associated owner, either an organizational affiliation or human ownership.

Agents also have zero or more *Agent-attributes*. Such attributes are one of the components on the *agent-directory-entries* in *agent-directories*. These attributes allow other agents to search the directory for agents that meet special properties (that is, agents that understand certain ontologies or provide certain functionalities). Agents have an *agent-locator*, with one or more *transport-descriptions* (one per supported protocol). The agent-locator is used by the *message-transport-service* to select a transport for communicating with a given agent.

2.1.1.6 Agent Platform

An *Agent Platform* provides the infrastructure in which *agents* can reside. The *Agent Platform* contains the *agents*, support software (such as the operating system, security algorithms, support tools and other) and the management components (that is, one *Message Transport Service*, one or more *Directory Facilitator* and one *Agent Management System*). An agent system can be composed of several *Agent platforms*.

Note that FIPA-ABSTRACT only defines how communication between agents in different platforms is performed. Agents on the same platform can interchange their messages by any means they have available.

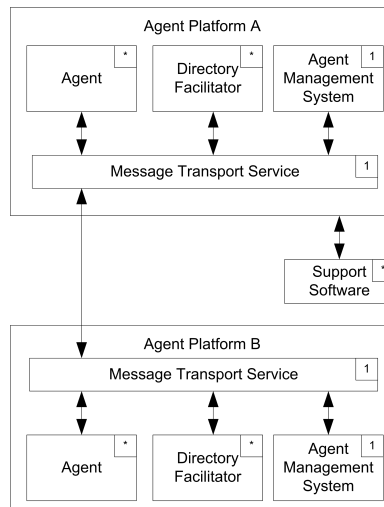

Figure 2.5: FIPA Agent Platform Diagram

2.1.1.7 Agent Naming

As seen in *Section 2.1.1.5*, agent names are unique identifiers for agents. However, this is true only between agents on the same *Agent Platform*. Notice that, *Section 2.1.1.6* does not define inter-platform *Agent Management System*, so nothing guarantees that two different platforms do not host an agent with the same name. Thus, the need for a more complex identifier arises.

These denotes an agent identifier through a collection of parameters known as *AID*. This parameters include:

- Name: Commonly this parameter contains the name of the agent (as defined in *Section 2.1.1.5*) followed by the character '@' and by the address of the platform containing the agent. This should be enough to identify an unique agent across all the available platforms, because it is guaranteed that the same platform will not host two agents with the same name, and two different platforms will not have the same adress.
- Address: List of *transport addresses* where agent can be reached via message sending.
- Resolver: List of services where *transport addresses* for reaching a given agent can be obtained. To be used in the case *Address* parameter of the *AID* does not provide this information. Notice that,typically, this parameter is the adress of a single service provided by the *Agent Management System* component of the platform where the agent associated to the *AID* resides.

If several *Agent platforms* are involved, the fields where *agent name* is used as identifier (*envelope*, *Directory Facilitator* and such) will use *AID* as identifier instead of just the name.

2.1.1.8 Agent Life Cycle

The *agent*, as a physical software process that resides in the hosts of a given *Agent Platform*, has a life cycle that must be managed by the platform. This management is performed by functionalities that allow the software process to start, stop, suspend or migrate to another host or platform. Such functionalities are typically provided by the *Agent Management System*.

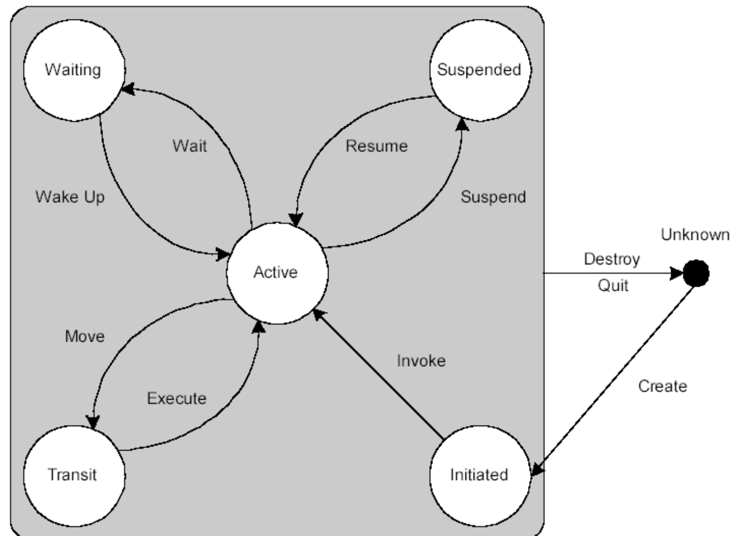

Figure 2.6: Agent Life Cycle Diagram

Here is the list of the functionalities provided by the *Agent Management System* for the agent life cycle. Notice such functionalities represent the nodes between the possible states of the agent as seen in *Figure 2.6*.

- Create: Creates a new Agent. The software process associated to the agent can be pre-started (for instance, loaded in memory) but is not run yet.
- Invoke: The agent is invoked. The software process associated to the agent starts to run.
- Destroy: The platform forces the agent to terminate. This process is initiated by the *Agent Management System*, and the agent is forced to finish its execution. After a small time-out, the process associated to the agent stops running, all the information related to the process can be removed from the host.
- Quit: The agent terminates by its own will. After a small time-out, the process associated to the agent stops running, all the information related to the process can be removed from the host.
- Suspend: The agent goes into a suspension state by its own will. After a small time-out, the process associated to the agent stops running, information related to the process must be maintained on the host because the process is supposed to go back into running state soon.
- Wait: The agent goes into a waiting state by its own will. The process associated to the agent keeps running, but the platform can apply some special measures to it (for instance, lowering its priority).
- Resume: The agent goes back from *Suspended* state into *Running* state again. The process associated to the agent starts running again.
- Wake Up: The agent goes back from *Waiting* state into *Running* state again. The process associated to the agent starts running normally again (for instance, priorities are restored).
- Move: The agent goes into a transition state by its own will. Platform starts required actions (such as stopping process, moving data associated to process, etc.) to move agent and the process from one *Agent Platform* to another.
- Execute: The platform brings the agent back from a *Transition* state. Platform starts required actions (such as starting process, loading agent and process data into the new host, etc.) to resume the process in the new platform.

The cycle also involves the *Message Transport Service* component, because different states of the software process associated to the agent will require different treatment of the messages addressed to the agent. Here is the list of how messages addressed to the agent should be processed according to the agent's states:

- Active: Messages are delivered to agent as normal
- Initiated: Messages are buffered. Will be sent to agent when it becomes *Active* again. Agents have the possibility to forward messages sent to them to another agent when they are not in *Active* state.
- Waiting: Messages receive the same treatment as in *Initiated* state.
- Suspended: Messages receive the same treatment as in *Initiated* state.
- Transit: Messages receive the same treatment as in *Initiated* state. Note that, in this case, messages can be forwarded to the same agent, on the new location.

- Unknown: *Message Transport Service* decides whether it will buffer the messages and send them to the agent when it becomes active, or reject them.

2.2 Summary

In this chapter the main requirements needed to develop FIPA compliant agent platforms have been reviewed. We have focused on the latest specification, FIPA-ABSTRACT, that defines *architectural abstractions*, i.e., common characteristics that should be found in agent technologies if they are to be able to inter-operate with each other. Even though FIPA-ABSTRACT lacks compatibility with older FIPA specifications, it has backwards compatibility with new ones (e.g. FIPA-2000).

While reviewing FIPA-ABSTRACT we have also gone through the main concepts and components that are common on all agent architectures.

FIPA-ABSTRACT specifications are the basis for the PAWS platform we will present in *Chapter 3*.

Chapter 3

The PAWS agent platform

This chapter introduces the PAWS (Platform for Agentified Web Services) [24] agent platform. The PAWS platform aims to provide an agent platform where agents are acting as proxies for web services, effectively providing web-services with agent capabilities such as coordination, protocol-based communication, etc.

It is relevant to understand the PAWS platform for two reasons. First of all, an important part of the effort invested in developing the work presented in this master thesis has been devoted to creating the PAWS platform. Second, as we will see in *Chapter 5* the PAWS platform supports the implementation of the theoretical planning framework. [1]

The PAWS platform parts from the agent platform [4] developed for the IST-CONTRACT project [3], which will be referred to as CONTRACT platform from now on. The CONTRACT platform is well-suited for distributed, contract-oriented applications, but it is not suited for more general applications. Thus, in this chapter the CONTRACT agent platform is generalized in order to create a more flexible (and widely usable) agent platform, the PAWS platform.

This chapter first explains the architecture of the CONTRACT platform, as well as the components of the architecture. It will also go over the components of the agents in the platform with special mention to the behaviors of the agents.

The chapter will then go over the architecture and components of the PAWS platform making special emphasis in the parts that have been developed by the author of this master thesis. This section will explain the components of the PAWS platform based on the explanation provided for the component on the CONTRACT platform. Thus it will clarify (w.r.t. the CONTRACT platform) if the component remains unchanged, has been adapted for the PAWS platform, has been removed on the PAWS platform or is completely new in the PAWS platform.

Once the PAWS platform has been introduced it is evaluated following the standards defined in other works that compare agents platforms (e.g. [21]).

3.1 The CONTRACT Agent platform

This section starts by giving a quick overview of the components available in the CONTRACT platform, explaining their purpose, functionalities and how these components interact between each other. In this way the reader can get a better understating of the CONTRACT platform.

[1] Presented in this document in *Chapter 4*

3.1.1 Agent architecture definition

The CONTRACT platform is developed with the purpose of supporting the software systems (i.e. software agents) that play an active role in establishing, fulfilling and executing the contracts. Such systems include both *business* contract parties and *administrative* contract parties. Whereas the first ones are the parties directly involved in the contract (e.g. *seller* and *buyer* parties) the second ones can be seen as internal software components that belong to the contractual environment and provide support functionalities (e.g. *observer*, *contract manager*, *notary*).

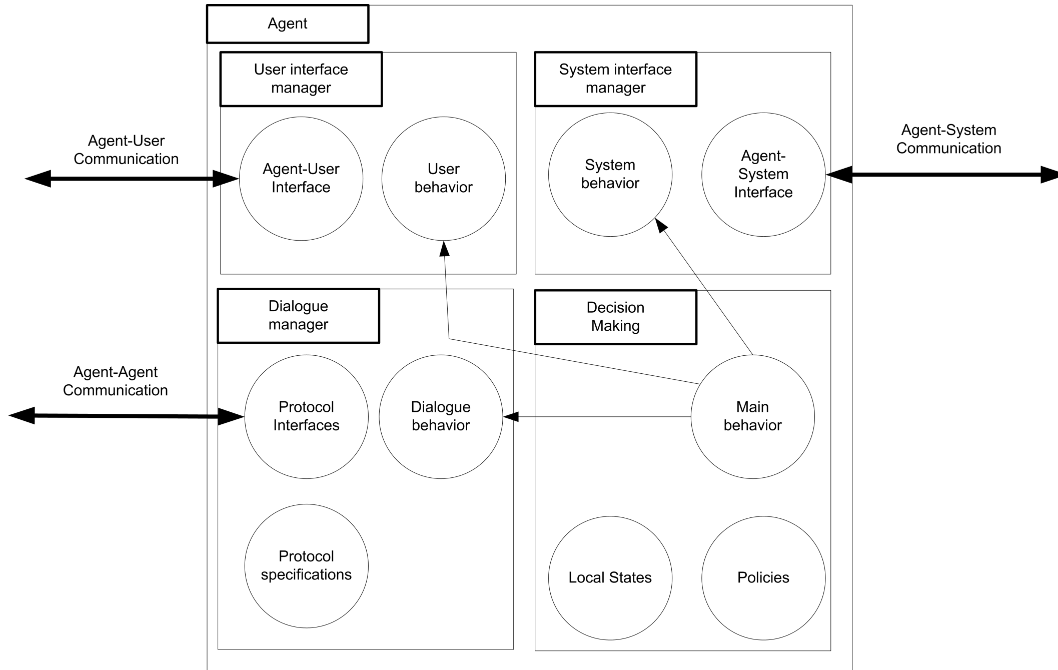

Figure 3.1: IST-CONTRACT project Agent architecture overview diagram

In the IST-CONTRACT project, an agent is a software component that can be instantiated to play the role of a *business* contract party, or one of the *administrative* contract parties (i.e. *observer*, *contract manager*, *notary* or *contract storer*). *Figure 3.1* shows the architecture of a generic IST-CONTRACT agent. Please note this architecture depicts four main modules:

1. Agent-Agent interface: Supports communication between agents in the system. Provides parsing and serialization mechanisms for agent messages, as described in *Section 2.1.1.4*, supporting just a single encoding service and a single message transport service. Includes interface specifications for the available protocols. This interface has been designed to able to keep track of multiple concurrent interactions.

2. Agent-System interface: Gives access to the facility components provided by the platform, as described in *Section 3.3*

3. Agent-User interface: Supports communication between agents and the external world. Is hardly focused to communication with human users.

4. Decision making module: Agent's control system, containing both agent's knowledge base and configuration parameters. This is the main module, the interfaces mentioned above are attached to this module.

Please note each of the four mentioned modules contains a behavior. The behavior of the decision making module is known as *main behavior*, as it controls the other three behaviors, the *dialog behavior* (associated to the Agent-Agent interface), the the *system behavior* (associated to the Agent-System interface) and the the *user behavior* (associated to the Agent-User interface). The functionalities of each of these four behaviors are as follows:

1. dialog behavior: defines both available performatives and how to combine them in order to achieve a particular communication behavior. Via dialog behaviors one can define which steps of the protocol are allowed and which are the most significant.
2. system behavior: consists in a set of low level (i.e. Java) APIs that define the access to the components of the platform.
3. user behavior: consists in a set of low level (i.e. Java) APIs that define the access to user interface (i.e. GUI and text interfaces) components.
4. main behavior: specifies what the agent can do and the procedures to be followed to do it. Such procedures are based on interaction protocols, parameters of the agent's organization (that is a simple way of depicting organizational constraints and preferences) and state of the world, taking into account both global state (i.e. system's state) and local state (i.e. agent's state). They can be either *contract-related* or *application-related*. *Contract-related* behaviors are static behaviors related to contracts (e.g. how to initiate a contract, how to terminate a contract, etc.) whereas application-related behaviors are related to domain-associated sets of actions specified in the contract (e.g. how to perform a payment or a shipping).

3.2 CONTRACT Agent components

This section explains the components that form an agent in the CONTRACT platform. The platform is composed by agents focused on contract management, this fact reflects on agent's internal architecture, as agents will include a component designed specifically for contract management purposes.

The architecture is formed by the following components, each of them will be introduced in depth later in the section:

- Decision maker: Core component of the agent containing its intelligence. Allows agent to deliberate about contracts (e.g. how to achieve their clauses). The decision maker makes use of the rest of the components.
- Contract Manager: Contains the knowledge about contracts. Aware of the contract clauses that apply to the agent, is responsible of notifying *Decision Maker* about facts related with them (e.g. pending obligations of the agent according to the contract clauses).
- Workflow Manager: Embeds an execution engine that supports the execution of pre-defined workflows. Able to monitor workflow execution (as it will satisfy contract conditions) is responsible of sending notifications both to *Decision Maker* and *Contract Manager* components.

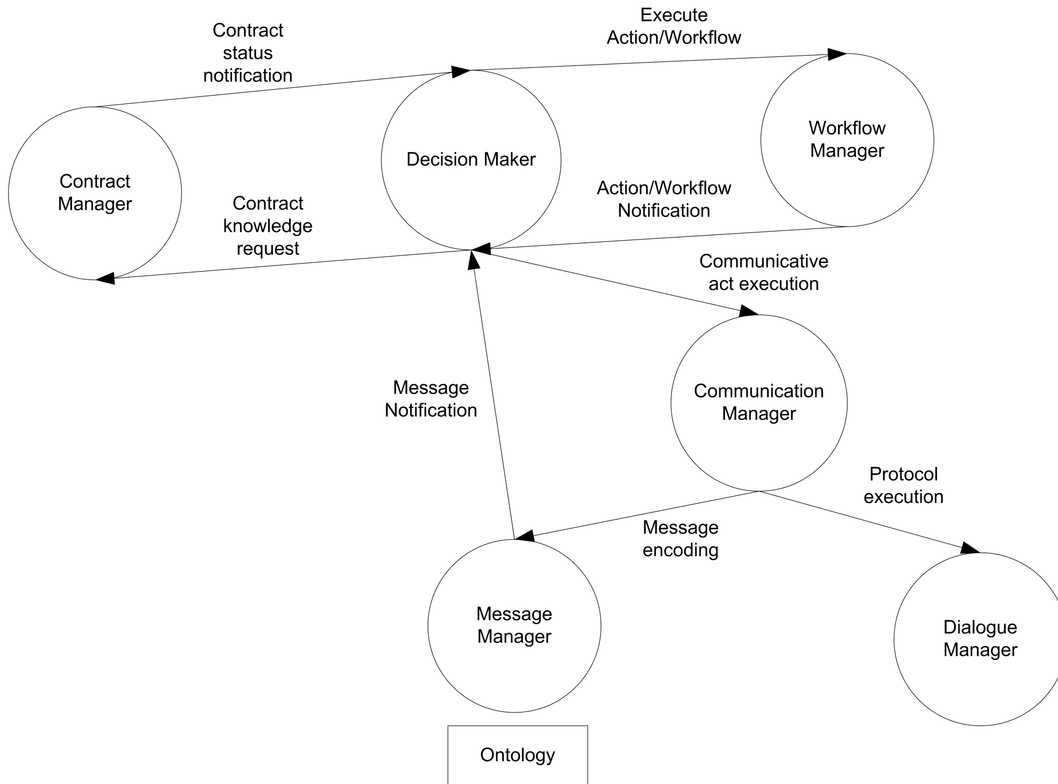

Figure 3.2: CONTRACT Agent components, hierarchical view

- Communication Manager: Takes care of communication between agents .Makes use of both *Dialogue Manager* and *Message Manager* components.
- Dialogue Manager: Implements a set of contract-oriented interaction protocols, specifying for each of them the set of sequences of messages that are acceptable w.r.t the protocol.
- Message Manager: Responsible of processing the content of messages (either incoming or outgoing ones).

Figure 3.2 gives a hierarchical overview of the agent's components. Please notice this picture denotes a design philosophy similar to the one seen on *Figure 3.1* where the decision making component is the core of the architecture, controlling the execution flow of all the other modules.

3.2.1 Decision Maker

This component contains the core intelligence and reasoning cycle (i.e. agent's control loop) of the agent. In CONTRACT it allows to program the agent using contract-related behaviors that model agent's deliberation about achievement and violation of contract's clauses.

The *Decision maker*, as depicted on *Figure 3.2* is assisted by the *Contract manager*, *Communication manager* and *Workflow manager* components for handling low-level actions

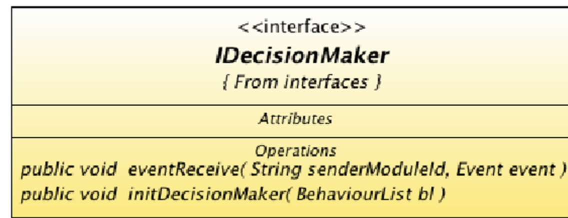

Figure 3.3: Decision maker interface

such as querying the state of a contract, sending a message to another agent or enacting some actions as specified by the domain, respectively. The components that assist the *Decision Maker* communicate with it by sending events (e.g. the reception of a message, the violation of a contract clause or the failure in the enactment of an action) that will update Agent's global state. *Figure 3.3* depicts the interface other components use for communicating with the *Decision maker*.

3.2.2 Contract manager

The *Contract Manager* contains the contract knowledge and business logic of the contracts. It is expressed in terms of predicates and actions, that must be formally defined in the ontology. It must be aware of the contract clauses that apply to a given agent, and their status (either active, inactive or violated). The *Contract Manager* notifies the *Decision Maker* about pending obligations, risk of violating a given contract clause, of the fulfillment of the contract.

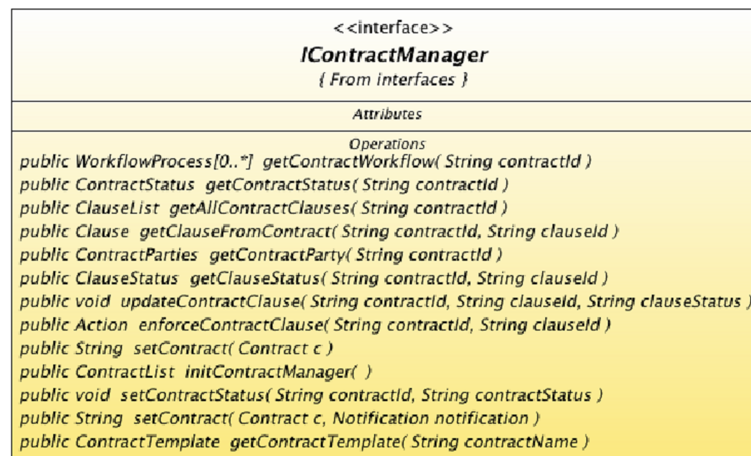

Figure 3.4: Contract manager interface

The *Contract Manager* exposes an interface with operations that allow the *Decision Maker* to submit queries about contract knowledge (e.g. clauses of a contract). *Figure 3.4* depicts this interface.

3.2.3 Workflow manager

On the one hand, the *workflow manager* embeds an execution engine that supports the execution of pre-defined contract workflows. These workflows are defined in languages such as BPEL or XPDL. On the other hand the *workflow manager* contains the operation knowledge required to create the contract workflow, that is, the order in which the actions have to be carried on. Actions are defined by inputs, out-puts, pre-conditions and post-conditions. The execution of a workflow results in the execution of a sequence of actions that eventually satisfy contract obligations. The *workflow manager* monitors and controls action execution, signaling fulfilled actions both to *Decision maker* (that can decide which step to perform next) and to *Contract manager* (which marks active obligations associated to the execution of the action as fulfilled).

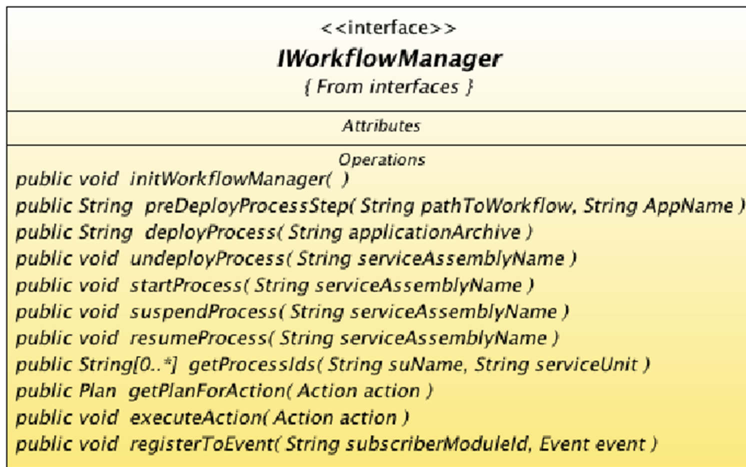

Figure 3.5: Workflow manager interface

The *Worflow Manager* exposes an interface with operations that allow the *Decision Maker* to manage and register workflows. *Figure 3.5* depicts this interface.

3.2.4 Communication Manager

The *communication manager* takes care of agent-agent communication. It includes querying the *Directory Facilitator* (as seen on *Section 2.1.1.2*) to find out the active agents in the system, along with their capabilities, in order to contact them. It also knows which interaction-protocols to choose according to the needs of the contract behavior being run. Such protocols include, among others, the set up of a new contract. This module is assisted by the *dialogue manager* and the *message manager*.

The *communication manager* exposes an interface that allows the *decision maker* to find out if a given communicative goal has been fulfilled (i.e. a communication interaction has ended). *Figure 3.6* depicts this interface.

3.2.5 Dialogue Manager

The *dialogue manager* implements a fixed library of interaction protocols, storing for each of them the set of sequences of messages that are acceptable for fulfilling the goal of the

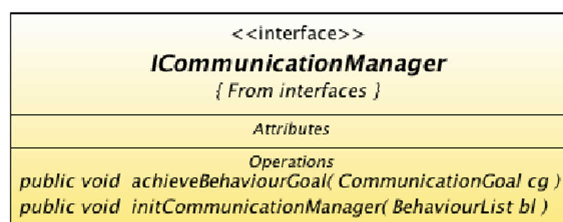

Figure 3.6: Communication manager interface

protocol. These sets of sequences are stored as finite-state machines, this allows keeping track of the current state of the protocol, and ensures the interactions are compliant with protocol's definition.

It must be remarked the set of available interaction protocols is focused on contract management. That is, there are protocols available to create contracts, notify about contract clauses, terminate or initiate contracts and such. Such protocols are fairly simple (e.g. they do not contain message loops) thus, the implementation of the finite-state machines that model the protocols is limited to supporting the most simple cases.

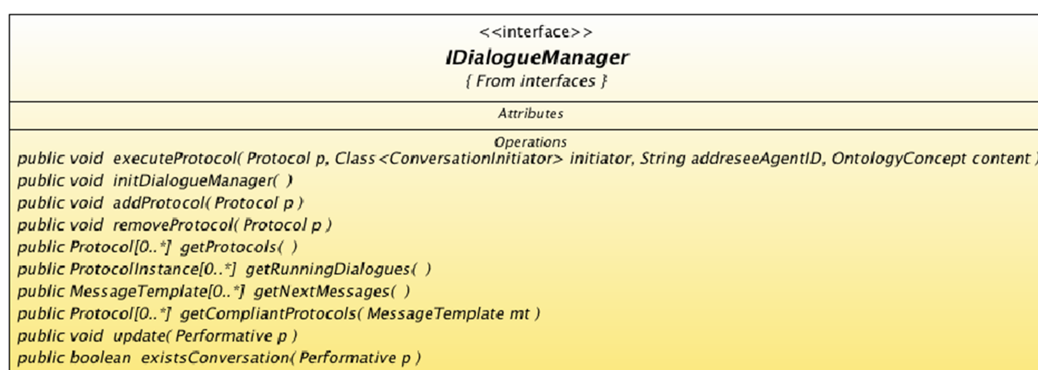

Figure 3.7: Dialogue manager interface

The *dialogue manager* exposes an interface that allows the *decision maker* to manage (add, remove or update) and initiate interaction protocols. *Figure 3.7* depicts this interface.

3.2.6 Message Manager

The *message manager* processes the content of incoming and outgoing messages. Regarding incoming messages, the *message manager* semantically interprets them, translating them into an RDF representation and integrating them into a knowledge base that can be queried later by the *decision maker*. Regarding outgoing messages, the *message manager* will code them from agent's internal representation to a common understandable format shared by all agents. The rules to perform this coding can be defined based on ontological relations. If this is done, an internal component of the *message manager* known as *ontology manager* will assist in the coding process.

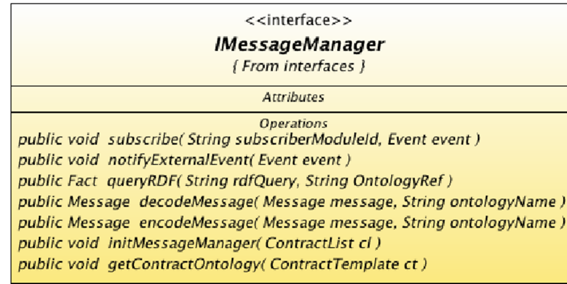

Figure 3.8: Message manager interface

The *message manager* exposes an interface that allows the *decision maker* to query the RDF knowledge base and subscribe to events associated to message (typically, notifications when a given message is received). It also allows the *communication manager* to perform message translations. *Figure 3.8* depicts this interface.

3.3 CONTRACT platform components

This section describes the CONTRACT platform components, that is the components of the architecture that help the agents perform advances functionalities. These functionalities are out of the scope of generic agent's purposes, and focus more on domain's purposes. For instance, in the case of the CONTRACT agent platform, platform components are focused on contract management, monitorization or storing, whereas agent components focus on agent communication, platform integration (allowing access to platform's facilities) and decision taking.

However notice that, in the case of the CONTRACT platform, agent's components also include modules for contract management (in this case, the *contract manager* as depicted on *Figure 3.2*).

3.3.1 Contract Storer

The *contract storer* is a generic agent that has the goal of providing access to a persistent and controlled storage of contracts. It acts as an intelligent wrapper between the *contract repository* component and the other agents. Thus, other agents do not need to know the *contract repository* interface or its implementation details. What's more, *contract storer* is able to check contracts provided for storing, making sure they are well formed before storing them. The *contract storer* stores, retrieves and updates contracts in behalf of other agents. It also responsible of keeping track of changes in the contracts, notifying observer or notary agents.

In the CONTRACT agent platform the *contract repository* is an *eXists* [11] database.

Please notice that, as this component is wrapped by an intelligent agent, it does not expose any interface. It exposes a set of interaction protocols to store, retrieve and update a contract (or a set of contracts) as well as a protocols to query the contracts store or receive notifications when they are updated.

3.3.2 Observer

The *Observer* acts as an event tracker in the contracting environment. Usually agreed by the parties participating in the contracts, they represent neutral sources of trusted contract-related information. The *Observer* gathers data without acting, that is, without altering the state of the world. Listeners (i.e. contract parties) can attach to observers, receiving notifications on the events of interest happening in the contracting environment.

Please notice that, as this component is an intelligent agent, it does not expose any interface. It exposes a set of interaction protocols to subscribe and de-subscribe to events of interest.

3.3.3 Manager

The *manager* is the complement of the *observer*, it takes actions in the contracting environment, when certain conditions are met. It gets environmental information from the observer, and maps it to contract states, taking actions (e.g. sending notifications) when certain states are met.

Please notice that, as this component is an intelligent agent, it does not expose any interface. It just will start interaction protocols with two purposes. First, subscribe to the observer in order to be able to obtain environmental information. Second, notify the agents affected by the contract states that are reached.

3.3.4 Notary

The *notary* is a generic agent allowed to certify contract management processes. Other agents in the system agree that contract creation, updating and cancellation process approved by a *notary* are valid.

3.3.5 Context Service

The *context service* captures and stores information about the configuration of the deployed platform. For instance, the *context service* can be used to obtain the location of *ontology service* and to initialize the *directory service*, as it will contain information about the agents present on the system at boot time.

3.3.6 Domain Ontology Service

The *domain ontology service* provides a registry based service for finding and accessing domain ontologies. This component provided a mapping between ontology identifiers and their locations (e.g. URLs).

3.4 The PAWS Agent Platform

This section explains the PAWS Agent Platform. The platform is based on the CONTRACT Agent Platform, but focused to generic purpose agents. Thus, all the components to be re-used that are bound to contract management are to be generalized. Some of the components do not make sense anymore in a general agent platform and some new components must be added in order to achieve a fully general platform.

In order to generalize the agents in the architecture, the business contract parties present in the CONTRACT platform must be generalized to agents. Agent's purpose will depend on

the domain the PAWS platform is applied to, rather than having a fixed purpose: managing contracts.

When generalizing the architecture of the agents the first step is to unbound the modules in the agent, allowing to change the ones implemented for the IST-CONTRACT project (which are static) for another modules. Thus, more generic modules can be implemented and replace the ones existing in the CONTRACT platform in order to develop the PAWS platform. What's more, additional modules can also be added to the architecture, making it more flexible. *Figure 3.9* shows the new architecture we have designed with the generalization applied.

In the new, generalized architecture, the following actions are performed for each component:

1. Agent-Agent interface: The most important generalization required in this interface comes from the usage of protocol interfaces that rule agent-agent interactions. On the one hand, in CONTRACT platform these interfaces are rather simple and do not support some features (such as message loops, or extensions of performatives).

