Numéro d'identification :

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

THÈSE

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

SPÉCIALITÉ   : **INFORMATIQUE**
Formation Doctorale   : **Informatique**
École Doctorale   : **Information, Structures, Systèmes**

Dynamic Service Generation:
Agent interactions for service exchange on the Grid

par

Clément Jonquet

Soutenue le 16 novembre 2006 devant le Jury composé de :

Michael N. HUHNS, Professeur, University of South Carolina, USA . . . . Rapporteur
Luc MOREAU, Professeur, University of Southampton, UK . . . . Rapporteur
Domenico TALIA, Professeur, Università della Calabria, Italy . . . . Rapporteur
Amal EL FALLAH SEGHROUCHNI, Professeur, Université Paris 6, France . . . . Examinateur
Jacques FERBER, Professeur, Université Montpellier 2, France . . . . Examinateur
Jean-Luc KONING, Professeur, Institut National Polytechnique de Grenoble, France . . . . Examinateur
Stefano A. CERRI, Professeur, Université Montpellier 2, France . . . . Directeur

Numéro d'identification :

<div align="center">

Académie de Montpellier

Université Montpellier II
— Sciences et Techniques du Languedoc —

Thèse

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

</div>

Spécialité	: **Informatique**
Formation Doctorale	: **Informatique**
École Doctorale	: **Information, Structures, Systèmes**

<div align="center">

Dynamic Service Generation:

Agent interactions for service exchange on the Grid

par

Clément Jonquet

Soutenue le 16 novembre 2006 devant le Jury composé de :

</div>

Michael N. Huhns, Professeur, University of South Carolina, USA . . . . Rapporteur
Luc Moreau, Professeur, University of Southampton, UK . . . . Rapporteur
Domenico Talia, Professeur, Università della Calabria, Italy . . . . Rapporteur
Amal El Fallah Seghrouchni, Professeur, Université Paris 6, France . . . . Examinateur
Jacques Ferber, Professeur, Université Montpellier 2, France . . . . Examinateur
Jean-Luc Koning, Professeur, Institut National Polytechnique de Grenoble, France . . . . Examinateur
Stefano A. Cerri, Professeur, Université Montpellier 2, France . . . . Directeur

This thesis is dedicated to Marcel and Esteban …

Contents

Acknowledgements

I would like to express my gratitude to my supervisor Professor Stefano A. Cerri. His guidance and his long experience have enhanced my working capabilities. His methods and advices have inspired me and will continue to help me.

My gratitude also goes to Pr. Christian Queinnec, Pr. Marc Eisenstadt, Dr. Marc-Philippe Huget for they remarks about some piece of my work.

I would like to express my thanks to John Domingue and Enrico Motta for providing me an opportunity to work at the Knowledge Media Institute (KMi – Open University, UK) for a few weeks during my thesis.

Thanks also to Monica Crubézy for receiving me at the Stanford Medical Informatics (SMI – Stanford University, USA). Thanks also to the France-Stanford Center for Interdisciplinary Studies.

Thanks also to the GT-MFI (Groupe de Travail - Modeles Formels pour l'Interaction) members for interesting discussions and presentations.

A very special thank goes to my friends and colleagues from the Laboratory of Informatics, Robotics, and Microelectronics of Montpellier (LIRMM): Maria-Augusta S. N. Nunes, Philippe Lemoisson, Mehdi Yousfi Monod, Patitta Suksomboon, Lylia Abrouk. A special acknowledgement to Pascal Dugénie for the elicitation of GRID concepts and the work we realized together on agent-Grid integration; to Nik Nailah Binti Abdullah for her remarks; to John Tranier for the constructive discussions for years; to Christophe Crespelle for his rigourous help; to Mathias Paulin for some other reasons nevertheless important. Other members of the LIRMM, teachers, professors and students.

I would also like to thank Jason Elric for his help on English revision.

Deep thanks to the teachers, from primary school to university, who have motivated and guided me during my studies.

This thesis has been made possible through encouragement from my family and friends. In particular, I would like to thank my parents and sisters for their perennial support. Special thanks and deep gratitude to Isabelle, my wife, for her support during these three years of work; without her nothing would have been possible.

Introduction

Computer Science or Informatics is often considered as a synthetic discipline because most of the phenomena studied have been created by a person rather than being 'given' by nature [Cou94]. The work of researchers is especially difficult because there are more things to create rather than to discover. Their responsibility is very important but they benefit from the large advantage that nothing is definitive and everything should be done and redone. Following this statement, this thesis participates to a new dynamics in Informatics that considers a new way of conceiving the concept of service.

Since ages, human beings have exchanged services one another. Material or immaterial, free or not, concrete or abstract. Naturally, when computing became a new communication and cooperation means for people, the concept of service has followed and we have seen a first way of realizing services in computer-mediated contexts. One interesting idea in this computerization of the concept of service is that the question 'how to request a service automatically?' became immediately important. As a consequence, people were concerned about how services may be used by programs (eventually representing other services), without any human intervention. However, because the implementation primitives adequate for services did not really exist yet, the concept of service was limited to a remote execution of a functionality. A service was considered as a program to run somewhere (on the machine, on the network) able (i) to answer a parameterized question to a human or artificial user (e.g., a remote procedure call); (ii) to execute a task independently (e.g., a service daemon). One important way by which human beings exchange services, i.e., conversations, was at this time forgotten.

> ### INFORMATICS
>
> We rather prefer the term 'Informatics' to the term 'Computer Science'. As the School of Informatics at the University of Edinburgh defines, it is the study of the structure, behaviour, and interactions of natural and engineered computational systems (i.e., representation, processing and communication of information in natural and artificial systems.). The central notion is the transformation of information – whether by computation or communication, whether by organisms or artifacts.

Indeed, human beings more often provide services one another by means of conversations. You have a conversation with your hairdresser in order to have your haircut done; you have a conversation with your architect in order to have your house built; you have a conversation with a friend in order to request his/her help ... and so on. The conversation is the method which helps you, as a service user, and your interlocutor, as a service provider, to express your own constraints, requirements and wishes about the

way you want the service to be provided. The service exchange is therefore customized and adapted for you according to your needs and to what your interlocutor may do in order to understand and satisfy them.

In this thesis, we will say that a service is *dynamically generated* for you. Putting the conversation at the centre of the service exchange in computer-mediated contexts is the overall objective of this thesis. Therefore, dynamic service exchange modelling represents the practical perspective of our thesis (i.e., the what to do). However, one may ask the question: why is it so important to model the way services are exchanged in computer-mediated contexts (i.e., the why to do it)? To answer this question we may divide it in two sub-questions: why is it important to model service exchange interactions that a human user may have with a machine? Why is it important to model service exchange interactions artificial entities may have one with another?

- The first question obviously refers to the way *human and machine interact one another*. A better modelling of service exchange means a better interaction with the machine and thus a better usability of computers and computing systems.

- The second question refers to the way we decide to implement this usability. The distributed nature of computing systems has made the service approach a good one for implementing the way *applications interoperate in a collaboration perspective*. This choice is explained in detail in chapter 2.

The last question (table 1) is therefore: how to develop computational artifacts to address this challenge of service modelling (i.e., the how to do it). In this thesis, we will explain our choices and our results in order to develop these artifacts. These results are mainly presented in the three last chapters of the manuscript. They consist briefly in:

- The proposal of a **new agent representation and communication model, called *STROBE***, that enables agents to develop different languages for each agent they communicate with. STROBE agents are able to interpret communication messages and execute services in a given dynamic and dedicated conversation context;

- The proposal of a **computational abstraction, called *i-dialogue*** (intertwined dialogues) that models multi-agent conversations by means of fundamental constructs of applicative/functional languages (i.e., streams, lazy evaluation and higher-order functions);

- The proposal of a **service-oriented GRID-MAS integrated model** based on the representation of agent capabilities as Grid services. In this model, concepts of GRID and MAS, relations between them and the rules are semantically described by a set-theory formalization and a common graphical description language, **called *Agent-Grid Integration Language* (AGIL)**.

Table 1: Thesis context: the 'what', 'why' and 'how'

WHAT	Modelling dynamic service exchange interaction in computer-mediated contexts for both human and artificial entities.
WHY	Enhancing the way these distributed entities work in collaboration to solve the problem of one of them.
HOW	Proposing proof of concept artifacts mainly based on agent communication and Grid service integration.

These results are what [Cou94] calls proofs of concepts (opposed to the other major class of Informatics results: proof of performance). They represent the methodological perspective of our work. We

show that using our approach is a good means to reach the identified objective. We illustrate it on simple experimentations or example scenarios that represent a large class of situations:

- Meta-level learning and dynamic specification by communicating, with STROBE (section 3.4). These two scenarios were experimented. They validate the feasibility of the model and demonstrate its potential;

- A travel agency conversation example, with i-dialogue (section 4.3.4). It illustrates an example of service composition;

- A multi-agents 'looking for a job' scenario, with AGIL (section 5.5). It exemplifies the use of AGIL as a description language for GRID-MAS integrated systems.

Addressing the question of service exchange modelling in computer-mediated contexts faces a double complexity. The first one (i.e., space) is related to the number of domains of Informatics that address this question: e-learning, knowledge modelling, service-oriented computing, distributed systems, etc. In this thesis we specifically concentrate on three of these domains because they have developed, as we demonstrate, significant and relevant aspects and complementarities for enhancing the way services are used and provided in computer-mediated contexts. These domains are Service-Oriented Computing (SOC), Multi-Agent Systems (MAS) and GRID. The second one (i.e., time) concerns the validity in time of the choice made i.e., what has been done and what will be done. In this thesis, we tried to make a contribution to the development of SOC, MAS and GRID keeping in mind what already exists and proposing solutions always compliant with other approaches. Our method often consists (e.g., in the STROBE model or

> ### COMPUTATIONAL ARTIFACTS
>
> In 1994, the American National Research Council Committee on Academic Careers for Experimental Computer Scientists wrote a report about Experimental Computer Science and Engineering (ECSE) research and careers [Cou94]. They defined ECSE as the building of, or the experimentation with or on, computational artifacts. Artifacts are implementations of one or more computational phenomena (hardware or software systems). An artifact can be the subject of a study, the apparatus for the study or both.
>
> Artifacts serve at least three primary purposes in ECSE: A given implementation can
>
> - seek performance or seek improvement and enhancement of prior implementations (**proof of performance**) e.g., a better algorithm;
>
> - demonstrate that a particular configuration of ideas or an approach achieves its objectives (**proof of concept**) e.g., an experimental model;
>
> - demonstrate a fundamentally new computing phenomenon (**proof of existence**) e.g., the Web, the mouse.
>
> Proof of existence artifacts are quite rare, where as proof of performance ones are most common.

with the i-dialogue abstraction) in taking already existing solutions such as the one found for example in programming languages and applying them in an agent and Grid framework.

A new consideration in service exchange

The word service is one of the buzzword of today's Informatics: service-based application, service-oriented architecture, service-oriented computing, Web service, Grid service, service level agreement,

service composition, service business process, service invocation, quality-of-service, network services, communication services, semantic services, dynamic services . . . and so on. Computer functionalities are more and more viewed as services provided to computer users and other programs. However some questions about services have not found any answer yet: What is exactly a service in the world of computer and information? Is it something intelligent? What are the means for service exchange in these computer-mediated contexts? Why do many services never understand what we are talking about? Why do we have to fit service requirements in order to benefit from service utility? From a certain viewpoint these questions may be summed-up in one:

- What kind of services do we want for tomorrow Informatics?

Answering this question represents a significant stake for research in Informatics because of several reasons:

- the importance the concept of service took inside and outside Informatics;

- the growing set of computing applications that take place in our lives and sometime substitute old or traditional methods (e.g., e-mail);

- the influence that service exchange can have on collaboration and multi-party remote problem solving;

- the influence that service exchange has on human-human relationships in the society;

- the intrinsic pedagogical dimension of service-exchange that facilitates knowledge creation and diffusion;

- the role played by Informatics to favour a better access to knowledge and know-how for human beings all around the world.

In this thesis, we adopt an integration approach that aims to encapsulate in one theoretical framework all the relevant aspects, often complementary, related to the concept of service. This framework is called **Dynamic Service Generation (DSG)**. It corresponds to a new way of viewing future service exchanges in computer-mediated contexts for human and artificial entities. Dynamic service generation is the global scientific objective which this work is aimed towards. In DSG, the service user constructs step-by-step along the conversation, what he wants as the result of the service provider's reactions. The conversational process is what distinguish a dynamically generated service from another one: services are not simple product deliveries. In Informatics, one of the fundamental differences between a product and a service is that when a user asks for a product, he exactly knows what he wants and a typical procedure call (with the correct parameters) is then activated. In other words, the user asks a fixed algorithm (procedure) to be executed. At the opposite, when a user requires a service,[2] he does not exactly know what he wants and he is not supposed to know exactly what the service provider can offer him. His needs and solutions appear progressively with the conversation with the service provider until he obtains satisfaction. Actually, *DSG shifts the service exchange modelling approach from specifying what the user wants, to specifying how the user will specify what he wants*. The thesis statement is summarized in table 2.

In the expression 'dynamic service generation', the verb 'to generate' means 'to bring something into existence'. The term 'generation' is the name given to the process that occurs in service exchange in order to generate (or engender, or create) a new service. The term 'dynamic' means that such a generation

---

[2]This is a language abuse because we can not say that 'a user requires a service'. The service is characterized by a service generation process. We would have say 'when a user asks a service to be generated' or 'when a service is generated for/to a user'.

Table 2: Thesis statement

PRESENTATION	*A service exchange is not a simple delivery of product; it is based on conversation. SOC, MAS and GRID together bring concrete solutions for implementing real services.*
JUSTIFICATION	The first part of this statement is justified in chapter 1 (reflection about the concept of service, definitions, scenarios, comparison with products). The second part is justified in chapter 2 which makes a state of the art of SOC, GRID and MAS and their interconnections.
DISCUSSION	We try to demonstrate this statement is true in the the three next chapters of the thesis. Each contribution addresses different requirements of DSG.

cannot be (and thus was not) pre-programmed; it occurs dynamically. We sometime use the expressions 'during execution', 'at run time', or 'on the fly' to underline this dynamic aspect.

Chapter 1 and 2 proposes a *characterization* of the DSG concept, instead of a definition or a specification, which fits better with the scientific process (described after) that led to the thesis results. We do not define in this thesis future DSG systems, but rather, we propose a set of characteristics that today's systems must have in order to implement this new concept of service exchange.

Thesis scientific process

This section details the scientific process that led this thesis work during the past three years. As the right hand side of figure 1 shows, this process led us to be interested in different domains; they are here reported chronologically. During these three years, we tried to *explore* research contexts in SOC, MAS and GRID, keeping in mind the pedagogical perspective of our work.[3] Each time we tried to *experiment* and then *model* our work. These steps were repeated recursively, being enriched by *information* we got to enhance our model(s). An important aspect of the scientific process occurred during this step: *dissemination*. This was done through the publication of scientific articles and communications (e.g., seminars, presentations, working group, etc.). At a certain time we needed to *contextualize* our work both theoretically and practically. This led us to *characterize* our theoretical approach and choose one-by-one the characteristics we wanted to add to our model(s). This characterization method allowed us to: (i) establish an open set of requirements or aspects (called characteristics) that (non exhaustively) defines our scientific objectives; (ii) continue our first research ideas as they corresponded to identified characteristics; (iii) have a contextualization of our work, helping us to adapt it; (iv) show some perspectives both at a short term (i.e., the next characteristics to study, how to connect studied characteristics together, etc.) and at long term (i.e., domains of Informatics that will have to be really enhanced or created). Finally, we adopt an *integration* approach that binds all the studied aspects together.

This scientific process is step-by-step detailed in the thesis. Chapter 1 and 2 establish a list of DSG characteristics by reflecting on the concept of service and by studying the states of the art. Chapter 3, 4 and 5 present our results in order to address some of these characteristics.
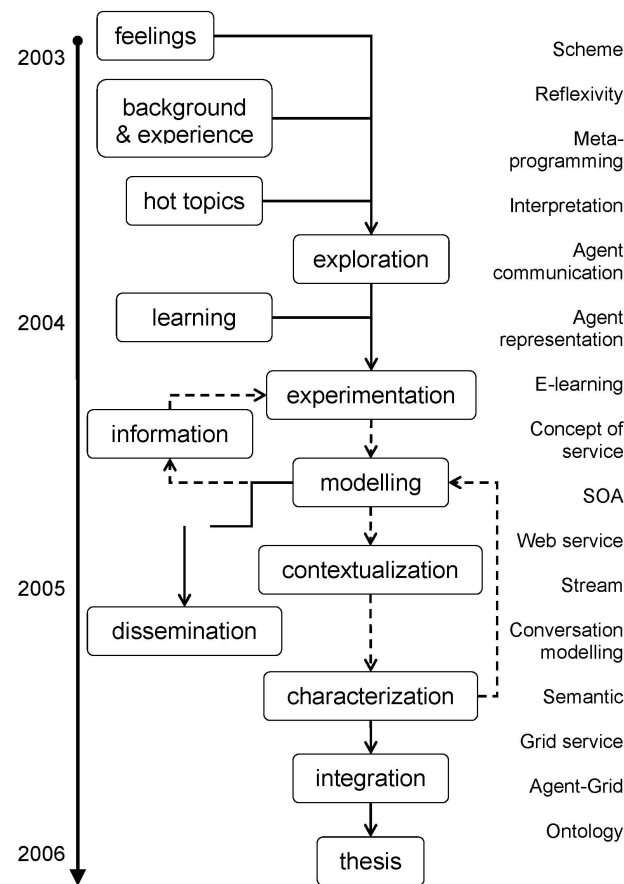
Figure 1: Thesis scientific process

Agent interactions for service exchange on the Grid

- The expression *service exchange* identifies the relation binding two entities, denoted *service user* and *service provider*, aiming to find a solution, identified and chosen among many possible ones, offered to the problem of one of them. These entities are later called agents.