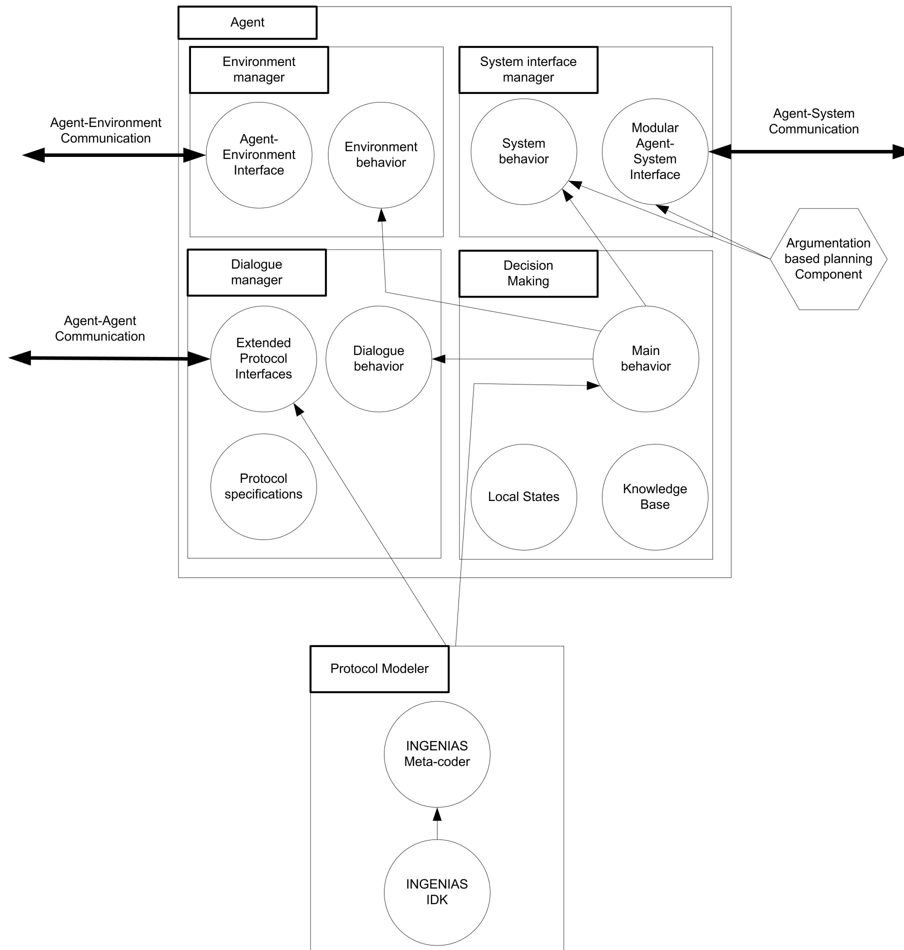

Figure 3.9: PAWS Agent architecture overview diagram

Figure 3.10: CONTRACT and PAWS agent configuration files example

These interfaces have been extended in order to support such features. Extending these interfaces implied modifying the *Protocol tree* class, which is the core class of the interfaces. On the other hand, only a few sample protocol specifications are available on the CONTRACT platform. To tackle this generalization, rather than implementing new specifications, a procedure to allow PAWS programmers to easily define them is provided now via the *Protocol Modeler* module.

2. Agent-System interface: This generalization is rather simple. Apart from keeping the interfaces with the available components, new interfaces must be added when a new component is included in the system. This process is already rather straightforward on the CONTRACT platform, so no generalization is required on this sense. Also, agent configuration files (and configuration file parsers) must be updated to allow the inclusion of new components, and the replacement of the existing ones. *Figure 3.10* shows how agent configuration files have evolved due to this generalization.

3. Agent-User interface: This interface must be generalized to allow not only Agent-User interaction, but also Agent-Environment interaction. Fortunately, the implementation of this interface in the CONTRACT platform (i.e. a message queue) is general enough to support both interactions without requiring further generalization. Future work can include an improvement on the implementation that takes into account message priorities, thus perceptions with higher priority can be processed before than perceptions with lower priorities. This would allow using the PAWS platform in applications with soft-realtime constraints [28].

4. Decision making module: This module's generalization is also focused on available sample protocols, just like *Agent-Agent interface*. The process that generates protocol interfaces will also generate *main behavior* stubs that can be easily adapted to fit

programmers needs. Minor generalizations are also performed, such as transforming the policies base into a more general knowledge base.

5. Protocol Modeler: This module is not generalized, but added from scratch to the platform. In this picture, it appears as an agent module as an example, but in the last version of the PAWS platform it is implemented as a platform component. The module is fully described in *Section 3.6.4.1*

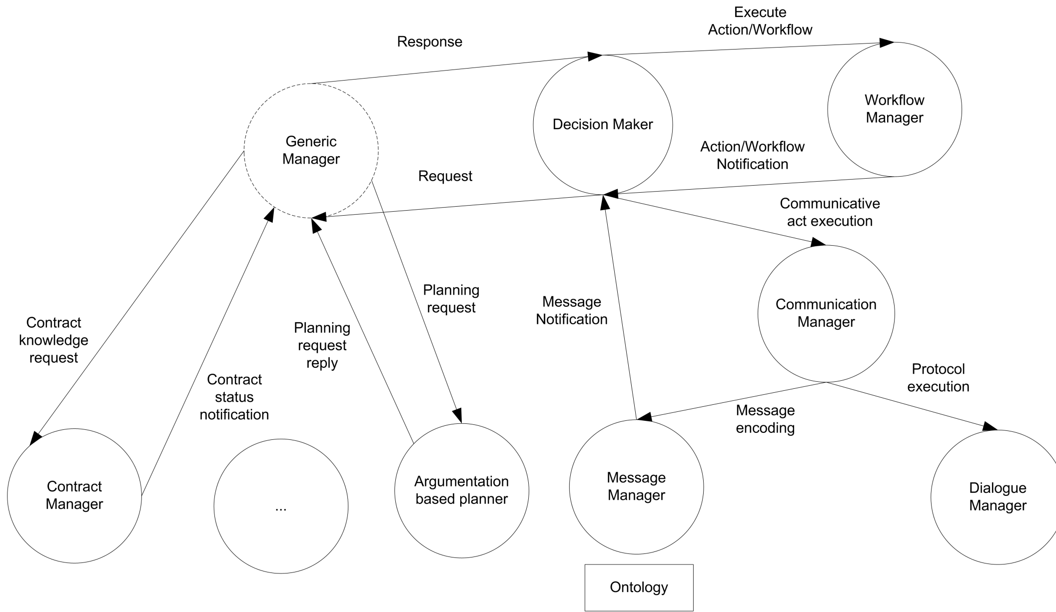

Figure 3.11: PAWS Agent components example, hierarchical view

Figure 3.11 gives a hierarchical overview of an example of the generalization of the architecture. In the generalization example, the *Contract manager* has been exchanged for a *Generic Manager* which is an interface that supports the interaction with concrete managers such as the old *Contract manager* (in case PAWS is applied to a domain where contracts are required) or the freshly added *Argumentation based planner*[2]. Please note this figure shows just an example, due to its modular design PAWS programmers can add or remove components at will. As summary, architecture is formed by the following components, each of them will be introduced in depth later in the following section:

- Decision maker: Core component of the agent containing its intelligence. Allows agent to be programmed to deliberate about the domain. The decision maker makes use of the rest of the components.

- Generic Manager: This module is an empty interface implemented in PAWS to deattach the domain dependent *Contract Manager* component present in the CONTRACT platform. Domain dependent modules can be attached to this module (or even replace it). In *Figure 3.9* both *Contract Mananger* and *Argumentation base planner* components are attached to this module, as example.

[2]This component is introduced on *Section 5.3*

- Workflow Manager: Embeds an execution engine that supports the execution of predefined workflows. Able to monitor workflow execution and send send notifications to *Decision Maker*. Thus, *Decision Maker* is aware of workflow's execution result and able to take decisions consequently.

- Communication Manager: Takes care of communication between agents .Makes use of both *Dialogue Manager* and *Message Manager* components.

- Dialogue Manager: Implements a set of generic interaction protocols (i.e. FIPA standard procotols), specifying for each of them the set of sequences of messages that are acceptable w.r.t the protocol. Domain-dependent protocols can be modeled by user and implemented using the *protocol modeler* architectural component.

- Message Manager: Responsible of processing the content of messages (either incoming or outgoing ones).

3.5 PAWS Agent components

This section explains in depth the PAWS agent's components that have been introduced before, when providing an overview of the architecture in the previous section. As the PAWS platform is based on the CONTRACT platform agent components can be divided in four types:

- Components directly re-used from the CONTRACT platform: these components have been implemented as exact copies of the CONTRACT platform components. For each of these components, its functionality is explained and a reason for having implemented them as direct copies given.

- Components generalized from the CONTRACT platform: these components have been implemented inspired on the CONTRACT platform components. However, they have been modified: the components where too oriented to contract management and have been generalized in order to fit in an agent platform with generic purposes. For each of these components, its functionality and the modifications performed to make them more general are explained.

- Components removed from the CONTRACT platform: these components where present on the CONTRACT platform but are not on the PAWS platform. For each of these components, its functionality is explained and a reason for having removed them is given.

- Components added to the PAWS platform: these components where not present on the CONTRACT platform but are on the PAWS platform. For each of these components, its functionality is explained and a justification for including them is given.

3.5.1 Agent components directly re-used from the CONTRACT platform

This section introduces components that have been implemented in the PAWS platform as exact copies of the CONTRACT platform component.

3.5.1.1 Workflow manager

This module does not need to be generalized. However it might need to be extended if some action executions are to be notified to modules different than the *Contract manager*. An example of such modules is the *norm monitor*[3] that might need to be notified of action executions to control if executing such actions results in the violation of an active norm. Another possible extension to be done in the future is including support for workflow languages different than BPEL or XPDL.

3.5.1.2 Communication Manager

This module has not been generalized for the PAWS platform. Future improvements of the platform could include using federations of *Directory Facilitator* components in order to support inter-platform agent discovery.

3.5.1.3 Message Manager

This module has not been generalized for the PAWS platform. Indeed, it was more general than required on the CONTRACT agent platform, because it is performing message translation when the scope of the project expects all agents to use the same encoding and language. Thus, this module is integrated directly on the PAWS platform. Future improvements of the PAWS platform could implement more complex processes of ontological translation, such as trying to perform a translation between two different ontologies with some common points.

3.5.2 Agent components generalized from the CONTRACT platform

This section introduces components that have been reused from the CONTRACT platform. Being too contract-oriented for a generic purpose agent platform they have been modified to make them more generic.

3.5.2.1 Decision Maker

No actions are to be performed when generalizing this component for the PAWS platform. However, the behaviors inside this component must be coded and adapted to the domain PAWS is applied to.

In PAWS, the *Decision maker* will be supported by other modules. These modules can implement functions to access *Decision maker's* interface when required or return results via custom interfaces. The *Decision maker* will get support from these modules by invoking them on the behavior. Please, remember *Figure 3.12* shows an example of such integration between the main component (the *decision maker*) and the new modules. Also, in PAWS stubs of the behaviors controlling the *Decision maker* will be available for programmers to adapt them. Such stubs will be generated from a graphical interface.

3.5.2.2 Dialogue Manager

This module has been heavily modified for generalizing it and making it suitable for the PAWS platform.

Fist of all the implementation of the finite state-machines must be extended, in order to support more complex protocol descriptions. It includes allowing loops on the finite-state

[3] As introduced in *Subsection 3.5.3.1*

machines (this will enable modeling protocols with messages that can be sent multiple times in a row) as well as backwards arcs to states that have been already visited (this will enable modeling protocols with loops).

Second, the interface must be extended, adding a function that supports updating the state of a protocol. This will enable more complex interaction protocols, where a participant can send several messages before receiving a reply. The interaction protocols implemented in the CONTRACT platform, allow a participant to send a message and then another participant to reply by sending another message. In the interaction protocols of the PAWS platform, a participant can send two or more messages to any number of parties before receiving a reply.

Last but not least, the set of interaction protocols on the library must be extended, making it contain not only contract-management oriented protocols. Apart from creating new protocols (e.g. contract-net and iterative-contract-net protocols) this process includes the integration of the protocol library with the *protocol modeler* component. This will allow PAWS programmers to define new protocols in a graphical and intuitive way.

3.5.3 Agent components removed from the CONTRACT platform

This section introduces components that where present on the CONTRACT platform, but are not one the PAWS platform. They are too contract-oriented for a generic purpose agent platform.

3.5.3.1 Contract manager

This module does not need to be generalized. It will be available for including it in PAWS-based applications, in case they have to deal with contracts. In the future, it can be used as a basis for other modules that need to keep track of events and notify them to *Decision*

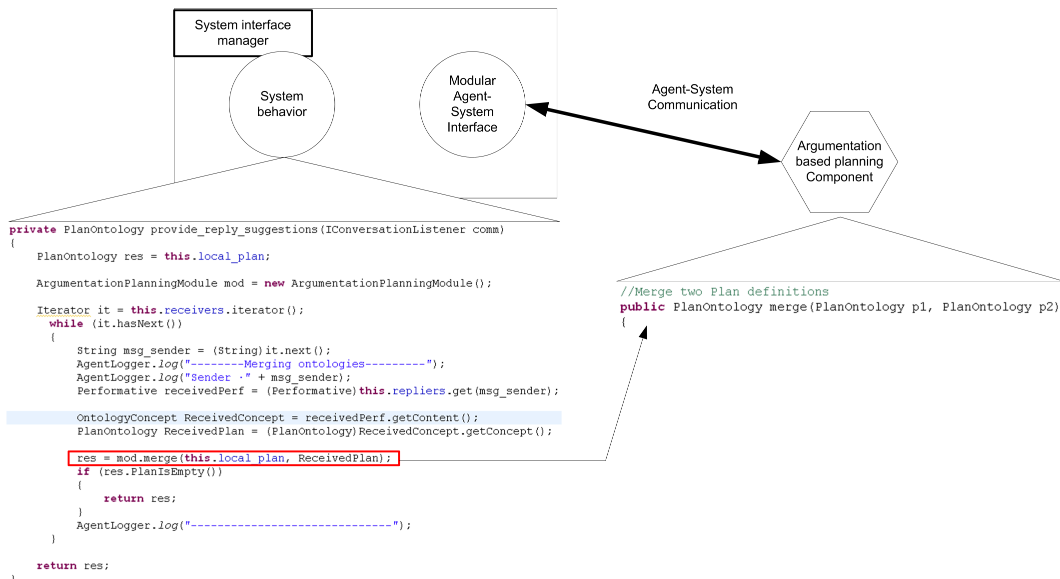

Figure 3.12: Module integration on behavior example

Maker. An example of such component is a *norm monitor* that, given a set of norms and an engine to process them, notifies the *Decision Maker* when a norm is violated, going to be violated (such as in the case of deadlines) or becomes active.

3.5.4 New agent components on the PAWS platform

No new components have been implemented for the agents in PAWS. PAWS users can add new components to the agents if they consider agents should incorporate new functionalities. For instance, in the example provided in *Chapter 5* a planning module has been included to provide planning functionalities to the agents in the system.

Please note that, when adding new components, new behaviors are to be attached to the *main behavior*. This fits in the concept that the *main behavior* of the agent will make use of the new modules. An example of such behavior extension (integrating the *main behavior* with a planning component) is available on *Figure 3.12*.

3.6 PAWS Platform components

This section describes PAWS platform components. Platform components are the generalization of the administrative parties present in the CONTRACT platform.

PAWS platform components can be implemented either as sets of PAWS agents or as Java modules[4]. PAWS platform components provide functionalities to the agents in the platform and to the human users of the platform. Modules to help users in the implementation of the system are present on the PAWS platform, something that was missing in the CONTRACT platform.

When adding new components as Java modules, if they are to be used by agents, its interface must be published. What's more, component's invocation details can be made opaque to agents by extending the *Agent-System* interface. Please notice that, in the case of components implemented via PAWS agents, no interface is to be published, as they will be accessed via the *Agent-Agent* interface.

When providing new functionalities to agents, the platform components fulfill the same purpose as the agent components explained in the previous section. However, platform components are outside agent's architecture, they are accessed through the *Agent-System* or the *Agent-Agent* interfaces, and thus, accessing to them is far less efficient than accessing to agent's internal components. What's more, in the case of agent components there will be one component per agent, whereas platform components are expected to be unique (although, they can be replicated if required).

In PAWS it is up to programmers who adapt the agent platform to their domain whether to implement new components as platform or as agent components, and weather to move them from one type of component to the other. As a rule of thumb, if a component is to be used extensively by all agents due to domain requirements, it should be implemented as agent component. Otherwise, a platform component implementation fits better in the requirements. Please notice that components to be used by users in the platform, or implemented as sets of PAWS agents and not as Java modules, are always platform components.

Just like agent components, PAWS platform components can be divided in four types:

- Components directly re-used from the CONTRACT platform: these components have been implemented as exact copies of the CONTRACT platform components. For each

[4] This feature was already supported by the CONTRACT platform

of these components, its functionality is explained and a reason for having implemented them as direct copies given.

- Components generalized from the CONTRACT platform: these components have been implemented inspired on the CONTRACT platform components. However, they have been modified: the components where too oriented to contract management and have been generalized in order to fit in an agent platform with generic purposes. For each of these components, its functionality and the modifications performed to make them more general are explained.

- Components removed from the CONTRACT platform: these components where present on the CONTRACT platform but are not on the PAWS platform. For each of these components, its functionality is explained and a reason for having removed them is given.

- Components added to the PAWS platform: these components where not present on the CONTRACT platform but are on the PAWS platform. For each of these components, its functionality is explained and a justification for including them is given.

3.6.1 Platform components directly re-used from the CONTRACT platform

This section lists the components that have been implemented in the PAWS platform as exact copies of the CONTRACT platform component.

3.6.1.1 Observer

The *observer* component has been implemented as a direct copy of the CONTRACT platform's *observer* component. A generic *observer*, gathering information about the environment in general, rather than about the contracting environment, is provided along with the PAWS platform, ready to be used by any agent that can't directly observe the environment, or wants to focus on reasoning, delegating observation to another component.

3.6.1.2 Context Service

The *context service* component has been implemented as a direct copy of the CONTRACT platform's *context service* component. It provides PAWS with a simple implementation for the *context service*, that reads the configuration information from a file formatted in XML style and stores it in a Java class. Users of the PAWS platform are encouraged to modify this class, adapting it to their needs, and providing more complex implementations if required.

3.6.1.3 Domain Ontology Service

The *domain ontology service* component has been implemented as a direct copy of the CONTRACT platform's *domain ontology service* component. It provides PAWS with a simple implementation for the *domain ontology service* that does not support registering ontologies. All the available ontologies must be specified at boot time, no more ontologies can be added later.

3.6.2 Platform components generalized from the CONTRACT platform

This section introduces components that have been reused from the CONTRACT platform. Being too contract-oriented for a generic purpose agent platform they have been modified to make them more generic.

3.6.2.1 Contract Storer

This component has been generalized from the *contract storer* for its implementation in PAWS. The *generic storer* wraps the *generic repository* component, and includes protocols to store, retrieve and update a generic component (or a set of them) as well as protocols to perform queries on the *generic repository*. Please, notice that currently the *generic storer* does not perform any checking on the objects store. In future versions, a set of rules that any object to be stored must comply with can be specified in a *storing rules base*. Then *generic storer* can then check that the restrictions modeled by these rules are met before storing objects in the *generic repository* (or even retrieving them, which can be used to model query permissions).

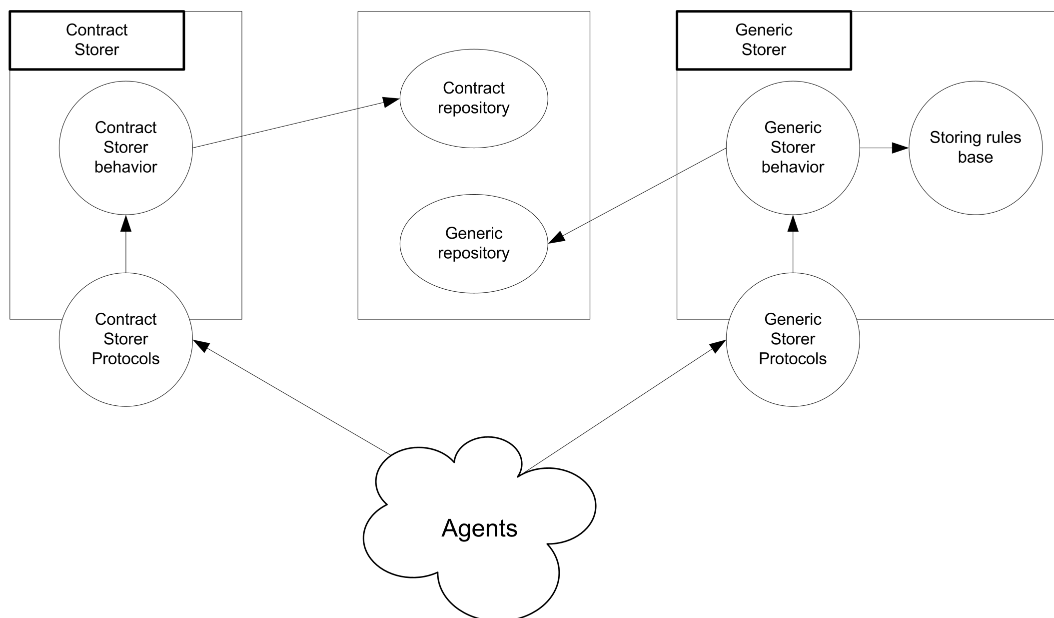

Figure 3.13: CONTRACT's contract storer and PAWS' generic storer

Figure 3.13 shows the hierarchical set of components on *contract storer* and *generic storer*.

3.6.3 Platform components removed from the CONTRACT platform

This section introduces components that where present on the CONTRACT platform, but are not one the PAWS platform. They are too contract-oriented for a generic purpose agent platform.

3.6.3.1 Manager

CONTRACT platform's *manager* component is strongly coupled to the contractual domain, and thus, it has been removed in PAWS. Future implementation of the PAWS platform are not likely to include any component with a functionality similar to the one of the *manager*, as this component would be only useful on very specific domains.

3.6.3.2 Notary

CONTRACT platform's *notary* component is hardly coupled to the contractual domain, and thus, it has been removed in PAWS. Having a neutral component able to validate actions that are key in the business process can be very convenient in some domains, as it forces agents to agree in the validity of the actions performed. However, we think it is better PAWS programmers code these component, adapting them to their domains (and thus, having components strongly coupled with the domain, like the *notary*) than providing a general component for this purpose.

3.6.4 New platform components on the PAWS platform

This section introduces components that are new to the PAWS platform.

3.6.4.1 The protocol Modeler

The *protocol modeler* is the only platform module available for users in the PAWS platform. As it has been mentioned before, the purpose of the protocol modeler is to provide PAWS users with an intuitive and graphical way to generate PAWS protocols. This need comes from the limited (and contract oriented) protocol-interface library CONTRACT agent platform provides.

Apart from generating protocol interfaces (so the protocol library is completely generic and easily adaptable to any domain), the protocol modeler is able to generate *main behavior* stubs that can be easily adapted by PAWS programmers to the needs of the domain where the PAWS platform is applied to. These stubs tackle another generalization problem, going from the closed *main behaviors* defined in the CONTRACT platform to open behaviors.

Here, a quick overview of the functionalities and the architecture of the *protocol modeler* component is provided. An in-depth guide about how to use the *protocol modeler*, based on a complex example, can be found at *Section 5.1*). Also, *Section 5.1* will explain how a protocol model can be transformed to PAWS protocol interfaces and code stubs.

Figure 3.14 shows the architecture of the protocol *modeler component.* The user defines the protocols to be implemented via the software *'INGENIAS Development Kit'* [17]. Then, a meta-coding program created for the PAWS agent platform generates, for each protocol defined, a set of protocol interfaces, a set of behavior stubs and an ontology stub.

'INGENIAS Development Kit', intends to guide and facilitate the designing and building of multi-agent systems. A part of this design is the *'Interaction Model'*. This model defines a set of interaction units that represent a single information exchange between two agents. In other words, each unit represent an agent sending a message (which contains some information) to one or more agents. Each interaction has an initiator and several collaborators assigned. The initiator can be seen as the agent that sends the message, whereas the collaborators can be seen as agents willing to receive and process the information contained in the message. An interaction unit is defined for each message in the protocol, and then, assigned initiator and collaborators according to the responsibilities in the protocol.

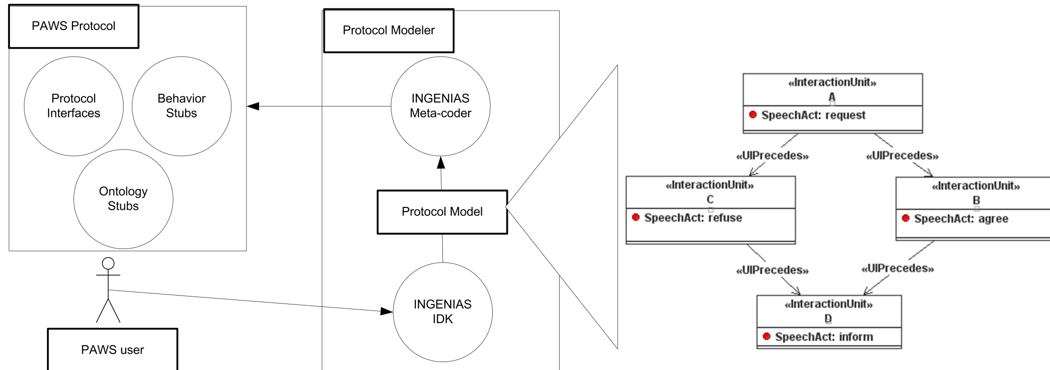

Figure 3.14: Protocol modeler architecture

The protocol interfaces define the protocol in terms of responsibilities and message precedences. These files are used to ensure the correct execution of the protocol from two points of view. First, a message can only be sent by the agent responsible of sending it and received by an agent responsible of receiving it. Second, a message cannot be sent if the preceding message is not sent. The only exception to this second rule is the message that starts the protocol, as it has no preceding message. Protocol interfaces are used to constraint the execution of the protocol, preventing it from taking execution paths that are not the ones specified in the *interaction model*. Protocol interfaces contain the following elements:

- A Java file for each message in the protocol. These files are extensions of PAWS performative files, that are implementations of FIPA performatives. For instance, in the example presented in *Figure 3.14*) message *B* is an extension of PAWS *agree* performative.

- A single protocol Java file. These file sets up roles in the protocol, specifying initiator and participant roles. Then, sets up messages in the protocol, specifying responsibilities (i.e. which agent will send the message and which agent or agents will receive it) and precedences (which message can follow each message in the protocol). Finally, the file specifies which is the starting message of the protocol.

- A role file for the initiator of the protocol. This file specifies which messages can receive the initiator of the protocol, and which responses can be provided to each of these messages.

- A role file for each participant in the protocol. This file specifies which messages can receive the participant, and which responses can be provided to each of these messages.

The *main behavior* stubs are Java classes where user must add parts of the code in order for the class to function properly. The *protocol modeler* component generates one stub for the initiator of the protocol, and one per participant. Stubs take care of inter-agent communication issues, including code for receiving messages from other agents and sending messages to them. However, the intelligence to decide the content of these messages, or which message to send if there are several alternatives is missing. For instance, the code for deciding if message *'B'* or *'C'* are to be sent in reply to message *'A'*, as seen on *Figure 3.14* will not be included in the stub of the participant agent.

The *ontology* stub is a Java class with a simple ontology with just one String-type field. The stub can be extended by the user, adding additional fields, or replicated, in the case different messages require different ontological contents.

3.7 PAWS platform evaluation

Agent development is becoming more common, even in commercial domains, so it is worth considering agent platform's strengths and weak points. Furthermore there is an important number of agent platforms available, this fact makes it more important to evaluate our PAWS with respect to other available platforms.

This comparison is performed based on the following set of neutral criteria as used in other papers (e.g. [21]):

- Standard compatibilities. Focusing on FIPA compatibilities.
- Communication supporting inter-platform messaging.
- Support for strong (code and process thread) and weak (just code) mobility. Migration method must be clean and efficient. Component of the architecture (i.e. daemon) to stop and restart threads on strong mobility.
- secure intra-platform and, specially, inter-platform communication.
- platform usability and documentation provided.
- state of the development. Open issues, possible future improvements of the platform, etc.

The PAWS agent platform throws the following results according to the comparison criteria.

- Compatible with FIPA specifications. Some components of the platform (e.g. the performatives in the protocol interfaces, the *Directory Facilitator*, etc...) have been designed based on FIPA specifications.
- Full support for inter-platform messaging. Support for inter-platform communication is limited by the simplicity of the *Directory Facilitator* component. This is because *Directory Facilitator* components cannot query each other to find out if a given agent is hosted in a remote platform. That is, in order for the agents in one platform to communicate with the agents in another platform, the configuration file that initializes the *Directory Facilitator* component needs to explicitly specify the remote agent is hosted on another platform.
- Both strong and weak mobility are supported, because PAWS agents are based on Java processes and Java classes. However the migration process shows some complicated points, such as the agent having to update the *Directory Facilitator* components of the platform it is leaving and the platform it is moving to, in order to be reachable by the rest of agents.
- Current version of the PAWS platform provides no message encoding. However, due to the modular design of the platform it is not complicated to substitute the *message manager* component for a *encoded message manager* component if the domain requires it.

- Graphical interfaces provide an easy experience when setting-up the platform, run agents on it, or even start coding them. Documentation on available interfaces is sufficient, so PAWS users will be able to develop their own modules and connect them to the platform without much problem. However, if some available component of the platform is to be changed, the process can be hard and tedious, due to the lack of documentation available on how the internal components of the architecture work. For the same reason, locating and fixing an existing bug can also be a thought task.
- The platform is mature enough. Most of the components have already been tested in an European project due to PAWS inheritance from the CONTRACT agent platform. Because of its modular design, new modules (or improvements on the existing ones) can be easily developed in the future.

3.8 Summary

In this chapter we have presented the PAWS platform, a generic agent platform aimed to host agents that serve as proxies for web-services.

As the PAWS platform is based on a generalization of the CONTRACT agent platform, the chapter has started by introducing the CONTRACT agent platform. With reference to this, we have seen an agent platform that presents the components defined on the FIPA-ABSTRACT specifications defined on *Chapter 2*, but when analyzing them, we have realized that most of them are bounded to contract-management purposes. What's more, we have identified some components (e.g. *contract manager*, *Notary*, *Manager*, etc.) which are fully-coupled with contract-management environments and not defined in the FIPA-ABSTRACT specifications.

As it goes on, the chapter aims to provide a better understanding of the PAWS architecture and the functionality of its components. With reference to this, the chapter shows a generic and modular agent platform, where users can attach new components easily. The chapter has explained how the PAWS platform differs architecturally from the CONTRACT platform. It also has gone over the components of the CONTRACT platform that have been re-used or removed on the PAWS platform, as well as the components of the PAWS platform that are not present on the CONTRACT platform. Reasons for adding, removing or re-using components between the two platforms have been given.

At the end of the chapter, the reader should have a good idea about the architecture of the PAWS platform, knowing which are the available components, their functionalities and how do they interact between them.

In the next chapter we will describe an *argumentation based planning framework* from a theoretical point of view. The information given in this chapter and on the next one will come in hand on *Chapter 5* where the PAWS platform will be used for implementing the theoretical planning framework.

Chapter 4

A theoretical planning framework

Dynamic service composition is an uprising trend in the Web services community, being the subject in many recent works [32]. Such works typically approach this issue via distributed planning systems (such as the GPGP/TAEMS framework [18]) where actors build partial plans that are then shared in order to generate a global plan. However, it is common in such works to neglect the issue of how to manage resource conflicts between the plans. In fact, only a few works tackle this [5].

This chapter presents a work that aims to help filling this gap, by developing a system able to detect and handle resource restrictions at runtime, during global plan generation process, and applying it to dynamic and distributed service composition. It does so by using the concepts of 'shared resource' and 'utility'. Assigning resources to Web services (via, e.g., OWL-S [19] annotations) arises the need of making sure that the workflow forming the global plan does not have resource conflicts, as resources might be limited and shared between actors. In our approach, such restriction is ensured using the idea of *conflict-freeness* introduced in Argumentation Theory [9]. Assigning utilities to Web services (via agents' knowledge bases) enable the agents to rationally choose a set of local workflows when several alternatives (either different workflows, or different paths inside the same workflow) are available in order to perform a given composite task.

The planning framework is introduced along with an example, where interactive community displays provide touristic information and services. This information is adapted taking into account elements such as user preferences, current location, weather forecast and such. The displays have the capability of suggesting amusement tours, formed by a set of touristic activities. These set of activities are to be composed into a tour, that needs to be conflict-free (the user will not want to include two activities that over-lap in time on the same tour) and adapted to the user's preferences (more than one tour is expected to be available). Thus, in the example, the tours can be seen as workflows and the touristic activities as services that compose the workflow. Resources are time slots assigned to each activity, representing the amount of time the activity will take for user to execute.

This chapter is structured as follows: it starts by providing some background on planning and argumentation theory. Later, it introduces the scenario used for the example. Then, it goes on by formalizing the elements that form a *plan*, and an *agent*, along with their properties. Later, this formal definition is used to explain the concept of *conflicting action*, and how can it be detected using Argumentation Theory. These definitions are also useful

to explain how agents can exchange their global plan proposals using a negotiation protocol. Then, it presents means to allow agents to decide, as a group, which global plan proposal is to be accepted if there are several available. Finally, all these elements are put together in order to introduce a general protocol to generate a global plan from the local plans.

4.1 Background

This section introduces some basic notions on concepts that are useful for understanding the theoretical planning framework introduced later in *Section 4.3*.

The section starts by introducing some notions of planning systems, focusing on explaining how plans can be modeled as trees of tasks and sub-tasks. Then the section introduces some concepts of *Argumentation Theory*. These concepts include the formal definition of *Argumentation Framework* as well as the concept of *Attack Relation*.

4.1.1 Planning systems

Classical planning is seen as a method for finding the set of actions that bring the world from one state to another. In other words, planning characterizes problems as an initial state and a goal state. In general, planning systems take into account the following elements:

- A description of the world, based on states.
- The actual (or initial) state of the world
- The goal (or final) state of the world.
- A set of actions that change the state of the world. Based on pre-conditions (i.e. set of states of the world from where the action can be performed) and effects (i.e. state where the world will be once the action has been performed).

If more than one agent is able to perform the actions in the planning problem, two main approaches can be identified, centralized planning and distributed planning.

In centralized planning one agent is responsible of building the plan that specifies all the actions each agent has to enact. For instance, agents can negotiate in order to choose a coordinator, who will be responsible of distributing the actions to be performed among the set of agents.

Distributed planning allows splitting the problem to be solved into sub-problems that are divided among the group of agents. Each agent will be responsible of solving its own sub-problem. There are two main methods of tackling this approach, task-driven planning and plan coordination[1]. In task-driven planning, the main goal is split into sub-goals that are distributed among the agents. Then, agents build plans to achieve this goal. In plan coordination agents have pre-existing plans, and the issue to be tackled is how to achieve a plan that can be performed in common.

This section will focus in a planning framework for distributed planning known as GPGP (Generalized Partial Global Planning) [18] framework. GPGP was developed as a framework for coordinating teams intelligent agents that cooperate to archive high-level goals. GPGP parts from the idea of generalizing and making domain independent the coordination techniques developed for the PGP (Partial Global Planning) framework [18]. GPGP understands agent coordination as a distributed search in a dynamic goal-tree. This tree is based on TAEMS hierarchical task network representation [6].

[1]The framework introduced in *Section 4.3* is a distributed plan coordination framework

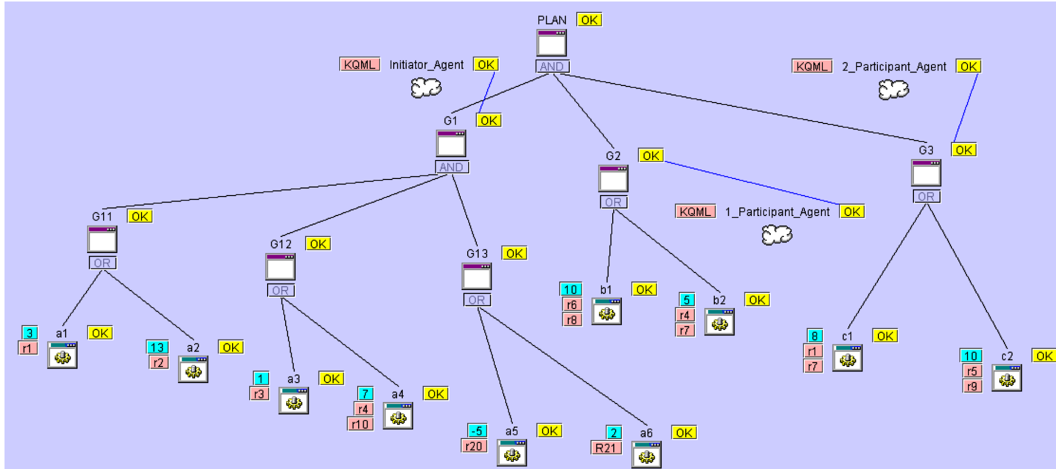

Figure 4.1: Planning problem representation example

Thus, influenced by TAEMS, GPGP represents the problem to be tackled as a tree of task-subtask dependencies. This tree models the problem, from the high level goals shared among the agents participating in the coordination process, to the most basic actions that are to be enacted by the agents, linking them via task-subtask relationships in a tree-like structure.

In the example shown in *Figure 4.1* the high-level goal is shown as the task *plan*. This task is de-composed into three simpler tasks known as *G1*, *G2* and *G3*. Each of these tasks is assigned to an agent(note the link between the task and the agent), and agents collaborate in enacting them. Once all agents have enacted the task they have been assigned, the high-level goal *plan* has been fulfilled. Note that each of these tasks are de-composed into simpler tasks. This de-composition can be performed via different types of task-subtask relation, including *AND* links (in order to enact the task, all the subtasks are to be enacted), *OR* links (in order to enact the task, at least one of the subtasks is to be enacted) and *XOR* links (in order to enact the task, exactly one of the subtasks is to be enacted). These task-subtask relations go on until tasks that cannot be de-composed anymore, denoted as *leaf-tasks* in this document, are met (in the example *a1,a2,a3,a4,a5,a6,b1,b2,c1,c2*). These *leaf-tasks* are the simple actions that the agents will enact in order to enact the more complex actions and, in the end, fulfill the common goals.

This representation of the problem allows alternative groups of *leaf-tasks* to be enacted in order to fulfill the common goals. In the example of *Figure 4.1* enacting both sets of actions *a1,a3,a5,b1,c1* and *a1,a4,a5,b2,c2* will result in the common goals being fulfilled. GPGP deals about enabling the agents to decide which subtasks will be enacted in order to fulfill the high-level goals. The planning framework presented in this document also tackles this issue, from a different approach and adding some extra restrictions (i.e. resource-sharing among actions) in order to make it more appealing.

4.1.2 Argumentation

The study of argumentation is concerned about how assertions are proposed, discussed and resolved in a given context when several diverging opinions are held. Argumentation development is the process by which parties engaging in debate undermine contrary stances and

arguments advancing in their respective positions. Argumentation is specially fit on environments where distributed intelligence, autonomous components or synchronous interaction are required.

So, analysing the outlines of the preceding paragraph one can identify the following core components of argumentation:

- Parts that compose an argument and the relationship between them.
- Rules and protocols that describe an argumentation process.
- How to distinguish legitimate from invalid arguments (or argument lines).
- How to distinguish an ending state, from where further argumentation is useless or redundant.

These components resemble the ones of formal logic reasoning, and indeed, logic and formal deductive reasoning have provided the basis to model and analyse argumentation from a computational point of view. However, several key differences can be found between both concepts. One can easily notice that sentences *'X is black' is a formal proof that holds 'X is darker than any other colour'* (as in reasoning) and *'Andorra holds nuclear weapons' is a persuasive enough argument for accepting 'Spain must attack Andorra'* (as in argumentation) hold several differences:

- On logical and mathematical reasoning premises are consistent and premises can be defined in terms of closed concepts. That is, on the first sentence there is a complete ordering among colours to define the concept 'darkness', and all parties share this ordering.
- Reasoning takes place in a closed and fully defined context, there is no such thing as *uncertain* or *incomplete* information.
- On reasoning, conclusions are final. On the first sentence, the statement 'X is darker than any other colour' is valid and will always be valid, without accepting later amendment or retraction. However, on the second sentence, the statement 'Andorra holds nuclear weapons' can become invalid or be retracted when further information becomes available (such as 'Andorra has never bought Uranium') or when a new view-point becomes available (such as 'Spain must buy this weapons and attack France').
- Reasoning is completely objective, whereas argumentation is susceptible to subjective views. It drives us to the notion of audience as introduced by Perelman [25]

Proof is demonstration, whereas argumentation is persuasion. Thus, whereas the first sentence tries to demonstrate a fact, the second one tries to convince of a fact. And whereas the first will be accepted in all possible worlds, the second one can change in the presence of different view-points or new information.

In the case of reasoning, correctness is a key factor, argumentation does not care about correctness, as long as the argument is persuasive. A simple analogy between argumentation and politics can be made, what a politician says does not need to be correct or true, as long as it succeeds in persuading us.

The first main motivation to apply argumentation theory to AI arises from the issues and problems that reasoning theories present when having to deal with incomplete or uncertain information, and in general non-monotonic logics [20], where conclusions drawn in the presence of incomplete information can be withdrawn when additional information becomes

available. Thus argumentation, at this stage, is studied as a way of using reasoning theory on the presence of non-monotonic logics, rather than as field independent from reasoning theory.

It was not until the work of Dung [9] when exploitation of argumentation models starts to be seen as independent from reasoning models. Two important ideas are put forward on this work.

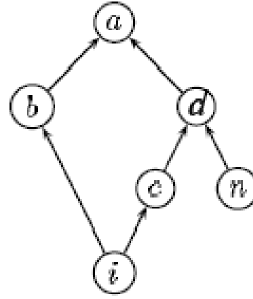

Figure 4.2: Argumentation Framework Example

Argumentation is modeled as a set of atomic arguments (x,y,z,...) and a binary relation between them, interpreted as argument x attacks argument y. Thus, an *Argumentation Framework* is composed by a pair of elements: a set of arguments, and a set of attack relations between arguments, that are, at the same time, pairs of arguments. Formally, an *Argumentation Framework* $AR = \langle Arg, Att \rangle$. The framework shown as example on *Figure 4.2* is denoted by $Example_Framework = \langle\langle a, b, c, d, i, n\rangle\langle(i, b), (i, c), (b, a), (c, d), (n, d)\rangle\rangle$

The subset of the full set of arguments known as justified arguments can be modelled via extension-based semantics, defining various properties of such subset on an argumentation framework. These properties vary from liberal big subsets (credulous) to extremely restrictive and small subsets (sceptic). Regarding this properties, Dung's initial property conflict freeness that is, there is no argument in the subset attacking other argument in the subset (correspondent to credulous semantics) is extended by Dung and other authors defining properties such as:

- Grounded extension: Correspondent to skeptical semantics. Can be defined as the smallest complete extension (one that is preferred and stable).
- Preferred extension: Maximum admissible set of arguments in the argumentation framework. That is, if S is a preferred extension, there is no admissible set of arguments U that complies with this property: 'The set of arguments $S \cup U$ is an admissible extension'.
- A set of arguments is admissible if it complies with the following property: 'If a given argument in the set A attacks another argument in the set B, then B attacks A'
- Stable extension: Where exists a conflict free set of arguments S, where every argument in S attacks all the arguments that are not in S.
- A complete extension is the set of arguments that is admissible (as defined before) and every argument which is acceptable, according to the set, belongs to the set.

- Semi-stable extension: The set of arguments S is a semi-stable extension if S if a set of arguments compliant with complete extension properties and is maximal (that is, there is no other complete extension S' that contains S).

- Prudent extension: Re-defining basic Dung's extensions (such as Grounded, Preferred, complete and stable, among others). These extensions are based in taking into account both direct and indirect conflicts (instead of being limited only to direct conflicts like in Dung's extensions) adding the restriction that two given arguments a and b cannot belong to the same extension, if a attacks indirectly b.

- An indirect conflict is defined in the following terms. Taking the example shown on *Figure 4.2*, one can see that argument a is attacked by argument b. b is attacked by argument i , so the set [a,i,n] is a stable extension, however, a prudent approach will not consider this set as extension, because even though argument i is defending a, it is also, indirectly, attacking a.

4.2 Description of the example

This section presents a detailed description of a scenario that will be used in later sections as example of the concepts in the framework proposal. The example is inspired on a real scenario presented in [14].

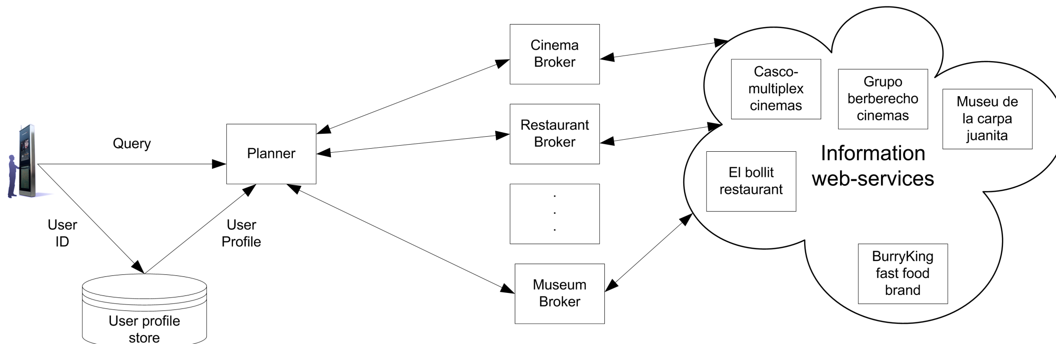

Figure 4.3: Scenario architecture Description

A personalized recommendation tool for entertainment and cultural activities is the basis for the scenario used in the example. The recommendation is offered via interactive community displays (ICDs) , multimedia information points offering interactive services in public areas. This scenario brings city services closer to residents and tourists by interconnecting people, service providers and locations. The services and information provided, and how user information is stored, processed and distributed, are all subjected to various municipal, national and European regulations. This means, for instance, that the system will not suggest adult movies to underage users.

As depicted in *Figure 4.3* the starting point of the scenario in this example is a user interacting with the systems interface (the ICD) in search for entertainment and cultural activities around the city. The user identifies herself. Then the system accesses the user profile from a remote repository. This profile will contain, among other elements, a model of user's interests, allowing the system to decide which touristic activities are more adequate

for the user. In the next step, the system composes an initial recommendation, using ratings and reviews about restaurants, cinemas, shops and amusement sites, and considering user preferences and location. Proposed activities are presented located on a map with basic information such as a brief description, address and pictures. When the user requests information of a venue, for instance a cinema, the system shows its detailed description, such as movies, sessions and prices. Moreover, the system informs on the required transportation (such as bus or underground) to reach the venue. Moreover, if it is lunch time, the system can even suggest a restaurant along the way, composing information from different web-services (cinemas, restaurants, maps and buses).

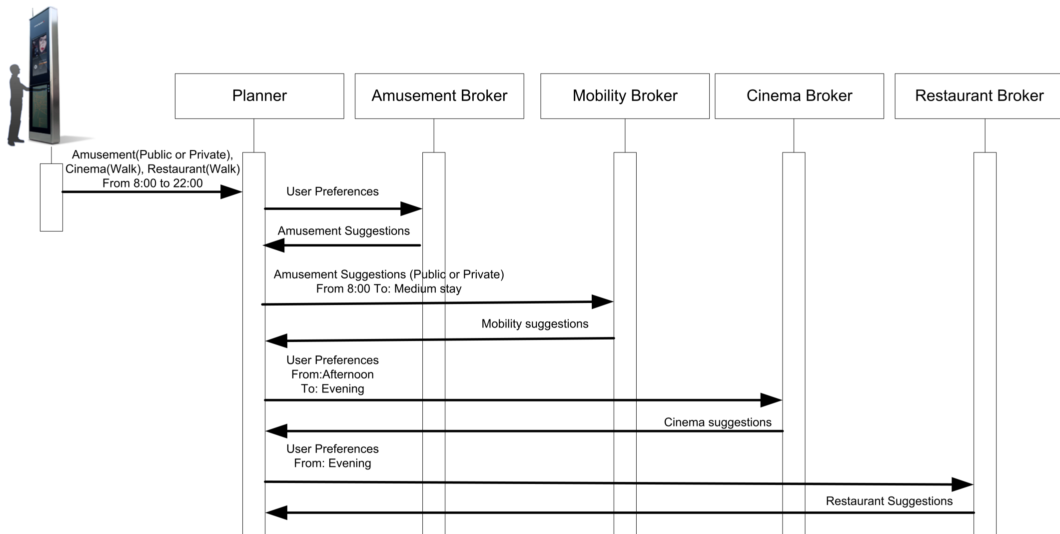

Figure 4.4: Workflow Diagram

4.2.1 The scenario: activity recommendation

In the presented scenario the user requests a tour via the graphical interface on one of the ICDs. The petition reaches the *planner* agent that contacts *broker* agents to gather information about the touristic activities that are available on the city. For efficiency, and taking into account that the planner agent is aware of the user's profile (there is one planner agent per user), one can consider that this information has already been pre-filtered, so *brokers* provide only information that matches the user's interests.

Each contacted *broker* returns a set of activities which are gathered and processed by the *planner* in order to obtain a set of conflict-free tours. Then, the set of tours is pondered and ordered based on the user's preferences, and the tour (or tours) with highest scores, returned to user.

In this example, the user requests a tour including an activity in an amusement park, a movie in a cinema near his hotel, and a dinner in a restaurant near the cinema. The user wants to start the tour early (at 8.00 h) and end it early too, at 22.00 h the latest.

The user wants to stay in the amusement park a medium amount of time (that is, enough time to see most of it). Smaller parks will require less time than bigger parks to be visited. One can consider this information is provided by *broker agents*, because they

Table 4.1: Activities proposed by each broker

Broker	Activities
Amusement sites	{*Port_Aventura*, *Tibidabo*, *Aquarium*}
Mobility	{*Port_Aventura_Private_T*, *Tibidabo_Private_T*, *Aquarium_Private_T*, *Port_Aventura_Public_T*, *Tibidabo_Public_T*, *Aquarium_Public_T*}
Cinema	{*The_hobgoblin_Soon*, *The_hobgoblin_Medium*, *The_hobgoblin_Late*, *P._movie_Soon*, *P._movie_Medium*, *P._movie_Late*}
Restaurant	{*Le_Remanguille_Soon*, *Le_Remanguille_Late*, *Casa_Pepe_Soon*, *Casa_Pepe_Medium*, *Casa_Pepe_Late*, *BurryKing_VSoon*, *BurryKing_Soon*, *BurryKing_Medium*, *BurryKing_Late*, *BurryKing_VLate*}

can gather it from each park information service. Suggested transport systems (to move between activities) have fixed times, being typically shorter in private transport than in public one, due to service frequencies and commuting times. The same stands for movies: it can be assumed that the *cinema broker* knows exactly the movie's start and end times by extracting this information from available cinema information services. In the case of restaurants, good restaurants tend to provide a slow service, and having lunch in them will take longer. However, fast food restaurants provide a nearly immediate service at the expense of a lower quality food.

4.2.2 Workflow

This section explains how the planner agent interacts with other components in the system in order to fulfill an user's request (depicted in *Figure 4.4*). First of all, the planner contacts with the *amusement broker* in charge of amusement sites and gets proposals to visit 'Tibidabo', 'Port Aventura' and 'Aquarium'. Then, asks the *mobility broker* for ways to get to each of the amusement sites. User contemplates public transport (best combination to be provided) or car. At this point, the *mobility broker* is asked for proposals starting at 8.00 h. The *mobility broker* is responsible of providing ways to go from the amusement sites back to user's hotel as well. Next, proposals from the *cinema broker* are retrieved. Such proposals include 'The hobgoblin', a 6 hour long movie and 'Programmer movie' a light-weight comedy. Finally, proposals from *restaurant broker* are also retrieved, including a French restaurant ('Le remangillue'), a Spanish 'tapas' bar ('casa Pepe') and a fast food brand franchise ('BurryKing'). Proposals provided by the *restaurant broker* must end, as much, at 22.00h.

4.2.3 Activities

The *Mobility broker* is able to generate a set of proposal activities from a single amusement activity. For instance, in this example, given the amusement activity 'Tibidabo', the *mobility broker* is able to generate two activities, that is: *Tibidabo_Private_T* (go to amusement activity 'Tibidabo' by private transport, and come back by private transport) and *Tibidabo_Public_T*. Thus, *Port_Aventura_Private* denotes the set formed by two activities {*Port_Aventura*, *Port_Aventura_Private_T*}. The same stands for other combinations of mobility-amusement activities. As a summary, proposed activities are listed on *Table 4.1*.

4.2.4 Resources

Each activity has a set of resources associated, typically, money and time each activity will take. For instance, going to the cinema to see 'The hobgoblin' can take 5 euros and 14 time slots. This information is provided by *broker* agents, along with the activity proposal. For simplicity, in this example it is assumed the user has an unlimited budget, so no money

Table 4.2: Resources of each alternative

Activity	Resources	Activity	Resources
$Port_Aventura_Private$	$\{8.00 - 18.00\}$	$Tibidabo_Private$	$\{8.00 - 16.30\}$
$Aquarium_Private$	$\{8.00 - 14.00\}$	$Port_Aventura_Public$	$\{8.00 - 19.00\}$
$Tibidabo_Public$	$\{8.00 - 17.30\}$	$Aquarium_Public$	$\{8.00 - 15.30\}$
$The_hobgoblin_Soon$	$\{16.00 - 19.30\}$	$The_hobgoblin_Medium$	$\{17.00 - 20.30\}$
$The_hobgoblin_Late$	$\{18.30 - 22.00\}$	$P._movie_Soon$	$\{17.00 - 18.30\}$
$P._movie_Medmium$	$\{18.30 - 20.00\}$	$P._movie_Late$	$\{19.30 - 21.00\}$
$Le_Remanguille_Soon$	$\{19.30 - 21.30\}$	$Le_Remanguille_Late$	$\{20.00 - 22.00\}$
$Casa_Pepe_Soon$	$\{20.00 - 21.00\}$	$Casa_Pepe_Medium$	$\{20.30 - 21.30\}$
$Casa_Pepe_Late$	$\{21.00 - 22.00\}$	$BurryKing_VSoon$	$\{19.30 - 20.00\}$
$BurryKing_Soon$	$\{20.00 - 21.00\}$	$BurryKing_Medium$	$\{20.30 - 21.30\}$
$BurryKing_Late$	$\{21.00 - 21.30\}$	$BurryKing_VLate$	$\{21.30 - 22.00\}$

resources are assigned to activities. As a summary, resources associated to each proposed activity (or sets of activities in the case of amusement-transport pairs) are listed on *Table 4.2*.

Resources on this example are time slots of twenty-five minutes (e.g., 9.30 denotes the time slot going from 9.30AM to 9.59AM). For any two timestamps 'X' and 'Y' , '$\{X - Y\}$' denotes the set of time slots between the two timestamps. (e.g., '$9.30 - 10.30 = \{9.30, 10.00, 10.30\}$, in other words, from 9.30AM to 11.59AM)'

4.3 Formal definition of the framework

This section presents a formal definition of the framework. It explains the basic concepts of the framework and their properties, so that these concepts can be used in subsequent sections.

Definition 4.1 *An action is represented by a single positive literal* a

$$Action = a$$

This literal will have the value true *if the action has been performed by the agent, and* false *otherwise. Let* performed(a) *be a function that, given an action, returns true if this action has been performed by an agent. Let* $\mathcal{L}_{action}$ *be the set of all actions of the domain.*

Remark 4.1 *It is assumed agents have means to know when they have successfully performed an action.*

Example 4.1 *Following the scenario presented, actions are the touristic activities proposed*

- $Action1 = Port_Aventura_Private_T$
- $Action2 = BurryKing_Soon$

Definition 4.2 *A goal is represented by a single positive literal* g

$$Goal = g$$

Remark 4.2 *This literal will have the value* true *if the goal has been fulfilled, and* false *otherwise. Let fulfilled(g) be a function which, given a goal, returns its truth value. Let* $\mathcal{L}_{goal}$ *be the set of all goals in the domain. When all these goals are fulfilled, the objective the global plan represents is reached.*

Example 4.2 *Following the scenario presented, the goals are the abstract activities requested by user, as seen on Figure 4.4.*

- $Goal1 = Amusement$
- $Goal2 = Cinema$
- $Goal3 = Restaurant$
- $\mathcal{L}_{goal} = \{Amusement, Cinema, Restaurant\}$
- $fulfilled(Amusement) \wedge fulfilled(Cinema) \wedge fulfilled(Restaurant) \rightarrow$ *user request satisfied*

Table 4.3: Existing alternatives

Code	Alternative
A1	$Port_Aventura_Public = amusement \leftarrow \{Port_Aventura, Port_Aventura_Public_T\}$
A2	$Port_Aventura_Private = amusement \leftarrow \{Port_Aventura, Port_Aventura_Private_T\}$
A3	$Tibidabo_Public = amusement \leftarrow \{Tibidabo, Tibidabo_Public_T\}$
A4	$Tibidabo_Private = amusement \leftarrow \{Tibidabo, Tibidabo_Private_T\}$
A5	$Aquarium_Public = amusement \leftarrow \{Aquarium, Aquarium_Public_T\}$
A6	$Aquarium_Private = amusement \leftarrow \{Aquarium, Aquarium_Private_T\}$
M1	$The_hobgoblin_Soon = cinema \leftarrow \{The_hobgoblin_Soon\}$
M2	$The_hobgoblin_Medium = cinema \leftarrow \{The_hobgoblin_Medium\}$
M3	$The_hobgoblin_Late = cinema \leftarrow \{The_hobgoblin_Late\}$
M4	$P._movie_Soon = cinema \leftarrow \{P._movie_Soon\}$
M5	$P._movie_Medmium = cinema \leftarrow \{P._movie_Medmium\}$
M6	$P._movie_Late = cinema \leftarrow \{P._movie_Late\}$
R1	$Le_Remanguille_Soon = restaurant \leftarrow \{Le_Remanguille_Soon\}$
R2	$Le_Remanguille_Late = restaurant \leftarrow \{Le_Remanguille_Late\}$
R3	$Casa_Pepe_Soon = restaurant \leftarrow \{Casa_Pepe_Soon\}$
R4	$Casa_Pepe_Medium = restaurant \leftarrow \{Casa_Pepe_Medium\}$
R5	$Casa_Pepe_Late = restaurant \leftarrow \{Casa_Pepe_Late\}$
R6	$BurryKing_VSoon = restaurant \leftarrow \{BurryKing_VSoon\}$
R7	$BurryKing_Soon = restaurant \leftarrow \{BurryKing_Soon\}$
R8	$BurryKing_Medium = restaurant \leftarrow \{BurryKing_Medium\}$
R9	$BurryKing_Late = restaurant \leftarrow \{BurryKing_Late\}$
R10	$BurryKing_VLate = restaurant \leftarrow \{BurryKing_VLate\}$

An alternative for performing a goal is represented by a rule, that is, an ordered list of positive literals, with at least one literal.

Definition 4.3 *An alternative has the following form:*

$$l_1 \wedge l_2 \wedge \cdots l_n \ \leftarrow l_{n+1} \wedge l_{n+2} \wedge \cdots l_{n+m} \ : n \geq 1 \wedge m \geq 1$$

Where the body of the rule is a set of actions:

$$l_i \in \mathcal{L}_{action} \quad n+1 \geq i \geq n+m$$

The tail of the rule can be a goal or a set of goals put together via conjunctions. For simplicity, in most definitions (indeed, in all of them but complemented alternative definition), it is assumed the tail of the rule to be a single goal.

$$l_i \in \mathcal{L}_{goal} \quad 1 \geq i \geq n$$

Example 4.3 *For instance, the following alternative Aquarium_Public is an alternative for fulfilling the goal Amusement via the actions Aquarium and Aquarium_Public_T: Aquarium_Public = Amusement ← Aquarium, Aquarium_Public_T. To summarize, table Table 4.3 shows all the alternatives that exist in the example introduced.*

Proposition 4.1 *Let* $\mathcal{L}_{alternative}$ *be the set of all alternatives. Let* Actions(alt) *be a function that, given an alternative, returns the set of actions in the alternative. Let* Goal(alt) *be a function that, given an alternative, returns the set of goals in the alternative.*

For any given alternative alt, the following condition holds:

$$(Actions(alt) = \bigcup_{i=1...m} l_i) \wedge (Goal(Alternative) = l) :$$

$$alt \text{ is of the form } (l \leftarrow l_1 \wedge \cdots l_n \ \ : n \geq 1)$$

Example 4.4 *Following the example:*

- $Goal(Casa_Pepe_Soon) = restaurant$
- $Actions(Tibidabo_Public) = \{Tibidabo, Tibidabo_Public_T\}$

Taking into account that alternatives are defined as rules, it can be deduced: if the actions in the body of the rule are true, then the goal in the head of the rule is true. Then, using functions *fullfilled(goal)* and *performed(action)* as defined before, the following statement can be derived:

Definition 4.4 *Given an alternative of the form* $l \leftarrow l1 \wedge \cdots l_n$

$$fulfilled(l) \leftarrow performed(l_1) \wedge \cdots performed(l_n) \ \ : n \geq 1$$

Proposition 4.2 *Function performed can be extended to alternatives. Informally, performing an alternative means performing all actions in the alternative. For any alternative* alt*:*

$$performed(alt) = \bigwedge_{act \in actions(alt)} performed(act)$$

Example 4.5 *Following the example:*
$performed(Aquarium_Public) =$
$performed(Aquarium) \wedge performed(Aquarium_Public_T)$ ***AND***
$fulfilled(amusement) \leftarrow performed(Aquarium_Public)$ ***THEN***
$fulfilled(amusement) \leftarrow performed(Aquarium) \wedge performed(Aquarium_Public_T)$

Remark 4.3 *One can deduce that fulfilling a goal is equivalent to performing an alternative with the goal in the head of the rule. Formally, for any goal* g*:*

$$fulfilled(g) =$$
$$\exists alt \in \mathcal{L}_{alternative} : (Goal(alt) = g) \wedge performed(alt))$$

In order to fulfill a given goal, several alternatives might be available. That is, several alternatives might share the same goal. Following the example, both $Le_Remanguille_Late$ and $BurryKing_Medium$ are alternatives for fulfilling the goal $restaurant$. The set of alternatives in the plan sharing the same goal is known as *action plan.* Therefore, an action plan is just a set, where every element in the set is an alternative, and where every alternative has the same goal.

Definition 4.5 *Let S be a set of alternatives. S is an action plan iff:*

$$\forall alt1, alt2 \in S \ goal(alt1) \ = \ goal(alt2)$$

Example 4.6 *Following the example, it can be stated that:*

- $\{Port_Aventura_Public, Aquarium_Private\}$ *is an action plan, because:*
  - $goal(Port_Aventura_Public) = amusement$
  - $goal(Aquarium_Private) = amusement$
  - $goal(Port_Aventura_Public) = goal(Aquarium_Private)$
- $\{P._movie_Soon, Casa_Pepe_Soon\}$ *is not an action plan, because:*
  - $goal(P._movie_Soon) = cinema$
  - $goal(Casa_Pepe_Soon) = restaurant$
  - $goal(P._movie_Soon) \neq goal(Casa_Pepe_Soon)$

Proposition 4.3 *Function performed can be extended to accept action plans. Informally, performing an action plan means performing at least, one of the alternatives. Formally, for any action plan* ap*:*

$$performed(ap) = \bigvee_{alt \in ap} performed(alt)$$

Example 4.7 *Therefore, following the example:*
$performed(\{Port_Aventura_Public, Aquarium_Private\}) =$
$(performed(Port_Aventura_Public) \cup performed(Aquarium_Private))$

In order to fulfill the common goal of the action plan, at least one of the alternatives in the plan (and ideally only one) should be performed. Thus, performing the action plan will fulfill the goal of the action plan.

Definition 4.6 *Let goal(ap) be a function which, given an action plan, returns the common goal of the alternatives in the action plan. For any action plan* ap*:*

$$performed(ap) = fulfilled(goal(ap))$$

Example 4.8 *Following the example:*
$performed(\{Port_Aventura_Public, Aquarium_Private\}) = fulfilled(amusement)$

Definition 4.7 Resources *represent the set of resources that are required and consumed, in order to execute a given action. Therefore, resources can be represented as positive literals that are associated to an action via a rule, known as* resource association rule*, of the following form:*

$$resource\ association = a \leftarrow r_1 \wedge r_2 \wedge \cdots r_n \ \ : n \geq 1$$

Where the elements in the body of the rule are resources, and the element in the head is an action.

Let $\mathcal{L}_{resource}$ denote the set of all resources, and $\mathcal{L}_{resource_association}$ denote the set of all *resource association rules*. Let $resources(act)$ be a function that, given an action, looks for a *resource association rule* that has the provided action in the head of the rule and returns the set of resources in the body of this rule.

Using *Definition 3* the function *resources(act)* can be extended to accept alternatives as parameters. Informally, the set of resources used by an alternative is the union of the set of resources used by every action in the alternative.

Proposition 4.4 *Formally, given an alternative alt:*

$$resources(alt) = \bigcup_{act \in actions(alt)} resources(act)$$

Example 4.9 *Following the example, given the alternative* $Tibidabo_Private$ *such that* $actions(Tibidabo_Private) = \{Tibidabo, Tibidabo_Private_T\}$ *it can be stated:* $resources(Tibidabo) = resources(Tibidabo) \cup resources(Tibidabo_Private_T)$*.*

Please, notice that, for simplicity, *Table 4.2* shows the resources associated to each alternative, rather than the resources associated to each action.

Definition 4.8 *A local_plan is a set of elements in the agent's knowledge structure. It has the following form:*

$$local_plan = \{g, ap, r, ra\}$$

Where:

- *g is the goal pursued by the agent.* $g \in \mathcal{L}_{goal}$
- *ap is a set of alternatives, such that all of them fulfill g, i.e. 'goal(ap) = g'. Therefore, ap is an action plan.*
- *r is the set of all resources, i.e.* '$r = \mathcal{L}_{resource}$'
- *ra is the set of all* resources association rules*, i.e.* '$ra = \mathcal{L}_{resource_association}$'

An agent will have a local plan for each goal it pursues. In this framework, typically, an agent will pursue only one goal, and therefore, will have a single local plan.