- The expression *agent interaction* identifies the mechanism by means of which agents would achieve this service exchange. From a certain point of view, the thesis contribution may be viewed as a formalization of service exchange interactions in MAS using GRID mechanisms.

- The term *service* refers to the 'SOC' domain. It is a quite new acronym for Service-Oriented Computing. It was proposed during the first International Conference on SOC held in Trento, Italy in December 2003 (www.icsoc.org). This term refers to the new domain of Informatics concerned with service-based applications: Web service, service-oriented architecture, business process management, semantic services, etc. The term was recently strengthen by the title of Singh and Huhns's recent book on those topics [SH05]. This book is a state of the art and a review of challenges in SOC to which we often refer to.

- The term *agent* refers to the 'MAS' (Multi-Agent Systems) domain. We both use this term to represent the MAS concept and the MAS research domain including autonomous agents, agent

---

[3]The PhD is often say to be a training TO research, FOR research and BY research.

architecture, agent communication, distributed AI, etc. Agents are the entities, human or artificial which exchange services. They do it in a collaboration, learning and problem solving perspective.

- The term *Grid* refers to the 'GRID' domain. It is the name given to the recent domain of distributed systems interested in sharing resources and services in a easy, secure and scalable manner. The Grid is sometime viewed as a logic continuation for the Web. We use to say 'on the Grid' like we do for the Web. In this thesis, we will use the word Grid in this sense. The term Grid was chosen for the analogy with the power grid (explained in section 2.4.1). The expression 'Grid computing' is also often used to refer to this new computing paradigm, however, we do not use it because it is too strongly related to the first domain of application of GRID which was only concerned with sharing computing resources in order to achieve large scale computations. Quite recently, the French GRID research community proposed the following acronym: Globalisation des Ressources Informatiques et des Données[4] in order to map with the English term. In this thesis, we use the term 'GRID' (analogous to SOC and MAS) to refer to the research domain and 'Grid' to refer to the computing concept i.e., the Grid, Grid service, Grid resource, Grid user, etc. GRID is described and used in this thesis at a quite theoretical level. We do not go into detail concerning core GRID functionalities, mechanisms or specifications. We consider the Grid as a computing infrastructure interesting for service exchange in which agent exchange services among virtual communities. Our analysis is based on the work of post-modelling realized by our team, in order to understand, use and develop GRID. Especially, key GRID concepts formalized in the context of the Agent-Grid Integration Language (chapter 5) were suggested by Pascal Dugénie as part of his on going PhD thesis.

As already mentioned, the thesis deals with three domains. However, the major contribution of the

---

[4]Which can be translated as 'Data and Computing Resources Globalization'.

thesis can be analyzed following two angles:

- The theoretical contribution and the perspectives of the thesis brought by (i) our reflection about the concept of service, (ii) the list of DSG characteristics and (iii) the proposed service based GRID-MAS integration, belongs mainly to SOC;

- The technical and concrete contribution of the thesis brought by STROBE (an agent representation and communication model), i-dialogue (a computational abstraction for agent conversation) and AGIL (a GRID-MAS integrated model), belongs mainly to MAS.

<div style="border:1px solid; padding:1em">

SIMILAR RESEARCH DIRECTIONS

Our research directions were confirmed during this thesis work (2003-2006) by a set of recent publications sharing the same scientific objectives or supporting the same vision. Among them:

1. Singh and Huhns's 2005 book about the challenges of SOC [SH05];

2. Huhns et al.'s December 2005 paper on research directions for service-oriented multi-agents systems [HSB$^+$05];

3. Foster et al.'s July 2004 paper on why GRID and MAS need each other [FJK04];

4. Foster et al.'s May 2004 specifications of the Web Service Resource Framework as a concrete definition of Web and Grid services merging [FFG$^+$04];

5. Goble and De Roure August 2004 'myth busting and bridge building' paper confirming the stakes of the Semantic Grid [GR04];

6. European (WSMO) and US (OWL-S) research work on Semantic Web services [CDM$^+$04];

7. The first publication, in 2005 of the 'International Journal of Multiagent and Grid Systems'. The 2001-2003 'Agent-Based Cluster and Grid Computing' workshops, and the 2005-2006 'Smart Grid Technologies' (SGT) workshops;

8. The 2003-2004 'Web Services and Agent Based Engineering' (WSABE) workshops, and the 2005-2006 Service-Oriented Computing and Agent-Based Engineering (SOCABE) workshops.

</div>

Chapter overview

The manuscript content integrates our results and show that they are all part of the same research perspective. The chapters are inspired from articles or research reports[5] we wrote in three years. [JC06] for chapter 1; [JC05b] for chapter 3; [JC05a] for chapter 4; [JDC06b] and [JDC06a, JDC07] for chapter 5. Chapter 2 is inspired from all of them.

Chapter 1 – What is Dynamic Service Generation?
This chapter presents a reflection about the concept of service. What is a service? We try to show the limit of the current vision of services, by illustrating some scenarios, such as travel planning or job search, that need a better engagement of provider and user, because the service exchange is

---

[5]Research reports are under submission.

absolutely not generic and needs to be customized and adapted for a specific user. By analyzing some definitions, we extract some criteria that intrinsically define the concept of service (table 3).

Table 3: Dynamic service generation features

RESULT	Services imply the creation of something new
MEANS	Services are associated with processes
	Services are constructed by means of conversations
SIDE-EFFECT	Services have a pedagogical dimension
	Services create relationships between members of communities

Following these criteria, we introduce the concept of *dynamic service generation* as a different way to exchange services in a computer-mediated context. This process allows services to be more accurate, precise, customized and personalized to satisfy a non predetermined need or wish. We present dynamic service generation systems key elements: *services* are dynamically provided and used by *agents* (human or artificial) within a *community*, by means of a *conversation*. We propose an extension of the economic taxonomy, that considers every computer-assisted performance as a service, by distinguishing between product delivery and dynamic service generation. In particular, the chapter proposes a list of 19 differences between product delivery systems and dynamic service generation systems.

The ambition of this chapter, and the next one, is not to specify and formalize DSG systems, but rather to identify a non exhaustive list of characteristics DSG systems should have. This list promotes a progressive transition from product delivery to dynamic service generation systems by transforming one-by-one the identified characteristics into requirements and specifications. Two aspects are in particular demonstrated important because they are very close to many of the characteristics:

- The agent oriented aspect of DSG systems, since agents uniformly model artificial and human, autonomous, intelligent and interactive entities that possess many abilities for service providing and using. Reasoning, communication, negotiation, conversation, organization, are identified as characteristics that promote a substitution of the current object oriented kernel of services by an agent oriented kernel.

- The Grid oriented aspect of DSG systems, since the Grid is the first infrastructure really developed in a service perspective. GRID acquires major importance in SOC by augmenting the basic notion of Web Service with two significant features: service state and service lifetime management, thanks to the concept of Grid service.

Most of the characteristics, related to SOC, MAS and GRID, are identified in chapter 2. However, the end of chapter 1 deals with some domains of Informatics such as interaction or learning that we want to independently detail because they seem important for DSG. The chapter ends with a set of programming techniques, such as lazy evaluation, dynamic typing, interpreted languages, etc. that may be interesting for DSG systems implementation.

Chapter 2 – State of the art

This chapter presents a state of the art of the three domains this thesis deals with: SOC because it corresponds to the new research interest related to services; MAS and GRID because they appear as the two major domains addressing DSG characteristics. The chapter presents these three domains as well as their interconnections (i.e., MAS-SOC, GRID-SOC, GRID-MAS). This state of the art is done through the prism of DSG, and each time an element seems interesting for DSG, it is expressed as a characteristic. The analysis highlights, for example, the importance of:

- being able to have a state;
- being able to dedicate a part of this state to an interlocutor;
- being able to intelligently and dynamically modify this state.

Another example of an aspect that appears to be very important for the three domains is message based communication. Indeed, SOC, MAS and GRID are three different approaches of distributed systems that use communication to enable different entities to interact in order to address a common objective e.g., service exchange. Analogies on the way these three domains address communication are clearly evident. It encourages the integration perspective taken by this thesis because the challenges are intrinsically the same. For example, the challenge in MAS of modelling agent conversation not by a fixed structure (interaction protocol) but by a dynamic dialogue, becomes the same one as the challenge of dynamically composing Web/Grid services in business processes.

The first two chapters identify the problem, the concept and a set of characteristics of the scientific objective. The conclusion of chapter 2 proposes a synthesis of all the identified DSG characteristics and briefly clarifies what are the characteristics addressed by the rest of the thesis chapters. Afterwards, the three last chapters present our solutions and results in order to go towards an implementation of DSG systems.

Chapter 3 – The STROBE Model

This is the first chapter which presents our results to address some identified DSG characteristics. As DSG strongly depend on conversations, this chapter presents the STROBE model, an agent communication and representation model developed in a DSG perspective. A part of the work done in this chapter is based on powerful concepts of applicative/functional programming applied in the context of agent communication and service exchange. The key idea consists in enabling an agent to develop a dedicated language for each of its interlocutor or group of interlocutors, by means of dedicated conversation contexts, called Cognitive Environments (CEs). These conversation contexts represent the interlocutor models. The term 'conversation context' is inspired from the term 'execution context' that exists in traditional programming languages and means the context in which an expression of a language (i.e., a program) is evaluated to produce some results. In the STROBE model, agents are able to interpret messages in a given environment, with a given interpreter both dedicated to the sender of these messages. Communication between agents enables to dynamically change these environments and these interpreters in order to change their way of interpreting messages. We are used to talk about data, control and interpreter levels learning. Some simple concrete experimentations illustrate the potential of the model. In particular: (i) meta-level learning by communicating i.e., how an agent can change at run time the way it interprets an interlocutor's messages by re-interpreting its dedicated interpreter; (ii) dynamic specification i.e., how an agent can express one by one its constraints on the way a service is provided by another agent thanks to non-deterministic interpreters.

Chapter 4 – I-dialogue

This is the second chapter addressing with concrete means the question of DSG. It proposes a computational abstraction, called i-dialogue, to model conversations that could occur in DSG when an agent has multiple conversations with several agents. It takes the form of a recursive function, producing and consuming streams of messages, run by each agent in a conversation. The i-dialogue abstraction is inspired by the *dialogue* abstraction proposed by [O'D85] to model process interactions and by [Cer96b] which suggests to use streams for agent communication. In this chapter, we extend the dialogue abstraction to more than two communicating processes and consider it for agent communication. I-dialogue (intertwined dialogues) models conversations among STROBE agents by means of fundamental constructs of applicative/functional languages. (i.e., streams, lazy

evaluation and first-class functions). Part of the chapter is used to present a general implementation of the i-dialogue abstraction for functional/applicative languages such as Lisp or Scheme. The i-dialogue abstraction deals directly with different sequences of inputs, different sequences of outputs, and state of the agents concerned. It avoids the drawbacks of classical approaches in conversation modelling such as interaction protocols, because (i) it does not presuppose anything about the internals of the interlocutor agent and only deal with output streams of messages; (ii) it deals with the entire conversation (expressed by potentially infinite and not predetermined streams of messages) and not simply with a message alone. The i-dialogue abstraction is adequate for representing intertwined dialogues, executed simultaneously and for which inputs and outputs depend on each other, such as those that can occur in service composition as it is illustrated on an example of scenario.

Chapter 5 – Agent-Grid Integration Language

This is the last chapter presenting the thesis results. It does not propose (as the two previous ones) a mechanism or a set of techniques, but it rather proposes a way to achieve the integration of GRID and MAS as they are the two key characteristics of DSG systems. The integration of GRID and MAS was already suggested and is motivated by the fact that both domains developed significant complementarities, as it is detailed in chapter 2. The chapter 5 addresses this question by means of a service-oriented approach because the concept of service is demonstrated to be a unifying concept for these two domains: services are exchanged (i.e., provided and used) by *agents* through the *Grid* mechanisms and infrastructure. The model of integration of these domains is formalized rigorously. Concepts, relations between them and rules of GRID-MAS integrated systems are semantically described by a set-theory formalization and a common graphical description language, called Agent-Grid Integration Language (AGIL). The key GRID concepts presented in this chapter have been established by the Open Grid Service Architecture specification. The key MAS concepts have been established by different approaches in the MAS literature, especially the STROBE model. The two main underlying ideas of the model are:

- the representation of agent's capabilities as Grid services in a service container, i.e., viewing a Grid service as an 'allocated interface' of an agent's capability by substituting the object oriented kernel of Web/Grid services with and agent oriented one;
- the assimilation of the service instantiation mechanism – fundamental in GRID as it allows Grid services to be stateful and dynamic – with the dedicated cognitive environment instantiation mechanism – fundamental in STROBE as it allows an agent to dedicate to another one a conversation context.

The AGIL's model for GRID-MAS integrated systems integrates all the previous results proposed in the thesis by addressing the 'what' identified in the beginning of this introduction: it is a formalization of service exchange interactions between agents on the Grid.

How to read this thesis?

This section helps to understand how to read this thesis by explaining the different presentation styles the reader will find in the manuscript:

'This is a *quotation* from authors. We use it each time it is appropriate because it perfectly express what we want to.'

DEFINITION: DEFINITION
This is a *definition* of a term or concept, it generally occurs after introducing the concept in order to sum up the meaning taken into account in the thesis. Definitions are summarized at page 211.

Characteristic 1 (keyword)

This is an identified DSG characteristic. *Characteristics are referenced in the rest on the manuscript by their number and keyword (e.g., C13[semantics]). They are summarized at page 215.*

In the chapters 3 and 5, the reader will also find *relations* and *rules* defining a correspondence between the STROBE model or AGIL's concepts (relations are summarized in appendix B). They are both denoted as follows:

$$corresponding : DOMAIN \rightarrow RANGE \text{ (type of relation)}$$

Rule 1 *This is a* rule *related to the previous relation.*
Mathematical expression of the rule.

SIDEBAR

This is a *sidebar*. It specifically details a concept or an idea related to the page's content. It often refers to related work or inspirations.

Each chapter begins with an abstract as well as a table of contents. When a concept is met for the first time, it is generally written in *italic*. The `type writer` font is reserved for code or specification terms and URLs. A glossary of all acronyms is available at page 219. The list of figures and tables are respectively available at page 199 and 201.

The bibliography is available at page 215. Notice a significant number of references for mainly two reasons: the work realized (i) is at the crossing of three important domains; (ii) was done within the French research communities, and we keep the concern of referencing French work. Each time it was possible, an English version or related reference is also cited.

Chapter 1

What is Dynamic Service Generation?

In the future, independently developed services will be dynamically selected,
engaged, composed, and executed in a context-sensitive manner.
Huhns et al. [HSB⁺05]

THE GOAL of this first chapter is to reflect the concept of service in Informat-
ics. In particular, we introduce the concept of *dynamic service generation* as
a different way to exchange services in a computer-mediated context: services are
dynamically provided and used by agents (human or artificial) within a community,
by means of a conversation. The chapter presents an overview of the concept of
service from philosophy and economy to Informatics. A strict comparison with the
current popular approach, called *product delivery*, is carried out. The main result
emerging from these reflections is a list of 'characteristics' of dynamic service gen-
eration, starting in this chapter and continuing through to the next one. This list pro-
motes a progressive transition from product delivery to dynamic service generation
systems by transforming one-by-one the identified characteristics into requirements
and specifications.

Contents

1.1 Introduction

Clancey [Cla05] reflects on a personal experience which occurred when he had to organize a journey. Using a travel agency service, he did not get satisfaction because his goal was to plan and have a pleasant journey, whereas the travel agency employees and Web sites simply wanted to sell him tickets. His need was feasible, but none of the agents (human or artificial) that interacted with him considered his real problem. All of them concentrated on the product that they had to sell: tickets ('piece-meal services'). From this scenario, Clancey explains that 'constraints and preferences usually cannot be formulated up front; the process of articulating constraints is necessarily interactive and iterative' and 'joint construction is required, a conversational interaction that bit-by-bit draws out from me what I am trying to do'. He raises some principles that should be taken into account in (human or artificial) service systems such as for example:

- the co-construction of the obtained product, based on a conversation (interaction and iteration);

- the real understanding and consideration of the user's problem(s);

- the entrustment of task that must lead to satisfaction;

- the pro-activity towards the user.

As another real life scenario, imagine someone looking for a job in a job agency. Finding or choosing a job has such important consequences on an individual's life that it cannot be generalized, it must be done on an individual basis. Requirements, constraints and conditions are all interleaved with one another. The service delivered by the person in the job agency is 'dynamically generated': the job seeker explains his/her qualifications, diplomas, interests, wants as well as gives his/her private constraints (family, culture, etc.), during an interactive conversation. The employee of the job agency and the job seeker construct, step-by-step, a solution that fits both of them. This solution will have relevant external consequences on the rest of the world. This is a kind of dynamically generated service.

The notion of service is now at the centre of distributed system development; it plays a key role in their implementation and success. One of these successful systems is of course the Internet, but nowadays, surfing on the Web is equivalent to asking another computer on the Internet to execute a computation specified by an algorithm, or to give an answer to a well specified question. Actually, the Internet allows to deliver 'ready made informational products' to human or artificial entities, rather than true 'on the fly generated smart services'. For the moment, we can store and retrieve 'products' (e.g., pages, documents, images, function application results, etc.) but we cannot yet participate to an interactive and collaborative society. These 'products' are not sufficient as demonstrated by Clancey's scenario failure. More generally, modern computers, operating systems and software provide the user with precise products for which they have been conceived for. A file system enables to create, copy, move, or delete files, a music player permits to listen, record, copy, download music, a travel guide or yellow pages Web sites allows to buy a train ticket or to find somebody's address and phone number. All these 'services' correspond to well identified and precise needs or wants and they are specifically implemented to answer them. As the needs or wants evolve, new versions of these systems are updated. It is the classic software engineering life cycle. However, can we really speak of services?

We will see in this chapter that to provide somebody with a service means to identify and offer a solution (among many possible ones) to his/her problem. We will explain how a service in a computer-mediated context must not be reduced to a simple 'method invocation', how it is unique and how it needs a real engagement of the entities using and providing this service. We argue that a real service exchange is not a simple product delivery, because a service is adapted, customized and personalized by a special provider for a special user. A very important aspect of the concept of service is that it should

be based on a conversation. The service exchange process should be viewed as a kind of collaboration. The chapter introduces the concept of *Dynamic Service Generation* (DSG) as the process of obtaining services constructed on the fly by the provider according to the conversation it has with the user. These services are called *dynamically generated services*. In DSG, service providers and service users are both *agents* (human or artificial) members of a *community*, both of which have *conversations*.

DSG is opposed to the classical *Product Delivery* (PD) approach. The main difference between a product and a service is not the obtained result, but the process to obtain it (delivery vs. generation). In one hand, the notion of product refers to the idea of obtaining the delivery of something already existing, or already designed, pre-identified as a need or want. On the other hand, the notion of service refers to the idea of creation, discovery, access or elicitation of something new, not pre-identified. The difference is very easy to understand in human everyday life. For example, someone looking for clothes may look for *ready-to-wear clothes* (prêt-à-porter) which is analogous to asking for a product, or may either look to have *clothes made by a tailor* which is analogous to asking a service to be generated. However, in the two cases, the result is the same i.e., clothes.

As another example of DSG, take for instance the one of Singh and Huhns in their recent book on service-oriented computing [SH05]. This is how they refer to service *engagement*:

> 'Imagine going to a carpenter, a human service provider, to have some bookshelves built in your home. You will typically not instruct the carpenter in how to build the shelves. Instead, you will carry out a dialog with the carpenter in which you will explain your wishes and your expectations; the carpenter might take some measurements, check with some suppliers of hard-to-get materials (maybe some exotic woods), and suggest a few options; based on your preferences and your budgetary constraints, you might choose an option or get a second estimate from another carpenter.'

Rethinking the principle of service exchange. Providing this kind of service in a computer-mediated context is a real challenge. In spite of the fact that most current literature concerned with services apparently comes from the Web communities (i.e., Web services), we are convinced that the challenge cannot be seriously taken-up unless profiting from results emerging from several domains in computing. The notion of service has to surpass HTTP protocols, Service-Oriented Architecture (SOA) standards, Remote Procedure Call (RPC) and eXtensible Markup Language (XML), and has to be enriched by other research domains such as information systems, concurrent systems, knowledge engineering, Artificial Intelligence (AI), interaction, and especially by distributed systems approaches such as GRID or MAS. Simply delivering products is no longer sufficient, a service provider cannot simply execute a single task for a given kind of user as it is in the client/server case or with Web services (section 2.2.2.1). DSG does not assume the service user to know exactly what the service provider can offer him. He[1] (the service user) finds out and constructs step-by-step what he wants as the result of the service provider's reactions. He does not exactly know what he wants and needs, therefore the solutions appear progressively during the conversation with the service provider until he obtains satisfaction. Actually, DSG is an interactive process between two agents in a community having a conversation. These elements form what we called a *DSG System*.

Considering the interaction between users and information systems, three presuppositions are generally held to be true [Cer05]:

- The user knows exactly what he wants;

- The user knows that the system can answer him;

- The user knows how to formulate his request.

---

[1]DSG users are agents, Artificial Agent (AA) or Human Agent (HA), the terms he/him will be used hereafter generically for she/her and it/its.

However, none of these presuppositions are verified in DSG. They all lie on the user's knowledge and not on the information system ability to take the initiative, to understand or to adapt to a specific request. DSG systems should not presuppose skills or abilities of their agents (human or artificial).

The ambition of this chapter, and the overall thesis, is not to specify and formalize DSG systems, but rather to identify a non exhaustive list of characteristics DSG systems should have. The proposed methodology for DSG systems development consists in a progressive evolution from current PD systems to DSG systems by integrating these characteristics one-by-one. Two aspects in particular are demonstrated as important:

- The substitution of the current object oriented kernel of services by an agent oriented kernel. The agent paradigm is the only approach in computing able to deal with conversation based services;

- The use of a GRID based service-oriented architecture and infrastructure to host agent service exchanges. The Grid is the only approach in computing able to deal with stateful and dynamic services.

Chapter overview. The remainder of the chapter is organized as follows: Section 1.2 presents the concept of service i.e., what is a service? what is a product? This section makes an overview of the intrinsically social ideas behind the concept of service from philosophy and economy to Informatics. Section 1.3 introduces dynamic service generation system elements and gives some DSG examples. A definition of DSG is done by opposition to PD. Finaly, section 1.4 starts a DSG characteristics list by indentifying a set of fundamental domains of Informatics that have a role to play in dynamic service generation system implementation. A set of programming techniques that may be interesting for DSG is also addressed (section 1.4.3.5).

1.2 The concept of service: theories and definitions

From a 'human' viewpoint, human beings need to help each other, they need to accomplish many tasks together. One may say that the more animals cooperate, the best their societies perform and evolve. Needless to say that cooperation among humans has been boosted by the emergence of several kinds of 'languages' (i.e., communication means), from the press to the Internet. Since the whole of information and knowledge cannot be held by only one individual, collaboration is crucial for both information and knowledge sharing. The notion of service operates here: education, health, justice, and trade are the results of services that men use and provide to one another. From a 'computing' viewpoint, the end of the previous century was marked by automation and computerization of various tasks and processes. Among them, we may distinguish service-oriented scenarios. Nowadays, computational artifacts are able to replace our human fellows in several tasks such as mathematical calculations (more quickly than with tables or abacuses), train ticket selling (without waiting in a queue), money withdrawal (without even speaking to the banker), one click buying on the Web, one click googleing to obtain any kind of information that someone decides to publish somewhere on the Web, etc. Different interests (cultural, educational, economic, etc.) brought these services to be computerized in order to be more available and to reach a wider audience. However, can we really speak of *services*? What is a service? What differentiates the exchange of services from the delivery of simple products?

1.2.1 Dictionary definitions

Etymologically, the word 'service' comes from the Latin *servitium* (slavery, servitude), and from *servus* (slave) and means the 'act of serving' (to serve). However, most usages of the word service are not concerned with subordination of one by another – except in some service expressions such as 'à votre service'. According to the dictionary, the term service is most of the time associated to a noun or an

adjective, and the pair carries the meaning. Thus, one will speak of national service, public service, weather service, community service, customer service, health service, postal service, information service, transportation service, etc. The word is also used with a verb that specifies the service delivery action: services may be offered, provided, invoked, fulfilled, delivered, performed, exchanged, rendered, built, achieved, realized, etc. The idea of service is so pervasive and ambient that we are used to say that we live in a 'service society'. The elements specifying the service exchange are themselves called: quality of service, terms of service, condition of service, denial of service, service contracts, service actors, etc. All these expressions indicate a general definition of service that could be: *use that one can make of something, or someone*; and also the expressions such as *in/out of service*, used for a functional device or person exercising his/her/it role (or not). Actually, this definition assumes the pre-existence of a tool/person that should be used. The service is therefore associated to a functionality, a role. However, the dictionary gives another set of definitions that seem more relevant:

- *what one makes for someone*;

- *work done by one person or group that benefits another*;

- *the contribution of an answer to the problem of another.*

They neither define a server nor a user, but rather entities accomplishing tasks for one another. In this second definition, the service is viewed as a kind of collaboration. In economy, the service, called immaterial good, is a good produced and consumed at the same time; it is seen as a transformation. We should also note that a service can also be qualified as continuous, interrupted, intermittent, periodic, temporary, etc. These qualifiers (as well as the past participles cited in the previous paragraph) indicate that a service does not exist (or pre-exist) alone; it is inevitably associated to a process achieving an action. It is important to note that this process should answer a need or a wish. In this definition (see also the next section), the notion of service should be viewed as a relationship between two (or several) entities whose goal is to solve a problem of some among them. The service user should be able to specify this action which must be created and adapted to his needs. The key element of the service notion is not therefore the product or the good resulting from the service, but the relationship created through-service exchange interactions.

DEFINITION: SERVICE

A solution, identified and chosen among many possible ones, offered to the problem of someone. This solution may be physical, psychological, concrete, abstract, real, virtual. This service exchange may be viewed as a kind of collaboration and establishes a specific relationship between a service user and a service provider.

We will see how DSG tries to implement this kind of service in a computer-mediated context. From this first definition (and from the discussion in the Introduction), we extract preliminary ideas that characterize the service exchange and that we keep in mind in the subsequent DSG description:

- Services imply the creation of something new;

- Services are associated with processes;

- Services are constructed by means of conversations.

1.2.2 Nifle's active appropriation services

Nifle, in his sociological and philosophical papers [Nif04a, Nif04b], reflects on the meaning of service. He proposes a service typology based on four aspects:

Assistance. The service is seen as a presence by the one(s) who have(has) a problem. It supposes a monitoring; the service provider comes with the service user although without completely taking care of his/her problem for him/her;

Substitution. The service is seen as a complete taking care of the service user's problem by the service provider. The problem is reduced to its objective, independently of the user;

Mastery. The service is seen as the exercise of a profession characterized by a mastery in a particular problem domain. It is an answer to a problem, proposed by the service provider, in a problem domain on which the service user lacks mastery;

Subcontract. The service is seen as an answer to a precise (and imperial) need. In this aspect, there is a relation of servitude as the need has to be satisfied.

As shown in figure 1.1 (a simplified version of Nifle typology [Nif04a]), the combination of these aspects makes two classes of services emerge. The first one, *active appropriation services*, consists in improving user's mastery of his/her problem. The service provider should help the user to progress in the resolution of his/her problem. The pedagogical dimension of this kind of service is very important as well as the way the user is taken into account. This kind of progress is similar to Socrates' Maieutics.[2] The problem is related to the service user (as the user expressed it), the service performance is related to the service provider (as the provider previously described it) and the solution is related to and benefits to both of them. The second type, *administrated services*, consists in delivering a pre-composed solution. There is no dialogue, and no user adaptation or consideration; there are only abstract needs and concrete impersonal and standard solutions. The service provider is seen as a specialist who demonstrates his know-how in obtaining results, by applying cognitive or technical procedures. The objective is absolutely not the improvement of the user's mastery.
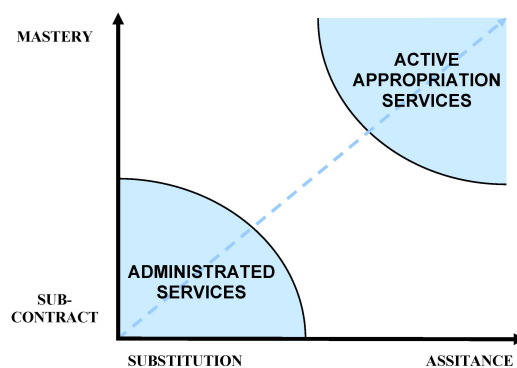
Figure 1.1: Nifle's service topology

Nifle asks the question [Nif04b]: Which concept of service is suitable for representing the Internet phenomenon? For him, the Internet deals with relationships. Information and communication are the means. Of course, the Internet conveys information but it is not its fundamental role (intrinsic meaning). The main interest of the Internet is to help establish remotely close relationships. Besides, the Internet is a new medium because it fundamentally differs from classical (passive) media. For Nifle, all services *based on a passive consumption are doomed to failure*. Through the Internet, users are in the driver's seat, they are active. Moreover, he explains that the Internet is based on a community paradigm, it is absolutely not individualist. Users participate in multiple communities and emancipate in virtual worlds. Therefore, Nifle's suitable services for the Internet are those which facilitate relational games (which are

---

[2]It is a method of teaching introduced by Socrates based on the idea that the truth is latent in the mind of every human being due to his innate reason, but has to be 'given birth' by questions asked by the teacher and answers given by the learner.

ECONOMIC TERMS

Economists are used to talk about a 'service economy', usually contrasted with a 'goods-producing economy'. They refer to an economy based on service exchanges rather than (manufactured or physical) goods. In economics, the triplet good/service/product defines manufacturing exchanges (sales) between economic actors:[a]

- A (manufactured) *good* means a physical, tangible object (natural or man-made) used to satisfy people's identified wants and needs and that upon consumption, increases utility. It can be sold at a price in a market (if the purchaser considers the utility of the object more valuable than the money).

- A *service* is an activity that provides direct satisfaction of wishes and needs without the production of a tangible product or good (i.e., non-material equivalent of a good). It is claimed to be a process that creates benefits by facilitating either a change in users, a change in their physical possessions, or a change in their intangible assets. Examples include information, entertainment, healthcare and education.

- A *product* is a generic term for a tangible good, or an intangible service. This is the output of any production process. A product is anything that can be offered to a market that might satisfy an identified want or need. It is the complete bundle of benefits or satisfactions that buyers perceive they will obtain if they purchase the product.

---

[a]These definitions comes from Wikipedia, the free encyclopedia (`www.wikipedia.org`) and the AmosWEB Economic GLOSS*arama (`www.amosweb.com`).

inscribed and ease human relationships) within virtual communities. Active appropriation services are suitable for the Internet.

From Nifle's reflection we extract two other aspects of the concept of service that we also find important for DSG description:

- Services have a pedagogical dimension; they aim to improve user and provider's mastery;

- Services create relationships between members of communities (social dimension).

1.2.3 Economic definitions and terminology

In this section, we propose to connect our terms to an existing taxonomy: the economic one. Following the economic definitions, information processing is systematically considered as a service. Indeed, services that computers provide are by definition virtual and do not have tangible/physical form e.g., information retrieval, weather forecast, computations, communication tools, etc. Even if services produce 'physical products' (e.g., a book purchased on the Web), they are systematically considered as service (e.g., we are used to talk about the Amazon e-commerce service). We claim that the economic definitions, adapted to a world where physical abstractions exist have limited applications in computer-mediated contexts. In Informatics, a first differentiation is made in services by considering *real world services* i.e., services exchanged in a non computer-mediated context (e.g., school) and *virtual services* i.e., services exchanged in a computer-mediated context or any information systems (e.g., e-learning). In this thesis, we propose to go further by refining the differentiation between virtual services (figure 1.2):

- *Product Delivery*, which are basic virtual goods produced by a pre-determined mechanism answering a parameterized question; akin to function;

- *Dynamic Service Generation*, which are complex virtual services based on a conversation between user and provider which aims to resolve the user's problem.
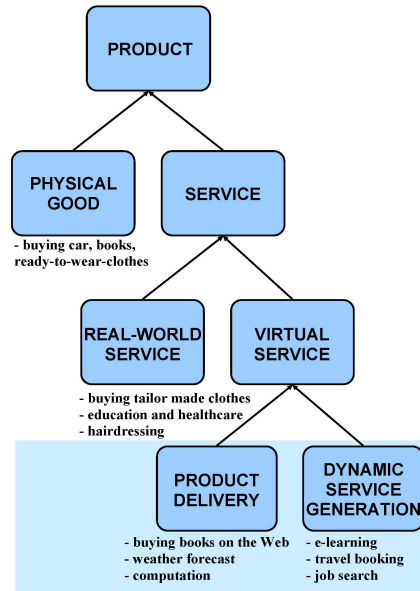
Figure 1.2: Economic taxonomy extension

Notice 'books' appear twice on figure 1.2. Actually, for us buying an identified book at Waterstones or on the Amazon Web site is exactly the same process. But in economic terms, the second case is called a service, because it occurs in a computer mediated context. We will call this second case a product delivery and distinguish it from a dynamically generated service.

The term production applies to both the product and the service, and puts in evidence the idea of process i.e., a composition of activities: *the act or process of producing something*. Production is the set of operations that allows one to get, by combining and composing existing but indirectly usable products, a new (manufactured or not) good/product adapted to fulfil a specific need. Production was firstly (19th century) limited to the process of creation (e.g., agriculture) but was extended to the idea of transformation. In order to detail the production process, we will use the term *generation* for the act of bringing something into being when talking about service (or DSG), and the term *delivery* for the act of delivering or distributing something (as goods) when talking about product (or PD).

Concerning service exchange actors, the terms client and seller fit well with the commercial situation. The terms teacher and learner fit well with the (e)-learning situation. We will use the generic terms *service user* and *service provider* to identify respectively the actor, latter called agent, which has a problem or a unprecise need or wish, for whom the service is realized and the agent that offers the service.[3] The term *service exchange* is used to regroup the terms provide and use. Two agents exchange a service when an agent uses a service another agent provides. We will say that a user 'asks' or 'requests' a product and that a product is 'delivered'. On the other hand, we will not say that a user asks or requests a service, but rather asks for a DSG or that a service is generated by a provider for a user.

The systems (i.e., computing systems), realized and used both by human and artificial agents, that allow DSG and PD are respectively called *Dynamic Service Generation Systems* (DSGS) and *Product Delivery Systems* (PDS). PDS allow the production of one and only one well-defined product, whereas

---

[3]In the literature, depending on the situation, the *service provider* could be called server or service producer, and the *service user* could be also called client, consumer, customer, recipient, requestor, etc.