Example 4.10 *Following the example, a valid agent could be the one formed by the set* $\{amusement, \{M1, M2, M3, M4, M5, M6\}, \mathcal{L}_{resource}, \mathcal{L}_{resource_association}\}$*. However, as it will be remarked later, the example has some requirements (e.g. there is only one planning agent) that make agents have some special values on the data structures that define the agents.*

Definition 4.9 *A set of local plans is a global plan if the following conditions hold:*

1. *All the local plans have different goals.*
2. *The union of the goals of all the local plans is* $\mathcal{L}_{goal}$*.*

Let global_plan be the global plan:

$$\forall p1, p2 \in global_plan \quad p1 \neq p2 \Rightarrow goal(p1) \neq goal(p2)$$

$$\left(\bigcup_{lp \in global_plan} goal(lp) \right) = \mathcal{L}_{goal}$$

Example 4.11 *Following the example, let these plans exist:*

- $p1 = \{Tibidabo_Public, BurryKing_Medium\}$
- $p2 = \{Tibidabo_Public, The_hobgoblin_Soon, BurryKing_Medium\}$
- $p3 = \{Tibidabo_Public, The_hobgoblin_Soon, Casa_Pepe_Medium, BurryKing_Medium\}$

Table 4.4: Utilities of each activity

Activity	Resources	Activity	Resources
Port_Aventura	20	*Tibidabo*	10
Aquarium	5	*Port_Aventura_Public_T*	−5
Tibidabo_Public_T	0	*Aquarium_Public_T*	2
Port_Aventura_Private_T	0	*Tibidabo_Private_T*	5
Aquarium_Private_T	−1	*The_hobgoblin_Soon*	30
The_hobgoblin_Medium	19	*The_hobgoblin_Late*	1
P._movie_Soon	15	*P._movie_Medmium*	8
P._movie_Late	3	*Le_Remanguille_Soon*	20
Le_Remanguille_Late	9	*Casa_Pepe_Soon*	15
Casa_Pepe_Medium	7	*Casa_Pepe_Late*	2
BurryKing_VSoon	5	*BurryKing_Soon*	2
BurryKing_Medium	0	*BurryKing_Late*	−2
BurryKing_VLate	−5		

The plan p3 is not a global plan, because two elements in the set fulfill the same goal '$goal(Casa_Pepe_Medium) = restaurant = goal(BurryKing_Medium)$'. The plan p1 is not a global plan, because the union of the goals fulfilled by the alternatives in the set, is not the set of all goals in the system '$((goal(Tibidabo_Public) \cup goal(BurryKing_Medium)) = \{amusement, restaurant\}) \neq \{amusement, cinema, restaurant\}$'. The plan p2 is a global plan, because **(**$goal(Tibidabo_Public) \neq goal(The_hobgoblin_Soon) \neq goal(BurryKing_Medium)$**) and (**$goal(Tibidabo_Public) \cup goal(The_hobgoblin_Soon) \cup goal(BurryKing_Medium) = \{amusement, cinema, restaurant\}$**)**.

As it has been stated before (*Definition 5*) several alternatives to fulfill a given goal might be available. However, on efficiency's sake, only one of them should be performed. Therefore, a procedure allowing agents to decide which alternative to perform (among the set of available ones) in order to fulfill a goal is required. This framework provides such means via preferences. *Preferences* represent the set of preferences of the agent regarding the actions in the plan. Therefore, the concept is not defined in the plan, but on agent's knowledge bases, and (unlike the plan) can vary from agent to agent.

Definition 4.10 *A* preference *is just a pair of elements. One of them is an action of the plan, the other is the value assigned to this action by this agent.*

$$preference = \langle a, i \rangle : a \in \mathcal{L}_{action} \wedge i \in \boldsymbol{Z}$$

Remark 4.4 *Notice that i is an integer that will have a positive value, if the action goes with the agent's preferences, a negative value, if the action goes against the agent's preferences, and 0 value, if the agent has no preferences towards this action.*

In the example, a preference is just a way to model the feelings of the user against a given touristic activity.

A set of preferences containing no repeated actions (that is, each action has an unique preference value) and containing a value for all the actions (that is, all the elements in $\mathcal{L}_{action}$) is known as *preference_set*.

Table 4.4 shows a valid preference set for the example.

Proposition 4.5 *Let* utility(act) *denote a function which, given an action, returns the utility value of this action according to the preference_set of the agent.*

$$(utility(act) = i) \text{ such that } (\langle act, i \rangle \in preference_set)$$

Example 4.12 *In the example, and as shown in Table 4.4, $utility(P._movie_Late) = 3$ and $utility(BurryKing_VLate) = -5$*

In order to be useful to choose among a set of available alternatives to fulfill a goal, function *utility(act)* must be extended to accept alternatives, instead of actions, as parameter. Intuitively, one can see that, the utility of an alternative is just the sum of the utilities of the actions involved in the alternative.

Definition 4.11 *Let alt be an alternative*

$$Utility(alt) = \sum_{act \in alt} utility(act)$$

Example 4.13 *In this example, given the alternative $Port_Aventura_Private$ as defined on Table 4.3, and using the utilities defined in Table 4.4, it can be stated:*

$Utility(Port_Aventura_Private) = Utility(Port_Aventura)+$
$Utility(Port_Aventura_Private_T) = 20 + (-5) = 15$

This concept allows to define an order among alternatives. Informally, if a given alternative $alt1$ has a higher utility value than another alternative $alt2$, then it can be stated that $alt1 \geq alt2$ and therefore, $alt1$ is preferred over $alt2$ by the agent. Then, if both alternatives are available for fulfilling the same goal, the agent will be able to choose only one of them.

Definition 4.12 *Formally, for any two alternatives $alt1$ and $alt2$, and for any goal $g1$, if $goal(alt1) = g1 = goal(alt2)$ then:*

$$alt1 \geq alt2 \;\; iff \;\; utility(alt1) \geq utility(alt2)$$

$$alt1 \; is \; preferred \; over \; alt2 \; for \; fulfilling \; goal \; g1$$

Example 4.14 *Following the example, taking alternatives $Port_Aventura_Public$ and $Aquarium_Private$, it can be stated that*
$Port_Aventura_Public$ is preferred over $Aquarium_Private$ for fulfilling the goal amusement because:

- $goal(Port_Aventura_Public) = goal(Aquarium_Private) = amusement$
- $utility(Port_Aventura_Public) \geq utility(Aquarium_Private)$
  - $utility(Port_Aventura_Public) = 15$
  - $utility(Aquarium_Private) = 4$
  - $15 \geq 4$

Proposition 4.6 *Let $Max(altSet)$ be a function that given an ordered set of alternatives returns the alternative with the highest utility in the set. Formally, for any set of alternatives altSet:*

$$Max(altSet) = alt1 : alt1 \in altSet \; \wedge$$
$$\neg(\exists alt2 \in altSet : utility(alt1) < utility(alt2))$$

At this point, the representation of an agent's local plan is already available, as well as means to enable the agent to rationally pick up an alternative among the set of available ones.

Definition 4.13 *An agent is formed by a local plan and the set of preferences the agent has towards the actions in the domain.*

$$agent = \{local_plan, preference_set\}$$

Table 4.5: Agents in the system

Agent	Goal	Action Plan	Utilities
Planner	*None*	{ }	*SeeTable* 4.4
Amusement + MobilityBrokers	*amusement*	{*A*1, *A*2, *A*3, *A*4, *A*5, *A*6}	*None*
CinemaBroker	*cinema*	{*M*1, *M*2, *M*3, *M*4, *M*5, *M*6}	*None*
RestaurantBroker	*restaurant*	{*R*1, *R*2, *R*3, *R*4, *R*5, *R*6, *R*7, *R*8, *R*9, *R*10}	*None*

The system is formed by a set of agents, each of them with a different *local_plan*, in a combination of local plans that fit *global_plan* definition.

Definition 4.14 *Let plan(agent) be a function that, given an agent, returns its local plan. Given the following global plan:*

$$gp = \{local_plan_1\ , local_plan_2\ , \ldots\ , local_plan_n\} : n \geq 1$$

Then, a framework is of the following form:

$$framework = \{agent_1\ , agent_2,\ \ldots,\ agent_n\} : n \geq 1$$

Where:

$$plan(agent_i) = local_plan_j\ \wedge i = j\ \forall i \geq n\ \forall j \geq n$$

In the example presented, *planner agent* has an empty set of plans (it receives plan proposals from brokers) and a set of utility values that model user's interests regarding the activities proposed by brokers. Broker agents have plans to fulfill the objective they pursue. Thus, *Cinema Broker* has plans for fulfilling objective *Cinema*, *Restaurant Broker* has plans for fulfilling objective *Restaurant*, and the entity composed by the collaboration of both *Amusement Broker* and *Mobility Broker* [2] plans for fulfilling objective *Amusement*. The set of resources available can be inferred from user's request (in this case, the set of time slots between 8.00 and 22.00). Resource association rules reside on broker agents and are provided to the planner along with the activity suggestions. The full summary of the agents in the system can be seen in *Table 4.5*.

4.4 Conflicting actions

The process of detecting conflicting actions is key in our approach, as it enables an agent to detect non-acceptable global plan proposals when negotiating which global plan is going to be executed by the set of agents. The conflict detection process makes use of Argumentation Theory, and it is based in the concept of *action collision*. Bearing in mind that resources are limited and consumed when actions are executed (thus, they cannot be used any longer) it can be stated that two actions collide when they make use of the same resource. This is because if one of the actions uses the resource the other will not be capable of execution, as it requires to consume the resource in order to execute, and the resource is not available anymore.

Definition 4.15 *Let collide(act1,act2) be a function which, given two actions, returns true if they collide, and false otherwise. Formally, using Definition 6, for any two actions act1 and act2:*

$$(collide(act1, act2) = true)\quad iff$$

$$(resources(act1) \cap resources(act2) \neq \emptyset)$$

[2] remember amusement activities are complemented with mobility options in a single alternative as explained on *Section 4.2*

Example 4.15 *Following the example presented it can be stated that* $collide(P._movie_Late, Le_Remanguille_Soon) = true$ *because* $(resources(P._movie_Late) \cap resources(Le_Remanguille_Soon) = \{19.30 - 21.00\} \neq \emptyset$ *and* $collide(The_hobgoblin_Soon, BurryKing_VLate) = false$ *because* $(resources(The_hobgoblin_Soon) \cap resources(BurryKing_VLate) = \emptyset$

At this point, an argumentation framework can be adapted to this planning framework. According to Dung's [9] definition an argumentation framework is a pair of sets: one set is a set of arguments, the other is a set of attack relations between arguments.

Definition 4.16 *Let arguments and attack_relations be sets of arguments and attack relations respectively. An argumentation framework is a pair of sets of the form:*

$$\langle arguments, attack_relations \rangle$$

The set of arguments is the set of all actions in the domain, i.e. $arguments = \mathcal{L}_{action}$. *The attack relation is given by the collide function, as seen on Definition 12. For any two actions act1 and act2*

$$attack(act1, act2)\ iff\ (collide(act1, act2) = true)$$

$$Notice\ that\ : attack(a1, a2)\ \ iff\ \ attack(a2, a1)$$

Once the argumentation framework has been defined and adapted properly, the concept of conflict-free set of arguments (in this case, conflict-free set of actions) can be defined. Following Dung's [9] work, a set of arguments can be said to be conflict-free if there are no arguments a and b in the set, such that a attacks b.

Definition 4.17 *Formally, for any set of arguments Set*

$$ConflictFree(Set) = true\ iff\ \neg(\exists a, b \in Set \mid attack(a, b))$$

Using actions as arguments in the set and via the adapted attack relation, one can deduce that:

$$ConflictFree(Set) = true\ iff$$

$$\neg(\exists act1, act2 \in Set \mid collide(act1, act2))$$

The conflict-free condition is easily applicable to a special set of actions, that is, the alternative. If local plans have been properly defined, so that in every local plan there are no resource conflicts between actions in the same alternative, all alternatives in all agent's action plans comply with conflict-free condition.

4.5 Building the global plan

Local alternatives are enough to satisfy agent's local goals. However, in order to reach the objective that the global plan represents one has to execute several alternatives, linked to different goals and thus to different agents, in parallel.

In this work, the concept of a set of alternatives that are to be executed together is known as *complemented alternative*. This *complemented alternative* can be seen as an alternative that has these two properties. First, the actions of the *complemented alternative* are the union of the actions of the alternatives that are complemented. Second, the goals of the *complemented alternative* are the union of the goals of the alternatives that are complemented. Thus, *complemented alternatives* are the only multi-goal alternatives, that is, the only to show more than one element on the head of the rule according to *Definition 3*

Definition 4.18 *Given two alternatives alt_1 and alt_2, and a function complement which, given a set of alternatives, returns the resulting complemented alternative, the result of the function is an alternative of the following form:*

$$complement(alt_1, alt_2) =$$

$$(\bigwedge_{g1 \in goal(alt1)} g1) \wedge (\bigwedge_{g2 \in goal(alt2)} g2)$$

$$\leftarrow (\bigwedge_{a1 \in actions(alt1)} a1) \wedge (\bigwedge_{a2 \in actions(alt2)} a2)$$

Then, the following properties can be derived:

Proposition 4.7

$$goal(complement(alt_1, alt_2)) = goal(alt_1) \cup goal(alt_2)$$

$$actions(complement(alt_1, alt_2)) =$$

$$actions(alt_1) \cup actions(alt_2)$$

Example 4.16 *In the example presented, a complemented alternative is just a tour that fulfills several goals. For instance, the tour composed of going to 'Port Aventura' by public transport and going to see the movie 'The Hobgoblin' in the sooner session fulfills both 'amusement' and 'cinema' goals. Formally* $goal(complement(Port_Aventura_Public, The_hobgoblin_Soon)) = \{amusement, cinema\}$ *and* $actions(complement(Port_Aventura_Public, The_hobgoblin_Soon)) = \{Port_Aventura, Port_Aventura_Public_T, The_hobgoblin_Soon\}$. *Intutively, this makes sense, because it is clear that performing the actions 'go from hotel to Port Aventura by public transport', 'enjoy Port Aventura', 'go from Port Aventura to hotel by public transport' and 'see movie The Hobgoblin' will fulfill the requests 'amusement' and 'cinema'.*

Even though agents will execute only one of these alternatives, they need to have means to reason about the concept of *complemented alternative*, because the alternatives executed by other agents can affect the availability of one agent's alternative. Therefore, the need to see the set of alternatives to be executed as a whole is clear, as a *complemented alternative* acceptable by all agents will be the final result of the coordination process. And in order to be executable by several agents, and therefore acceptable as a coordination proposal, the *complemented alternative* must be conflict-free.

The example presented does not use the framework as a coordination tool for several planning agents to agree on a single *complemented alternative*, but as a planning tool that allows a single planning agent to build a conflict-free set of touristic activities based on the proposals sent by other agents.

Definition 4.19 *Formally, a given complemented alternative 'calt' is conflict-free if it satisfies the following property:*

$$\neg(\exists act_1, act_2 \in actions(calt) : collide(act_1, act_2))$$

Example 4.17 *Following the example, it can be stated that* $complement(Aquarium_Public, The_hobgoblin_Soon)$ *is conflict-free but* $complement(Port_Aventura_Public, The_hobgoblin_Soon)$ *is not conflict-free.*

If a complemented alternative is conflict-free all actions in the alternative can be executed, no matter resource restrictions. Therefore, the following proposition will be true:

Proposition 4.8 *For any two alternatives alt_1 and alt_2:*

$$performed(complement(alt_1, alt_2))\ iff\ performed(alt_1) \wedge performed(alt_2)$$

$$resources(complement(alt_1, alt_2)) = resources(alt_1) \cup resources(alt_2)$$

$$utility(complement(alt_1, alt_2)) = utility(alt_1) + utility(alt_2)$$

Definition 4.20 *A complemented alternative 'calt' is an alternative for the global plan iff :*

- *The goals of the alternative is the set of all goals. i.e. $goal(calt) = \mathcal{L}_{goal}$*
- *Is conflict-free. i.e. $ConflictFree(calt)$*

Example 4.18 *Following the example, it can be stated that:*

- $complement(Port_Aventura_Public, P._movie_Late, BurryKing_VLate)$ *is an alternative for the global plan because:*
  - $goal(Port_Aventura_Public) \cup goal(P._movie_Late)$ $\cup goal(BurryKing_VLate) = \mathcal{L}_{goal}$
  - $resources(Port_Aventura_Public) \cap resources(P._movie_Late) \cap$ $resources(BurryKing_VLate) = \emptyset$
- $complement(Aquarium_Public, Casa_Pepe_Soon, BurryKing_VLate)$ *is not an alternative for the global plan because:*
  - $goal(Aquarium_Public) \cup goal(Casa_Pepe_Soon) \cup goal(BurryKing_VLate) =$ $\{amusement, restaurant\} \neq \mathcal{L}_{goal}$
- $complement(Port_Aventura_Public, The_hobgoblin_Soon,\ BurryKing_VLate)$ *is not an alternative for the global plan because:*
  - $resources(Port_Aventura_Public) \cap resources(P._movie_Late) \cap$ $resources(BurryKing_VLate) \neq \emptyset$

Notice that *complement* function can also be applied to *action plans*. Informally, complementing two *action plans* is the set of alternatives that results of complementing the *alternatives* in the plans in a Cartesian product. Therefore, it can be seen that the result of the function is no more than a set of alternatives with a common goal (set of goals in this case), that is, an action plan.

Definition 4.21 *Given two action plans ap_1, ap_2:*

$$complement(ap_1, ap_2) =$$
$$\forall alt1 \in ap1 \ \ \forall alt2 \in ap2 \ \ complement(ap1, ap2)$$

Example 4.19 *Following the example, let the following action plans exists:*

- $ap_{amuse} = \{Port_Aventura_Public, Port_Aventura_Private\}$
- $ap_{rest} = \{BurryKing_VSoon, BurryKing_Soon\}$

Then, $complement(ap_{amuse}, ap_{rest}) = \{\{Port_Aventura_Public, BurryKing_VSoon\},$
$\{Port_Aventura_Public, BurryKing_Soon\},$
$\{Port_Aventura_Private, BurryKing_VSoon\},$
$\{Port_Aventura_Private, BurryKing_Soon\}\}$

Definition 4.22 *A complement of an action plan is a global plan proposal if all the alternatives in the set of alternatives are alternatives for the global plan as seen on Definition 17.*

Example 4.20 *Thus, the complemented alternative (*$complement(ap_{amuse}, ap_{rest})$*) used in the example of the previous definition is not a global plan proposal, because the alternatives in the plan fulfill the set of goals* $\{amusement, restaurant\}$ *and not the set of goals* $\{amusement, cinema, restaurant\}$*. However, complementing the action plan* $complement(ap_{amuse}, ap_{rest})$ *with the action plan* $ap_{cine} = \{The_hobgoblin_Soon, The_hobgoblin_Medium\}$ *will result in a global plan proposal.*

Algorithm 1 :Ponders alternatives in action plan using agent's $preference_set$ and taking into account other agent's preferences (provided by ponderation)

```
function ponder, in {ap: action_plan, po: ponderation}, out {ponderation}
BEGIN
ponderation := ∅
for all alt ∈ action_plan do
   ponderation := ponderation∪ < alt, po.value(alt) + utility(alt) >
end for
return ponderation
END
```

Algorithm 2 : Complements action plans contained in *received_proposals* with agent's action plan

---

```
function complement_proposals, in {received_proposals : set_of_proposal}, out {action_plan}
BEGIN
actual_proposal := own_action_plan
for all prop ∈ received_proposals do
   actual_proposal = complement(actual_proposal, plan(prop))
end for
return actual_proposal
END
```

Algorithm 3 : Removes from *actual_proposal* non conflict-free alternatives

---

```
function purge_proposals, in {ap: action_plan}, out {action_plan}
BEGIN
res := ∅
for all alt ∈ ap do
   if conflictFree(alt) then
      res = res ∪ alt
   end if
end for
return res
END
```

Algorithm 4 : Gets the alternatives that appear in all action plans in the set $own_action_plan \cup received_proposals$

```
function merge_proposals, in {received_proposals : set_of_proposal}, out {action_plan}
BEGIN
actual_proposal := own_action_plan
for all prop ∈ received_proposals do
   actual_proposal = actual_proposal ∩ plan(prop)
end for
return actual_proposal
END
```

Algorithm 5 : Ponders the set of alternatives in *actual_proposal* using the set of utility values provided by *received_proposals*. Then, gets the alternative with the maximum utility

```
function pick_max, in {actual_proposal : action_plan, received_proposals : set_of_proposal}, out
{alternative}
BEGIN
choice := null
choice_valoration := 0
for all alt ∈ actual_proposal do
   for all ponderation ∈ ponderation(received_proposals) do
      alt_valoration := alt_valoration + ponderation.value(alt)
      if choice_valoration < alt_valoration then
         choice := alt
         choice_valoration := alt_valoration
      end if
   end for
end for
return choice
END
```

4.6 Global utilities for a global plan

Just as *alternatives* fail to capture the higher objectives the *global plan* represents, and *complemented alternatives* are defined to fill this gap, *utilities* fail to capture the notion of coordination and global consensus between agents which this work aims to model. Therefore, the notion of *global utility* is defined to fill in this gap. The need comes from the fact that all the *alternatives* in the chosen *alternative for the global plan* are to be executed, so the framework must provide means for the agents to evaluate, as a whole, all the alternatives, not only the one the agent has to perform. Therefore, agents can infuse their own preferences in the alternatives the other agents will execute, as well as on the alternatives they have to execute. In the end, it will result in a global evaluation of the *alternative for the global plan*. Furthermore, this method is very flexible, because if an agent wants to evaluate only its own alternatives, all it has to do is ponder the alternatives of the other agents with a neutral value, that is 0. Intuitively, the global utility of an *alternative for the global plan* can be seen as the sum of the utility each agent has about this alternative, for each agent in the domain.

Table 4.6: Utility values for the alternatives per agent

Alternative/Agent	A_1	A_2	A_3	B_1	B_2	B_3
$Agent_1$	10	3	2	0	-13	0
$Agent_2$	-15	0	0	4	12	3

Table 4.7: Utility values of complemented alternatives for agents and global utility values

Agent	$Utility_{A1B1}$	$Utility_{A2B2}$	$Utility_{A3B3}$	*Choice*
$Agent_1$	10	-10	2	A1B1
$Agent_2$	-11	12	3	A2B2
Global	-1	2	5	A3B3

Definition 4.23 *Given an alternative for the global plan 'calt', a function AgentUtility which returns the utility of an alternative according to an agent's preference set and $\mathcal{L}_{agent}$, the set of all agents in the domain:*

$$global_utility(calt) = \sum_{ag \in \mathcal{L}_{agent}} AgentUtility(ag, calt)$$

In the example provided, the only agent that has a set of utility values for the touristic activities proposed is the *planner agent*. Thus, the example lacks a real use for global utilities, because local ones (the ones the *planner agent* has) are enough for fulfilling the needs of the system described in the example.

However, a system where multiple agents have different utility values can benefit from the ussage of a global utility function rather than a local one, as the following case shows.

Example 4.21 *Let $\{Agent_1, Agent_2\} = \mathcal{L}_{agent}$. Let $AX, BX \in \mathcal{L}_{alternative}$. Let the following global plan proposal exist: gpp = $\{complement(A1, B1), complement(A2, B2)$ $, complement(A3, B3)\}$. Let XY denote $Complement(XY)$*

If $Agent_1$ makes the choice, it will choose $A1B1$ with an utility for $Agent_1$ = 10, Utility for $Agent_2$ = -11 and Global utility = -1.

If $Agent_2$ makes the choice, it will choose $A2B2$ with an utility for $Agent_1$ = -10, Utility for $Agent_2$ = 12 and Global utility = 2.

If a consensus choice is made, agents will choose $A3B3$ with an utility for $Agent_1$ = 2, Utility for $Agent_2$ = 3 and Global utility = 5.

An agent has no access to other agent's *preference set*, therefore, only an agent can ponder an *alternative for the global plan* with its preferences. Therefore, in order to build this global utility function the *global plan proposal* must be exchanged over all the agents in the system.

4.7 Coordination protocol

As it has been seen on previous sections, the agents in the domain are to build a set of common alternatives to fulfill the global plan (that is, *global plan proposal*) and ponder it. In order to do so, partially built proposals should be exchanged between the agents in the domain. This exchange, along with the internal actions the agents have to perform at each step, are depicted in the coordination protocol.

1. An *Initiator* agent [3] sends a requests for *action plan* proposals to all agents in the domain.

2. *Participant* agents process request and prepare their proposals.

[3] This role can be played by any agent in the domain, therefore, the protocol can be considered to be non-centralized

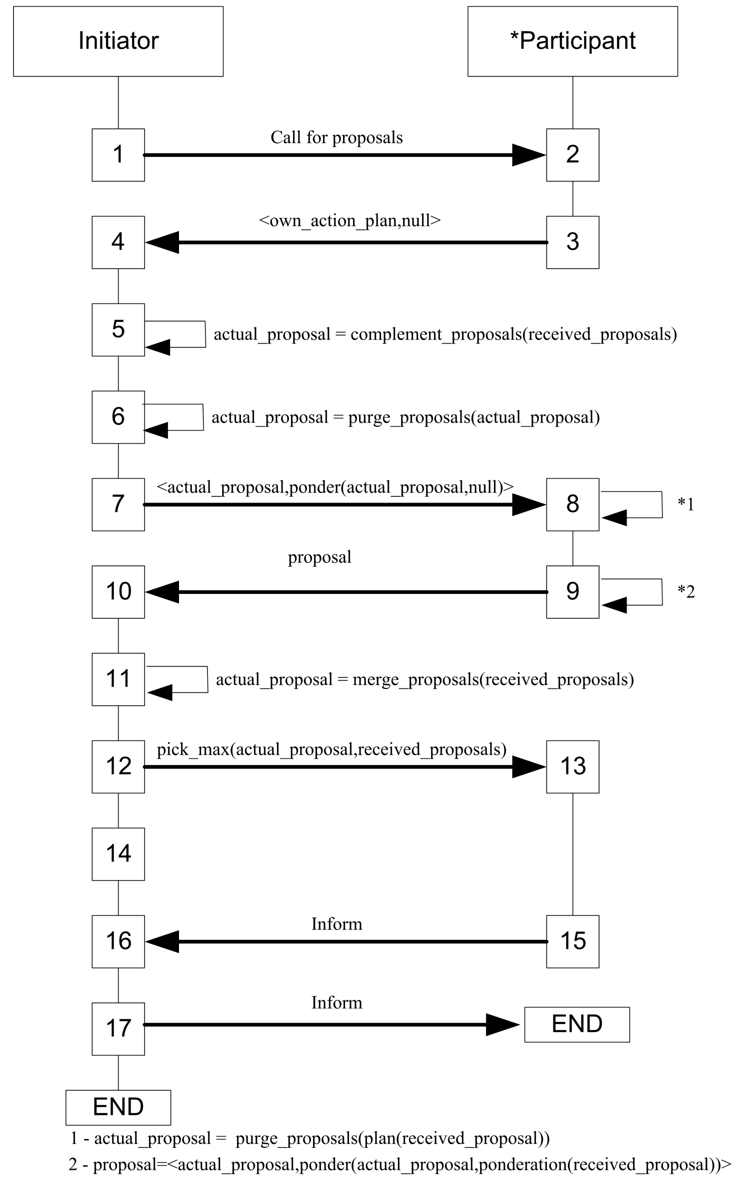

Figure 4.5: Basic protocol diagram

3. *Action plan* proposals are sent to *Initiator* agent.

4. All proposals are received. A time-out can be set here in case any *Participant* agent does not send a proposal.

5. Received *action plan* proposals are complemented in a single *global plan* proposal. At this step, the global plan proposal has alternatives to fulfill all the goals, but it presents two problems. First of all, it has not been pondered using any utility values, and second, it can have non conflict-free alternatives.

6. The *global plan* is filtered, removing non conflict-free options, and pondered based on utilities. At this step the alternatives in the *global plan* are conflict-free, but only according to *Initiator* agent point of view. Different resource association rules on other agents can find conflicts in the alternatives proposed. What's more, the different alternatives in the plan have been pondered only with local utility values (that is, the utilities *Initiator* agent has). Finally, it must be remarked that, in case none of the alternatives in the *global plan* are conflict-free (from *Initiator*'s point of view) the protocol will end at this step.

7. The *global plan* proposal is sent to *Participant* agents.

8. *Participant* agents receive the *global plan* proposal. It is filtered, removing non conflict-free options. At this point, the alternatives in the global *global plan* are conflict-free from *Initiator* agent's point of view, and from one *Participant* agent's point of view. Finally, it must be remarked that, in case none of the alternatives in the *global plan* are conflict-free (from at least one *Participant*'s point of view) the protocol will end at this step.

9. The filtered *global plan* proposal is pondered based on *Participant's* utilities. At this point, alternatives have been pondered using the utilities of two agents in the system: *Initiator* and one *Participant*. Then, the proposal sent back to *Initiator* agent as a *global plan* counter-proposal.

10. All *global plan* counter-proposals are received. A time-out can be set here in case any *Participant* agent does not send a counter-proposal.

11. *Initiator* agent merges all *global plan* counter-proposals into a single common *global plan* proposal. All alternatives should be conflict-free (from all agents points of view) at this point. Also, all alternatives should have been pondered according to all agents utilities values at this point, that is pondered using global utilities.

12. *Initiator* agent chooses the alternative with a higher utility score. It sends the alternative to all *Participant* agents.

13. Each *Participant* agent starts executing the actions of the selected alternative that belongs to him.

14. *Initiator* agent starts executing the actions of the selected alternative that belongs to him.

15. Each *Participant* finishes performing its part of the alternative. Upon finishing, it notifies *Initiator* agent.

16. All notifications are received. A time-out can be set here in case any *Participant* agent does not send alternatives.

17. Once *Initiator* agent has finished performing its own actions, it informs *Participant* agents that the global plan has been performed. If one action of the plan cannot be performed or fails, *Initiator* can command *Participants* to take actions to revert the system to a coherent state (that is, actions to undo actions that have been already performed) and try another alternative (the one with the second-best score according to global utilities) following protocol from step 12.

Actions in the protocol presenting a higher complexity are depicted on the *Algorithms* using meta-code. For understanding the meta-code provided, the following premises are to be taken into account:

- *ponderation* is a hash set where the keys are alternatives and the values integers. Given an alternative *alt*, *ponderation.value*(*alt*) accesses the value associated to this alternative, returning 0 if this value does not exist.
- *proposal* is a pair where one element is an *action plan*, and the second a *ponderation*. Given a proposal *prop*, *plan*(*prop*) and *ponderation*(*prop*) return the *action plan* and the *ponderation* of the proposal respectively.
- *own_action_plan* denotes an agent's *action plan*
- *received_proposals* denotes the set of proposals that Initiator agent receives from participants.

In the example presented, *broker agents* lack planning capabilities. Thus, as seen on the original protocol, they will perform task 3 to provide their own *action plans* (that is, the activity proposals) but do nothing at steps 8 and 9 (steps are kept for compliance with the original protocol). On the other hand, they also do not need to perform any action at steps 13 and 15, because activities will be performed by user, not by agents.

4.8 Related and further work

Dynamic service composition has been covered by several works. Typically, these works tackle dynamic service composition via pre-defined workflow models (such as in [2]) or AI planning techniques (such as in [32]). In the former, an abstract composite service is defined at design time. In this abstract composition nodes are not bound to services, but to generic *search recipes*. At run time, this generic recipes are bound to concrete services. The ALIVE architecture[4] uses a similar approach (via *Service templates*) but does not use pre-defined abstract workflows, but dynamically generated ones. In the latter, existing methods tend to assume that each service is an action that alters the state of the world. Our framework uses a similar approach (as actions map to service invocations). However, most of existing approaches use centralized planning. In centralized planning, a single component (therefore, a single failure point) is the one that dynamically generates the global plan, and it does so based on its perception of the world, which is typically centralized. The work presented in this chapter uses a distributed planning (even though the example shown is a centralized planner, for simplicity) inspired on GPGP where all agents build local plans that are then shared in order to generate the global plan.

Argumentation approaches have been applied for handling conflicts in other works, such as the one by Hulstijn and van der Torre [29] . Conflicts between complex elements are derived from the conflict between the simpler elements that compose them, that is, the actions. The framework presented in this chapter uses the same idea, but takes it a bit further, completely specifying the reason why two given actions attack each other, and applying this idea to practical ideas such as planning and service composition.

The detection of resource conflicts when generating plans is an issue that has also be tackled in Decker's work '*Coordinating mutually exclusive resources in GPGP*'([5]). Decker's work focuses on preventing the use of mutually exclusive resources (known as *mutex*) via

[4]The ALIVE architecture is introduced in *Appendix A*

a *resource agent*. Resources, as defined in Decker's work, are not consumed by tasks, but used for a certain period of time. When this period expires, the resource can be used by another task. When agents want to execute tasks, they send a bid for the resources that will be occupied by the task to the *resource agent*, who decides which task will make use of the resource first. Our planning framework has several advantages over Decker's work. First of all, is more versatile, as it allows the inclusion of both *mutex* resources and resources that are consumed when the task is performed. It is unclear how Decker's work deals with resources that are consumed upon action execution. Second, it is able to deal with multi-criteria resources (for instance, using both money and time as resources on the example presented), which can supose a problem under Decker's approach, due to having to start multiple bids (one per resource criteria). Last but not least, it is fully distributed. In Decker's work, a centralized component is added to a distributed planning system, and this is not consistent with the idea of distributed plan generation. In the framework presented, both plan generation and resource-conflict detection and management are performed in a distributed fashion.

In future steps it is planned to extend the theoretical framework presented to include i) negative goals (that would represent sets of actions to be avoided), ii) negative resources (that could either indicate that performing an action will generate a resource or be used to model synergy between actions, that is, actions that should be performed together) or iii) actions appearing in several action plans (this would include a negotiation process between involved agents in order to decide who is to perform this shared action). The potential improvement provided by using a global utility function instead of a local one is also to be tested in the future.

4.9 Conclusion

In this chapter a resource conflict management approach to detect and solve conflicts between service invocations in dynamically generated workflows has been presented. Conflict detection is achieved via an argumentation-based approach, while conflict solving is achieved via a distributed negotiation protocol.

One idea under exploration is using resource assignation to control the amount of time a workflow will take to execute or to balance the work load on a set of services. Services can be dynamically linked to resources representing the computational load they experience, such as 'overload', 'high-load', 'medium-load', 'idle'. Then, depending on available resources, the system would be able to discard workflows that are expected to take too much time to execute (e.g., they attempt to use too many overloaded -e.g. a resourceless - services).

Using utilities to model the trust (local utilities) and reputation (global utilities) agents have towards available service is a promising approach. Agents can keep a local utility value (this is a trust value) for services they invoke. During global plan negotiation process, this value can be passed to other agents (via a global utility function) that propose this service in their plan. Once all agents annotate plan proposals with their utility values, the global utility function will become a reputation value. Therefore, once an agent invokes a service, it can share information about its performance with other agents.

Chapter 5

Implementing the planning framework in PAWS

This chapter explains in depth the implementation of the formal planning framework introduced in *Chapter 4*. The chapter makes use of the concepts introduced during the presentation of the formal framework, so reader is encouraged to go through *Chapter 4* before reading this one. Even then, references to the definitions of the formal concepts are used when required.

The framework is implemented under the PAWS platform [24] , that has been already analyzed and described in *Chapter 3*. One of the objectives of this chapter is to demonstrate that, once generalized, the framework can be used for tasks that vary from the original purpose, supporting electronic business systems interacting on the basis of dynamically generated, cross-organizational contracts.

The chapter starts by explaining how the formal negotiation protocol introduced in *Chapter 4* has been implemented. It goes on by detailing the implementation of the functions used in the protocol, that are supported by the implementation of the formal framework. Then, it introduces the process to be followed in order to initialize framework's elements (that is, plans, actions, resources, agents, etc.) via a graphical interface. The chapter goes on by introducing a simple test case with non real elements. Finally the chapter presents a real-world test case (the one presented in *Section 4.2*) implemented using the framework.

As an introduction, the diagram of the classes that form a protocol can be seen in *Figure 5.1.* The diagram of the classes that are used by the behaviors to support decision making can be see in *Figure 5.2.* The components of these diagrams are explained in depth later on this chapter.

5.1 Protocol Modeling

This section explains the formal modeling of the informal negotiation protocol introduced in *Section 4.7.* The protocol applies to the scope of an interaction, that is, from the point where the first message is sent by protocol's initiator (where the protocol starts) to the point where the last message is received, either by the initiator or by the participant agents (where the protocol ends). The main use of the protocol is to simplify the communication process. As the set of agents in the domain know beforehand which protocol will be used for interactions, they can deduce which messages can be sent at each point of the protocol, and which messages can be expected to be received.

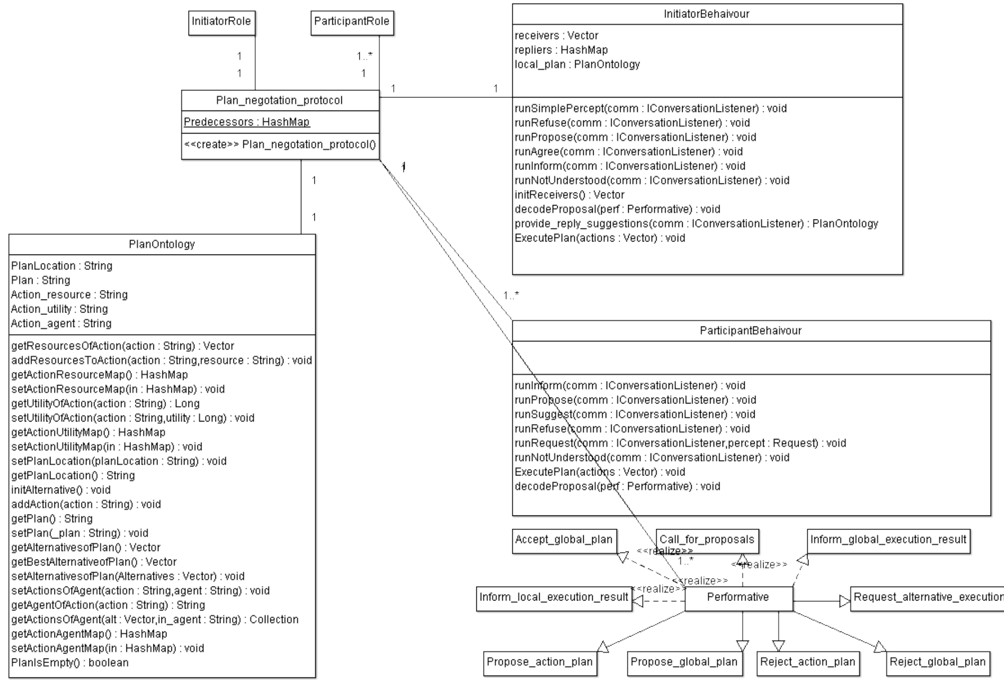

Figure 5.1: Diagram of protocol's classes

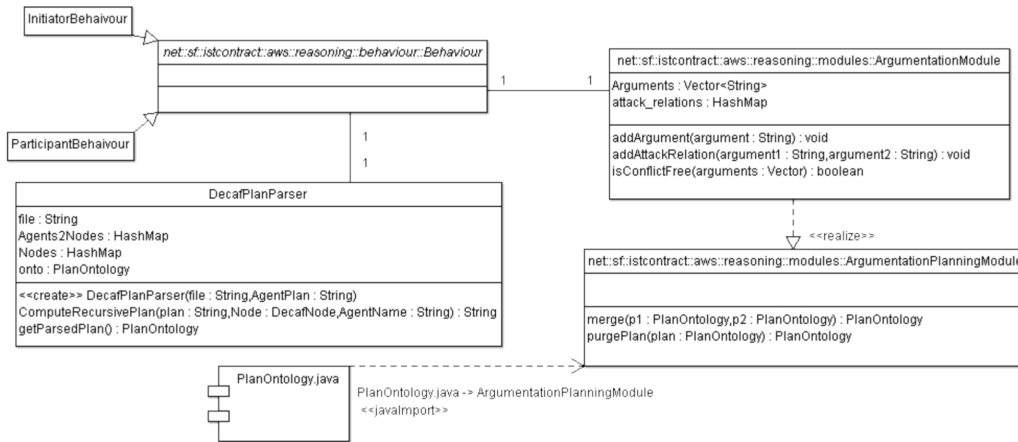

Figure 5.2: Diagram of decision making support classes

The main idea of the protocol is to provide agents with a component that takes care of communication details (from a high-level point of view), enabling other components to focus on their functionality and forget communication details such as which messages should be

sent to start a given interaction or which messages should be expected once the interaction has started. In other words, the functionalities the protocol provides to other components of the agent are:

1. Upon message reception, deduce (from the history of messages received) which is the state of the current interaction, and thus, deduce which deliberative process should be started (that is, decide which alternative is conflict-free, start performing actions in a alternative, etc.). In other words, when a message is received, the agent can infer which action should the other modules perform. In this sense, the protocol acts as the main thread of the agent's execution, demanding functionalities to the other modules when messages are received, in a reactive way.

2. When a message is to be sent to another agent, decide which is the set of possible messages to be sent depending on the state of the current interaction. Also, when protocol's initiator execution is started, decide which message (and which protocol if many of them are available) is to start the execution of the interaction.

5.1.1 Protocol Basic Concepts

The presented protocol (see *Figure 5.3*) is modeled using 'INGENIAS Development Kit' [17]. This editor, developed by the *'GRASIA'* group intends to guide and facilitate the designing and building of multi-agent systems. A part of this design is the 'Interaction Model'. This model defines a set of interaction units that represent a single information exchange between two agents. In other words, each unit represent an agent sending a message, that contains some information, to another agent. Each interaction has an initiator and a collaborator assigned. The initiator can be seen as the agent that sends the message, whereas the collaborator can be seen as a agent willing to receive and process the information contained in the message. In order to model the presented protocol, an interaction unit must be defined for each message in the protocol, and then, assigned initiator and collaborator according to the responsibilities in the protocol.

Due to the resemblance between the protocol and the 'Interaction model' designs, it is possible to perform a transformation, generating stubs of the PAWS protocols from the information included in the 'Interaction Model' design. Thus, it is possible for developers to design PAWS protocols via a graphical editor and transform them into PAWS code stubs using a program created for this purpose. The program, known as *'Ingenias2ContractCodeGenerator'* has been developed by the author of this document under the scope of the IST-CONTRACT [3] European project, and is included as delivered code in the CD that accompanies this document.

5.1.2 High level model

To start with, the protocol must be modeled as a high level flow of messages between the agents involved. This flow will define which messages can be sent (according to the protocol) by which agent, and when. Then, this simple model, which is rather far away from the model INGENIAS requires (as well as far away from the implementation) is divided into three more simple models, that correspond to INGENIAS interaction model. First of all, a model defining the actors in the protocol, that is, the agents participating in the protocol. Second, a model defining the responsibilities of the actors defined regarding each message in the protocol, that is, which is the actor that will send the message, and which is the actor that will receive it. Finally, a model defining the flow of the messages, that is, which is the message that starts the protocol, which messages can follow, and how.

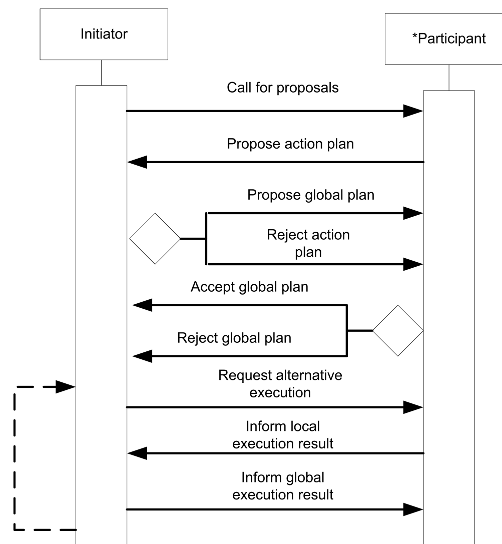

Figure 5.3: Coordination protocol High level design

These are the messages of the protocol as defined seen in *Figure 5.3*:

1. Call for proposals: Message sent by the *Initiator* agent to ask the *Participant* agents to send their *action plan* proposals.

2. Propose action plan: Message sent by the *Participant* agents to provide their *action plan* proposals to the *Initiator* agent.

3. Propose global plan: Message sent by the *Initiator* agent to provide the *Participant* agents with a *global plan* proposal, when there is at least one available.

4. Reject action plan: Message sent by the *Initiator* agent to inform the *Participant* agents there is no conflict-free *global plan* proposal.

5. Accept global plan: Message sent by the *Participant* agents to inform the *Initiator* agent the proposed *global plan* is conflict-free. Includes the modified *global plan.*

6. Reject action plan: Message sent by the *Participant* agents to inform the *Initiator* agent the proposed *global plan* is not conflict-free.

7. Request alternative execution: Message sent by the *Initiator* agent to ask the *Participant* agents to start executing an *alternative for the global plan.*

8. Inform local execution result: Message sent by the *Participant* agents to inform the *Initiator* of the local result of executing the *alternative for the global plan.* That is, the result of executing the actions the *Participant* agent has to perform from the whole alternative.

9. Inform global execution result: Message sent by the *Initiator* agent to inform the *Participant* agents of the global result of executing the *alternative for the global plan.* That is, the result of executing the set of all actions in the alternative. If any alternative failed, the *Initiator* agent can include in the content of the message some actions to be

executed in order to undo already performed actions in the alternative that has failed. Then, the *Initiator* agent can request the execution of a different alternative, via the dotted-line path.

5.1.3 INGENIAS Role model

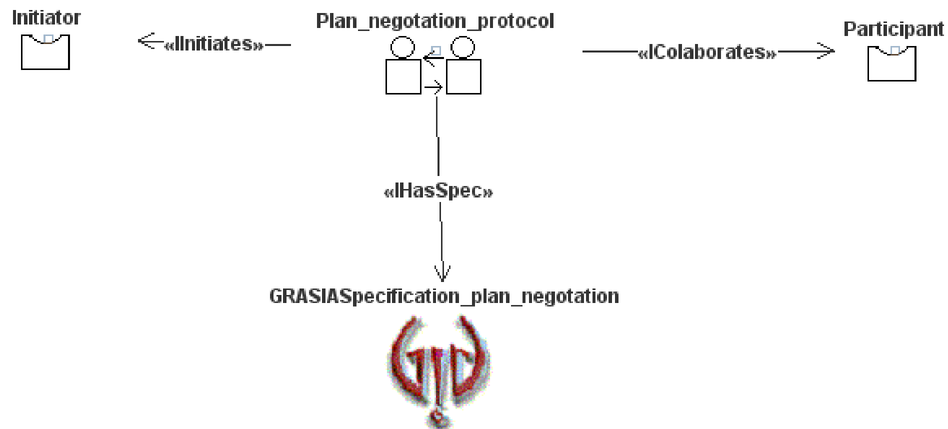

Figure 5.4: Coordination protocol, INGENIAS design: Roles

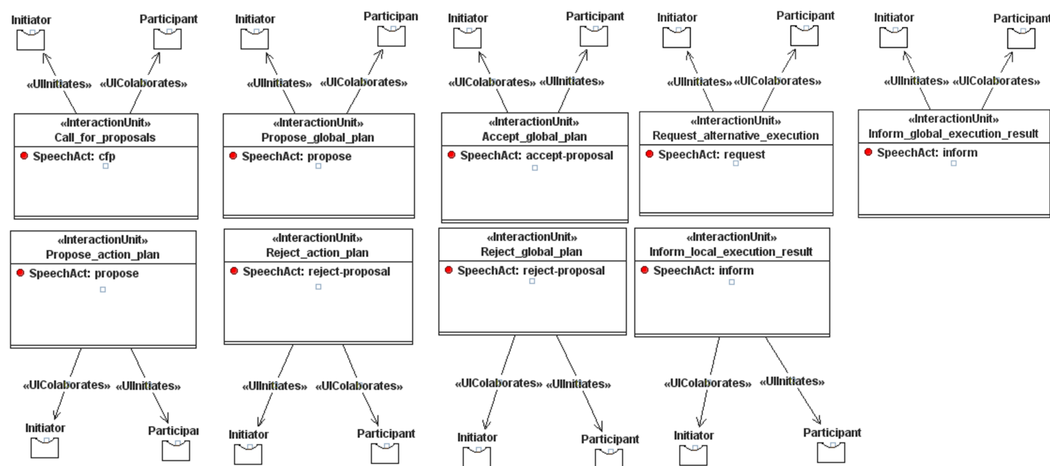

Figure 5.5: Coordination protocol, INGENIAS design: Responsabilities

Figure 5.4 shows the modeling of the actors involved in the protocol, that is, the *roles model*. Each kind of actor is modeled as a role. In this case, only two roles are available, the *Initiator* and *Participant*. Note that the *Initiator* role is modeled as the initiator of the protocol, and the *Participant* role as the collaborator of the protocol. Also, note the protocol can be assigned a name at this point. Finally note how protocol is assigned to

a 'Grasia' specification, however this association has no implications on the scope of this document.

5.1.4 INGENIAS Responsibilities model

Figure 5.5 shows the modeling of message responsibilities as an INGENIAS interaction model, that is, the *responsibilities model*. Each interaction unit represents a message with one or more initiator and collaborators, a name (which is assigned in alignment to the message for simplicity) and a performative. As specified in *Section 2.1.1.4*, each performative represents a communicative intention. Please note that when an agent has the tip of the arrow of a message in *Figure 5.3* it has a collaborates relationship with the interaction unit that represents the message, whereas the other agent has an initiates relationship with the interaction unit.

5.1.5 INGENIAS Precedences model

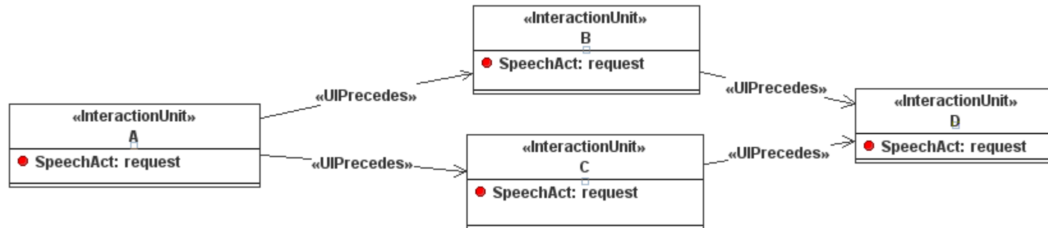

Figure 5.6: Coordination protocol, INGENIAS design: Precedences example

Figure 5.7 shows the order between the different interaction units defined, that is the *precedences model*. This ordering is modeled via 'ui-precedes' relationships. If a given interaction unit 'A' precedes another unit 'B', unit 'B' cannot be exchanged between the participants in the protocol until 'A' has been exchanged. Alternative paths can also be specified, taking the example in *Figure 5.6*, unit 'D' cannot be exchanged between the participants of the protocol until either unit 'B' or unit 'C' have been exchanged. By defining these precedences some restrictions that are inherent to the protocol can be modeled. Such restrictions are of the form '*Propose_action_plan' message cannot be send if message 'Call_for_proposals' has not been received before*, or *'Propose_global_plan' can only be followed by either 'Accept_global_plan' or 'Reject_global_plan' message.* Notice messages that are not linked by 'precedes' relationship to another message are finishing messages, that is, the protocol ends once these messages are sent. Example of these messages are 'Inform_global_execution_result' or 'Reject_global_plan'.

5.1.6 INGENIAS Interaction Model

As seen on *Figure 5.8* all the models defined before (that is, roles, responsibilities and precedences) are put together in a INGENIAS *Interaction Model*.

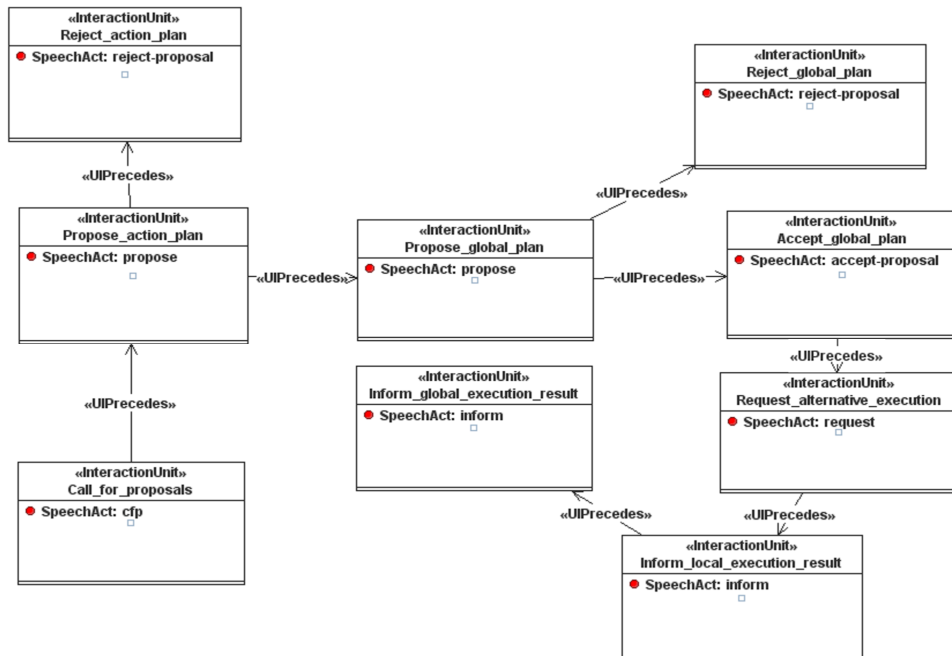

Figure 5.7: Coordination protocol, INGENIAS design: Precedences

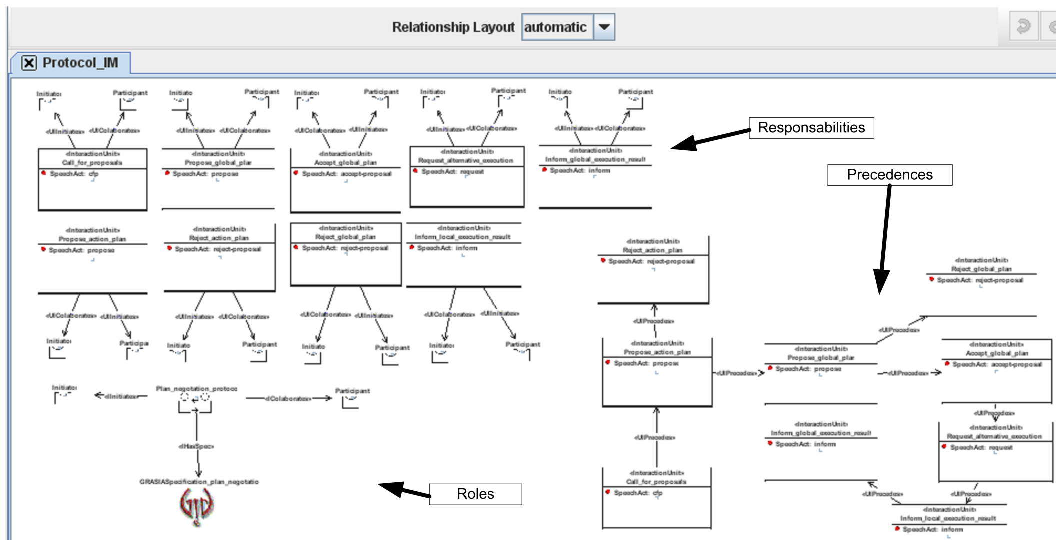

Figure 5.8: Coordination protocol, INGENIAS general design

5.2 Protocol Implementation

INGENIAS *Interaction Model* is able to fully specify the interaction protocol that will be used in the framework. However, in order to be able to use the protocol, it must be integrated

in the platform that will support the framework. To do so, a protocol compatible with the PAWS protocol system must be generated. In PAWS, a protocol is composed by several components, that are grouped under a common Java package. This section goes over all these components, explaining their functionality, and showing an example of the generated file based on the INGENIAS model presented in *Section 5.1*. Once all the components have been introduced this section finishes by stating the process of generating them from the INGENIAS model.

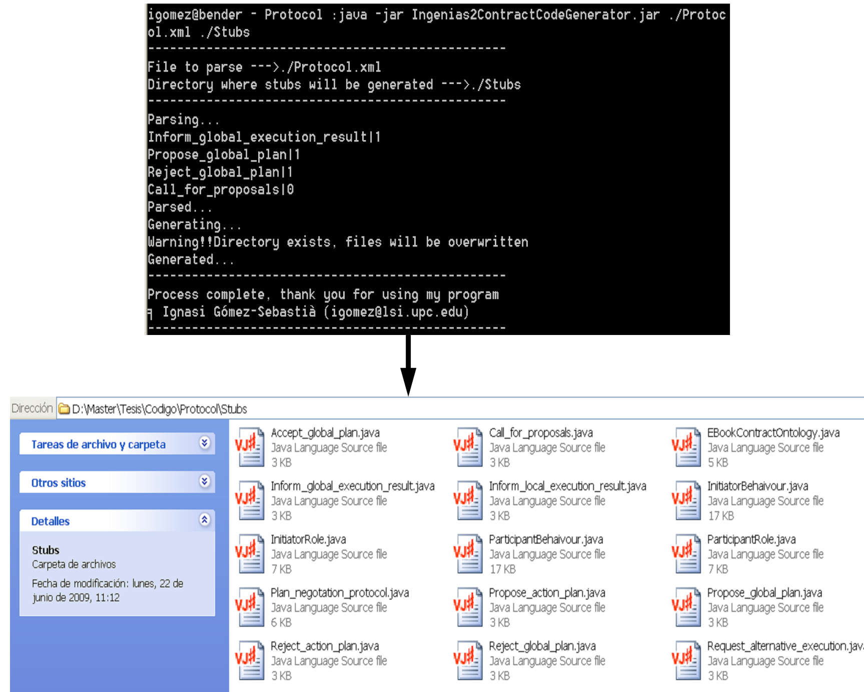

Figure 5.9: INGENIAS and PAWS: Stub generation process

As mentioned before in *Sub-Section 5.1.1* all the components are stubs generated using the software 'Ingenias2Contract CodeGenerator' a meta-coder developed by the author of this document under the scope of the project IST-CONTRACT [3] that is able to transform an INGENIAS interaction model specification into PAWS protocol stubs. In order to generate the stubs all that is to be done is to run the software specifying as parameters the INGENIAS model (saved as an XML formated file) and the system directory where the stubs will be generated. *Figure 5.9* shows this process.

Once the stubs have been generated, all that has to be done is creating a Java package that fits the one used in the stubs (it will be net.sf.istcontract.aws. communication.protocol.*PROTOCOL* where *PROTOCOL* is the name assigned to the protocol) and import the stubs into the package. Once the stubs have been imported, the protocol is ready to be adapted and used.

5.2.1 PAWS protocol components: the protocol

The protocol is the core file of the package. It is an extension of PAWS's *protocol* file and is used to:

```
    //Define and init roles
    Role    Initiator,Participant;
    Initiator = addActor(InitiatorRole.class.getName());
    Participant = addActor(ParticipantRole.class.getName());

    //Set roles on protocol
    setConversationInitiator(InitiatorRole.class);
    setConversationParticipant(ParticipantRole.class);

    //Define and init interaction unit

    MessageTemplate Propose_global_planInitiatorParticipant =
        addMessage(Initiator , Participant , Propose_global_plan.class , EBookContractOntology.class);
    MessageTemplate Reject_global_planParticipantInitiator =
        addMessage(Participant , Initiator , Reject_global_plan.class , EBookContractOntology.class);
    MessageTemplate Accept_global_planParticipantInitiator =
        addMessage(Participant , Initiator , Accept_global_plan.class , EBookContractOntology.class);
  //Set starting messages of protocol
  setStartMessage(Call_for_proposalsInitiatorParticipant);

  //Set precedences between messages of protocol
  LinkFollowingMessage(Propose_global_planInitiatorParticipant, Reject_global_planParticipantInitiator);
  LinkFollowingMessage(Propose_global_planInitiatorParticipant, Accept_global_planParticipantInitiator);
```

Figure 5.10: INGENIAS and PAWS: Protocol stub example

- Initialize the roles participating on the protocol, and assign them as protocol *Initiator* or *Participant*. In order to initialize the roles, generated role classes are used.
- Initialize the messages that are to be used in the protocol. Each of them will have:
  - An initiator, that is, the role responsible of sending the message.
  - A participant, that is, the role responsible of receiving the message.
  - A message type, that is, one of the interaction units from the INGENIAS model. Please note that, according to this assignation, a single interaction unit can map to more than one message in the protocol. It will happen in the case where an interaction unit is linked to more than one role via either 'Initiates' or 'Collaborates' relationships. If it happens, a message is created per interaction unit and per initiator - participant pair.
  - An ontology class. This is a Java class that represents the ontological content that is sent along with each message. A dummy ontological class (known as 'EBookContractOntology') is assigned when the protocol class stub is generated. The programmer can freely adapt this class to its own needs.
- Assign the precedences to the initialized messages. First of all, define which message is to start the interaction. Then define which messages can follow each message in the interaction.

Figure 5.10 shows a small part of the protocol stub file generated from the INGENIAS model presented in *Section 5.1*

5.2.2 PAWS protocol components: the behaviors

The behaviors are the most complex component of the protocols. There is one auto-generated behavior stub per role. Programmers must adapt these stubs, and they can duplicate them to create multiple behaviors per role. This is useful in the case where several agents (possibly with different behaviors) play the same role.

Behaviors provide the protocol with decision making capabilities (that is, if several messages can be sent at any point of the protocol, which one is to be chosen) and the programmer with stubs to facilitate the communication.

First of all, behaviors include procedures to react to incoming messages, enabling programmers to perform some actions upon message reception. The stubs provide functionalities to find out in which exact point of the protocol the interaction is by analyzing the performative contained in the incoming message.

Second, they also include stubs to handle the generation and sending of messages in response to the message that has been received. These stubs include the control of the type of message to be sent (that must be compliant with the protocol being followed), the creation of an ontology content for the message, the creation of a performative and finally the creation of an envelope (following the specifications explained on *Section 2.1.1.3*) to include all the elements mentioned before. Formally, a given message 'MesB' can be sent in response to a received message 'MesA' if the INGENIAS Interaction Model is linking 'MesB' with 'MesA' via a precedence relationship. Apart from that, the role linked to the behavior must have the responsibility of sending the reply message. For instance, following the specification of the protocol presented on *Section 5.1* it can be stated that the Initiator agent can respond either with 'Reject_global_plan' or with 'Accept_global_plan' messages to message 'Propose_global_plan'. This is because both 'Reject_global_plan' and 'Accept_global_plan' are linked via precedence relationship to 'Propose_global_plan', and also because both 'Reject_global_plan' and 'Accept_global_plan' are to be initiated by the Initiator agent.

Last but not least, the behavior stubs are structured in such a way that they ease the process of including the procedures deciding which message to send. Following the prior example, deciding whether to accept or reject a *global plan* when a *global plan proposal* is received.

The behaviors are divided in two main types of functions that can be easily modified by programmers to adapt the generate code to their own needs. It must be noted that, unlike in other components, these modifications are mandatory. Otherwise, the original behaviors cannot be assured to work properly. Apart from these main functions, additional support functions are provided. These support functions include logging of errors during protocol execution or the possibility of sending a *not-understood* message at any point of the protocol.

The first type of main function provided by the protocol is the 'runSimplePercept' function. A stub of this function is depicted in *Figure 5.11*. Only behaviors linked to agents of type *Initiator* include it. The function allows the agents responsible of starting the protocol to send the first message of the protocol to one or more agents when a given external percept is received. The programmer can adapt the following parameters in the function:

- Percept: External percept that makes the agent start the protocol. Other percepts will be logged for debugging purposes and ignored. The original value for this parameter is 'PUT_YOUR_PERCEPT_HERE'.

- Onto: Ontological content to be sent along with the first message. The original value for this parameter is null, so no content is sent.