DSGS enable the generation of many kinds of different, dynamically constructed services for/by human or artificial users/providers. One may say that PDS exist. DSGS do not exist for the moment. A PDS is generally formed only by a service provider (i.e., a PDS can reduce to a service provider), whereas a DSGS includes all the elements introduced in the subsequent section 1.3.2 and in figure 1.3.[4] The main objective of the rest of the chapter is to propose a set of characteristics that will help to progressively shift from PDS to DSGS.

> DEFINITION: DYNAMIC SERVICE GENERATION
> The process of exchanging services, constructed on the fly, and created specifically following the conversation between a service provider and a service user. This unique dynamic process establishes relationships within communities and improve user and provider's mastery by solving a problem or answering a not necessarily clearly specified need or wish.

> DEFINITION: PRODUCT DELIVERY
> The process of supplying products, obtained as the result of the activation of a provider's functionality in order to answer to a well specified user's need or want. This one-shot invocation generic process does not improve user or provider's mastery and hardly help to answer a need or wish not expressed following a specified set of parameters.

1.3 Dynamic service generation

1.3.1 DSG examples of scenario

In order to have a real image of what DSG is, let us consider some examples. As explained in the beginning of the chapter, human beings have always generated services to one another, so let us start with human-oriented scenarios and continue with computer-oriented ones.

1.3.1.1 Human-oriented service scenarios

- A **travel planning** requires a dynamically generated service, most of the time accomplished by a travel agency, allowing a user to express his/her real objectives, possible constraints on dates, his/her centre of interests, the preferred transportation, priorities, family/professional constraints, etc. It is much more than simple ticket buying (cf. the scenario reported by Clancey [Cla05]).

- When a client goes to an automatic **train ticket** machine he/she knows exactly what he/she wants and expects to obtain quickly the product. Sometimes, clients wait half an hour at the train ticket office because they know that the automaton would fail in satisfying their request, or would not be able to answer a specific question. They want to talk to a person. They want a service to be generated by this person.

In general, in everyday life, it is not difficult to make a difference between PD and DSG. However, these examples show that the two approaches are not opposed but complementary. Sometimes user's needs are perfectly addressed by automatic machine or a simple product selling Web site.

---

[4]Therefore, we will say for example 'a PDS has the property P' where as we will say 'a service provider in a DSGS has the property P'.

1.3.1.2 Computer-oriented service scenarios

Of course, the previous examples may be viewed as future computing scenarios. More generally, most domains of Informatics (agent and MAS, SOC, e-learning, remote collaboration, distributed systems, etc.) that deal with the concept of service will benefit from DSG.

- In some aspects, DSG is analogous to **information retrieval** with the following properties: (i) the database is so large that users cannot specify a single, simple database query for which there is a single answer; (ii) users may not necessarily have a definite set of constraints in mind when starting the search task, or may change constraints during the message exchange, and (iii) there may be multiple search goals; what satisfies a user may not satisfy another one.

- DSG could as well be a base for development of future **Intelligent Tutoring Systems** (ITS) in e-learning. For the moment, ITS are most of the time limited to information transfer: the 'tutor' teaches what it knows to the learner. There is no real interaction (i.e., changes of state) between them: the tutor hardly learns anything. DSG should support exchanges in ITS and allow them to be more dynamic and thus to learn and create more knowledge available to both the tutor and the learner. This aspect is more precisely studied in section 1.4.3.3.

- **Pervasive computing and ambient intelligence** are two domains of application of DSG. An ambient intelligence system necessarily includes human and artificial agents and the service exchanges in such a system are strongly adapted to specific users. Ambient intelligence systems need to be dynamic, pro-active, evolutive, etc.

- **Collaboration** not only allows a community to perform a set of joint activities effectively, but also allows the members of that community to learn new knowledge and skills and to improve shared understanding overall. A crucial requirement of collaborative environments consists in enhancing learning as a result of communicating and exchanging services in a collaborative perspective. DSGS may support such collaborative environments, because of the pedagogical dimension underlying DSG (section 1.2.2 and section 1.4.3.3).

- As a last example, we simply mention the progress that actors such as Amazon make when they introduced marketing concepts in their **e-commerce** Web sites. For instance, they propose on-line selling services more adapted to users and strategically designed in order to improve sales. The scope of DSG seems to satisfy the same need for adaptability than e-commerce scenarios.

1.3.2 Dynamic service generation system elements

1.3.2.1 Agent

In order to develop interaction based service exchange models, the first obvious step is to adopt a unified view of interacting entities. PD and DSG involve two kinds of entities. The first one is most of the time called: human user, human being, natural entity, etc. The second one is most of the time called: computer, machine, agent, software system, etc. In order to model uniformly these two different kinds of user/provider, we will adopt a unified view of these entities as it is currently done within the MAS community [HS98, Fer99, Jen01, BD01, Woo02], by means of one of the most advanced autonomous, intelligent and interactive entities: agents. The *agent* concept is for the moment the best operational metaphor of humans in Informatics. In the rest of the thesis, for the sake of simplicity, we consider two types of agents: *Human Agent* (HA) and *Artificial Agent* (AA).[5] Three types of interactions occur in PD/DSG: AA-AA, AA-HA, HA-HA. A PD/DSG may occur provided by any kind of service provider

---

[5] The (artificial) agent notion is more precisely detailed in section 2.3.1.

agent for any kind of service user agent. We use the term *conversational process* to identify the conversation among these two agents. We use the term *community* to identify the social group in which agents evolve and generate services to one another. Figure 1.3 summarizes DSGS elements.

DEFINITION: AGENT

Computational metaphor that represents active entities (i.e., human or artificial) involved in dynamic service generation or product delivery. Agents are supposed autonomous, intelligent and interactive; they can play the role of service provider or service user.
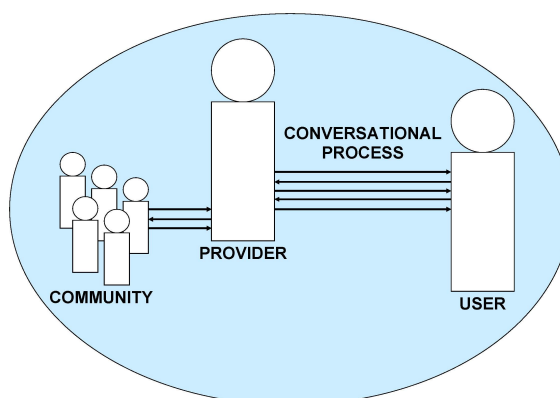
Figure 1.3: Dynamic service generation system elements

1.3.2.2 Community

A notion that appears in DSG is that of community. A PDS may be considered as an independent system, isolated and disconnected from other PDS, simply in relation with users. On the contrary, a DSGS is composed of a set of members of a virtual community, allowing a service provider to dynamically construct the service it has to generate, both by using its own capabilities and by asking to other community agents to generate services. The community is the virtual place in which complex service exchanges occur by means of a set of simple two agent's service exchanges. The community reflects the set of roles played by community agents and relationships created between agents at the time of service exchanges.

These communities may be of little or large scale. They may be persistent for a long time or very ephemeral. They may be explicit (e.g., a working group) or implicit (e.g., all Web users). They are highly dynamic and evolve according to interactions and services generated.

The notion of community is strongly influenced by GRID and MAS organizational structures. Section 2.4.5.3 details the analogy between them.

DEFINITION: COMMUNITY

The dynamic social group (virtual or not) in which the agents evolve by using and providing different dynamically generated services with one another. The community is the context of conversational processes and is defined by the set of relationships created between agents.

1.3.2.3 Conversational process

PD can be considered as the result of a one-shot interaction process between a pair: user – provider, whereas a DSG may be viewed as the result of the activation and management of a unique process defined by the triplet: user – conversational process – provider. As shown in section 1.2.1 services are (i) associated with processes; (ii) constructed by means of conversations. We call the merging of both a conversational process (figure 1.4). The agents playing the roles of provider and user pre-exist to the DSG, whereas the conversational process represents the conversation these two agents have to realize the dynamically generated service. A conversational process is a long lived interaction that allows user and provider to dynamically and respectively express their need or wish and capabilities or constraints. Interaction plays a central role in DSG.

A *process* is defined as a naturally occurring (or designed) sequence of operations or events, which produces some outcome. A process can be identified by the changes it creates in the properties of one or more objects under its influence. In DSG, when the sequence of operations (i.e., messages) is designed, or follow a protocol, the conversational process is called a *conversation*. When the sequence of operations (i.e., messages) occurs naturally and dynamically without being predetermined, the conversational process maybe called a *dialogue* (section 2.3.3.2 details these aspects of agent conversation modelling). The outcomes produced by conversational processes are the result of the DSG itself and the elements produced at each step. Changes created on the object under the influence of the conversational process (i.e., agents) are changes of their internal states. Each move in the state produces changes that are directly influenced by the previous state and the interpretation of the last message as figure 1.4 shows. Indeed, a conversational process is a set of interactions and an interaction between two entities implies an action to occur on the interacting entities. Changes of state is one of the reasons that explains why each DSG is unique (changes of state are stored in a history).
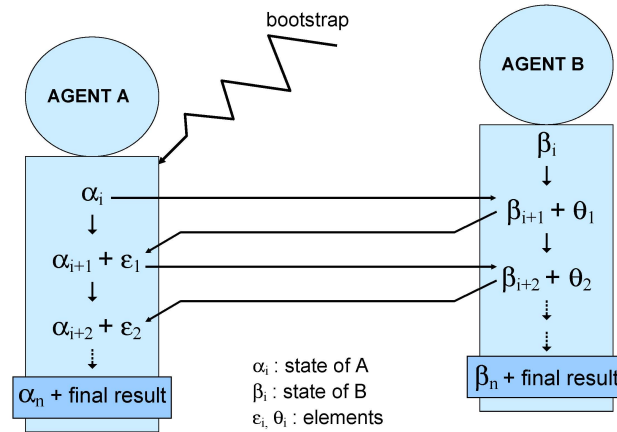
Figure 1.4: Conversational process

The beginning and the end of a conversational process implies the beginning and the end of the DSG. We talk about DSG *bootstrap* to identify the beginning of the conversational process. We call *elements*, the different changes provoked by interactions occurring during the conversational process. Elements are local to each agent implicated in the conversational process. The end of the DSG is characterized, for each agent, by a *final result* as the integration of all process elements. This final result may be a physical or a virtual good, but also knowledge, emotion, information, advice, qualification, or any kind of answer. The final result may be 'agent internal' (e.g., emotion, knowledge), or 'agent external' (e.g., a physical/virtual good).

DEFINITION: CONVERSATIONAL PROCESS

A long-lived, dynamic, unique, and not a pre-determined set of interactions (i.e., conversation or dialogue) allowing user and provider to respectively express their needs and constraints during the service generation. A conversational process starts with a bootstrap, produces elements local to agents and finishes by a final (internal or external) result.

1.3.3 Dynamic service generation vs. product delivery

As told before, DSG is opposed to the classic PD approach. We outline here after some fundamental differences between PDS and DSGS (summarized in table 1.1); that permits to characterize DSGS a contrario (i.e., by opposition). Besides, these differences are not necessarily all and always true in DSG. For example, a user may know his need (D1), but may not know the way to express them (D3).

D 1 (Need expression) *In PD the user exactly knows what he wants. His need is clearly specified. He knows which provider can help him to find it. In DSG, on the contrary, a user may want a service to be generated for him without knowing exactly what he wishes; he can even be unaware of the fact that he needs a service to be generated (D18). The user is simply in a bootstrap situation, some of his needs appear during the generation process, other ones disappear.*

D 2 (Offer expression) *In PD, the user knows what the system can offer him. He knows it because of previous PD, or recommendation, or advertisement, or precise description. Besides, a registry defines the provider's available capabilities. In DSG, the user elicits and understands what the service provider can offer him by interacting with it. He discovers some of his needs in the same time as he discovers the service provider's capabilities (he discovers also other capabilities that would be used in future DSG (bootstrap)). On the other side, the service provider constructs something to offer to the user, according to the conversation it has with him. In DSG, registries or any kind of index are difficult to set up as each dynamically generated service is unique (D4) and fits a very specific need (D7).*

D 3 (Request expression) *In PD, the user knows how to express his request. He should adapt to the provider's language. This is not the case in DSG. The service provider should help the user to express his request and try to understand his language. Notice the important aspect of learning (D11).*

D 4 (Uniqueness) *Several requests processed in a PDS, for the same or another user, produce the same type of delivery. Some products may be exactly the same if we consider the purely functional capability of* stateless *service provider. On the contrary, generated services can never be the same as they depend on a unique conversational process. Moreover, each DSG is also unique because of D5.*

D 5 (History) *Dynamically generated services depend on one another; they are part of a history. A dynamically generated service will of course depend on the conversational process, but also on the current states of the agents, themselves changed by previous DSG. Three generated services in a special order would not have been the same in another order. PDS do not have history; most of them do not even have a state.*

D 6 (Subsumption) *PDS cannot realize DSG; PD may be composed of other PDs even if it is difficult (see for example business process management, section 2.2.1.3). On the other hand, DSGS subsume PDS, which means that a system allowing DSG allows also PD. A dynamically generated service can be the result of the aggregation/composition of other dynamically generated services but also of product deliveries. It is important to note that the goal of DSGS is not to substitute PDS, but to be complementary. Very often, a PD perfectly answers the user's needs and a DSG is not necessary. If a user has precise needs or precise ideas of what he wants the system to give him, then interactions are simple (e.g., getting a person's phone number from a Web site shall always be a PD).*

D 7 (Development) *A PDS is pre-developed with a clearly predefined goal for the potential user and is supposed to correspond with a well established and clearly identified need (a market). On the contrary, a DSGS is offered within a service domain, so the user's specific objectives have to be defined during the DSG conversational process.*

D 8 (Lifetime) *A PDS has a long lifetime as it is developed once and never changes. On the other hand, DSGS life cycles are ephemeral as they change and evolve with each DSG (which does not mean they have a short lifetime: D5, D9, D10, are only compatible with a long lifetime).*

D 9 (Evolution) *A PDS evolution is slow, as it requires modifications in the conception, design and development - a revision of the whole life cycle. A DSGS evolves naturally according to change implied by different dynamically generated services. DSGS evolve also because of D10.*

D 10 (Reasoning) *PDS are inactive when not engaged in a delivery phase. They only wait for the next requests (as for example Web servers). Conversely, a DSGS is never inactive; it is perpetually evolving. Agents composing it learn and reason on previous generations to improve the next ones. This evolution is based on the DSGS history (local to agents) (D5). A part of this reasoning is done dynamically (during service generations) and another part is done statically (after generations phases) for example analyze and change according to statistics. See also section 1.4.3.3.*

D 11 (Pedagogy) *A PD does not create knowledge. Users know what they want and there is no creation of new knowledge. Inversely, there is a pedagogical dimension in DSG. The fact of generating and using a service generates knowledge both for the service provider and for the service user. If D1 is merely verified, then users discover at least what they really want or do not want. See also section 1.4.3.3 and section 1.2.2.*

D 12 (Satisfaction) *The added value by a PDS increases with the number of products delivered by the system, but the satisfaction of the user stays the same. The added value by a DSGS increases proportionally with the users' satisfaction of the final result, which entails an indirect publicity (reputation) for service providers and thus stimulates new users ready to have similar services (bootstrap).*

D 13 (Retraction) *During a PD, the user cannot change his mind. He cannot retract and must wait for the end of the delivery (generally fast) to express a new request with his new need or want. Conversely during a DSG the user can change his mind at anytime during the process. Especially, he does that according to the service provider's reactions.*

D 14 (Psychology) *A PD does not provoke emotional reactions. The user knew what he wanted beforehand, no surprise or any other emotions are implied by the PD. Some emotions such as happiness, disappointment, etc. may occur, but they are related to the product received and not to the process of delivering this product. On the contrary, a dynamically generated service has a psychological effect on the agents involved; especially, as it implies and provokes emotions (positive and/or negative). The dynamicity of the process (everything is not previously clearly defined) lets unforeseen events occur generating emotional reactions. These emotions play an important role in the agents DSG valuation (D15).[6]*

D 15 (Valuation) *The use of a PDS is easily valuable and billable. Since several PDs are the same (D4), corresponding to a pre-defined need (D7), and since retraction is not possible (D13), the valuation (price) is calculated as a function of offer and demand. Besides, remote procedure calls executions may be anticipated and resources (time and space) may be previously reserved (algorithm complexity can be predetermined). Inversely, a generated service is neither easily valuable nor billable. Since generated*

---

[6]D14 applies directly to human agents, but we can also notice that 'emotion modelling' for artificial agents is a domain of research in its infancy.

services only have interests for a given user (at a precise time) and since they will never be regenerated in the same way, a function of offer and demand to help to define the value, is not easily found. It is impossible to really estimate how much a user should 'pay' for a generated service. A dynamically generated service may also be viewed as an investment whose potential return is very difficult to evaluate. A DSG is also hardly valuable because of D16.[7]

D 16 (Predictability) *PDS are able to announce the result of their use; they can 'show' future results (i.e., obtained products). A PDS can for example explain situations before and after the PD. On the contrary, a service provider in a DSGS must gain the trust of its users. It cannot announce a final result, nor guarantee it. Where a PDS cycle of use is defined by: estimation, order, delivery; a DSGS cycle of use is based on a trust delegation. A service provider takes intelligent decisions for the user by inferring on its intentions. It must sometimes take the initiative (D18).*

D 17 (Determinism) *PD is a deterministic process that leads to a product. Computing principles behind PDS are deterministic: algorithms, execution, etc. On the other hand, DSGS have non-deterministic behaviour; we should not really anticipate their final results. Computing principles should be non-deterministic (section 1.4.3.2).*

D 18 (Behaviour) *PDS are passive, and never take the initiative; they systematically wait to be called. They are able to publish and advertise their capabilities but not to start the delivery process. On the other hand, a service provider in a DSGS is active and proactive; it may start a DSG process without the user's previous solicitation. It must be able to note or detect bootstrapping situations when they occur (i.e., potential users). Pro-activity should be appreciated if it is necessary (D16) and allows the DSGS to increase its added value. Besides, a user cannot appreciate a non desired solicitation. For these reasons, DSGS are good models of ambient intelligence.*

D 19 (Description level) *PDS simply deal with the syntactic level (i.e., the basic interpretation of expressions) in their interactions among agents. DSGS must also deal with semantic (i.e., meaning of expression) and pragmatic levels (i.e., the context of interpretation) within interactions.*

1.4 A list of dynamic service generation characteristics

1.4.1 Methodology

This section starts establishing a set of DSG characteristics (continuing through chapter 2). This is not an exhaustive list, but it helps us to concretely represent the path towards DSGS. Actually, there is not a formal specification of DSGS, but there is a continuum of systems that start from today's PDS (Web services, MAS, distributed systems, the Grid, Semantic Web services, etc. as presented in chapter 2) and converge to tomorrow's DSGS. The more a system achieves the characteristic of the list, the more it tends towards DSGS.