- receiverName: Name of the agent (as seen on *Section 2.1.1.7*) that will receive the

```
//Behaivour when initiator wants to start protocol
@Override
  protected void runSimplePercept(IConversationListener comm) throws AgentException
  {
          //Variables
              OntologyConcept onto = null;
              //TODO:Add your ontological-content here, if required
              //Uncomment ONLY if ontology is added
              //onto = new OntologyConcept(PLACE_YOUR_ONTOLOGY_HERE);
              //END-OF TODO:Add your ontological-content here, if required
              String percept = "PUT_YOUR_PERCEPT_HERE";
              String receiverName = "PUT_YOUR_RECEIVER_NAME_HERE";
      SimplePercept sp = (SimplePercept)percept;
      //TODO: Adapt conditions and code to your needs
              if ((sp.getStInput().equalsIgnoreCase(percept)))
              {
                  try
                  {
                      receiverName = "PUT_YOUR_RECEIVER_NAME_HERE";
                      //Send the first message of the protocol
                      comm.SendMessageStartNewConversationInitiator(Plan_negotation_protocol.class
                  }
```

Figure 5.11: PAWS Protocol Stub: Run simple percept function

first message of the protocol. More than one receiver can be specified, in this case, an instance of the protocol is started for every receiver agent.

```
//Variables
Performative receivedPerf = (Performative)this.percept;
String ID = comm.buildID(receivedPerf, "Initiator");

//Update state of protocol
comm.UpdateStateOfConversationReceivedMessage(receivedPerf, Propose.class, ID);
//And get current node name
currNode = comm.GetCurrentNodeName(ID);
```

Figure 5.12: PAWS Protocol Stub: Update state of protocol on message reception

The second main type of function is the 'runPERFORMATIVE' function, where *PERFORMATIVE* is a simple PAWS performative. These functions are executed when a message including the corresponding performative is received. The functions include stubs to update the state of the protocol, finding out, exactly, the state of the interaction. The procedure 'UpdateStateOfConversationReceivedMessage' will update the state of the interaction, and the function 'GetCurrentNodeName' retrieves the current node (that is, state) of the protocol. As nodes of the protocol map exactly to INGENIAS interaction units, and such units are unique (they will not appear twice in different steps of the protocol) it can be stated that knowing the current node identifies the current state of the protocol. *Figure 5.12* shows the procedure followed to identify the current state of the protocol when a message is received. Once the current state of the protocol has been received, if it is a final state of the protocol [1] the stub will perform actions to log the end of the protocol. Programmers can add their own code here to perform some actions when the protocol ends in a given state. For instance, in a negotiation protocol, an agent could update the trust it has towards the agent it is negotiating with, raising it if the negotiation ends in a satisfactory state and lower it otherwise. *Figure 5.13* shows the stub corresponding to the treatment of a final message of

[1] That is, no messages can follow these messages according to INGENIAS design

```
if (currNode.equalsIgnoreCase("Reject_global_plan"))
{

        //Agent received a finishing message of protocol
        //TODO: replace with desired code
        AgentLogger.log("Agent is at runRefuse: Current node is ------->'"
        + currNode + "' this is the end, my only friend, the end");
        //END-OF TODO:replace with desired code
}
```

Figure 5.13: PAWS Protocol Stub: Detection of protocol's final state

```
//Uncomment to send Propose_global_plan
/*
        if (1==1)
        {
                replyMessage = this.getBehaivoralChoice("Propose_global_plan", NextMsg);
                Propose replyPerformative = new Propose(    receivedPerf.getIdMessage(), receivedPerf.getIdDialo
                //Message is sent
                comm.sendMessage(replyPerformative, replyMessage, onto, ID);
        }
*/
//Uncomment to send Reject_action_plan
/*
        if (1==1)
        {
                replyMessage = this.getBehaivoralChoice("Reject_action_plan", NextMsg);
                Refuse replyPerformative = new Refuse(  receivedPerf.getIdMessage(), receivedPerf.getIdDialog(),
                //Message is sent
                comm.sendMessage(replyPerformative, replyMessage, onto, ID);
        }
```

Figure 5.14: PAWS Protocol Stub: Detection of protocol's non-final state

a protocol. If it is not the final state of the protocol, a stub to create and send the messages that can follow the message received is provided. There is one stub per possible message. It is responsibility of the programmer to decide which of the stubs is chosen, that is, which of the messages is to be sent. For instance, in a negotiation protocol where a selling prize proposal can be accepted or rejected by the buyer, an *accept* message can be sent if the prize suggested is lower than the utility perception of the item by buyer, and a *reject* message sent otherwise. *Figure 5.14* shows the stub corresponding to the treatment of a non-final message of a protocol

5.2.3 PAWS protocol components: the message types

The message types are representations of the interaction units that appear in the INGENIAS model. Each message type is an extension of the PAWS performative that corresponds to the INGENIAS performative assigned to the interaction unit in the model. Notice that message type stubs generated are rather simple, they just contain a constructor method that calls the constructor of the superclass. However, the possibility to extend these stubs with custom code is a powerful tool for PAWS programmers. By extending the code one can, for instance, easily include procedures that are to be executed every time an instance of a given performative is generated.

It must be remarked that performatives used in PAWS are not the same as the ones used in INGENIAS. However, a direct mapping between both can be easily generated. This mapping has been designed taken into account FIPA's performative definition as seen on *Section 2.1.1.4*. To summarize the mapping can be seen in *Table 5.1*

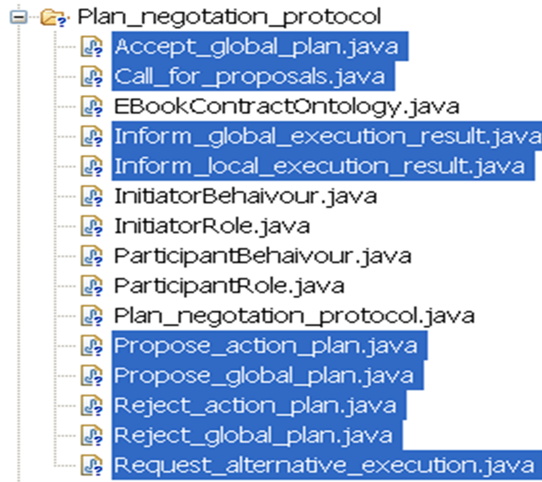

Figure 5.15: INGENIAS and PAWS: Message types generated

Table 5.1: PAWS-INGENIAS performative mapping

INGENIAS performative	PAWS performative
$accept - proposal$	$AcceptProposal$
$agree$	$Agree$
$confirm$	$ConsentSuggestion$
$disconfirm$	$DismissSuggestion$
$failure$	$Failure$
$inform$	$Inform$
$propose$	$Propose$
$query$	$Query$
$refuse$	$Refuse$
$request$	$Request$
$subscribe$	$Subscribe$
cfp	$Suggest$
$cancel$	$UnSubscribe$

Figure 5.15 shows all the message type stubs (highlighted components) generated from the INGENIAS model presented in *Section 5.1*.

5.2.4 PAWS protocol components: the roles

The roles are representations of the different types of agents that can interact via the protocol. Roles are used to filter the messages that, at any point of the protocol, a given agent is allowed to receive. By definition, a given message is allowed if it contains a performative such that there is, at least, one interaction unit containing this performative and having a 'collaborates' relationship with the role. In this sense, role's functionality is rather obsolete, because behavior components are able to perform this very same task with more flexibility, and keeping track of the interaction state, enabling and disabling messages at different points of the protocol rather than doing it for the whole protocol. In other words, unlike roles, behaviors can ensure not only that interaction unit responsibilities are followed but also that interaction unit precedences are. However, roles are kept for backwards compatibility with older versions of protocols' implementations, and because they can be used to include some procedures that are to be invoked when a given message is received.

Role implementations can be divided in two main procedures:

- processIncomingMessage: procedure invoked when a message is received. If the performative in the message is permitted, as defined before, the message is forwarded to

```
@Override
public void startConversation(IConversationListener owner, Performative perf, ProtocolNode rootNode)
throws ProtocolViolationException {
    this.owner = owner;
    protocolNode = rootNode;
    if (1==2) {/*this is a dummy code, so all performatives are inside else if*/}
    else if ( (perf instanceof Refuse)) {processIncomingMessage(perf, owner);}
    else if ( (perf instanceof Propose)) {processIncomingMessage(perf, owner);}
```

```
        private void handleRefuse(Performative perf, MessageTemplate[] possibleReplies)
        {
            //NOTICE: Intelligence and message processing moved to behaivour
        }

        private void handlePropose(Performative perf, MessageTemplate[] possibleReplies)
        {
            //NOTICE: Intelligence and message processing moved to behaivour
        }
```

Figure 5.16: INGENIAS and PAWS: Role code stub example

a handler function. Otherwise, an exception is thrown.

- handlePERFORMATIVE: where *PERFORMATIVE* is a permitted performative. Indeed, there is one handler function for each permitted performative (i.e handlePropose, handleAgree, etc..). In the role stubs, these functions are empty, but ready for programmers to add their own code here if they require so.

Figure 5.16 shows a small part of a role stub generated from the INGENIAS model presented in *Section 5.1*.

5.2.5 PAWS protocol components: the ontology

The ontology is the component of the protocol that allows agents to include meaning about the entities in the domain and their relationships into the messages. In practical means, in PAWS the ontology is translated into a set of Java classes (one per entity to be represented) with a set of attributes. The class will also present two functions per attribute, one to set the value of the attribute and one to retrieve it. Attributes can be references to another classes (effectively modeling relationships between entities) or simple data types (such as the common types provided by Java, Integer, String, etc...). The ontology class is initialized in the behavior components, before sending a message, and parsed back (via Java class casting) upon message reception. A dummy ontology is automatically generated from the INGENIAS model and included in protocol's Java package. Programmer can adapt the ontology later to its own needs.

Figure 5.17 shows a small part of a dummy ontology generated from the INGENIAS model presented in *Section 5.1*.

5.2.6 PAWS component generation procedure

In order to generate the stubs for the components mentioned before in this section, two procedures are to be followed. First, the INGENIAS interaction model must be parsed,

```
public class EBookContractOntology {
    @XmlElement(required = true, nillable = true)
    protected String name;
            public String getName() {
                return name;
            }
        public void setName(String value) {
            this.name = value;
        }
```

Figure 5.17: INGENIAS and PAWS: Role code stub example

retrieving information about the important components in the model. Then, this information must be transformed to Java code implementing the PAWS protocol components stubs.

The parsing process is performed by reading the text file (formatted in xml) that stores the INGENIAS interaction model. The following elements are retrieved from the file.

```
<object id="Call_for_proposals" type="ingenias.editor.entities.InteractionUnit">
<objectproperty id="TransferredInfo" collection="true">
</objectproperty>
<mapproperties>
<key id="SpeechAct">cfp</key>
<key id="_view_type">UML</key>
</mapproperties>
</object>
```

Figure 5.18: INGENIAS parsing: Message structure

Messages: Each interaction unit is a message. Both name and performative can be retrieved from the interaction unit. INGENIAS interaction units are stored as objects of type 'ingenias.editor.entities.InteractionUnit'. The name of the message can be retrieved from the 'id' property of the object. The performative of the message can be retrieved from the value of the 'SpeechAct' key of the object. This information is useful for building both protocol and message type components. *Figure 5.18* shows an example of a message as stored in an INGENIAS xml file.

```
<object id="Initiator" type="ingenias.editor.entities.Role">
<mapproperties>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</object>
```

Figure 5.19: INGENIAS parsing: Role structure

Roles: Each role is modeled as an INGENIAS role. Only the name of the role is relevant for the stubs. Roles are stored as objects of type 'ingenias.editor.entities.Role'. The name of the message can be retrieved from the 'id' property of the object. This information is useful for building protocol, role and behavior components. *Figure 5.19* shows an example of a role as stored in an INGENIAS xml file.

Protocol: The name of the protocol can be retrieved from the INGENIAS model. The protocol is stored as an object of type 'ingenias.editor.entities.Interaction'. The name of the protocol can be retrieved from the 'id' property of the object. This information is used for

```
<object id="Plan_negotation_protocol" type="ingenias.editor.entities.Interaction">
<objectproperty id="TransferredInfo" collection="true">
</objectproperty>
<mapproperties>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</object>
```

Figure 5.20: INGENIAS parsing: Protocol structure

the name of the protocol stub file and for the name of the package. *Figure 5.20* shows an example of a protocol as stored in an INGENIAS xml file.

```
<relationship id="68" type="ingenias.editor.entities.UIPrecedes">
<object id="68" type="ingenias.editor.entities.UIPrecedes">
<mapproperties>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</object>

<role idEntity="Call_for_proposals" class="ingenias.editor.entities.InteractionUnit" roleName="UIPrecedessource"
type="ingenias.editor.entities.UIPrecedessourceRole" dgcid="18781313">
<mapproperties>
<key id="attributeToShow">0</key>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</role>

<role idEntity="Propose_action_plan" class="ingenias.editor.entities.InteractionUnit" roleName="UIPrecedestarget"
type="ingenias.editor.entities.UIPrecedestargetRole" dgcid="21695081">
<mapproperties>
<key id="attributeToShow">0</key>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</role>
</relationship>
```

Figure 5.21: INGENIAS parsing: Message precedence structure

```
<relationship id="39" type="ingenias.editor.entities.UIColaborates">
<object id="39" type="ingenias.editor.entities.UIColaborates">
<objectproperty id="Condition">
</objectproperty>
<mapproperties>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</object>

<role idEntity="Reject_action_plan" class="ingenias.editor.entities.InteractionUnit" roleName="UIColaboratessource"
type="ingenias.editor.entities.UIColaboratessourceRole" dgcid="18245896">
<mapproperties>
<key id="attributeToShow">0</key>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</role>

<role idEntity="Participant" class="ingenias.editor.entities.Role" roleName="UIColaboratestarget"
type="ingenias.editor.entities.UIColaboratestargetRole" dgcid="32789224">
<mapproperties>
<key id="attributeToShow">0</key>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</role>
```

Figure 5.22: INGENIAS parsing: Message-role collaborates responsibility structure

Message precedence relationships: Each precedence relationship between interaction units is stored in a 'relationship' of type 'UIPrecedes'. These relationships contain two interaction units: one unit with property 'roleName' set to 'UIPrecedessource' and the other set to 'UIPrecedestarget'. The unit with the value 'UIPrecedessource' is linked to the unit with value 'UIPrecedestarget' via a precedence relationship. In other words, the message corresponding to the unit with value 'UIPrecedessource' is followed, according to the protocol specification, by the message corresponding to the unit with value 'UIPrecedestarget'. This information is useful for building protocol and behavior components. *Figure 5.21* shows an example of a message precedence relationship as stored in an INGENIAS xml file.

Message-role collaborates relationships: Each responsibility relationship of type 'collaborates' assigned to an interaction unit is stored in a 'relationship' of type 'UIColaborates'. These relationships contain two components: one is an interaction unit with property 'roleName' set to 'UIColaboratessource'. The other component is a role with property 'roleName' set to 'UIColaboratestarget'. In other words, the message corresponding to the component with value 'UIColaboratessource' can be received, according to the protocol specification, by the role corresponding to the component with value 'UIColaboratestarget'. This information is useful for building protocol, behavior and role components. *Figure 5.22* shows an example of a message-role 'collaborates' relationship as stored in an INGENIAS xml file.

```
<relationship id="0" type="ingenias.editor.entities.UIInitiates">
<object id="0" type="ingenias.editor.entities.UIInitiates">
<objectproperty id="Condition">
</objectproperty>
<mapproperties>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</object>

<role idEntity="Call_for_proposals" class="ingenias.editor.entities.InteractionUnit" roleName="UIInitiatessource"
type="ingenias.editor.entities.UIInitiatessourceRole" dgcid="16607011">
<mapproperties>
<key id="attributeToShow">0</key>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</role>

<role idEntity="Initiator" class="ingenias.editor.entities.Role" roleName="UIInitiatestarget"
type="ingenias.editor.entities.UIInitiatestargetRole" dgcid="1935046">
<mapproperties>
<key id="attributeToShow">0</key>
<key id="_view_type">INGENIAS</key>
</mapproperties>
</role>

</relationship>
```

Figure 5.23: INGENIAS parsing: Message-role initiates responsibility structure

Message-role initiates relationships: Each responsibility relationship of type 'initiates' assigned to an interaction unit is stored in a 'relationship' of type 'UIInitiates'. These relationships contain two components: one is an interaction unit with property 'roleName' set to 'UIInitiatessource'. The other component is a role with property 'roleName' set to 'UIInitiatestarget'. In other words, the message corresponding to the component with value 'UIInitiatessource' can be sent, according to the protocol specification, by the role corresponding to the component with value 'UIInitiatestarget'. This information is useful for building protocol, behavior and role components. *Figure 5.23* shows an example of a message-role 'initiates' relationship as stored in an INGENIAS xml file.

Once the information provided by the parsing process is available, the stubs for PAWS protocol's components can be generated. The process for generating each stub is as follows:

Protocol: One protocol file is generated. Protocol's class name is the 'Protocol name' retrieved during parsing. This name is also used to give a proper name to class' constructor method. First of all, variables representing the roles are created based on 'Role names' retrieved during parsing. These variables are initialized using 'Role type' information, effectively assigning *Initiator* or *Participant* roles to them. Once these variables are ready, the Initiator and the Participants of the protocol can be assigned. Secondly, messages variables are created using 'Interaction units names'. In order to create these variables, the sender and receivers of the messages must be known, and this information has been retrieved during the parsing via both 'Message-role collaborates relationships' and 'Message-role initiates relationships'. In order to initialize these variables, sender of the message, receiver of the message and performative corresponding to the message are required. Also, the name of the ontology to be used must be know, but this information is static, and is not retrieved during the parsing process. In the third step, message variables are used to set the message to start the protocol. This information has been implicitly retrieved during parsing, because 'Message precedence relationships' state which message goes before each message. In order to know which message starts the protocol, all that has to be done is looking for the message that is not preceded by any message. Last but not least, message precedence relationships are set. This process is fairly easy, taking into account 'Message precedence relationships' are already available.

Roles: One role file per role in the INGENIAS interaction model is generated as retrieved in 'Role names' during parsing. Role's class name is the name of the protocol and the name of the role as retrieved via 'Protocol name' and 'Role name' during parsing. This class is an extension of class 'ConversationInitiator' if the role initiates the protocol and of class 'ConversationParticipant' otherwise. Whether the role starts the protocol or not can be inferred by finding out which is the message that starts the protocol, and looking at the role responsible of sending it. First of all, if the role is the starter of the protocol, a stub for the function 'startConversation' is generated, otherwise the stub is not included in the role file. Secondly, a stub for function 'processIncomingMessage' is generated. This stub will include a 'handler' function for each performative the role is permitted to receive, according to the definition of the protocol. To know this, it must be found which messages the role is linked to via 'colaborates' relationships, and which performatives do these messages have. All this information has been parsed beforehand via 'SpeechAct' component of the interaction unit and 'Message-role collaborates relationships'. Finally, stubs for handler functions must be generated. There will be a handler function for each performative the role is permitted to receive, as explained before. Handler functions are rather simple, and the name of the performative is used only in the name of the function.

Message types: One per interaction unit available in the INGENIAS interaction model is generated as retrieved in 'InteractionUnit' objects during parsing. Type's class name is the name of the protocol and the name of the interaction unit as retrieved during parsing. This class is an extension of a PAWS performative class. The exact class can be inferred using the performative associated to the interaction unit object and the translation table presented in *Table 5.1.*

Behavior: One role file per role in the INGENIAS interaction model is generated as retrieved in 'Role names' during parsing. Behaivour's class name is the name of the role as retrieved via 'Role name' during parsing and the constant word 'Behaivour'. First of all, if the role is the starter of the protocol, (that is, it has a 'initiates' relationship with the message starting the protocol) a stub for the function 'runSimplePercept' is generated, otherwise the stub is not included in the behavior file. Both protocol's and role's names are required in this function in order to send the first message of the protocol. Then a 'runPERFORMATIVE' function stub is generated for each performative the role can receive. As stated before this

can be found out looking at which units are linked to role via 'colaborates' relationships, and which performatives do these units have. These stubs are run when a message including the specified performative is received. The stubs include a set of functions to perform an update of the protocol, finding out the exact point where the protocol is after message reception. When updating the state of the protocol, it can happen that there is no reference to the protocol, because the message received is the first message of the protocol. In such case, where the role is the a Participant, and the message received is the first message of the protocol, a special set of functions is executed in order to initialize the protocol, setting the state of the protocol to the first message. Once the exact state of the protocol has been identified, the stub provides functions to guide the sending of reply messages to the message received. Regarding this, two cases are possible, either the received message is a final message on the protocol, and thus no messages can be sent in reply to this message, or it is not, and thus the role is allowed to send a message in reply to the message received. If the received message is a final message, a stub to log the termination of the protocol is created. Otherwise, for each possible reply to the received message, a stub for sending the reply is generated. A message is known to be a final because it is not linked to any other message via 'precendence' relationships in the interaction model. Which messages can follow a given message can also be found out by analyzing 'precendence' relationships. This information has been already retrieved during the parsing process.

Ontology: One ontology file is generated. The ontology is an exact copy of an example ontology. This ontology has several simple fields as well as functions to assign values to these fields, update these values and retrieve them. In order to generate the ontology a model file is read and streamed out directly, without any modifications, to the package where the protocol is generated.

5.3 Framework implementation

The framework is implemented using the stubs obtained from the INGENIAS protocol model. First of all, the stubs have to be adapted, to make them use the components provided by the PAWS platform. Then, in order to convert generic Java stubs into Java classes adapted, not only to the protocol specified, but also to the framework to be implemented. Later, to convert some of the dummy stubs provided (mainly the ontology) into classes useful to the framework. This section explains in depth the process followed in order to perform the mentioned tasks. Last but not least, classes that support the framework but are out of the protocol are to be developed, and integrated into the protocol (mainly into behavior classes) as a last step.

5.3.1 Use of agent directory

The original stubs generated from the INGENIAS protocol model contact an agent with a dummy name for sending the message that will start the protocol. This is not the desired situation, as ideally, the protocol should contact every participant agent in the system. This can be easily fixed by creating a set of agent names and contacting every agent in the set. In this case, the Java Vector structure 'receivers' contains the names of all the agents to be included, as participants, in the protocol. *Figure 5.24* shows the PAWS code required by the *Initiator* to send the starting message of the protocol to a set of participants.

The problem with the presented solution is that this set of participant agents known as 'receivers' should be created dynamically. If a new agent enters the system, it should be included in the set and contacted the next time the negotiation protocol starts. In order to

```
Iterator<String> it = this.receivers.iterator();
while (it.hasNext())
{
    receiverName = (String)it.next();
    AgentLogger.log("· Sending to agent '" + receiverName + "'");
    comm.SendMessageStartNewConversationInitiator(Plan_negotation_protocol.class,
            InitiatorRole.class, receiverName, onto);
}
```

Figure 5.24: PAWS Stub: Initator sending message to multiple participants

```
private Vector<String>  initReceivers()
{
    Vector<String> result = new Vector<String>();
    try
    {
        java.util.List AgentList = AgentConfigurationProvider.getConfiguration().getDirectoryFacilitator().
        getAgentScenario().getAgentList().getAgentInfo();
        Iterator<AgentInfo> it = AgentList.iterator();
        while (it.hasNext())
        {
            AgentInfo dummy = (AgentInfo)it.next();
            String AgentName = dummy.getAgentId();
            String AgentType= dummy.getType();
            AgentLogger.log("HEAIOU, agent:'" + AgentName + "' of type '" + AgentType +
                    "' is on the system!!!");
            if (!AgentType.equalsIgnoreCase("Initiator"))
            {
                result.add(AgentName);
            }
        }
```

Figure 5.25: PAWS Stub: Initializing participant's structure via Agent Directory

solve this problem and initialize the set dynamically and efficiently, the *agent directory* (as presented in *Section 2.1.1.2*) comes in hand. *Figure 5.25* shows how the Vector structure containing the names of all the participants can be initialized using the *agent directory* provided by the PAWS platform.

5.3.2 Enabling multiple participants

'runPERFORMATIVE' functions contained in initiator's behavior stub are designed to support the interaction with a single participant agent. These functions have two main problems when dealing with multiple participants.

On the one hand, the code in the functions that is to update the state of the protocol when a message is received can stop working properly if multiple participants are to be taken into account. This is because, the function always updates the state of the protocol, but if multiple participants are present, some other conditions might be taken into account before updating the state of the protocol. To illustrate this need, an example with a negotiation protocol is presented. In this protocol multiple participants reply to a 'propose' message with either 'accept' or 'reject' messages. On the *Initiator* stubs, function 'runReject' does not need any modification, if one 'reject' message is received at least one of the participants has rejected the proposal, so the negotiation has failed and the protocol is over. Thus, the state can be updated upon receiving a single message. However function 'runAccept' cannot update the state of the protocol when a single 'accept' message is received, it has to wait until all messages (that is, one per participant) have been received before updating the state of the

```
this.repliers.put(sender,receivedPerf);
if (this.repliers.size() == this.receivers.size())
{
    AgentLogger.log("All participants have informed of global plan fulfillment");
}
else
{
    AgentLogger.log("Not all participants have replied yet");
    AgentLogger.log("Some global plan execution notifications to be received");
}
```

Figure 5.26: PAWS Stub: Updating state of protocol, multiple participants

protocol. In other words, all agents have to accept before it can be considered the protocol is over and the negotiation has been a success, updating the state of the protocol when only one participant has accepted is a clear mistake. *Figure 5.26* shows the modification required on the stubs in order to handle multiple-participants protocol state update. Please, notice this code makes use of the Vector structure that keeps the names of all the participants, and that has been already used to send all the messages that start the protocol in the previous subsection.

```
Iterator it = this.receivers.iterator();
while (it.hasNext())
{
    String msg_sender = (String)it.next();
    receivedPerf = (Performative)this.repliers.get(msg_sender);
    Inform replyPerformative = new Inform(  receivedPerf.getIdMessage(),
            receivedPerf.getIdDialog(),
            receivedPerf.getInReplyTo(),
            receivedPerf.getSender(),
            receivedPerf.getReceiver(),
            onto,
            receivedPerf.getProtocolName(),
            receivedPerf.getReplyWith()         );
    ID = comm.buildID(receivedPerf, "Initiator");
    comm.sendMessage(replyPerformative, replyMessage, onto, ID);
    AgentLogger.log("·Plan fullfillment notification set to '" + msg_sender + "'");
}
```

Figure 5.27: PAWS Stub: Sending reply to a message, multiple participants

On the other hand, if the protocol state is found to be non-final one or several messages can be sent in reply to the message received. In this case, the stub must create, prepare and send the reply messages. This procedure must be modified in order to support replies sent to several participants. To do so the creation of messages can be maintained (the basic components of the message such as the performative or the ontological content will remain the same between receivers) but the sending of messages must be repeated several times, once per participant agent. Thus, the instructions that send the reply message must be embedded into a loop that will repeat them for each participant agent in the protocol. Apart from that, the message sent must be slightly modified, in order to update the filed 'receiver' to the participant agent contacted at each iteration of the loop. *Figure 5.27* shows the modification required on the stubs in order to handle multiple-participants replies. Please, notice this

code makes use of the Vector structure that keeps the names of all the participants, and that has been already used to send all the messages that start the protocol in the previous subsection.

The following handling functions are present on initiator's stubs. Some of them require the adaptations mentioned above in order to support multiple participants. The list of functions with their adaptations is as follows:

- runRefuse: This function requires no modification. If a single participant agent refuses the proposal, the protocol must finish.
- runPropose: This function must be adapted in order to support multiple participants when updating the state of the protocol. This is because the Initiator agent cannot take a decision about weather to accept or reject the proposals until all proposals have been received. Modifications are also required when sending reply messages (either 'Propose global plan' or 'Reject action plan').
- runAgree: This function must be adapted in order to support multiple participants when updating the state of the protocol. This is because the Initiator agent cannot ask the participant agent to start executing the agreed alternative until all participants have agreed. Modifications are also required when sending the reply message (that is, 'Request alternative execution').
- runInform: This function must be adapted in order to support multiple participants when updating the state of the protocol. This is because the Initiator agent cannot notify of successful plan execution until all participants have successfully executed their alternatives. Modifications are also required when sending the reply message (that is, 'Inform global execution result').

5.3.3 Adapting the ontology

The dummy ontology file generated along with the protocol stubs is fairly simple and clearly not suitable for supporting the complex information interchange the negotiation protocol requires. Thus, it must be adapted in order to make it more complete.

The ontology needs to represent all the information agents interchange during the progress of the protocol, as well as provide means to easily manage this information. Thus the ontology file must include fields to represent the required information along with methods to create, update and query these fields. Methods to perform simple operations on these fields can also be provided. The ontology represents the components in it as String structures. These simple, yet efficient, representations eases the process of serializing and interchanging the plan between agents over the network.

The core of the ontology is the plan. The plan represents all the available alternatives (separated by character '-') and all the actions on each alternative (separated by character '_'). The ontology class provides functions to allow agents to add actions and alternatives to the plan without having to worry about how the plan is represented. Function 'initAlternative' will insert a new alternative in the plan, whereas function 'addAction' will add a new action to the last alternative added to the plan. Function 'getPlan' will return the String representation of the plan, which is very convenient for logging purposes, whereas function 'setPlan' will assign a new plan given a String representation of the plan. Finally, function 'getAlternativesofPlan' will interpret the String representation of the plan to return a more complex representation, that is, a Vector of elements (alternatives) where each element is a Vector of Strings (actions). This last function is useful if the elements in the plan must be

processed, because the data structures provided can be easily iterated. It must be noticed that the reverse function 'setAlternativesofPlan' is also available, in case programmer needs to assign a plan from a Vector structure.

Another important component in the ontology is the *Action Agent* map. This data structure stores the relationships between actions and agents, that is, which agent is to execute which action if an alternative agreement is met at the end of the negotiation. The string representation of this component resembles an xml schema, with an action component having an agent field inside (e.g. $\langle action = ''Action1" \rangle Agent1 \langle /action \rangle$). The ontology class provides a function to set up this relationship ('setActionsOfAgent') as well as a function to query it ('getActionsOfAgent'). This relationship is queried passing a set of actions (typically, the alternative for the global plan that has been agreed) and an agent. The set of actions provided will be filtered, returning only the actions in the set that must be performed by specified agent, according to the relationship. It must be noticed that ontology also provides functions to query the agent responsible of an action ('getAgentOfAction') and to get or set the data structure using Java HashMap structures ('getActionAgentMap' and 'setActionAgentMap' to get and set respectively)

The *Action Utility* map is also a component of the ontology. This data structure stores the relationships between actions and utilities, that is, the preferences each agents has towards each action. The string representation of this component resembles an xml schema, with an action component having an utility field inside (e.g. $\langle action = ''Action1" \rangle 10 \langle /action \rangle$). The ontology class provides a function to set up this relationship ('setUtilityOfAction') as well as a function to query it ('getUtilityOfAction'). It must be noticed that ontology also provides functions to get or set the data structure using Java HashMap structures ('getActionUtilityMap' and 'setActionUtilityMap' to get and set respectively)

The last important component of the ontology is the *Action Resource* map. This data structure stores the relationships between actions and resources, that is, the set of resources that will be consumed when performing each action. The string representation of this component resembles an xml schema, with an action component having a set of resources fields (separated by ',' character) inside (e.g. $\langle action = ''Action1" \rangle Resource1, Resource2 \langle /action \rangle$). The ontology class provides a function to set up this relationship ('addResourcesToAction') as well as a function to query it ('getResourcesOfAction'). The first function will assign a resource to an action (initializing the set of resources if it is the first resource assigned) whereas the second one returns an iterable structure (a Vector) containing the set of resources of a given action. It must be noticed that ontology also provides functions to get or set the data structure using Java HashMap structures ('getActionResourceMap' and 'setActionResourceMap' to get and set respectively)

It must be noticed the ontology contains the field 'PlanLocation'. This field contains the URI of a file containing the specification of the plan. Upon receiving the 'call for proposals' message participants read this field, and initialize their ontologies using the information parsed from this file. This simple method is enough for testing the framework. The ontology can be modified and more complex methods for storing the plans can be provided (such references to database schemas) if required.

Finally, the ontology provides also two functions that do not assign or retrieve values from the fields on the ontology, but perform (somehow complex) queries on them. These functions are 'PlanIsEmpty' that will return a boolean value *true* if there is plan (that is, no conflict-free plan) available, and *false* otherwise, and 'getBestAlternativeofPlan' that making use of 'Plan' and 'ActionUtility' data structures, returns the alternative (as a Vector iterable structure) with the higher overall utility in the plan.

Figure 5.28 shows a sample initialization of the ontology corresponding to the branch of a plan.

```
in.initAlternative();
in.addAction("C1");
in.initAlternative();
in.addAction("C2");

in.setUtilityOfAction("C1", new Long(8));
in.setUtilityOfAction("C2", new Long(2));

in.addResourcesToAction("C1", "R6");
in.addResourcesToAction("C1", "R8");

in.addResourcesToAction("C2", "R4");
in.addResourcesToAction("C2", "R7");

in.setActionsOfAgent("C1", comm.getAgentId());
in.setActionsOfAgent("C2", comm.getAgentId());
```

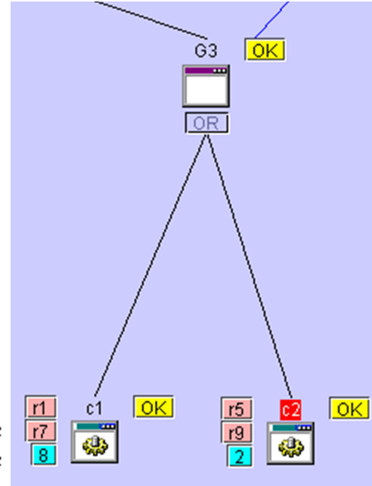

Figure 5.28: PAWS Stub: Initializing the Ontology

5.3.4 Framework implementation

The implementation of the framework starts with the implementation of a generic argumentation framework in a class known as 'ArgumentationModule'. This class is implemented as a reasoning module of the PAWS platform. The class has fields to model the two basic components of every argumentation framework, that is, the set of arguments and the set of attack relations between arguments. The class presents functions to fill this information (that is, add both arguments and attack relations between arguments) and, the most important, to determine if a given set of arguments is conflict-free. That is, given a set of arguments $InSet$ and a argumentation framework $AF = \langle AR, attacks \rangle$ such that $InSet \subseteq AR$:

$$\forall a \in InSet \; not \exists b \in InSet : (a, b) \in attacks$$

In order to determine if the set of arguments provided is conflict-free, the generic framework class goes over all the arguments in the set. For each of those arguments, it is checked weather it has attack relations with another arguments or not. If it does, it is checked if, at least, one of the attacked arguments is in the set provided. If it is, the function returns true. If the function has gone over all the arguments in the set provided without returning true, false is returned instead. False is returned also in the case where no attack relations are available. *Figure 5.29* shows the Java code of the function.