We have identified two main types of characteristics. Characteristics of the first type, introduced in this chapter, were identified thanks to a reflection about today Informatics. Some appear to us as an evidence, such as C1[web]. Some were influenced by reflections about Informatics done by researchers, such as for example Wegner, suggesting a radical shift in Informatics. This is the case of C2[interaction], C3[non-determinism]. Some are the result of our personal synthesis about a specific subject e.g., C4[learning]. Some are related to other domains of Informatics that seem major for service exchange (C5[reasoning], C6[human]). Finally, some are directly related to our practical experience

---

[7]We may quote the French writer Alexandre Dumas, 'Il y a des services si grands qu'on ne peut les payer que par l'ingratitude'.

Table 1.1: Summary of the differences between PDS and DSGS

Nb.	Name	Product Delivery	Dynamic Service Generation
1	*Need expression*	the user exactly knows what he wants	the user has an unclear wish (bootstrapping situation)
2	*Offer expression*	the user knows what the system can offer him	the user elicits progressively the provider's capabilities
3	*Request expression*	the user knows how to express his request	the provider adapts to the user's language
4	*Uniqueness*	same type of delivery	unique generation of service
5	*History*	no history	historical dependence of services
6	*Subsumption*	cannot realize DSG	can realize PD
7	*Development*	pre-developed with a clearly defined goal	offered within a service domain and constructed dynamically
8	*Lifetime*	long lifetime	ephemeral life-cycle
9	*Evolution*	slow evolution	dynamic and natural evolution
10	*Reasoning*	no reasoning	static and dynamic reasoning
11	*Pedagogy*	no knowledge creation	pedagogical perspective
12	*Satisfaction*	stay the same for each delivery	increase proportionally with each generations
13	*Retraction*	no possible retraction	anytime mind changing
14	*Psychology*	no emotions	implies (positive or negative) emotions
15	*Valuation*	easily valuable and billable	hardly valuable or billable
16	*Predictability*	result(s) predictable	based on trust delegation
17	*Determinism*	deterministic process	non-deterministic behaviour
18	*Behaviour*	passive	active and pro-active
19	*Description level*	syntax	semantics and pragmatics

(C7[programming]). Characteristics of the second type, introduced in the next chapter, were naturally identified during our examination and understanding of three important domains of Informatics. All these characteristics are formulated one-by-one along the next sections each time an important aspect for DSG is noticed. These are all summed up at page 215.

1.4.2 Foundational domains of Informatics

We adhere to the realist position taken by Singh and Huhns [SH05] that:

> 'It is not possible for computer scientists to develop an effective understanding of SOC by merely studying the basics standards for Web services but rather by examining several areas of computer science that come together in connection with services. Many of the key techniques now being applied in building services and service-based applications were developed in the areas of database, distributed computing, artificial intelligence, and multi-agent systems.'

Therefore, the first characteristics come from five research domains of Informatics that we further think to be the foundation of DSG. These five domains emerge from the distributed aspect that Informatics has taken in the last decades: i.e., distributed documents, resources, applications knowledge, and systems (figure 1.5). Working on distributed approaches is already a research challenge because it faces

with an intrinsic paradox: the more distributed objects are the greater is their potential value, but the harder it is to extract that value [SH05]. Taken one by one, these domains present many interesting and promising qualities to realize future DSGS. However, DSG takes-up the challenge of a common development and integration of these domains.
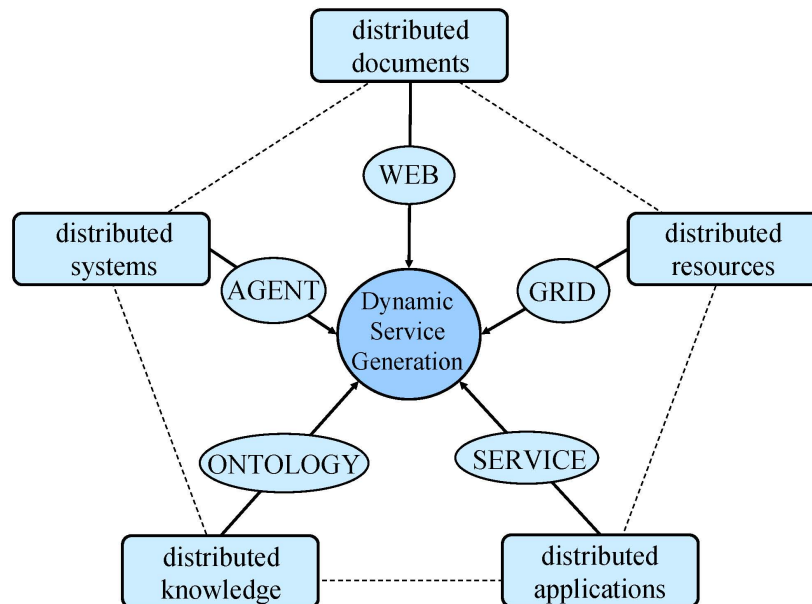
Figure 1.5: DSG as the integration of five domains of Informatics

Distributed documents: the Web. The World Wide Web is a network of servers linked together by common protocols, allowing access and exchange of millions of hypertext documents. The Web created a worldwide participatory movement towards a new world where everyone can start a relationship with everyone else, independently on the distance, following his initiative and creativity. Furthermore, the Web is an interaction space where agents in communities exchange information and knowledge. The challenge is to keep this statement true when implementing service exchange on the Web. For the moment, the Web is a non-structured set of information intended for humans, but unfortunately not equipped with any semantics. Nevertheless, the Web is a key element of DSG. DSG would enable the Web to become a real collaboration space by allowing all kind of agents to generate services with one another.

Characteristic 1 (web)

> *DSGS should be Web oriented. They have to be accessible through Web standards and technologies but the Web also has to evolve to fit other DSGS requirements (e.g., statefulness, semantics, conversation, etc.)*

In a sake of clearness other characteristics are described within sections that deal with domains they are related to: in section 2.2 for service, in section 2.3 for agent, in section 2.2.3 for ontology and in section 2.4 for GRID. In the next sections some important characteristics that are not directly related to service, agent or GRID are identified.

1.4.3 Other domains of Informatics

1.4.3.1 Interaction

Social intelligence is a phenomenon that can not be explained by a 'linear' model: the whole is not the sum of its parts. The emergence of social intelligence in human societies may well be accepted as a fact. However the same is not straightforward in societies of artificial agents. The whole is often considered as greater than the sum of its parts because of interactions between these parts. Humans interact since centuries, but artificial agents do not yet do in the same performing way as humans do. Interaction in Informatics is for the moment rather poor, compared to calculation. Since it was created, the computer is used to help humans: it can forecast the weather, it can calculate the resistance of a bridge, it can mine large repositories of data helping humans to take a decision, or it can solve many equations that humans need to resolve. In fact, humans do not really interact with their computers and these computers do not really interact together, humans simply ask them to answer some precise questions, to compute some algorithms! If we want to use computers in a non limited way, to profit of their knowledge and capabilities: we have to interact with them, let them interact one another and profit of these interactions in a way similar to the one used when profiting of human collaborative interactions.

Characteristic 2 (interaction)

DSGS should be interaction oriented as the main difference between PD and DSG is the conversational process. Interaction should be intrinsic to the underlying model of DSGS. The behaviour of a DSGS should emerge from interactions between agents of this DSGS.

WEGNER'S INTERACTIVE COMPUTATION

Several researchers consider interaction as a key element of Informatics: Turing, Milner, Hewitt, McCarthy, etc. We especially appreciate Wegner's approach that consists in shifting from an algorithmic view into an interactive view of Informatics (Interactive Computation) [Weg97, WG99, WG03, GW04]. Wegner's analysis returns to the original work of Turing about *choice machines*. Choice machines extended classical Turing Machine to interaction, allowing a human to intervene in the computation to make a choice that changes the rest of the computation. Unfortunately, choice machines were historically forgotten and the Turing thesis myth created: Turing Machines models all computation rather than just functions. However, as Turing himself said, Turing Machines could not provide in principle a complete model for all forms of computation i.e., computers cannot be reduced to Turing Machines [GW04]. That is why interaction needs to become more and more important in computing aiming to overpass calculation (shift from algorithms to interaction). Wegner explains that:

> 'It is time to recognize that today computing applications, such as Web services, intelligent agents, operating systems, and graphical user interfaces, cannot be modelled by Turing Machines; alternative models are needed.'

He explains that computation is no longer executed in a closed world, but in an open system where many inputs and outputs may be interleaved with one another without being all defined before computation begins. Wegner's analysis seems very important for DSG. DSGS are among those systems whose behaviour may not be modelled and represented by a Turing Machine.

1.4.3.2 Non-determinism

A function (or an algorithm or a system) is deterministic if the result (i.e., outputs elements) is only determined by the entry (i.e., inputs elements). Any deterministic algorithm produces the same result for the same entry. During a deterministic computation, at every step of the computation, there is only one possibility for the following computational move.

One should speak of a non-deterministic system if from a certain state of the system there are several possibilities of choice for the next state. Thus, non-determinism means something neither determined nor settled by previous information. A non-deterministic computation permits more than one choice of the next move at some stage in a computation. Besides, this computation may have different results. Non-deterministic program execution and termination are not new in Informatics since they were introduced by Dijkstra in 1975 [Dij75]. Non determinism is encouraged in languages such as ADA or Constraint Satisfaction Programming (CSP) languages and algorithms [Dec03]. The key idea is that expressions in a non-deterministic language can have more than one possible value. With non-deterministic evaluation, an expression represents the exploration of a set of possible worlds, each being determined by a set of choices. Non-determinism is strongly bounded to interaction. The limited perception of the world defines distributed approaches and implies interactions. It is the cause of the emergence of a non-deterministic behaviour. Section 3.4.2 presents an experimentation of the STROBE model using a non-deterministic interpreter that enables the dynamic specification of a service.

Characteristic 3 (nondeterminism)

> *DSGS should be non-deterministic as the service exchanges that may occur are non predictable and depend on the conversations. The non-deterministic behaviour of the system should not be predicted (thus expressed by a procedure) but should emerge from interactions.*

Remark – We should be careful with the meaning of 'non-deterministic'. It means that the process achieves i.e., finds an answer (a solution or a statement that there is no solution), but the way it is done is non-deterministic i.e., can be each time different and not foreseeable.

1.4.3.3 Learning

Section 1.2.2 raises the pedagogical aspect of service exchange. D10 and D11 respectively express the reasoning and knowledge creation aspect of DSGS. D1 supports the view that DSG implies learning. At the end of the process the user has at least learned the way to express his request and what he really wants. Next time he will ask for a service to be generated, he will use what he learnt during the first DSG. Therefore, we may say that DSG implies learning: new knowledge is (statically and dynamically) acquired by agents generating and using services. If we consider the two types of agents involved in DSG, we distinguish between (i) machine learning and reasoning for AAs; (ii) e-learning, i.e., knowledge, information, know-how, or skill acquisition for HAs. This section deals with the two as they are closely bound together.

Concerning e-learning, we do not consider DSGS as Intelligent Tutoring Systems (ITS) that means systems whose goal is to guide learners in their learning process thanks to a tutoring mechanism. There are two roles in ITS, tutor and learner, and learning is always done one sided (from tutor to learner). Actually, DSGS are rather highly interactive systems and interaction plays the key role in the learning process for both agents involved in the DSG. Interaction between two entities implies a change of state on the interacting entities. In DSG, this change of state is viewed as learning. The learning process should be seen as a co-construction of knowledge (social constructivism). For this knowledge creation process to be unlimited, and to (ac)cumulate upon time, the two entities have to learn from each other during the process and re-inject what they learnt into the loop. Consider, for example, an *interactive CD-Rom* about a specific domain. The knowledge that the user of the CD-Rom would get is inevitably finite and limited due to the fact that the CD-Rom is a fixed support unable to change and to evolve to acquire

new knowledge tailored to the user. Today this support is more or less obsolete in e-learning practices. However its principles are still quite diffused, and e-learning supports are most often content centred: little effort is reserved in e-learning to the evolution of knowledge, in the system, as a consequence of interactions with the users.

This paragraph details a simple overview of learning, considering agent learning i.e., reflecting about what an agent learns by itself and by/about its interlocutors. Three types of learning are distinguishable:

1. *Internal learning* (for HAs) or *machine learning* (for AAs). Considering AA, it is the classical approach as old as AI itself. Agents learn from a Knowledge Base (KB) by abstracting and generalizing, applying some abduction, deduction, induction, rules on the KB. The most widespread approach in machine learning is reinforcement learning [KLM96];

2. *Learning-by-being-told*. It comes from distributed AI and is issued from the instructional effect where an agent changes or creates representations according to received messages;

3. *Learning as a side effect of communication*. It is quite 'serendipitous'[8] as it occurs when an agent learns without being aware of it, when it learns because of a beneficed communication and collaboration context or any kind of a posteriori useful interaction with any agent not specifically called a priori for the purpose of mutual teaching or learning.

DSG considers these three types of learning. They can be all applied to any type of agent. Types 2 and 3 have been influenced by Conversational Framework of Laurillard [Lau99], who argues that learning can be viewed as a series of teacher-learner conversations taking place at multiple levels of abstraction. As summarized in [LSL+00]:

'At the most general level of description, the learning process is characterized as a 'conversation' between teacher and learner, operating on two levels, discursive and interactive, the two levels being linked by the twin processes of adaptation and reflection.

Learning happens in the context of collaboration and conversation as the side effect of activities and observations that have not learning itself as their aim [Cer94].

Characteristic 4 (learning)

DSGS should support learning, knowledge creation and sharing in order to fulfill the pedagogical dimension of DSG. This is both true at the agent individual level (service user and provider change of states) and at the social level (new knowledge common to and shared between agents emerges from DSG).

A part from this learning occurs dynamically during the generation process (type 1, 2, 3). A part occurs statically (type 1). DSGS agents have persistent states or KB they should enrich after every DSG. Insofar as this KB keeps the history of past experiences, it becomes a source of optimization in subsequent interactions.

Characteristic 5 (reasoning)

DSGS agents should be able to reason and learn according to their current state and history in order to evolve and improve the way they provide and use dynamically generated services. This reasoning process should be dynamic during DSGs, and static between DSGs.

---

[8]From serendipity: the faculty of making fortunate discoveries by accident.

This thesis details mainly the second type of learning. The STROBE model, introduced in chapter 3, implements a learning-by-being-told approach allowing agents to learn at the 'data', 'control', and 'interpreter' levels as presented in section 3.3.2. This thesis does not deal with type 1 and 3. However, section 3.3.4.4 explains how to address type 1 in the STROBE model, and [JEC05] proposes a theoretical bootstrap to address type 3 and tend towards 'serendipitous' learning with enhanced presence by HA and AA communication.

1.4.3.4   Human integration

The integration of humans in the loop strongly depends on domains such as human-computer interaction, natural language processing, dialogue and discourse processing, visualization, emotion modelling, etc. that improve the way humans and machines interact with each other. No matter the model or the architecture of a system, if human users are concerned, then these domains play an important role. The challenge of these domains for DSGS is to construct interfaces that do not limit the user's expressivity and, if possible, that enhance it. Adaptation should come from the service provider and as little as possible from the HA user.

Characteristic 6 (human)

> *DSGS should integrate human agents by means of highly suitable interfaces, visualization mechanisms and miscellaneous means of interaction. Moreover, they should be able to understand and deal with the languages employed by HA users. More generally, DSGS should deal with all domains of Informatics that tend towards a better integration of humans in the loop.*

1.4.3.5   Dynamic concepts of programming

DSGS should be highly dynamic systems (C2[interaction], C3[non-determinism], C4[learning] . . . and to come C7[programming], C9[registry], C10[negotiation], C12[process], C15[conversation], C16[dialogue], C21[stateful]). This section presents some dynamic concepts of programming that seem to be good, concrete tools to implement DSGS, assuming of course that they are better integrated in higher level design and implementation methodologies.

Dynamically interpreted languages.   Defining a computer language means to define an execution machine for this language. An execution machine takes as an input a program source and returns as an output the result of its evaluation. Two types of execution machines exist: either a program is compiled and then executed by a specific (eventually virtual) machine (e.g., C++, Java), or it is interpreted directly by another program, named interpreter, or evaluator [ASS96] (e.g., Lisp, Scheme, Prolog, Maple). The two techniques have both advantages and drawbacks (addressing, syntactic analysis, binding, etc.), but it is easily comprehensible that interpreted languages are more dynamic than compiled and executed languages. The compilation process ties a program content once for all. Compiled languages perfectly map with fixed and predetermined algorithms, whereas interpreted languages seem more adapted to shift from algorithms to interaction as presented in sidebar page 44. For example, the dynamic modification of the execution machine at run time (during the evaluation of an expression) is quite easily feasible with interpreted languages such as Scheme. This is detailed in section 3.2.3 where STROBE agents dynamically change their message interpreter while interpreting the stream of messages.