The implementation of the framework goes on with the extension of the generic argumentation framework class to a planning framework class. This new class, which is also implemented as a PAWS reasoning module, provides functions to merge plan ontologies. Given two ontologies $ont1$ and $ont2$, the following actions are performed in order to merge them:

- Merge action-resource map: For every action $a : a \in ont1.plan \wedge a \notin ont2.plan$ add the action-resource map of a in $ont2$ to $ont1$, that is: $ont1.resources(a) = ont2.resources(a)$. For every action $a : a \in ont1.plan \wedge a \in ont2.plan$ merge the resources of a in $ont1$ with the resources of a in $ont2$, that is: $ont1.resources(a) = ont1.resources(a) \cup ont2.resources(a)$.

```
private Vector<String> arguments;
private HashMap<String,Vector<String>> attack_relations;
//Find out if a set of actions if conflict free according to attack relations set-up
protected boolean isConflictFree(Vector<String> arguments)
{
    Iterator<String> arguments_it = arguments.iterator();
    while (arguments_it.hasNext())
    {
        String attacker_argument = (String)arguments_it.next();
        try
        {
            if (this.attack_relations.containsKey(attacker_argument))
            {
                Vector<String> attacked_arguments = this.attack_relations.get(attacker_argument);
                if ((arguments.contains(attacker_argument)) & (share_element(arguments,attacked_arguments)))
                    {
                        return false;
                    }
            }
        }
        //attack_relations is empty, there are no conflicts at all.
        catch (java.lang.NullPointerException E)
        {
            return true;
        }
    }
    return true;
```

Figure 5.29: PAWS Generic Framework: Conflict-free check

- Merge action-utility map: Initialize a new action-utility map. For every action in $ont1$ get the utility of this action and put it in the action-utility map initialized before. For every action in $ont2$ get the utility of this action and put it in the action-utility map initialized before. If the map did already contain an utility for this action, the utility is updated as the sum of the existing utility and the new one provided. The result of this operation is an action-utility map with all the actions in $ont1$ and $ont2$ with their respective utilities. If both ontologies share an action, the utility of the action will be the sum of utilities of the action in both ontologies.

- Merge action-agent map: For every element in the action-agent map of $ont2$, add it to the action-agent map of $ont1$. In this case, no action in the map of $ont1$ should be in $ont2$ and vice-versa. Indeed, if this is found to happen, an exception is thrown.

- Merge plans: For every alternative in the plan in $ont1$, merge it with every alternative in the plan in $ont2$. The procedure to merge two alternatives has been formally introduced in *Section 4.5*. The implementation of the procedure is to initialize a new plan, take the string representation of each alternative in $ont1$ and, for each alternative in $ont2$ append the actions of both alternatives. Then, add this alternative as a new alternative on the initialized plan. Thus, given two plans $plan1 = act1_act2 \| act3_act4$ and $plan2 = bct1_bct2 \| bct3$ merging them will result in the following merged plan $merge = act1_act2_bct1_bct2 \| act1_act2_bct3 \| act3_act4_bct1_bct2 \| act3_act4_bct3$. Now, all that is to be done is removing from this merged plan the alternatives that are not conflict-free. This can be done by initializing the generic argumentation framework presented before with arguments (the actions) and attack relations (add an attack relation whenever two actions share the same resource). Finally, initialize a new plan, and for every alternative in the merged plan, if it is conflict-free, add to the freshly initialized plan, otherwise, discard it.

5.4 Plan definition

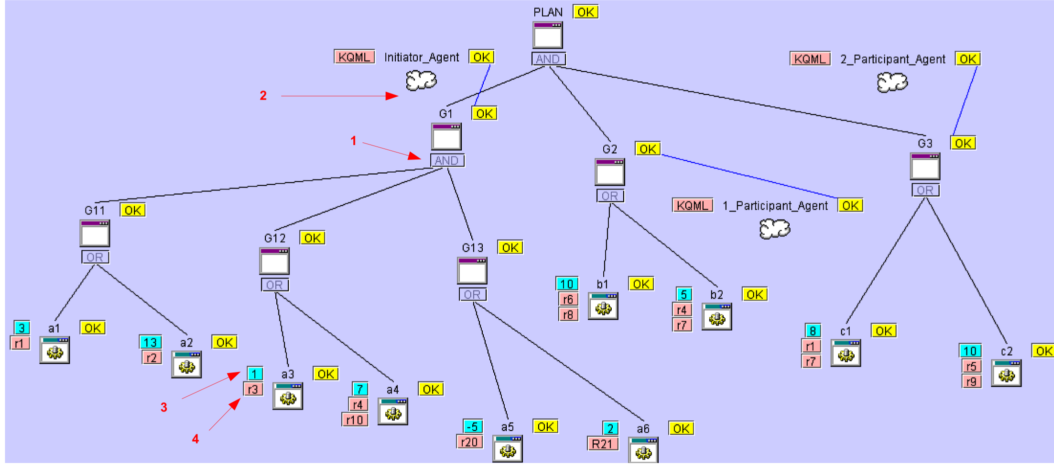

Figure 5.30: PAWS Plan GUI: Main window

Plans can be defined graphically using the plan editor GUI of DECAF system. DECAF (Distributed, Environment-Centered Agent Framework [15]) is a toolkit that provides a stable platform to design intelligent agents systems. DECAF focuses on the design of plans ,scheduling processes and coordination among the agents in the system. In the plan editor, actions are the basic building blocks that can be chained together to achieve a more complex goal. This way of defining and linking actions together is inspired on HTN (hierarchical task network) that defines plans in a tree-like structure, where nodes are actions and hierarchical links between nodes model action decomposition into sub-actions. Each node will have a *'CAF'* annotation (denoted by label 1 on *Figure 5.30*), specifying if, in order to perform the action, all the sub-actions are to be enacted (*'AND'* annotation) at least one of the sub-actions (*'OR'* annotation) or exactly one sub-action(*'XOR'* annotation). In this case *'OR'* and *'XOR'* annotations are equivalent, because agents will perform the minimal number of actions required to fulfill their goal, effectively minimizing the set of possible conflicts between plans. In the case of the plans defined for this framework, the root of the tree is the overall plan, and the nodes directly linked to the root, the goals that agents in the system pursue. Thus, these nodes are assigned to agents(denoted by label 2 on *Figure 5.30*) . Notice the root node will have an *'AND'* type *'CAF'* annotation, because, in order to execute the plan, all the goals are to be fulfilled, however agent's goals nodes can have any type of *'CAF'* annotation, depending on the alternatives available to fulfill the goal. Then, from agent's goals nodes to leaf nodes (the nodes that have no sub-actions assigned to them) there are zero or more intermediate nodes with any type of annotation, this nodes are useful to group leaf nodes allowing the definition of alternatives (*'OR'* annotations) or actions that are to be included in every alternative (*'AND'* annotation). Finally, leaf nodes are defined. This nodes model the actions that agents execute, and thus contain more information associated than other nodes. The main information associated to them is the utility of the actions (denoted by label 3 on *Figure 5.30*) and the resources consumed by each action (denoted by label 4 on *Figure 5.30*).

In the GUI non leaf nodes are added as *tasks*, whereas leaf nodes are added as *actions*. Agents must be added in order to link them to the goals (that is, the sub-actions of the root

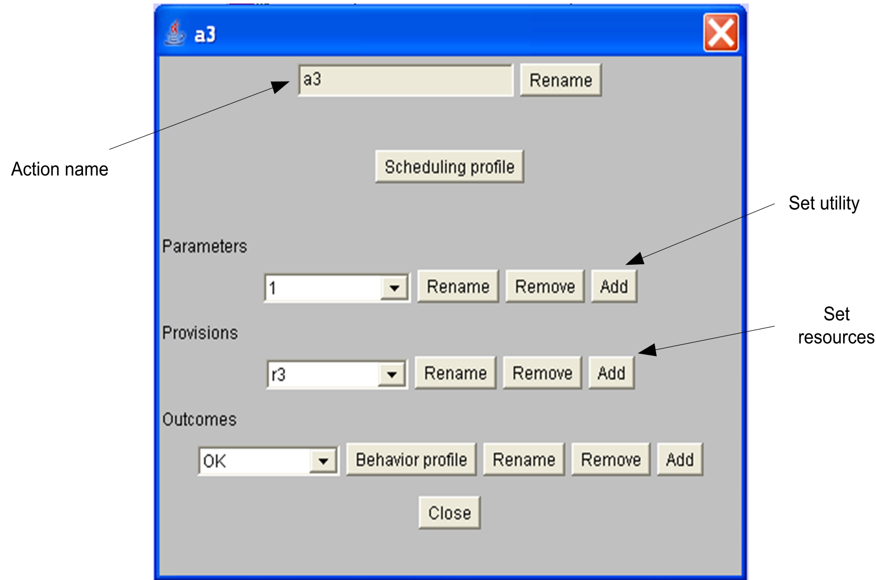

Figure 5.31: PAWS Plan GUI: Pop-up window

Table 5.2: Example plan

Agent	Alternative
$Initiator_Agent$	$\langle a1, a3, a5\rangle$
$Initiator_Agent$	$\langle a1, a3, a6\rangle$
$Initiator_Agent$	$\langle a1, a4, a5\rangle$
$Initiator_Agent$	$\langle a1, a4, a6\rangle$
$Initiator_Agent$	$\langle a2, a3, a5\rangle$
$Initiator_Agent$	$\langle a2, a3, a6\rangle$
$Initiator_Agent$	$\langle a2, a4, a5\rangle$
$Initiator_Agent$	$\langle a2, a4, a6\rangle$
$1_Participant_Agent$	$\langle b1\rangle$
$1_Participant_Agent$	$\langle b2\rangle$
$2_Participant_Agent$	$\langle c1\rangle$
$2_Participant_Agent$	$\langle c2\rangle$

node), and they can be added as *non-local tasks*. Notice that, in this case, the name of the agent assigned to the goal must match the name of an agent in the system. All the elements to be added are available under the *edit* menu, *add item* menu. Once these elements have been added, relationships between them (either node-node or agent-node) can be created by clicking on one element, and dragging an arrow to the second one. Last but not least, information on resources consumed and utility are to be assigned to leaf nodes. This can be done by double-clicking on the node and introducing resources on the *'provisions'* field and setting the utility on the *'parameters'* field on the windows that pops-up (as seen on *Figure 5.31*). Both operations are performed via the respective 'add' buttons.

As an example about how plans are graphically defined, *Figure 5.30* maps to the set of agents and plans for each agent seen on *Table 5.2*. Notice how the *'AND'* node on agent *Initiator_agent* maps to an alternative with 3 elements, each of which is chosen from a set of two elements due to the *'OR'* node linking them.

The integration of the planning framework with the plan definition GUI described is done via the DecafPlanParser Java class which is included in the protocol package. This class, given the path of a plan file and an agent name, returns the Ontology class corresponding to this plan. This class will have all the fields of the ontology (mainly, the plan and the action-

agent, action-resource and action-utility maps) filled up with the information described in the plan file. The process goes over all the nodes filling information about hierarchical relations between the nodes, and storing a reference to the node linked to the agent specified. Once all this information has been gathered, the nodes are processed recursively, from the node that references the agent to the leafs. This process allows the part of the global plan corresponding to the agent specified to be constructed, taking into account both *'AND'* and *'OR'* nodes. Going through an *'OR'* node implies starting a new alternative, whereas going through an *'AND'* implies adding actions to each alternative in the plan. Reaching a leaf node implies filling information about action-agent, action-resource and action-utility maps.

5.5 Simple test scenario

Simple unitary tests on the framework classes have been performed. This section will omit these tests for simplicity and focus on integration tests, to check, from a sample data entry, that the output provided by the framework is correct. The tests are performed using a sample test scenario, with non-real plans.

The tests are focused on checking that the components of the framework are able to guide the negotiation protocol properly, choosing the right path according to the scenario provided. Thus, the following scenarios are defined:

- There is , at least, a conflict-free proposal acceptable by all agents. In this case, the protocol reaches the end when all agents inform of their global execution results. *Figure 5.32* shows the path followed by the protocol in this case. It is useful to duplicate this scenario, defining another one that reaches the same state, but with a different plan being accepted. It can be done by changing the utilities set to actions. At the same time, this test will prove the framework reacts to changes on utilities by accepting a different plan at the end of the negotiation process.

- There is no possible proposal acceptable by all agents. In this case the protocol stops with a refusal message sent, usually, by the Initiator agent. *Figure 5.33* shows the path followed by the protocol in this case.

In order to test the framework when there are acceptable proposals, the scenarios shown on *Figure 5.34* and *Figure 5.36* are used. Using these scenarios as input, the framework throws the results shown on *Figure 5.35* and *Figure 5.37* respectively.

To test framework when there are no acceptable proposals the scenario shown on *Figure 5.38* is used. Using this scenario as input, the framework throws the results shown on *Figure 5.39*.

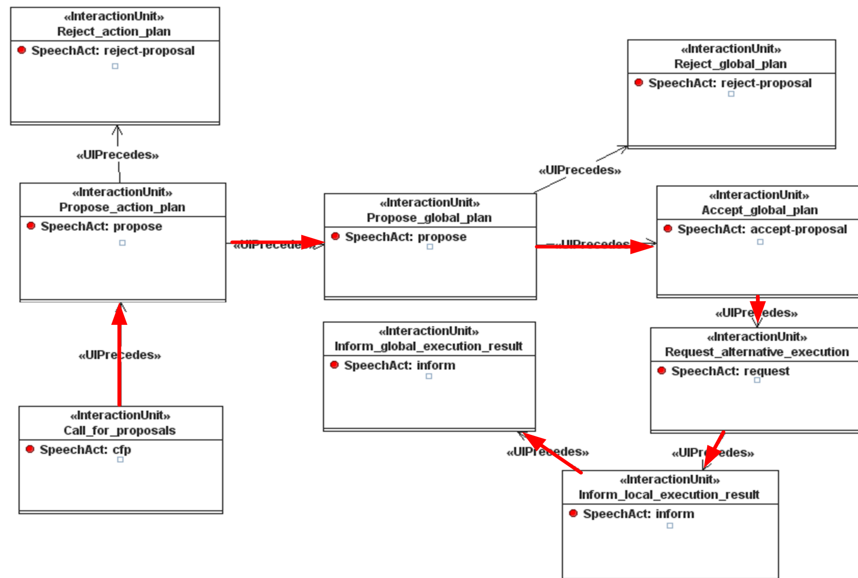

Figure 5.32: Framework: Plan acceptance protocol path

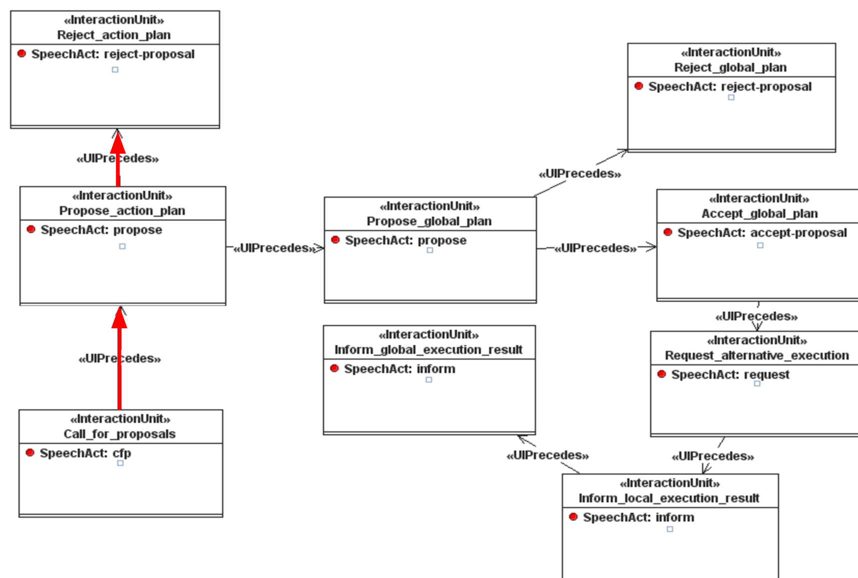

Figure 5.33: PAWS Plan GUI: Plan refusal protocol path

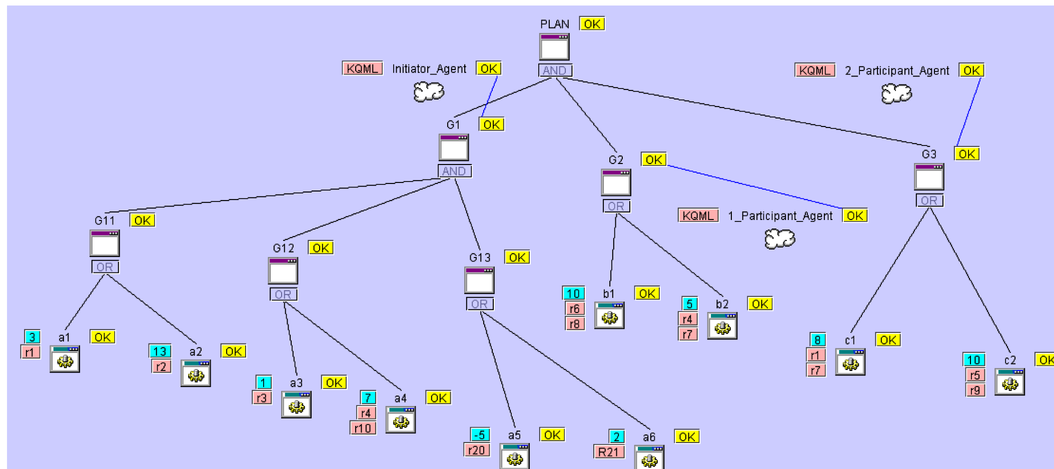

Figure 5.34: Framework: Acceptable plan 1

5:14:2	ParticipantBehaviour.runInform:-1	<----------------------------------->
5:14:2	ParticipantBehaviour.decodeProposal:-1	a2_a4_a6_b1_c2
5:14:2	ParticipantBehaviour.runInform:-1	Plan:
5:14:2	ParticipantBehaviour.runInform:-1	The plan has been succesfully executed
5:14:2	ParticipantBehaviour.runInform:-1	<----------------------------------->

```
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Call_for_proposals
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Call_for_proposals
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Propose_action_plan
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Propose_action_plan
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Propose_global_plan
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Propose_global_plan
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Accept_global_plan
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Accept_global_plan
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Request_alternative_execution
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Request_alternative_execution
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Inform_local_execution_result
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Inform_local_execution_result
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Inform_global_execution_result
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Inform_global_execution_result
```

Figure 5.35: Framework: Acceptable plan results 1

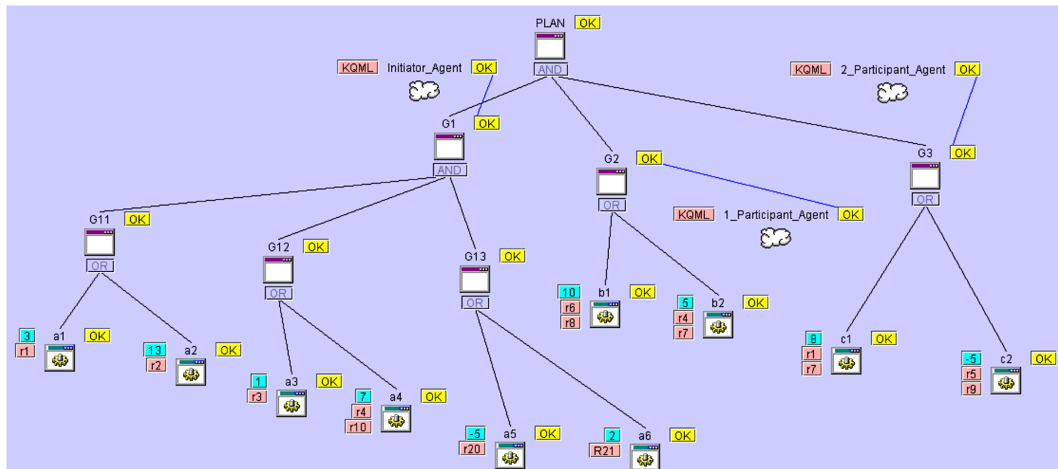

Figure 5.36: Framework: Acceptable plan 2

5:27:58 ParticipantBehaivour.runInform:-1	<------------------------------------>
5:27:58 ParticipantBehaivour.decodeProposal:-1	· a2_a4_a6_b1_c1
5:27:58 ParticipantBehaivour.runInform:-1	·Plan:
5:27:58 ParticipantBehaivour.runInform:-1	The plan has been succesfully executed
5:27:58 ParticipantBehaivour.runInform:-1	<------------------------------------>

```
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Call_for_proposals
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Call_for_proposals
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Call_for_proposals
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Call_for_proposals
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Propose_action_plan
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Propose_action_plan
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Propose_global_plan
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Propose_global_plan
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Accept_global_plan
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Accept_global_plan
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Request_alternative_execution
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Request_alternative_execution
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Inform_local_execution_result
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Inform_local_execution_result
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Inform_global_execution_result
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Inform_global_execution_result
```

Figure 5.37: Framework: Acceptable plan results 2

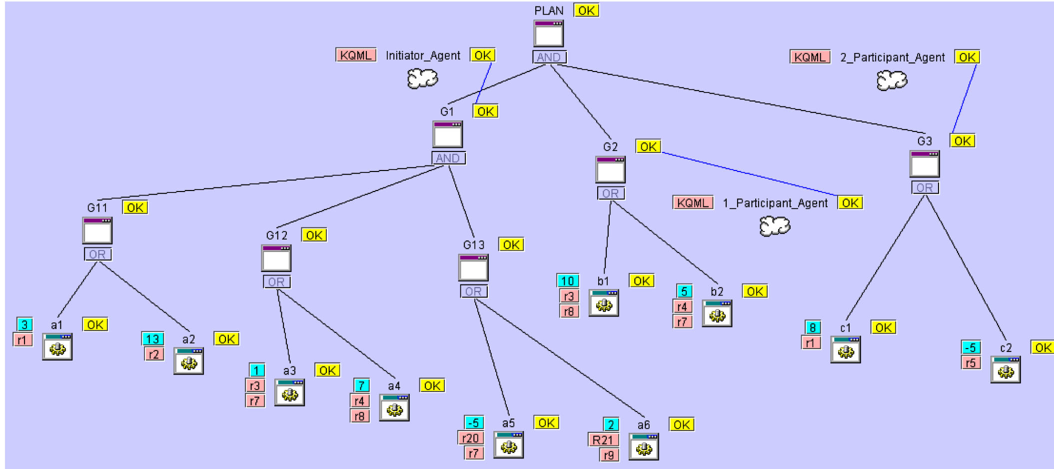

Figure 5.38: Framework: Rejectable plan

5:34:53 InitiatorBehaivour.runRefuse:-1	<------------------------------------->
5:34:53 InitiatorBehaivour.decodeProposal:-1	·
5:34:53 InitiatorBehaivour.runRefuse:-1	· Plan:
5:34:53 InitiatorBehaivour.runRefuse:-1	· Rejector: '2_Participant_Agent'
5:34:53 InitiatorBehaivour.runRefuse:-1	Global plan proposal has been rejected
5:34:53 InitiatorBehaivour.runRefuse:-1	<------------------------------------->

```
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Call_for_proposals
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Call_for_proposals
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Propose_action_plan
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Propose_action_plan
message(INITIATOR_AGENT,1_PARTICIPANT_AGENT,"Propose_global_plan
message(INITIATOR_AGENT,2_PARTICIPANT_AGENT,"Propose_global_plan
message(1_PARTICIPANT_AGENT,INITIATOR_AGENT,"Reject_global_plan
message(2_PARTICIPANT_AGENT,INITIATOR_AGENT,"Reject_global_plan
```

Figure 5.39: Framework: Rejectable plan results

5.6 Use case test scenario

This section presents a real test case based on the example presented on *Section 4.2*. The section presents the graphical plan corresponding to the example, and a test, providing an example on how the scenario presented in *Section 4.2* can be modeled using the planning framework implemented. The plan defined can be seen on *Figure 5.40* and corresponds to the tables *Table 5.3*, *Table 5.4* and *Table 5.5*.

Table 5.3: Alternatives proposed by each broker

Broker	Activities
Amusement and mobility	*{Port_Aventura_Private, Tibidabo_Private, Aquarium_Private}* *Port_Aventura_Public, Tibidabo_Public, Aquarium_Public*
Cinema	*{The_hobgoblin_Soon, The_hobgoblin_Medium, The_hobgoblin_Late,* *P._movie_Soon, P._movie_Medium, P._movie_Late}*
Restaurant	*{Le_Remanguille_Soon, Le_Remanguille_Late,* *Casa_Pepe_Soon, Casa_Pepe_Medium, Casa_Pepe_Late,* *BurryKing_VSoon, BurryKing_Soon, BurryKing_Medium,* *BurryKing_Late, BurryKing_VLate}*

Table 5.4: Resources of each alternative

Activity	Resources	Activity	Resources
Port_Aventura_Private	{8.00 − 18.00}	*Tibidabo_Private*	{8.00 − 16.30}
Aquarium_Private	{8.00 − 14.00}	*Port_Aventura_Public*	{8.00 − 19.00}
Tibidabo_Public	{8.00 − 17.30}	*Aquarium_Public*	{8.00 − 15.30}
The_hobgoblin_Soon	{16.00 − 19.30}	*The_hobgoblin_Medium*	{17.00 − 20.30}
The_hobgoblin_Late	{18.30 − 22.00}	*P._movie_Soon*	{17.00 − 18.30}
P._movie_Medmium	{18.30 − 20.00}	*P._movie_Late*	{19.30 − 21.00}
Le_Remanguille_Soon	{19.30 − 21.30}	*Le_Remanguille_Late*	{20.00 − 22.00}
Casa_Pepe_Soon	{20.00 − 21.00}	*Casa_Pepe_Medium*	{20.30 − 21.30}
Casa_Pepe_Late	{21.00 − 22.00}	*BurryKing_VSoon*	{19.30 − 20.00}
BurryKing_Soon	{20.00 − 20.30}	*BurryKing_Medium*	{20.30 − 21.00}
BurryKing_Late	{21.00 − 21.30}	*BurryKing_VLate*	{21.30 − 22.00}

Table 5.5: Utilities of each activity

Activity	Resources	Activity	Resources
Port_Aventura_Private	20	*Tibidabo_Private*	15
Aquarium_Private	4	*Port_Aventura_Public*	15
Tibidabo_Public	10	*Aquarium_Public*	7
The_hobgoblin_Soon	30	*The_hobgoblin_Medium*	19
The_hobgoblin_Late	1	*P._movie_Soon*	15
P._movie_Medmium	8	*P._movie_Late*	3
Le_Remanguille_Soon	20	*Le_Remanguille_Late*	9
Casa_Pepe_Soon	15	*Casa_Pepe_Medium*	7
Casa_Pepe_Late	2	*BurryKing_VSoon*	5
BurryKing_Soon	2	*BurryKing_Medium*	0
BurryKing_Late	−2	*BurryKing_VLate*	−5

Running the framework with this information throws the results shown on *Figure 5.41*. As a summary, the actions performed are:

1. Initiator agent sends a *Call for proposals* message to all participant agents.
2. Each Participant agent replies by proposing an action plan. Initiator agent waits until all replies have been received.
3. Initiator agent puts the action plans together in a global plan proposals. As it is conflict-free Initiator agent sends the proposals to each Participant agent.
4. Each Participant agent accepts the proposal. Initiator agent waits until all replies have been received.
5. Initiator agent scans the proposal looking for the alternative with the higher global utility. Then, requests the execution of the alternative to all Participants.
6. Each Participant agent performs its part of the alternative, then informs Initiator agent. Initiator agent waits until all informs have been received.
7. Initiator agent finishes performing its part of the alternative, then informs each Participant the global plan has been fulfilled.

Notice that, in order to ease the process of assigning resources to actions, the class parsing the plan definition file (that is, *DecafPlanParser*) is extended to class *DecafPlanParserResourceInterval* that interprets resources assigned to actions (typically 2) as an interval of resources rather than as single resources.

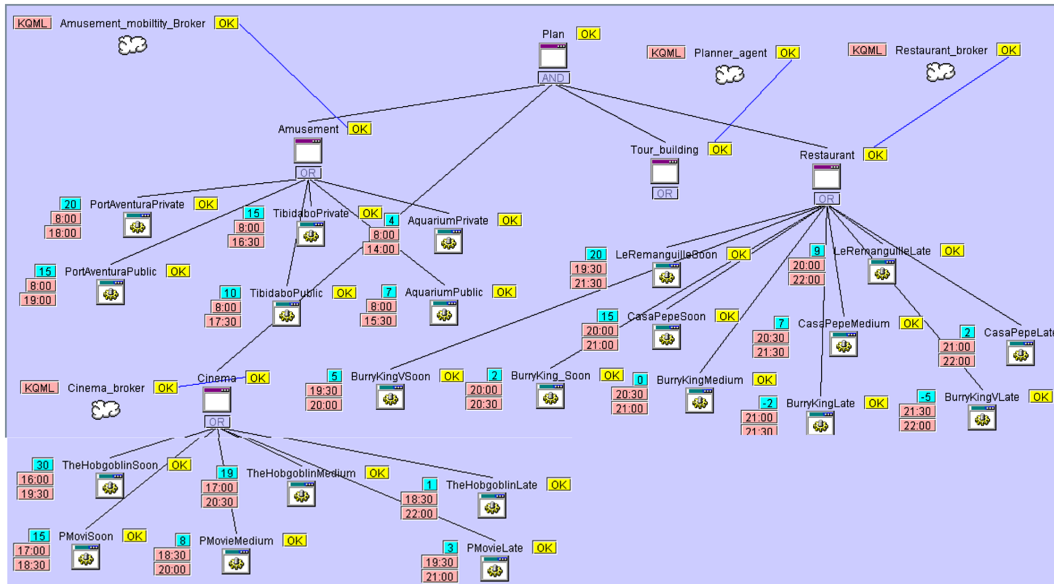

Figure 5.40: Framework: Real test plan

4:24:22	ThParseSchema.run:-1	http://www.ist-contract.org/schemas/ISTContract.xsd parsed and co
4:24:18	ParticipantBehaivour.runInform:-1	<-------------------------------------->
4:24:18	ParticipantBehaivour.decodeProposal:-1	· AquariumPublic_TheHobgoblinSoon_CasaPepeSoon
4:24:18	ParticipantBehaivour.runInform:-1	·Plan:
4:24:18	ParticipantBehaivour.runInform:-1	The plan has been succesfully executed
4:24:18	ParticipantBehaivour.runInform:-1	<-------------------------------------->

```
message(PLANNER_AGENT,AMUSEMENT_MOBILTITY_BROKER,"Call_for_proposals
message(PLANNER_AGENT,CINEMA_BROKER,"Call_for_proposals
message(AMUSEMENT_MOBILTITY_BROKER,PLANNER_AGENT,"Propose_action_plan
message(PLANNER_AGENT,RESTAURANT_BROKER,"Call_for_proposals
message(CINEMA_BROKER,PLANNER_AGENT,"Propose_action_plan
message(RESTAURANT_BROKER,PLANNER_AGENT,"Propose_action_plan
message(PLANNER_AGENT,AMUSEMENT_MOBILTITY_BROKER,"Propose_global_plan
message(PLANNER_AGENT,CINEMA_BROKER,"Propose_global_plan
message(PLANNER_AGENT,RESTAURANT_BROKER,"Propose_global_plan
message(AMUSEMENT_MOBILTITY_BROKER,PLANNER_AGENT,"Accept_global_plan
message(RESTAURANT_BROKER,PLANNER_AGENT,"Accept_global_plan
message(CINEMA_BROKER,PLANNER_AGENT,"Accept_global_plan
message(PLANNER_AGENT,AMUSEMENT_MOBILTITY_BROKER,"Request_alternative_execution
message(PLANNER_AGENT,CINEMA_BROKER,"Request_alternative_execution
message(PLANNER_AGENT,RESTAURANT_BROKER,"Request_alternative_execution
message(CINEMA_BROKER,PLANNER_AGENT,"Inform_local_execution_result
message(AMUSEMENT_MOBILTITY_BROKER,PLANNER_AGENT,"Inform_local_execution_result
message(RESTAURANT_BROKER,PLANNER_AGENT,"Inform_local_execution_result
message(PLANNER_AGENT,AMUSEMENT_MOBILTITY_BROKER,"Inform_global_execution_result
message(PLANNER_AGENT,CINEMA_BROKER,"Inform_global_execution_result
message(PLANNER_AGENT,RESTAURANT_BROKER,"Inform_global_execution_result
```

Figure 5.41: Framework: Real test plan results

5.7 Summary

In this chapter we have presented a practical implementation of the planning framework introduced in *Chapter 4* under the PAWS agent platform introduced in *Chapter 3*.