Streams.   As Abelson and Sussman [ASS96] explains, streams allow to model systems and their evolution in terms of sequences that represent the time histories of the systems being modelled. They are an alternative approach to modelling state. As a data abstraction, streams are sequences as lists. The difference is the time at which the elements of the sequence are evaluated: list elements are evaluated when the list is constructed, whereas stream elements are evaluated only when they are accessed. The basic

idea consists in constructing a stream only partially, and to pass the partial construction to the program that consumes the stream. We identified two reasons for using stream processing in DSGS:

- Stream processing allows to model stateful systems without ever using assignment or mutable data. It is often easier to consider the sequence of values taken on by a variable in a program as structure that can be manipulated, rather than considering the mechanisms that use, test, and change the variable. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment (side effects). Having stateful programs without the assignment problem should be very interesting for DSGS insofar as stateless services are easier to compose with one another in processes.

- Agent communication inputs and output flow of messages may be represented by streams. Streams are a smart data structure to model agent conversation as lazy evaluation (see below) is relevant to express the natural delayed aspect of interactions: an agent may delay the production of the next message until it interprets its interlocutor's reaction to its previous message.

In this thesis, we address, in the STROBE model, only the second reason. We propose, in chapter 4 a solution for modelling agent conversation by means of a computational abstraction based on stream processing. This chapter, also gives more fine grained explanations about streams. The stream approach is also defended in Wegner et al. [WG99]

Lazy evaluation.   Most applicative/functional languages as Scheme [ASS96] or Lisp are applicative order languages, that means that all the arguments of functions are evaluated when the function is applied. In contrast, normal order languages (such as Daisy [Joh89]) delay evaluation of function arguments until the actual argument values are needed. Delaying evaluation of function arguments until the last possible moment (e.g., until they are required by a primitive, or returned as an answer) is called lazy evaluation. A lazy evaluator only computes values when they are really required avoiding to compute values that are not really needed. For example, lazy languages allow to define the `if` special form as a classical function. Lazy evaluation often support the implementation of streams (sidebar page 140). This seems a good idea for the development of the DSGS in which one cannot anticipate, or foresee messages that are going to be exchanged during conversational processes.

Dynamic typing.   In dynamic/weak typing (opposed to static/strong typing) languages, neither the variable name, nor the binding between the variable name and its value contains any information of type. This information is only stored in the value itself. Dynamic typing in particular interesting for generic procedures i.e., procedures that can be applied to any type of argument. Such procedures are interesting for DSGS because they avoid to limit service generation to well specified remote procedure call. [Weg96] explains why strongly-typed languages limit interoperability (sidebar page 54). Dynamic typing and lazy evaluation has been defended as necessary for Web languages [Cer99a]: URI (Uniform Resource Identifier) may be considered variables names in a global repository while XML documents may represent variable values that include type information as the tree of tags.

Continuations.   Continuation is a fundamental notion in programming languages [Que96]. It is part of the execution context as well as the interpreter and the environment (section 3.2.2). When a program is interpreted, the continuation is the following move of the current process i.e., the next expression to evaluate with outputs from the current evaluation. Continuation passing style programming allows to dynamically manage the evaluation progress of programs and capture or change (by escape or resume) its future. In this sense, continuations should be very useful for DSGS. For example, in order to solve the problem of the Web memory, Queinnec [Que00] proposes to use Web continuations: instead of thinking in terms of state and transitions from page-to-page, an alternative view is proposed, where a program is

suspended and resumed, continuations automatically reifying the whole state of the computation. See section 3.5.1 for more detail about continuations.

Constraint satisfaction programming. Constraint satisfaction programming [Dec03] consists in assigning values to variables subject to restrictions (constraints) on the set of values they can take. A solution of a constraint satisfaction problem is an assignment of all variables of the problem so that all constraints are satisfied. A constraint network is defined by a set X, D, C that are respectively the set of variables of the problem, the set of domains for these variables, and the set of constraints on these variables, as well as a solver which executes a specific algorithms (retrospective methods: backtracks, backjumping and prospective methods: look-ahead, arc-consistence, propagation, etc.). The three sets X, D, C come from the problem modelling phase. During DSG, this modelling phase or simply constraint expression should be done interactively with the user. CSP seems interesting for implementing DSGS as they formalize a recursive non-deterministic approach to problem solving: a CSP solver try to find a solution by taking into account one-by-one the constraints expressed by the programmer. In DSG, this constraint expression may be accomplished in the conversational process, each agent executing a solver and taking constraints into account when they occur. This is the idea of dynamic specification presented in section 3.4.2.

Characteristic 7 (programming)

> *DSGS should use techniques and methods allowing a dynamic behaviour of the system as much as possible. Dynamic concepts of programming such as interpreted languages, stream processing, lazy-evaluation, dynamic typing, continuation passing style, constraint satisfaction programming are all promising examples.*

1.5 Conclusion

The main contributions of this chapter are:

- To reflect and define the concept of service in computing mediated contexts;

- To define the concept of Dynamic Service Generation;

- To enumerate a set of differences between product delivery and dynamic service generation;

- To start a list of DSG characteristic (continuing in the next chapter).

A discussion and synthesis about the complete list of DSG characteristics is done at the end of the next chapter in section 2.5.

Chapter 2

State of the art

The more things change, the more they are the same.
Alphonse Karr, French writer (1808-1890)

THIS CHAPTER presents a state of the art of the three main domains to which DSG is related, as well as their interconnections. The overview of SOC introduces the service-oriented architecture principles and their application in DSGS e.g., service creation, execution, registries, processes, etc. SOC evolution, in particular semantics, is discussed. The overview of MAS introduces agents and MAS concepts and specifically detail agent communication approaches, such as the challenge of modelling dialogues. The overview of GRID concentrates on GRID-SOC and GRID-MAS integration approaches preparing the integrated model of chapter 5.

Contents

2.1 Introduction

SOC, MAS and GRID are three very active domains in Informatics. In this chapter, we have decided to relate the different approaches, according to their influences or interests for our work on DSG. We give also an importance to the interconnections between these domains: MAS-SOC, GRID-SOC and GRID-MAS integrations approaches (figure 2.1). Many other elements related to SOC, MAS or GRID are not mentioned here. This state of the art aims to continue the list of characteristics started in chapter 1. Each time an important aspect for DSG is noticed, we express it as a DSG characteristic.
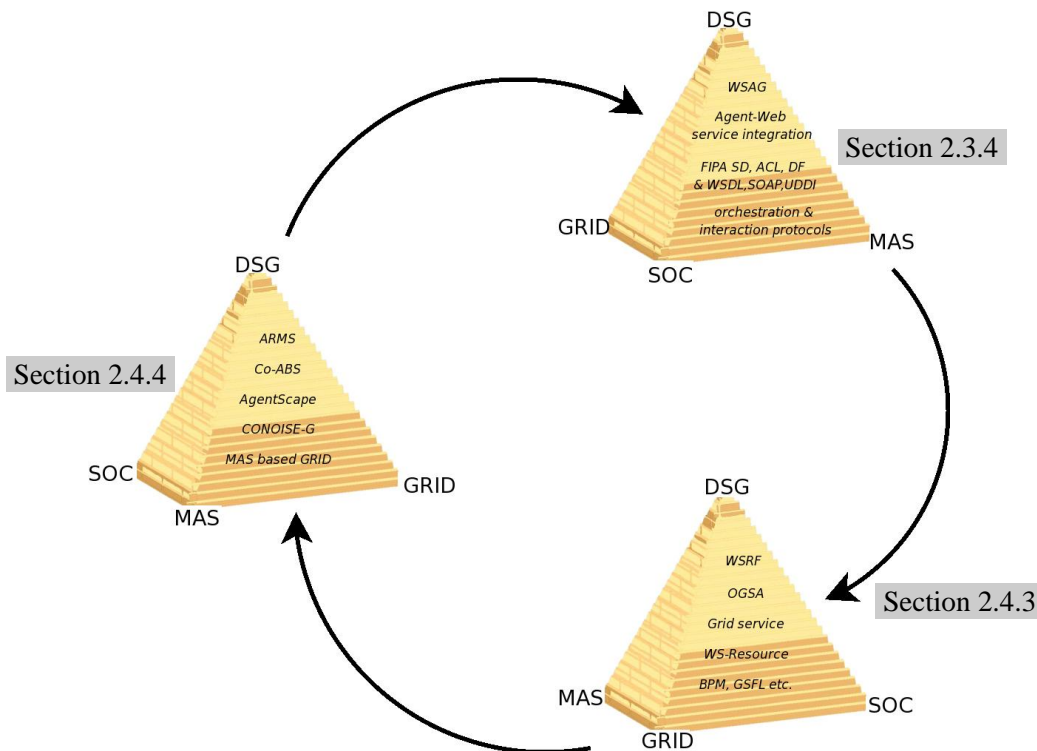
Figure 2.1: Interconnections between SOC, MAS and GRID

Chapter overview. The remainder of the chapter is organized as follows: Section 2.2 is dedicated to SOC and presents a state of the art of the main current framework that implements SOA: Web services. In this section, we detail SOA principles and the different phases of the service life-cycle e.g., service creation, service registry, service contracts, etc. Section 2.2.3 details in particular the semantic evolution of services. Section 2.3 is dedicated to MAS. We first introduce agent types and architectures, then we detail the communication aspect of MAS explaining the role of agent conversation modelling approaches. Section 2.3.4 details in particular related work on agents and Web services. Section 2.4 is dedicated to GRID. We first present GRID concepts and definitions, then GRID-SOC related work is specifically addressed i.e., Grid services (section 2.4.3) as well as GRID-MAS integration approaches (section 2.4.4). Section 2.4.5 explains three GRID-MAS analogies that demonstrate the complementarity of these domains and motivate the integration proposed in chapter 5. Finally, section 2.5 summarizes the discussion about DSG by concluding both chapter 1 and 2.

2.2 Service-Oriented Computing

The SOC community proposes a set of models for computing systems based on the concept of service; these models define the Service-Oriented Architecture (SOA). In this section we first make an overview of SOA principles, these are not necessarily the same principles for DSGS as dynamic service generation reconsiders a number of SOA presuppositions. Then, we detail the main framework implementing this architecture: Web services. Finally, we look at the extensions and future directions of SOA, in particular semantics.

2.2.1 Service-oriented architecture principles

2.2.1.1 Founding principles

SOA defines a model for the execution of distributed software applications. This is historically due, for example, to the work of the Open Group with Distributed Computing Environment (DCE), the Object Management Group (OMG) with Common Object Request Broker Architecture (CORBA), Microsoft with (Distributed) Component Object Model (D/COM) or Sun with Java Remote Method Invocation (RMI). The initial thrusts of these distributed software components approaches[1] were (i) to standardize invocation mechanisms; (ii) to make transparent the location of these components on the network. Subsequently, due to the success of XML languages, the concept of service detached from a common middleware emerged, a notion based only on standardized and interoperable protocols and technologies over the Internet. A service is neither software nor a process but an abstraction that implements a functionality. Today, SOA main framework is Web services (section 2.2.2.1).

2.2.1.2 Singh and Huhns definition of SOC

For Singh and Huhns, the benefits of SOC are both that: (i) SOC enables new kinds of flexible business applications of open systems that simply would not be possible otherwise; (ii) SOC improves the productivity of programming and administering applications in open systems [SH05]:

> 'SOC provides the tools to model the information and relate the models, construct processes over the systems, assert and guarantee transactional properties, add in flexible decision-support, and relate the functioning of the component software systems to the organizations that they represent (...) In addition, it provides the ability for the interacting parties to choreograph their behaviours so that each may apply its local policies autonomously and yet achieve effective and coherent cross-enterprise processes (...) SOC enables the customization of new applications by providing a Web service interface that eliminates messaging problems and by providing a semantic basis to customize the functioning of the application (...) SOC enables dynamic selection of business partners based on quality-of-service criteria (such as performance, availability, reliability, and trustworthiness) that each party can customize for itself (...) SOC provides support for dynamic selection of partners as well as abstractions through which the state of a business transaction can be captured and flexibly manipulated; in this way, dynamic selection is exploited to yield application-level fault tolerance (...) SOC facilitates utility computing, especially where redundant services can be used to achieve fault tolerance (...) Services offer programming abstractions where different software modules can be developed through cleaner interfaces than before. SOC provides a semantically rich and flexible computational model, for which it is easier to produce software.'

---

[1]They are some time called 'distributed objects' approaches, as the basic idea was to enable method invocation in remote objects.

Interoperability

Wegner defines interoperability as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform [Weg96]. He discusses the two mechanisms enabling a 'client' and a 'server' interoperation: (i) *interface standardization*: map client and server interfaces to a common representation; (ii) *interface bridging*: two-way map between client and server.

> 'Interface standardization is more scalable because m client and n servers require only m+n maps to a standard interface, compared with m*n maps for interface bridging. However, interface bridging is more flexible, since it can be tailored to the requirements of particular clients and servers. Interface standardization makes explicit the common properties of interfaces, thereby reducing the mapping task, and it separates communication models of clients from those of servers. But predefined standard interfaces preclude supporting new language features not considered at the time of standardization (for example, transactions). Standardized interface systems are closed while interface bridging systems are open.'

From this reflection, we can see that both mechanisms have weaknesses. In particular, in order to have open DSGS, standardization seems not appropriate. However, the position taken in this thesis is that a consensus is needed between pure close standardization and non standardization at all. For example, standardization in Web services is a good thing to enable better services semantic description, etc. (section 2.2.3). However, standardization is not a good thing in agent representation and agent communication models as they go in opposition with agent autonomy and heterogeneity (section 2.3.3.2).

They identify the following elements for SOA that we can analyze at the light of the DSG perspective:

- ***Loose coupling***. *No tight transactional properties would generally apply among the components. In general, it would not be appropriate to specify the consistency of data across the information resources that are parts of the various components. Focus should be on high-level contractual relationships.*

- ***Implementation neutrality***. *The interface is what matters. We cannot depend on the details of the implementations of the interacting components.*

  SOAs are said to be loosely coupled, meaning that: they are implementation independent, they do not have technological constraints, the failure of a part of the system does not imply a total failure of the service, communication is done by asynchronous message passing, etc. The most the architecture is loosely coupled, the most service providers and users are autonomous and heterogeneous. Loose coupling is therefore a strong aspect of DSGS. In SOA, the implementation neutrality is encouraged by standardized interfaces and protocols.

- ***Flexible configurability***. *The system is configured late and flexibly. The configuration can change dynamically.*

  A SOA is said to be dynamic when it is not completely pre-configured during the development and spreading phases and some parts of the service is specified and (re)configured at run time (on the fly). Dynamic SOAs are based on conversations that allow service providers to dynamically change and adapt the service by communicating. Loose coupling is naturally more dynamic than strong coupling. Of course this aspect is one of the most important of DSG. To dynamically change and

adapt the service by communicating is the main characteristic of DSG. This is also expressed by D7, D13, C2[interaction], C3[non-determinism], C4[learning] . . . and to come C7[programming], C9[registry], C10[negotiation], C12[process], C15[conversation], C16[dialogue], C21[stateful].

- ***Long lifetime***. *The components must exist long enough to be able to detect any relevant exceptions, to take corrective action, and to respond to the corrective actions taken by others. Components must exist long enough to be discovered, to be relied upon, and to engender trust in their behaviour.*

  Multiple ephemeral life cycle (D8) does not mean DSG elements exist for a short time. Stability is of course necessary for DSG elements: reasoning (D10), evolution (D9), history (D5) are only compatible with a long lifetime. Furthermore, C21[stateful] specifies explicitly that DSGS agents should be able to manage their lifetime autonomously. C19[trust] concerns the trust aspect.

- ***Granularity***. *The participants in an SOA should be understood at a coarse granularity. Interactions and dependencies should occur at as high a level as possible.*

  High level interaction as been defined as a strong aspect of DSG. It is specified by C2[interaction], C11[message], C15[conversation] and C16[dialogue].

- ***Teams***. *Instead of framing computations centrally, it would be better to think in terms of how computations are realized by autonomous parties. In other words, instead of a participant commanding its partners, computation in open systems is more a matter of business partners working as a team.*

  The community notion was already presented as a key element of DSG. This is expressed by C18[community].

DEFINITION: SERVICE-ORIENTED ARCHITECTURE

A model for the execution of loosely coupled and dynamic service-based software applications. A service is seen as a standardized and interoperable interface of a specific function. The Web service framework is the current main implementation of service-oriented architecture.

Characteristic 8 (SOA)

DSGS should respect fundamental SOA principles such as loose coupling, implementation neutrality, standardization, message based communication, interoperability, etc. The more SOAs are loosely coupled and dynamic, the more they will fit with DSG requirements.

2.2.1.3 Service-oriented architecture main aspects

In SOA, a service exchange life-cycle is generally composed of three steps: *information*, *negotiation*, *execution* (figure 2.2). These steps are more or less important according to the dynamicity of the service execution. The information step consists in the service user and provider discovery and selection, it occurs via service registries. The negotiation step consists in the establishment of agreements, called contracts, between service provider and user. The execution step consists of instrumentation and realization of the service performance. The more SOA is dynamic, the more this life-cycle is repeated during a service performance (figure 2.3). In DSG, these three steps are not clearly separated. They occur all simultaneously. The next paragraphs detail some aspects of service exchange life-cycle and their relation with DSG. They are summed up in table 2.1.
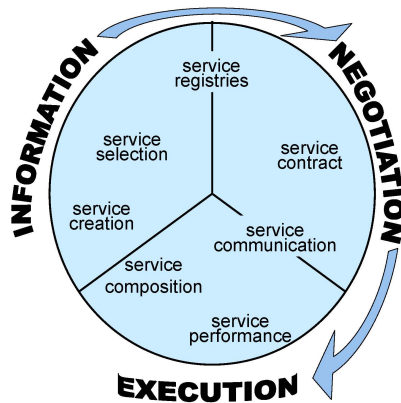
Figure 2.2: Service life-cycle

Service creation. There are three ways of creating a new service: (i) the *virtualization* of an application (i.e., the 'servicization') which consists in taking an already existing application and in transforming it into a service (e.g., interfacing a private application with the SOA standards); (ii) the *service dissemination* which consists in creating a new service that directly realizes the wanted functionality from scratch (e.g., writing a new SOA standards compliant service); (iii) the *service aggregation* which consists in creating a new service by aggregating, composing already existent services (e.g., writing a new business process). Considering D7, D9, D16, virtualization and dissemination may not be applied in DSG. Aggregation may apply, but dynamically. This is in particular discussed further in this section (C12[process]) and in section 2.4.5.1. In fact, considering DSG lead us to introduce a new way: *service generation*. The service is not pre-defined, but dynamically created, constructed, and specified during the service exchange.
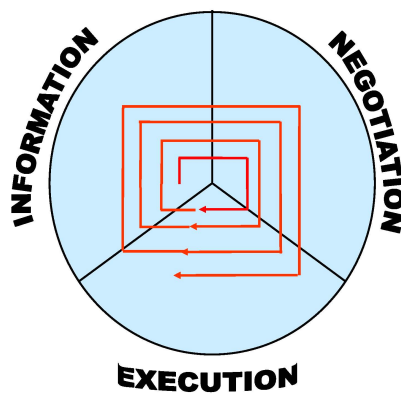
Figure 2.3: Dynamic SOA life-cycle

Service selection. In order to make a service widely used, it needs to be discovered by a user and a correspondence between the user's goals and the service functionalities has to be established. It is called the matchmaking problem. Semantically rich descriptions enable a more precise selection of services (section 2.2.3). The service selection may be: (i) *local* i.e., depends on reputation mechanisms, recommender techniques, referrals, trust or other social aspects directly connecting the service user and

provider; (ii) *global* i.e., occurs in a 'marketplace' (cf. asymmetric registries just below). At the end of this selection/negotiation phase the signature of the contract occurs. Service selection in DSGS is not easy because the matchmaking problem becomes very hard to solve. DSG is actually based on a trust delegation (D16 and C19[trust]).

Service registries.   Let us consider for example three cases in user/provider meeting:

1. No third party, no mediator, the service provider and the service user know each other; the service performance is normally executed;

2. Intervention of at least one third party: an asymmetric registry. The service provider publishes a service offer in a services registry and the service user finds this offer via the registry;

3. The same as (2), but both the service provider and the service user respectively publish an offer and a demand in a symmetric registry. The role of the registry is more important as it has to fit offers with demands.

These three cases show the different roles a registry should have. The purpose of a registry service[2] is to enable service providers and users to locate each other. Most of the time, registries are simple service directories in which service descriptions are available for potential users. They are called *asymmetric registries* [MBG03]. Classical functionalities of these registries are *yellow pages* (mechanism to find service providers by their characteristics and capabilities according to a standard taxonomy of domains), *white pages* (mechanism to found service providers by their identity (e.g., names)) and *green pages* (mechanism to find all the services provided for a given service provider). These basic registries are simple databases with a passive behaviour.

A more advanced registry might be more active by providing not only search services but also matchmaking, discovery or negotiation services. In these *symmetric registries*, offer and demand are respectively published by providers and users. Several offers for one demand or the opposite may exist. The role of the registry is therefore more active and consists in mapping these offers and demands together (matchmaking problem). This mapping could be realized by negotiation protocols e.g., auctions,[3] RFP (Request for Proposal), CNP (Contract Net Protocol), etc (section 2.3.3.3). The registry plays the role of *marketplace*,[4] and proposes a brokering or facilitating service added to simple search service. Criteria for the selection can be based on price systems or barter systems. Prices simplify a mechanism for service selection, because the many factors contributing to the value of a service are compressed into a single number. This kind of registry may be very useful for the services valuation (D15) as the registry may compare offers and demands.

Service selection and registries are one of the difficult aspect of DSG: a registry of potentially generated services is very hard to maintain because of D1, D2 and D4. However, some aspects such as reputation, evaluation occur also in DSG (D12 and D15). Notice however that some aspects of DSG, such as pro-activity (D18), can enhance service selection.

Characteristic 9 (registry)

> *DSGS need symmetric dynamic registries where service providers and service users publish respectively what they know, or what can be published from their offers and demands. These registries should play the role of marketplaces and facilitate negotiation in order to match offers and demands.*

---

[2]Sometime called a directory or index or list.

[3]An auction is a market in which prices are determined dynamically (first-price open-cry (English), first-price sealed-bid, second-price sealed-bid, Dutch, etc.).

[4]Market oriented programming is an approach to distributed computation based on the market-price mechanisms of buying and selling. It has the following features: (i) The state of the world is described completely by current prices, and communications are offers to exchange goods at various prices; (ii) Agents do not need to consider the preferences or abilities of others (iii) Under certain conditions, a simultaneous equilibrium of supply and demand across all of the goods is guaranteed to exist.

Service contract. Negotiation is a general mechanism for resolving conflicts and reaching agreements among autonomous entities. Negotiation is a process by which a joint decision is reached by two or more agents, each trying to reach an individual goal or objective (possibly contradictory with the other's). The agents first communicate their positions, which might conflict, and then try to move towards agreement by making concessions or searching for alternatives. In SOA service provider and user, after negotiation, come to a mutually acceptable agreement about the terms and conditions under which the desired service will be performed. These agreements are called *contracts* or *service level agreements* (SLAs). The contract describes the reciprocal engagements between the service user and provider. It formalizes the service exchange: a service performance results from the execution of a contract. The elements of the contract are said to be 'produced' and 'consumed'. These elements are [MBG03]:

1. The *service agents identification* contains at least the identification elements (e.g., a X509 certificate, a URI) of the service user, the service provider, but eventually identify elements of other agents such as third parties or mediators.

2. The contract uses a *service functional description* model which describes concrete service functionalities. It is called the service functional model and it contains the specification of the service objectives, actions realized, information needed, rules applied, etc. This model is of course not an implementation model in order to avoid that it is too strongly related to the service itself and that it reveals some information about the way the service is realized.

3. A service is often seen as a black box abstraction that interacts with users by message passing via an interface. The *service interface specification* therefore contains the message format description, and the message exchange protocols as well as the syntactic and eventually semantic description of the message elements. It also contains the definitions of bindings which formalize the relation between the service capabilities and the protocols and messages. Interaction with the service interface should be for example: simple message sending, request/answer messages (the main one), sequence (streaming) of messages, multiple request/answer messages.

4. The *service operational description* corresponds to the *quality of service* specifications. The quality of service is a set of operational properties that should be verified during the service performance. These properties are:

   - The service domain of application, with the limits of the performance, rights and obligations of agents involved in the service exchange, but also the conformity to standards, etc.

   - The service quality of performance such as dimensionality, effectiveness, accessibility, accuracy and precision of the service.

   - The service security conditions and level with authentication policies, authorization listings, privacy rules, integrity, etc.

   - The service robustness description with reliability, availability, continuity of the service, but also some technical properties such as transaction management,[5] etc.

   - The service management description which specifies who drives, and monitors, the service performance. What will be the follow-up, the warranty, etc.

   - The rules and adaptation to changes such as dysfunction, evolution, versioning, etc.

5. The *service contract life cycle* details the service performance lifetime as well as the eventual service performance frequency, the conditions of renewal, the conditions of service end over, etc.

---

[5]A *transaction* is a process (i.e., program in execution) that accesses and possibly modifies data. Transaction processing mechanisms ensure that all or none of the steps of the process are completed and that the system reaches a consistent state following the ACID properties: Atomicity, Consistency, Isolation and Durability.

6. The *service exchange description* specifies some properties about the service. For example if it is free or not. If not, which is the payment condition and whether it is an inclusive price or a performance unit price.

In DSG, some elements of the service contract are difficult or impossible to establish, such as service functional description and interface specification, for the same reasons that make service selection difficult.

Characteristic 10 (negotiation)

> *DSGS should allow agents to dynamically negotiate (and re-negotiate) the terms and conditions under which the desired service is dynamically generated. A service contract may exist, but some parts are fixed and some parts may evolve during the DSG.*

Service communication. In SOA, methods of the services are not directly invoked by users, but when messages are received, a service provider decides how to proceed by interpreting the user's message and invoking itself the good method(s). This is an important difference with software component based approaches where methods of components were directly called by external application. Message passing based communication represents a major class of interaction, which conceptually differ from invocation. It was originally suggested by Hewitt [Hew77], in the actor model. In SOA, message passing is generally done synchronously while the actor message passing and the agent's one are synchronous. Service communication occurs during both the negotiation and execution steps.

Communication is one of the DSG stake as conversation is the key of the arch of dynamically generated services. The main limit of service communication is synchronism. It limits agent autonomy and prevents multi-party conversations such as those that occur in DSG scenarios. C2[interaction], C15[conversation] and 16 are also related to communication.

Characteristic 11 (message)

> *DSGS agents should interact with an asynchronous direct message-passing mode of communication.*

Agent communication and negotiation is detailed in section 2.3.3. The analogy between service communication and agent communication is drawn in section 2.4.5.1.

Service performance. They are of three types [MBG03]:

1. *Stateless performance*. No state changes during the performance. The service provider executes some tasks that produce information results directly transferred to the user. Services are also said to be stateless if they delegate responsibility for the management of the state to another component (section 2.4.3.1). A stateless service can be restarted following a failure without concern for its history of prior interactions. The same occurrences of the service should be executed several times without other effects than resource allocation. When the service is re-executed, the information that it delivers may be either invariant or variable, however the underlying application should manage evolving states. Take for example a clock service or a weather forecast service.

2. *Internal stateful performance*. The service provider executes some state transitions on its set of data (e.g., Data Base Management System service). A stateful service has internal states that persist over multiple interactions. Stateful services should keep a state and evolve not simply with external messages (as simple objects) but also according to their own state. The same occurrences of the service produce successive state transitions. These internal state transitions are in principle reversible. Take for example a ticket reservation service.

Table 2.1: Elements of the service life-cycle

Step	Service ...	Today	Tomorrow
INFORMATION	creation	virtualization dissemination	aggregation generation
	selection	matchmaking between goals and functionalities	semantically enabled
	registries	asymetric	symetric marketplace
NEGOTIATION	contract	agents identification functional description interface specification operational description contract life cycle exchange description	based on agent negotiation
	communication	message passing based synchronous point-to-point	asynchronous multi-party
EXECUTION	performance	stateless internal stateful resource stateful	separation between stateless services and stateful resources
	composition	orchestration/workflow conversation/choreography	dynamic based on agent conversation

3. *Resource stateful performance*. The service provider executes some state transitions on an external resource (environment) or agent (including the service user) consistent with internal state transitions. These changes are called side effects assignments and are by nature irreversible. Take for example a ticket buying service.

The notion of state plays also a very important role in DSG. It is a pre-requisite at any form of conversation. This aspect will be detailed in section 2.4.3.1; this is expressed by C21[stateful] and C22[separation]. The real challenge of DSG is to preserve stateless service qualities and facilities without storing a static state at the service level, but rather change a dynamic state provided by another component (i.e., a stateful resource) with which the service interacts. That is the idea of WSRF presented in section 2.4.3.1.

Service composition - High level process of services.   Service interaction is often defined in terms of: business processes, composition, aggregation, orchestration, choreography, workflow, conversation, etc. This is sometime called Business Process Management (BPM). In this thesis, we prefer the term *high level process of services* to avoid confusion with the non-Informatics meaning of this expression. However, all the previously cited terms have to be further explained.

In SOA, a *business process* is a structure that defines logical and temporal relations between services. It is a service composition:[6] the composite service performance is the result of the combination of the composed service performances engaged in the process. One of the difficult facet of BPM is to maintain global coherence without explicit global control. In some aspect BPM is similar to the planning problem that has been investigated extensively in AI [AHT90]. However two assumptions are often

---

[6]Notice the composition is often used in SOC as aggregation: a business process is not a simple composition i.e., a series of service invocations in pipeline (where the output of a service is the input of another one). In the following, we will use these two terms in the same sense: to refer to any form of putting services together to achieve some desired functionality.

made in classical planning approaches: (i) the world is static except for the actions of the planner; (ii) the planner is omniscient. These assumptions are not valid in SOA. A business process design needs a coordination between composed services participating in the process. This coordination and cooperation of the tasks imperatively needs communication between composed services and may be static (i.e., done before the business process execution) or dynamic (i.e., realized at the same time as the business process is executed). The aggregation of services in business process is often described with the terms *orchestration/workflow* when it is static and with the terms *conversation/choreography* when it is dynamic [Pel03].

Orchestration defines interactions and their sequences (described in term of messages and their interpretation) between composed services participating to the business process; akin to a 'workflow' (as BPEL4WS in Web services). The description of this static composition of services is local and private to each participant to the process. A choreography may connect orchestrations with one another. Choreography specifies message exchanges as well as the sequence of these exchanges; akin to a 'conversation' (as WSCL or WSCI in Web services). It is a public description common to all participants (therefore the language of choreography must be standardized while the one of orchestration does not necessarily need to). Orchestration and choreography technologies are described in section 2.2.2.3.

DEFINITION: HIGH LEVEL PROCESS OF SERVICES

A (business) process that realizes the coordinated composition, or any form of putting together, of several services (composed) in a unique service (composite) performance. This composition may be static (orchestration/workflow) or dynamic (choreography/conversation).

The main weakness of high level process of services approaches is dynamicity. Most of the time, the business process or composite service is designed before its execution: it is pre-compiled and ready to be triggered. Even if the workflow engine can execute the workflow orchestration invocations asynchronously, the process is still centralized, which means that it suffers from the single point-of-failure weaknesses that characterize centralized systems. These are the important aspects that agent technologies may improve. In particular, section 2.4.5.1 details the analogy between BPM and agent conversation. It explains why this evolution from simple service exchange to high level processes of services could be done only by substituting the object oriented kernel of service-oriented applications by an agent oriented kernel able to have and manage conversations to realize dynamically generated services.

The realization of dynamic high level process of services is a challenge taken-up by DSG. But DSG includes also all other aspects described by chapter 1 and 2 characteristics. Actually, DSG addresses the question of high level process of services by decomposing these processes in a set of agent-agent interactions as it is suggested in choreography approaches. We further think that high level processes of services would be realized by enhancing the way agents communicate and represent each other, instead of enhancing the way of describing the processes. This is the choice made in this thesis. Chapter 3 suggests solutions for agent representation and communication rather than for business process modelling.

Characteristic 12 (process)

DSGS should enable dynamic high level processes of services by modelling agent communication. This may be true both for workflow/orchestration or conversation/choreography based approaches.

2.2.2 Service-oriented architecture standards and technologies

2.2.2.1 Web service

Nowadays, the main way of implementing a SOA is by means of Web Services (WS) (`www.w3.org/2002/ws`) [BHM$^+$04]. The identification of the concept is sometimes attributed to D. Winer who

SERVICE PROPERTIES

We may classify services with a set of 11 questions, according to a set of properties as represented in figure 2.4:

- Does the service communicate by asynchronous or synchronous message passing?

- Does the service manage an internal persistent state?

- Is the service accessible via a system oriented architecture or a service-oriented architecture? (if not, we should better talk about software components)

- Is the service compliant with SOA standards? Is it possible to publish it as a standard service?

- Does the service manage its lifetime? Is it transient or persistent?

- Is the service a collaborative one? Does the service can be used by several users?

- Is the service defined by a contract?

- Is the service execution a simple one shot interaction (e.g., request/answer) or a long lived conversation?

- Does the service have a semantic description? Is it able to deal with semantics and ontologies?

- Does the service usage involve only human or artificial users? Is it able to provide any type of user with the functionality? Is the service itself provided by a human or artificial service provider?

- Is the service a simple, or a composed or composite one?
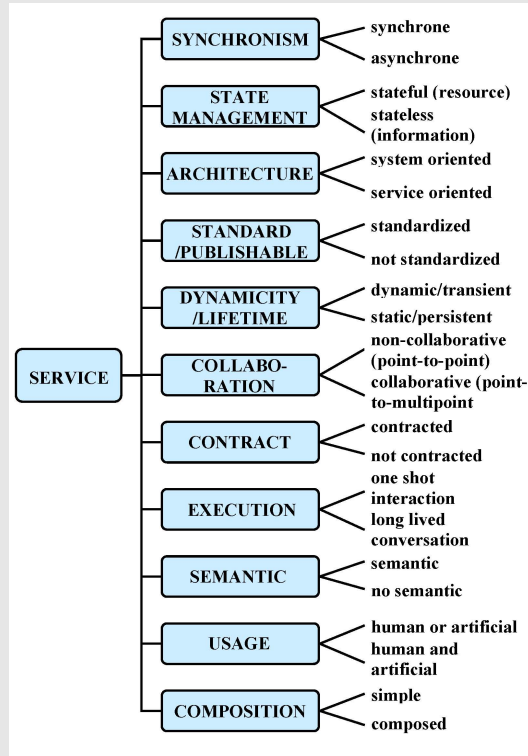
Figure 2.4: Service properties

proposed at the beginning of 1998 XML-RPC as a RPC mechanism based on XML (UserLand: `www.xmlrpc.com`). Web services inherit from work and experiences from distributed software component approaches (CORBA, RMI, etc.) [Vog03]. The World Wide Web Consortium (W3C) defines a Web service as:

> 'A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.'