The chapter has started by explaining in depth the components of the PAWS platform the implementation will deal with. It includes the protocol that supports plan negotiation between agents, and the planning module that supports agents on decision making, stating if a given plan proposal is acceptable (i.e. conflict-free) or not.

Regarding the protocols, an overview of the INGENIAS IDK is provided, focusing on how to model the protocols via the *Interaction Model*. The chapter gives a complete explanation on how to model the interaction protocol using INGENIAS, and how to generate and adapt PAWS code from the INGENIAS model. Each of the components generated from the INGENIAS model is explained in depth, providing examples based on the negotiation protocol implemented. The components explained include the protocol file, the behaviors, the message types, the roles and the ontology. Once the reader has gone over this explanation, he/she should be able to design his/her own protocols using INGENIAS and easily implement them in the PAWS platform.

The chapter also includes a deep explanation of the process that generates PAWS code from the INGENIAS specification, with the aim of providing a better and deeper understanding of the *protocol modeler* component.

Regarding the planning module implemented in the PAWS platform, the chapter has provided a deep explanation about how the module has been implemented, connected to other PAWS components (such as the ontology and the *directory facilitator*) and integrated on the behaviors. This last step implies profound changes on the behavior files, which has been explained in detail. We hope this provides reader with a better understanding on how PAWS behaviors are structured and implemented.

Then, the chapter has introduced the *plan definition* module. This includes a deep explanation on how to use the DECAF plan editor GUI for defining GPGP plans.

To finalize, the chapter has introduced two test cases. A simple one, and one based on the example provided on *Chapter 4.2*. For each of the examples, the plans defined via the *plan definition* module and the results provided by the implemented framework (showing both the results and the negotiation process between the agents) are provided.

Chapter 6

Conclusions

This chapter presents the conclusions extracted from the development of the work presented in this master thesis. It starts by analyzing if the objectives presented in *Chapter 1* have been met. The chapter goes on with a summary of the original contributions. Finally the chapter gives an overview about how the work could be continued, how could it be improved and how new lines of work can arise from it.

6.1 Conclusions

The main results exposed on this master thesis are:

- Development of the paws platform. The process of development includes an analysis of the original CONTRACT platform and explanation of its components, functionalities and the connections between them. This provides an overview on how agent platforms oriented to hosting agentified web-services look like. An analysis of the components that have been developed and modified in order to develop the PAWS platform is also included.
- Development of a theoretical planning framework that has the novel feature of using Argumentation Theory for detecting conflicts between available plans. The framework, based on the negotiation of a global plan from a set of local plans between a set of intelligent agents, shows two more concepts that are not commonly seen in other planning frameworks. The use of utilities for pondering plans, and the ability to deal with resource constraints when generating the global plan.
- Integration of the theoretical planning framework with the PAWS agent platform. This is the nexus between the previous ideas of the master thesis. On the one hand, it provides an implementation of the theoretical framework, on the other hand it provides an empirical proof that the platform is general.

In order to achieve these results, some analysis over the relevant state of the art on the following fields has been performed:

- Analysis of the state-of-the art in generic agent platforms, focusing on the FIPA standard for building them.
- Analysis of the state-of-the art on agent design methodologies. Analysis of the INGENIAS methodology, focusing on *interaction model*. Analysis of the INGENIAS IDK to find out how protocol definitions could be transformed to PAWS code.

Generalizing the CONTRACT platform in order to create the PAWS platform has been a thought task, because the platform was strongly coupled to contract-management purposes and some of the internal components of the architecture (which are not very well documented) have been modified. Even then, the effort has payed, because being based on the CONTRACT platform the PAWS platform presents a solid and simple solution for agentifying services. At the same time, this process has provided me with knowledge on how to generalize agent platforms (which requires a deep knowledge of agent platforms) and results in a guide on how to perform these generalizations. As these guides are not very common on multi-agent systems literature, I hope it is useful for researchers interested in applying generalization processes to other agent architectures.

At the same time, implementing some support components that make the platform more usable (such as the *protocol modeler*) has provided me notions on the INGENIAS methodology and a deep understanding of the INGENIAS IDK software, specially regarding the *interaction structure* part.

Integrating the planning module on the PAWS platform fulfills the purpose of providing a coordination mechanism for transforming local plans into global plans, but at the same time, as a residual result, provides the empirical demonstration that the PAWS platform has been generalized, as planning has little to do with contract-management.

As explained later in *Section 6.3* further improvements can be made, but I consider the objectives of this Master Thesis have been met.

6.2 Summary of original contributions

This master thesis presents two main contributions to the field of Artificial Intelligence.

- The development of the PAWS platform, an agent platform for agentified web-services. Agent platforms for agentified services are cutting edge research issues in the multi-agent systems world. AgentScape [31] and WSIG (Jade Web service Integration Gateway) [16] are good examples of such platforms. The current trend in platforms for agentified web-services consists in providing an agent-platform and an interface (for instance, a gateway) that can be used by agents to invoke the services. Thus, two separate platforms are provided, one for the agents, and another one for the services, and users have to administrate and maintain both of them. PAWS presents a novel approach, integrating both agents and services on the same platform. Two benefits arise from this integration: first of all, users have to take care of just one platform. This makes the process of administration, maintenance and even development easier. What's more a single platform will make use of less computational resources (memory, disk, CPU, etc.) on the host it is deployed on. Second, thanks to this integration, not only agents can make use of services, but also services can make use of agents and other components on the platform. Agents and components in the platform can publish services exposing their capabilities, then web-services can invoke them (just like they would invoke another web-service) to use them. An example of this use is a web-service that wants to be agentified. All it has to do is invoking the platform asking to create an agent, that will effectively agentify the service.

- The theoretical definition and the practical implementation of a planning framework that presents the novel contribution of using 'Argumentation Theory' for detecting non acceptable plans. This provides an alternative to existing approaches, and at the same demonstrates the versatility of Dung's Argumentation theory, showing it can be applied to research fields which are miles away from it, like planning systems. It also

contributes to the field of planning systems for resource constrained actions, which is a field that has not received much attention taking into account the research efforts invested in more general planning systems.

6.3 Future Work

Future work on this master thesis can be divided on three main points.

- Future improvements on the PAWS platform.
- Future improvements on the theoretical planning framework.
- Applying the PAWS platform and the theoretical argumentation framework to other scenarios.

6.3.1 Future improvements on the PAWS platform

The PAWS platform presents some room for improvement, specially when comparing it to more mature agent platforms. Weak points detected on the platform include:

- The platform lacks an operative and powerful sniffer, to allow PAWS users to check the execution flow of the agents on the platform, in order to identify (and fix) unexpected or incorrect behaviors. It would be desirable if this sniffer could also include debugging of the invoked services, in case they are coded by the same programmers of the agents. The sniffer currently provided with the platform is a very simple preliminary version.
- Some of the components of the platform (such as the *directory facilitator* and the *ontology manager*) are very simple. They support only static information (read when the platform starts-up) and should be improved in future versions of the platform. Specially, regarding the *directory facilitator*, an agent discovery service should be implemented and integrated with it.
- Even though the documentation on how to use the platform, code the agents and attach new modules to the platform is enough, documentation on the internal components of the platform is not extensive enough. This documentation should be improved, in case a user needs to change an internal component for adapting the platform, or to fix a bug.
- The connection between the INGENIAS IDK and the PAWS platform for protocol modeling is performed via an external meta-code generator. As INGENIAS allows developing new modules, attaching them to the IDK, this program should be coded as an INGENIAS module, fully integrating PAWS code generation features in the INGENIAS IDK.

The platform is highly oriented to supporting agentified services. Further work on the platform should include remarking this feature as it makes PAWS different from other agent platforms oriented to more general purposes. In order to focus the platform even more on service agentification, new modules should be implemented. This could include modules for service discovery or for interpreting the semantic annotations (i.e. OWL-S annotations) of the services, in order to enrich semantically the information provided by the service discovery component.

It is also planned to integrate into the platform a module for using more complex reasoning languages, in case the simple reasoning behaviors implemented in Java are not enough for some domains. This task plans integrating both 2APL and JASON agent languages in the PAWS platform.

The last work planned regarding the PAWS platform is trying to encapsulate in the agents REST services, as currently the only services available for the agents to invoke are SOAP services.

6.3.2 Future improvements on the theoretical planning framework

Future improvements on the theoretical planning framework presented in this master thesis include extending the framework with features other planning approaches already present. Such extensions should include:

- Negative goals. Representing sets of actions to be avoided. This would complement the use of utilities that is already present in the current definition of the framework.
- Negative resources. Representing the fact that performing an action will generate a resource rather than consume it. It can be used to link two actions, one with the negative resource, and the other with the positive one. This will represent the fact that the action consuming the resource cannot be performed if the action generating the resource is not performed before. Integrating this into the attack relations modeled via the argumentation framework is a challenging issue.
- Actions appearing in several local plans. This would effectively model actions shared between a set of agents. The negotiation protocol should be complemented with a sub-negotiation protocol to decide which of the agents sharing the action are to enact it.

Apart from that, the implementation of the framework provided with the PAWS platform should include its own plan editor, rather than re-using the plan editor provided with DECAF software.

Finally, a deeper study (e.g. via simulations) of the improvement obtained by using a global utility function to evaluate plans rather than a local one is to be performed.

6.3.3 Applying the PAWS platform and the theoretical argumentation framework to other scenarios

The modular architecture of the PAWS platform makes it useful for adding intelligent features to service composition. For instance, a module for Case-Based reasoning could be included in the platform for providing more efficient service composition. When composing services for fulfilling a given task, the Case-Based reasoning module can be queried. If a good enough match for the case is found, the case can be used as the resulting composition. Otherwise, the planning module can be used to generate a composition, storing it as a case. Thus, service composition would be very efficient if compositions for very similar tasks are to be performed often, because then, compositions are not to be generated from scratch, just retrieved from a base of cases. As future work, we have to try to apply the results of this thesis to the mentioned scenario.

Another line of research regarding intelligent service composition can be found in the theoretical planning framework introduced in this master thesis. The framework takes into account concepts such as utilities and resources assigned to actions. If these actions map to

services the utilities of the actions can be mapped to the trust (local utilities) and reputation (global utilities) agents have towards the services. That is, once an agent in the domain has invoked a service, it can share its own feedback about the service (regarding Quality of Service provided or availability) with all the agents in the domain. This will result in more efficient and reliable services being invoked more. Regarding resources, they can be assigned to services based on their load. For instance, no resources would be assigned to idle services, 5 medium-load resources distributed among medium-loaded services, and a single high-load resources assigned to busy services. The task of deciding how loaded is a service can be performed by a module attached to the PAWS platform for this purpose, using for instance a clustering algorithm that classifies a service in iddle, medium or high-loaded class based on the state of service's host (e.g. percentage of free memory, CPU load, etc.). Once all services have been assigned resources based on their load, the planning framework will ensure that each workflow provided as solution will contain only one busy service, at most five medium-loaded services and any number of idle services. This will ensure a minimum level of efficiency on the workflows enacted by the agents, as they will not contain too many invocations to over-loaded services. We find this idea very appealing and plan to investigate it further on the future.

The work presented in this master thesis can also be relevant for the ALIVE framework.

The ALIVE framework (introduced in *Appendix A*) presents several interesting issues ready to be tackled. Among them is the process of coordination between agents to allow collaborative workflow execution. As it is remarked in *Remark A.1*, several alternative sets of actions are available to make the system go from the state represented by a landmark to the state represented by another one. Thus, the agents enacting these sets of actions must decide which set to take when making the system pass from one state to another, advancing in the achievement of the objectives in the organization. This issue would be rather simple if the decision had to be taken by a single agent. However, as explained above, agents coordinate in order to execute the workflows in parallel, collaborating when executing them. Thus, it is not a single agent that has to decide which set of actions to execute, but a set of agents (the set of agents that will collaborate in the workflow) that have to agree on which set of actions are to be performed.

The planning framework presented in this master thesis can be used to tackle this issue, as it allows coordination between agents, whereas the PAWS platform can be used for hosting the agents at the coordination level, as they are required to invoke services on the service level and coordinate among them.

The mapping between the planning framework presented in *Chapter 4* and the ALIVE workflows can be as follows: Local workflows can be modeled by local plans, and the global plan resulting of the planning process will be the collaborative workflow to be enacted by the set of agents. Utilities can be used to model the trust (local utility) and reputation (global utility) of the agents towards a given task in a workflow. Regarding resources, they can be left unassigned, or can be assigned by the organizational level in order to model normative or organizational constraints. For instance, if two given tasks cannot be enacted together on the same workflow, all that is to be done is assigning them the same resource. Conflict-freeness condition guaranteed by the planning framework will prevent them to be present on the same workflow (i.e. global plan).

Appendix A

The ALIVE Framework

This appendix provides an overview of the ALIVE[1] framework [1].

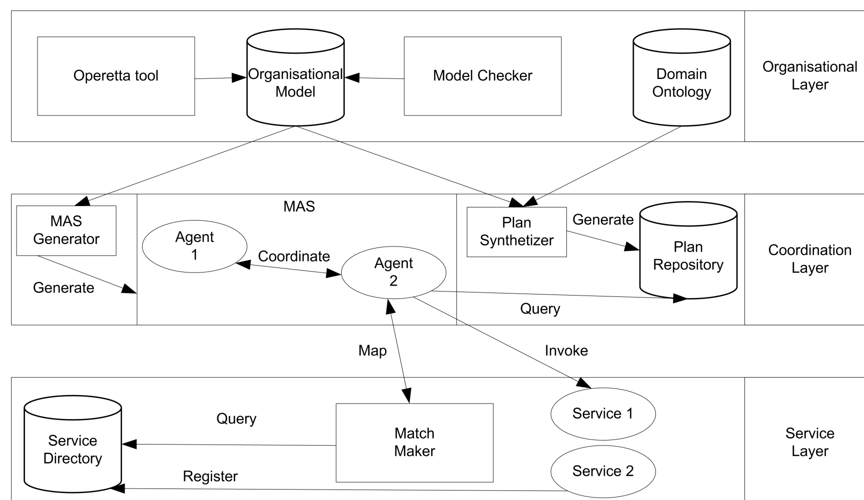

Figure A.1: ALIVE architecture diagram

The ALIVE framework is being developed in collaboration with several universities and enterprises within the frame of the FP7 project ALIVE (ICT-215890) funded by the European Commission. It combines Model Driven Design (MDD) and Agent-Based System Engineering with agent-based coordination and organizational mechanisms, providing support for 'live' (that is, highly dynamic) and open systems of services. The work presented in this document focuses on a part of the coordination mechanisms developed for the ALIVE project. ALIVE's multi-level approach helps to design, deploy and maintain distributed systems by combining, reorganizing and adapting services. ALIVE defines a multi-layered architecture divided in three levels (see *Figure A.1*) with connection between adjacent levels.

[1]For more detailed information on the ALIVE framework please see `http://www.ist-alive.eu/index.php?option=com_docman&task=doc_download&gid=7&Itemid=49`

A.1 Organizational Level

The Organizational Level provides an explicit representation of the organizational structure of the system. Stakeholders and their relations are represented, together with formal goals, requirements and restrictions.

The Organizational Model is the main component of the Organizational Level, representing the organization as a social system. This model is inspired in the one presented in Opera methodology [7]. The model includes *objectives*, the common goals for which the organization is created. It also includes *roles*, abstract groups of activity types identifying the necessary activities to achieve the objectives.

Objectives are assigned to roles, and relationships between pairs of roles (known as parent and children from now on) define the links through which objectives can be delegated. The set of all roles and the relations among them form the Social Structure. Three kinds of relations between roles are contemplated on ALIVE's Organizational Model:

- hierarchical relation: where a parent role delegates some of its objectives to its children.
- market relation: where children roles can request the assignment of objectives from the parent role.
- network relation: where both roles are authorized to request the objectives of the other.

ALIVE's Organizational Model also defines landmarks, that are important states in the achievement of a goal, and landmark patterns, that impose an ordering over landmarks. This will effectively define the order in which landmarks should be reached. A set of landmarks and their relations is known as scene. Relations between scenes (i.e. scene transitions) can also be included in the model. Via scene transitions, scenes can be organized into an Interaction Structure, enabling the representation of complex interactions.

The Organizational Level supports methods of design based on norms , rights and obligations of the actors in the organization. Norms complement the elements of the model introduced before, and can be applied to highly regulated scenarios where other approaches do not fit well.

The Operetta Tool [8] supports system designers in specifying and visually analyzing the Organizational Model, whereas, the Model Checker verifies its consistency. The Domain Ontology captures a shared understanding of certain aspects of the domain, providing a common vocabulary along with important concepts and their properties, definitions and constraints (intended meaning) in order to describe the domain knowledge, encoding it via an ontology language. Off-the-shelf ontology editors, such as Protege [26], can be used to define this ontology. Elements defined in the domain knowledge via the ontology can include, for instance, the abstract tasks available for agents to execute, or the resources consumed by each of this tasks. The Organizational Model is sent to the Multi-agent system (MAS) Generator component in the Coordination Level to generate the agents that will populate the MAS. One agent per role defined in the model is generated. The Organizational Model along with the Domain Ontology is sent to the Plan Synthesizer in order to generate the plans the agents will enact.

A.2 Coordination Level

The Coordination Level provides the patterns of interaction among actors, transforming the representations of the Organizational Level into coordination plans known as workflows.

Figure A.2: ALIVE workflow example

Workflows are defined as Generalized Partial Global Planning (GPGP) plans and chains of abstract TAEMS hierarchical tasks [18]. Tasks are semantically annotated, and contain both pre-conditions and post-conditions that describe the state in which the system is before and after the task is performed respectively.

The following statement defines a workflow: for any task in the workflow, if it has a preceding task, the pre-condition of the task is the post-condition of the preceding task. Otherwise, the pre-condition of the task is the state represented by the preceding landmark. The same stands for the succeeding task: the pre-condition of the succeeding task is the post-condition of the task, if no succeeding task exists, the post-condition of the task is the state represented by the next landmark.

Definition A.1 *Let the set of actions $\langle A_1, A_2, ..., A_N \rangle$ denote a workflow, and L_1, L_2 be two landmarks. Let $precondition(A_i)$ and $postcondition(A_i)$ denote the precondition and postcondition of a given action A_i respectively and let $state(L_i)$ denote the state represented by a given landmark L_i.*

$$\forall i \ 0 \geq i \geq N-1$$
$$((\exists j \ 0 \geq i \geq N-1 : precondition(A_i) = postcondition(A_j) \wedge i \neq j) \vee (precondition(A_i) = state(L_1))$$
$$\wedge$$
$$((\exists j \ 0 \geq i \geq N-1 : postcondition(A_i) = precondition(A_j) \wedge i \neq j) \vee (postcondition(A_i) = state(L_2))$$

Corollary A.1 *If $\langle A_1, A_2, ..., A_N \rangle$ is a workflow, given the landmarks L_1, L_2 and according to the the previous definition, then it can be stated that the worklow $\langle A_1, A_2, ..., A_N \rangle$ connects the landmarks L_1, L_2. In other words, enacting the workflow when the system is on the state represented by the first landmark (i.e. $state(L1)$) will bring the system to the state represented by the second one (i.e. $state(L2)$).*

Example A.1 *Figure A.2 shows an example of a workflow.*

Remark A.1 *Notice that several workflows might be available for connecting two landmarks. Notice also that, even inside the same workflow, alternative sets of actions (that is, alternative paths) might be available. For instance, in the example provided in Figure A.2, enacting the set of actions $\langle 1, 21, 31, 4 \rangle$ or the set of actions $\langle 1, 22, 32, 4 \rangle$ will result*

in enacting the workflow. Thus, it can be deduced that, in order to bring the system from a state to another state (that is, from landmark to landmark) different alternative sets of actions are available.

Workflows are derived from information defined on the Organizational Model. It must be noticed that, for efficiency, workflows are generated and kept in a persistent database (from where they can be retrieved when required) rather than generated 'on the fly' when they are to be enacted.

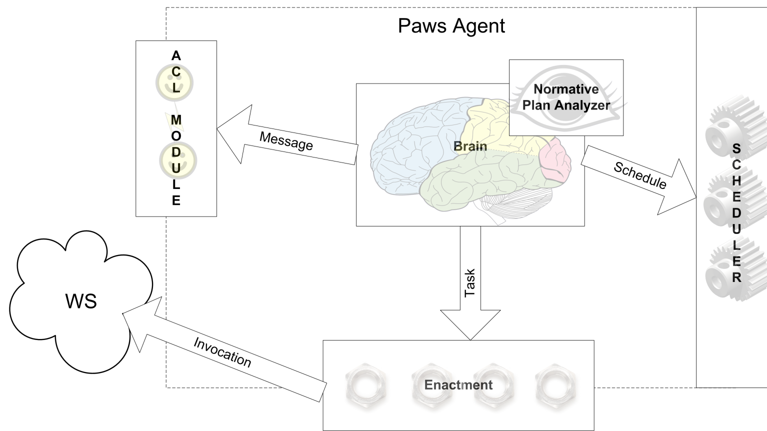

Figure A.3: ALIVE agent architecture

The MAS is a set of intelligent agents that coordinate among them to enact the workflows in a distributed fashion. Making intelligent agents enact the workflows has two main advantages. First of all it allows choosing the workflows preferred by the organization, if several are available. On the other hand, the agents analyze and monitor workflow execution, reacting to unexpected events, either by enacting other workflows, or by communicating the incident to other levels. What's more, agents can perform complex tasks that require intelligent reasoning. For instance, in order to model organizational constraints (such as Norms and restrictions) when enacting the workflows, agents include an intelligent component that can foresee if executing an available workflow will violate any of these constraints, deciding if it is worth running it despite the violation.

A schema of the agent's architecture can be seen on *Figure A.3.* This figure depicts the main components of ALIVE's agents. ALIVE's agents include the following components: the Brain Module is the core of the agent, and provides reasoning and decision-making capabilities; the Normative Plan analyzer is a part of the Brain Module. It scans the workflows in order to determine if enacting them will violate any of the norms defined in the organizational model; the ACL Module provides agents with the capability of communicating with other agents in the system by sending messages to them; the Scheduler component provides an interface from where the agents can coordinate and distribute tasks, enabling parallel execution of workflows; finally, the Enactment component facilitates the invocation of services, making the details of the invocation transparent to the agents.

A.3 Service Level

The Service Level selects an appropriate service for a given abstract task by using the semantic information contained both in the service description and in the task description. These descriptions facilitate the process of composing services and finding alternative services when a given service is not available. The Match Maker component receives an abstract task description from an agent in the Coordination Level and looks for services that can fulfill this task. The Match Maker component queries the Service Directory to go through all available services, selecting the most appropriate one, based on the task's semantic description and on Quality of Service parameters (such as average response time) that allow to choose the best service if several of them match the semantic query. Then, the service chosen is returned to the agent, and the task is executed and monitored. If a given service is no longer available (or is not the most appropriate anymore, due to better services entering the system), the Service Level is able to re-assign services to tasks "on the fly" without the Coordination Level noticing the change. The Service Level is also able to perform simple service composition; for instance, if a given task requires sending cinema information to a user via SMS, and there are two available services, one which retrieve cinema information and one which send SMS, the Service Level is able to bind the task to a composition of these two services.

List of Figures

2.1 Agent-directory UML relationships elements 9
2.2 Transport UML relationships element 13
2.3 Message Interchange Flow Diagram 14
2.4 Agent UML relationships element 16
2.5 FIPA Agent Platform Diagram 17
2.6 Agent Life Cycle Diagram 18

3.1 IST-CONTRACT project Agent architecture overview diagram 22
3.2 CONTRACT Agent components, hierarchical view 24
3.3 Decision maker interface 25
3.4 Contract manager interface 25
3.5 Workflow manager interface 26
3.6 Communication manager interface 27
3.7 Dialogue manager interface 27
3.8 Message manager interface 28
3.9 PAWS Agent architecture overview diagram 30
3.10 CONTRACT and PAWS agent configuration files example 31
3.11 PAWS Agent components example, hierarchical view 32
3.12 Module integration on behavior example 35
3.13 CONTRACT's contract storer and PAWS' generic storer 38
3.14 Protocol modeler architecture 40

4.1 Planning problem representation example 45
4.2 Argumentation Framework Example 47
4.3 Scenario architecture Description 48
4.4 Workflow Diagram 49
4.5 Basic protocol diagram 65

5.1 Diagram of protocol's classes 70
5.2 Diagram of decision making support classes 70
5.3 Coordination protocol High level design 72
5.4 Coordination protocol, INGENIAS design: Roles 73
5.5 Coordination protocol, INGENIAS design: Responsabilities 73
5.6 Coordination protocol, INGENIAS design: Precedences example 74
5.7 Coordination protocol, INGENIAS design: Precedences 75
5.8 Coordination protocol, INGENIAS general design 75
5.9 INGENIAS and PAWS: Stub generation process 76
5.10 INGENIAS and PAWS: Protocol stub example 77

5.11 PAWS Protocol Stub: Run simple percept function 79
5.12 PAWS Protocol Stub: Update state of protocol on message reception 79
5.13 PAWS Protocol Stub: Detection of protocol's final state 80
5.14 PAWS Protocol Stub: Detection of protocol's non-final state 80
5.15 INGENIAS and PAWS: Message types generated 81
5.16 INGENIAS and PAWS: Role code stub example 82
5.17 INGENIAS and PAWS: Role code stub example 83
5.18 INGENIAS parsing: Message structure 83
5.19 INGENIAS parsing: Role structure 83
5.20 INGENIAS parsing: Protocol structure 84
5.21 INGENIAS parsing: Message precedence structure 84
5.22 INGENIAS parsing: Message-role collaborates responsibility structure 84
5.23 INGENIAS parsing: Message-role initiates responsibility structure 85
5.24 PAWS Stub: Initator sending message to multiple participants 88
5.25 PAWS Stub: Initializing participant's structure via Agent Directory 88
5.26 PAWS Stub: Updating state of protocol, multiple participants 89
5.27 PAWS Stub: Sending reply to a message, multiple participants 89
5.28 PAWS Stub: Initializing the Ontology 92
5.29 PAWS Generic Framework: Conflict-free check 93
5.30 PAWS Plan GUI: Main window 94
5.31 PAWS Plan GUI: Pop-up window 95
5.32 Framework: Plan acceptance protocol path 97
5.33 PAWS Plan GUI: Plan refusal protocol path 97
5.34 Framework: Acceptable plan 1 98
5.35 Framework: Acceptable plan results 1 98
5.36 Framework: Acceptable plan 2 99
5.37 Framework: Acceptable plan results 2 99
5.38 Framework: Rejectable plan 100
5.39 Framework: Rejectable plan results 100
5.40 Framework: Real test plan 102
5.41 Framework: Real test plan results 102

A.1 ALIVE architecture diagram 111
A.2 ALIVE workflow example 113
A.3 ALIVE agent architecture 114

List of Tables

4.1 Activities proposed by each broker 50
4.2 Resources of each alternative 51
4.3 Existing alternatives 52
4.4 Utilities of each activity 56
4.5 Agents in the system 58
4.6 Utility values for the alternatives per agent 63
4.7 Utility values of complemented alternatives for agents and global utility values 64

5.1 PAWS-INGENIAS performative mapping 81
5.2 Example plan 95
5.3 Alternatives proposed by each broker 100
5.4 Resources of each alternative 101
5.5 Utilities of each activity 101

Bibliography

[1] EU-ALIVE project Home page
(http://www.ist-alive.eu/) *Visited 2009-August-23*

[2] Casati F., Ilnicki S., Jin L. 'Adaptive and Dynamic Service Composition in eFlow', In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, 2000

[3] IST-CONTRACT project Home page
(http://www.ist-contract.org/) *Visited 2009-June-12*

[4] CONTRACT Platform definition document
(`http://www.ist-contract.org/index.php?option=com_docman&task=doc_download&gid=13&Itemid=44`) *Visited 2009-June-12*

[5] Decker K. and Li J. 'Coordinating Mutually Exclusive Resources using GPGP', In *Autonomous Agents and Multi-Agent Systems, Volume 3*, page 133-157, 2006 - Jan -01

[6] Decker K., Lesser V. 'Generalizing the partial global planning algorithm'. IN *Int. J. Intell. Cooperative Inf. Syst.*, pages 319-346 Volume 1, 1992

[7] Dignum V. 'A Model for Organizational Interaction: based on Agents, founded in Logic'. IN *SIKS Dissertation Series 2004-1*. Utrecht University, PhD Thesis. 2004

[8] Dignum V., Okouya D. 'OperettA: a prototype tool for the design, analysis and development of multi-agent organizations'. IN *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers*. Pages 1677-1678 , Portugal, 2008

[9] Dung P.M. 'On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games', 1994

[10] Durfee E.H. 'Planning in Distributed Artificial Intelligence' In *Foundations of distributed artificial intelligence*, Chapter 8, 1996

[11] eXists project home page.
(http://exist.sourceforge.net/) *Visited 2009-August-20*

[12] Factory Design Pattern (http://en.wikipedia.org/wiki/Factory_method_pattern) *Visited 2009-March-12*

[13] FIPA specifications page (http://www.fipa.org/specifications/index.html) *Visited 2009-March-12*

[14] Gomez-Sebastia I., Manel Palau, Juan Carlos Nieves, Javier Vazquez-Salceda and Luigi Ceccaroni 'Dynamic orchestration of distributed services on interactive community displays: the ALIVE approach', In *Advances in intelligent soft computing, Vol. 55*, Jan. 2009, pages 450-459

[15] Graham J.R., Decker K.S. and Mersic M. 'DECAF - A Flexible Multi Agent System Architecture ', In *Autonomous Agents and Multi-Agent Systems, Vol. 7 Numbers 1-2*, July 2003, pages 7-27

[16] Greenwood D. 'JADE Web Service Integration Gateway (WSIG)', JADE tutorial on AMAAS 2005.

[17] INGENIAS project Home page
(http://grasia.fdi.ucm.es/main/?q=en/node/127) *Visited 2009-June-12*

[18] Lesser V. et al. 'Evolution of the GPGP/TMS Domain-Independent Coordination Framework' In *Autonomous Agents and Multi-Agent Systems*, pages 87-143 Volume 9, 2004-07-01.

[19] Martin D. et al. 'Bringing Semantics to Web Services: The OWL-S Approach' In *Semantic Web Services and Web Process Composition*, page 26-46, 2005

[20] Introduction to non monotonic logics
(http://plato.stanford.edu/entries/logic-nonmonotonic/) *Visited 2009-June-12*

[21] Nguyen G. et al. 'AGENT PLATFORM EVALUATION AND COMPARISON', In *Advances in intelligent soft computing, Vol. 55*, Jun. 2002

[22] OASIS Comitee 'Reference Model for Service Oriented Architectures 1.0' Available at `http://www.oasisopen.org/committees/download.php/19679/soarmcs.pdf` 2006

[23] Paurobally S., Tamma V. and Wooldridge M. 'Cooperation and Agreement between SemanticWeb Services'. IN *W3C Workshop on Frameworks for semantic web services.* Austria, 2005

[24] PAWS project Home page
(http://sourceforge.net/projects/paws-ai/) *Visited 2009-June-12*

[25] Perelman C., Olbrechts-Tyteca L. 'The New Rhetoric: A Treatise on Argumentation', University of Notre Dame Press, Notre Dame, 1969.

[26] Protege project home page
(http://protege.stanford.edu/) *Visited 2009-June-12*

[27] Searle J.R. , Kiefer F. and Bierwisch M. 'Speech act theory and pragmatics', 1980

[28] Soft real time constraint definition
(http://en.wikipedia.org/wiki/Real-time_computing) *Visited 2009-August-20*

[29] van der Torre L. and Hulstijn J. 'Combining Goal Generation and Planning in an Argumentation Framework', In *Workshop on Argument, Dialogue and Decision*, 2004

[30] Wooldridge M., Jennings N.R. In *Intelligent Agents: Theory and Practice*, January 1995

[31] Wijngaards N.J.E.,Overeinder B.J., van Steen M. Brazier F.M.T. In *Supporting Internet-scale multi-agent systems.* Data Knowledge Engineering, 41, pages 229245, 2002.

[32] Wu D. et al. 'Automating DAML-S Web Services Composition Using SHOP2', In *Proceedings of the 2nd International Semantic Web Conference*, page 195-210, 2003