Web services are describable, discoverable and message based software components that perform some function. Many Web service technologies exist (figure 2.6) but Web services are mainly based on the three XML based standard languages: (i) Web Service Description Language (WSDL) to describe software components i.e., functions that may be invoked; (ii) Simple Object Access Protocol (SOAP)[7] to describe methods for accessing these components i.e., message exchanges; (iii) Universal Description, Discovery and Integration (UDDI) to publish a service and to identify/discover a service provider in a service registry.[8] In Web services, providers publish their services on registries, and users find the service providers from registries and then invoke them such as illustrated in figure 2.5.[9]
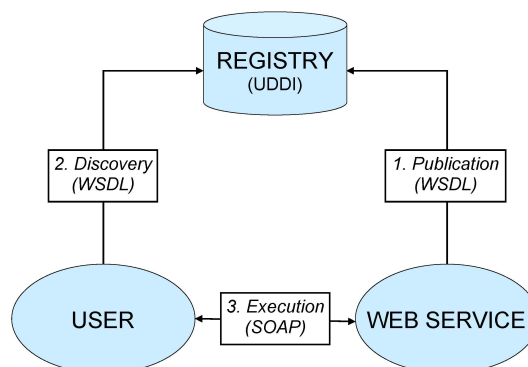
Figure 2.5: Web service life cycle

There are two main views of Web services: the *RPC-centric view* (Sun) that treats services as offering a set of methods to be invoked remotely and the *document-centric view* (Microsoft and Sun) that considers the documents as the main representations and purpose of the distributed computation. Each component reads, produces, stores, and transmits documents.

Web services denote both an already used and a promising approach under one simple important viewpoint: it has facilitated the integration of various heterogeneous components. The two main objectives of Web services are standardization and interoperability (intra-enterprise and inter-enterprise). Web services allow to access distributed functionalities on a network in a standardized way. With Web services, different applications can communicate (most of the time using HTTP and thus passing through

---

[7]SOAP is very bad named as it is not an object-oriented protocol of communication. In CORBA, for instance, a user invokes remotely a method by identifying the object with the Interoperable Object Reference (IOR). In SOA methods of the services are not directly invoked by users, but by the service itself interpreting the user's message and invoking itself the good method(s) or routine(s).

[8]Notice that according to specifications, only WSDL is really needed to describe a Web service, SOAP and UDDI are de-facto standards.

[9]The three steps (Information, Negotiation, Execution) of SOA architecture are realized with Web services in 2 and 3 on figure 2.5.

firewalls) with each other without knowing anything special about their implementation (operating system or programming language), but only dealing with a standardized interface where exchanges are expressed by XML documents. Standardization enables both services to be accessed by any kind of agent and interoperability between these services. It is clear that the main advantage of Web services is that we can compose them to create new services. So, from service invocations, which are single-shot two-party interactions, Web services started evolving into high level process of services that are typically long-lived multiparty interactions. From this interest, another set of languages, called Process Description Languages (PDL), emerged (section 2.2.2.3). For a recent overview of Web services and their future see for example [SvdAB$^+$03].

The two main Web service development platforms are J2EE (from Sun and JCP (Java Community Process)) and .Net (from Microsoft). They are able to generate WSDL and SOAP standard documents from platform dependent languages (Java, C#).

DEFINITION: WEB SERVICE

Web services are standardized and interoperable software components that perform some function and that use Internet protocols and technologies. They are describable, discoverable and executable according to SOA standards. Web services may be included in high level process of services in order to address specific collaboration problems.

2.2.2.2 Web services limits

Web services have some important drawbacks such as RPC like computing, object-oriented behaviour, client/server orientation, no user adaptation, no memory (stateless), no lifetime management, no conversation handling (simple request/answer interaction), synchronous communication. Web services are passive (non proactive) and they do not take into account the autonomy of components, neither they use a high level, history aware, communication language. In particular:

- SOAP is based on the HTTP protocol (one of the main Web protocols) which is volatile by nature: a connection is simply opened for a request/answer messages exchange. The communication is locking and synchronous. HTTP is stateless and thus unable to manage a long lived interaction such as a conversation.[10] Therefore SOAP is also a stateless protocol and each SOAP message is unrelated to any other message. One must use conversation identifiers at the application level to build a conversation.

- Web applications often adopt the client/server approach: the most centralized mode of distributed system. In client/server mode, most of the information resides on one side (server) and most of the intelligence on the other (client). The Web is still limited to embody high level, peer-to-peer mode architectures.

- A major drawback of Web services is semantics: WSDL allows definition of a service interface and service implementation but only at a syntactic level. There is no way to transfer transaction semantics across a SOAP call. This aspect is more precisely described in section 2.2.3.

- There are two major standards for directories: ebXML registries and UDDI registries (themselves described and interfaced as Web services). Unfortunately, neither support semantic descriptions/searching of/on functionality. Searches can only be based on keywords such as the service provider's name, location, or business category. ebXML registries have an advantage over UDDI

---

[10]Cookies used by webmasters to remember users settings are non sufficient for conversation based services! However, this is currently evolving with HTTP 1.1 which allows sequences of request/answer and HTTPR which starts dealing with state.

registries in that they allow SQL-based queries on keywords. Indeed, none of them provide high level discovery and selection mechanism. Therefore, they do not play the role of a marketplace.

There is many work addressing the question of registries, in particular in order to integrate semantics. For example, [MBB$^+$05] proposes to construct registries with KAoS description of services (including semantics, trust, information, usage policies, etc.). KAoS allows specifically to express authorization and obligation. [MPD$^+$03] presents an extension to the standard UDDI service directory approach that supports the storage of metadata (QoS information, semantic description, etc.) via a tunnelling technique that ties the metadata store to the original UDDI directory. As another example, the Internet Reasoning Service (IRS) [CLMM03, DCH$^+$04] is a service publication and discovery platform which integrate semantic description (Semantic Web services – section 2.2.3).

Even if DSGS should be Web oriented and compliant (C1[web]), we are aware that today's Web was originally proposed for document exchanges and has limits for SOA. Web services remain a 'vending machine' model [HNL02]. All these aspects and drawbacks imply that the Web service framework does not fulfil DSG requirements. Actually, for us, Web services are typical PDS: remote parameterized and standardized functions accessible by RPC. They differ from DSGS on all differences identified in section 1.3.3. For example, for D1 and D3, a user discovers a Web service in a UDDI registry and invokes it according to its WSDL description. One of this thesis objective is to show that enriching the Web service approach, with MAS and GRID, is the best solution to tend towards DSGS.

2.2.2.3   High level process of services technologies

As it is a quite recent interest, there is no high level process of services standard yet. Each consortium and/or company proposes its own framework (appendix A). Many papers address this question [SK03, SHP03, Pel03, RvSBS03, DCG05] (see also in section 2.4.5 where papers that address this topic with an agent approach are detailed.). We mention here three approaches. On the workflow/orchestration side, Business Process Execution Language for Web Service (BPEL4WS) (coming historically from IBM and BEA Web Services Flow Language (WSFL) and Microsoft XLang) seems to become the de facto standard. Structurally, a BPEL4WS process describes a workflow by stating who the participants are, what services they must implement in order to belong to the workflow, and what are the various orders in which the events must occur. That is, a BPEL4WS process describes the orchestration of a set of messages all of which are described by their WSDL definitions [BVV03]. BPEL4WS allows to express precedence constraints more complex than those expressed with Unified Modeling Language (UML). BPEL4WS uses WSDL to model both the process and the participating Web services. Therefore, a BPEL4WS business process is described as a new Web service that could be implicated in other business processes.

On the conversation/choreography side, Web Services Conversation Language (WSCL) specifies the sequencing of XML documents – as well as specifications for the documents themselves - being exchanged between a Web service and a user of that service. A WSCL conversation is a XML document. Web Service Choreography Interface (WSCI) provides a global, message-oriented view of the choreographed interactions among a collection of Web services. It describes the flow of messages exchanged by a Web service that is interacting with other services according to a choreographed pattern (protocol). A WSCI specification is part of a WSDL document.

2.2.2.4   Other technologies

We have detailed the main SOA standards and technologies. However, there exists many other ones related to different aspects of SOAs: security, transaction, quality of service, agreements, etc. Some of them may be more or less important for DSG, such as for example WS-Agreement [ACD$^+$05], recommended by the Global Grid Forum (GGF). It specifies a language and protocol for facilitating the

establishment of agreements and contracts between two parties in a service exchange. It is an implementation of the different elements of the service contract introduced in section 2.2.1.3. A WS-Agreement, is an XML document composed of: (i) `Name`, which identifies the agreement; (ii) `Context`, which contains the service agents identification and the service functional description; (iii) `Terms` (`service` and `guarantee`) which contain the service operational description. Languages such as WS-Agreement do not exploit the high potential brought by agents systems such as communication. For example, WS-Agreement has a poor set of message types and conversation models (simple offer/accept or offer/reject protocol) [PW05]. We will see in section 2.3.3 how agents may enhance the establishment of contracts by interaction and negotiation protocols and high level communication languages.

Figure 2.6 tries to sum up the current SOA technologies and standards. See appendix A for a description of acronyms and URLs. The concern of this thesis is the concept of service from a high level viewpoint. We are not so much interested in the plethora of SOA standards.

		Security Trust	Robustness Transaction	Management Infrastructure
Business Processes	BPEL4WS, WSCL, WSCI, WS-CDL, BPML …	WS-Security, SAML, WS-Trust, WS-Reliability	WS-Transcation, WS-Coordination, WS-AtomicTransaction, WS-BusinessActivity,	OMI, OGSA, WSRF
Discovery Publication	UDDI, WS-Inspection, ebXML Registries			
Description Semantic	WSDL, RDF, OWL, ebXML CPP, WS-Addressing			
Exchange Execution	SOAP, XML-RPC, REST, ebXML Messaging			
Base Language	XML, XML Schema, XML Namespaces			
Fondations Transport	TCP/IP, HTTP, SMTP, URI, MIME, SIP			

Figure 2.6: Service-Oriented Architecture technologies and standards

2.2.3 Semantics

The lack of semantics of Web services is one of the reason that motivate research on ontologies, Semantic Web and Semantic Web services.

Ontology. Ontology processing is a sub-domain of knowledge engineering which deals with knowledge representation and reasoning. An ontology is described [Gru93, GO94] as a 'formal specification of conceptualization'. However, this is a too large definition e.g., mathematics is also formal specification of conceptualization. More recently, the Web community proposed a new definition: 'an ontology defines the terms to describe and represent an area of knowledge' [Hel04]. Thus, we could say that ontologies deal with semantics: it is the set of terms for a particular domain, including the vocabulary, the semantic interconnections, some simple rules of inference and logic.

Ontologies are composed of *concepts*, *relations* and *instances*. For example, if you want to define a car, you should say: 'a car is a transportation object, with four wheels, and that you need a licence to drive it. MyCar is a car'. 'Car' is a concept, 'is a' is a relation, and 'MyCar' is an instance. Concepts are generally decomposed in several worlds: physical/real (natural concepts, physical objects),

mental/imaginary (mental abstraction of the physical world), roles (behavioural descriptions), abstract (without instances), concrete/occurrence (that have a real existence). Classical relations in ontologies are for example: subclass, superclass, part of, has part, sibling, equivalence. For a good (and quite complete) overview of ontologies see for example [SS04].

An important aspect of knowledge modelling and ontologies is reasoning. Reasoning is the real use of AI in knowledge engineering. In reasoning, we could distinguish two types of approaches: (i) the problem solving one, where problems are classified as well as Problem Solving Methods (PSM), methods to resolve these problems (PSMs are predetermined and triggered when the situation requires it) [FM01, Bre97, Mus98]; (ii) the (machine) learning one, where logic based methods, algorithms are executed on data/information the system has and by rules (abduction, deduction, etc.) to change it. Ontologies mainly applied the first approach; they are related to PSMs.

> LEVELS OF DESCRIPTION
>
> **Dictionary.** definitions associated to concepts;
>
> **Taxonomy.** specialization relationships (inheritance) between concepts;
>
> **Thesauri.** adding to taxonomies various lexical relationships (hyperonimy, synonymy, patronimy, etc.);
>
> **Ontology.** adding to thesauri other kinds of relationships (inheritance, aggregation, instantiation, ownership, causation, contains) and allowing reasoning.

Ontology creation and management[11] are difficult, especially if we consider that by definition, ontologies are made to be shared and therefore used and modified by several entities. Every entity should map its own resources onto the concepts of the shared ontology. Actually, collaborative ontology development is still a challenge in knowledge engineering [DLJC06]. Introducing time in ontologies is another challenge.

Ontologies are used each time we need a semantic description: knowledge systems, information retrieval (annotating/indexing documents), knowledge reasoning, etc. For example, in [DJDC], we formalize AGIL (chapter 5) by means of an ontology. It gives us a description, understandable and processable by agents, that defines the elements involved in service exchange scenario between agents on the Grid.

> DEFINITION: SEMANTICS
> Semantics is related to the meaning of things. Semantics is taken into consideration every time that a set of concepts, the relations between them and their rules need to be formalized. This formalization often takes the form of an ontology.

The Semantic Web. The recent main framework for ontologies is the Web. Today's Web is conceived to be used by people (HAs) who can interpret the content of the information because they have the necessary background knowledge, which they share with the creators of the given pages. Programs or AAs can only process this information in ad hoc ways. As Tim Berners-Lee (one of the Web pioneers) et al. state [BLHL01]:

> 'Most of the Web content today is designed for humans to read, not for computer programs to manipulate meaningfully (...) The Semantic Web is not a separate Web but an extension

---

[11]Tools for ontology development are very important, they support the information management by graphical interfaces that hide awful syntax and allow the user/designer to concentrate on the ontology itself and not on the representation formalism. Some ontology development tools are: Protégé, Triple-20, OILed, Webonto.

of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.'

From a simple point of view, the Semantic Web consists in adding all objects of the Web labels that represent the object. These labels are indexed in ontologies. They provide greater expressiveness when modelling domain knowledge and can be used to communicate this knowledge among agents of any type. In addition, the Semantic Web provides the necessary techniques for reasoning about ontology concepts, as well as resolving and mapping between ontologies. In order to allow semantic based services and information management, the Web needs protocols and standards that enable specification, access, and maintenance of the meaning of terms and objects belonging to Web pages (labels).

The current components of the Semantic Web framework are principally Resource Description Framework (RDF), RDF Schema (RDF-S) and the Web Ontology Language (OWL) [McG04] (previously DAML+OIL). RDF is built on top of the Web notion of a URI. URIs are employed in RDF to remove the ambiguity about terms and to enable multiple perspectives to coexist. OWL specifies classes and properties in a form of Description Logic (DL) with the terms in its expressions related using Boolean operators analogous to `and`, `not`, and `or`, as well as the constraints on various properties.

Semantic Web service. Web service discovery, invocation, composition, interoperation is limited to be human interpretable by their lack of semantics. Existing SOA standards only provide descriptions at the syntactic level. Since no explicit semantic information is normally defined, automated comprehension of the service description is limited to cases where the provider and user assume shared knowledge about operations. *Semantic Web services* (SWS) are instead semantically described Web services. They use ontologies to constitute the knowledge-level model of the information describing and supporting the use of the services. These ontologies enable automated understanding of their functionalities. More generally, this semantic layer helps Web service discovery, invocation, composition, or interoperation. For example, semantically rich descriptions of service facilitates the matchmaking problem. Web service discovery should be based on the semantic match between a description of a service demand, and a description of a service offer in a semantically competent registry.

For a nice overview of Semantic Web services see for example [LLA+03, CDM+04]. The second proposes to characterize Semantic Web services by three dimensions: usage activities, architecture and service ontology. *Usage activities* define the functional requirements, the *architecture* defines the components needed for accomplishing these activities and finally, the *service ontology* establishes the link with domain knowledge.

The two main current approaches for Semantic Web services development are proposed by the OWL-S (`www.daml.org/services/owl-s`) and WSMO (`www.wsmo.org`) working groups:

- OWL-S (OWL-based Web Service Ontology) [SMZ01, MPM+04] is an agent-oriented approach to Semantic Web services, coming from DARPA with DAML-S and providing fundamentally an ontology for describing Web service functionalities. The objective for the development of OWL-S are to enable reasoning about Web services, planning compositions of Web services, and automating the use of services by software agents. OWL-S provides an OWL ontology for describing Web services. An OWL-S description for a service consists of three components: service profile (what it does), service model (how it works) and service grounding (how it can be accessed).

- WSMO (Web Service Modeling Ontology) [RKL+05] aims at providing an appropriate conceptual model for developing and describing services and their composition, based on the principles of maximal decoupling and scalable mediation. WSMO is inspired from the Web Service Modeling Framework [FB02] originally motivated by knowledge modelling community and its concerns with problem solving methods and ontologies. WSMO four basic elements are ontologies, goals, services and mediators. An implementation of the WSMO ontology is done within the Internet Reasoning Service [CLMM03, DCH+04].

Semantic basis for dialogue. One of the most important role of ontologies consist in considering them as a reference for mutual understanding. A shared representation is essential to successful communication and coordination. The ontology plays the role of the 'common sense' in human discourse. For example, in MAS, ontologies are used to realize the semantic level of agent communication. Agent communication languages may use ontologies to support the interpretation of the content expression by the receiving agent. This is more precisely described in section 2.3.3.

Characteristic 13 (semantics)

DSGS should deal with semantics at the data, information and knowledge levels, through the use of ontologies. The elements of DSG messages (i.e., descriptions, requirements, constraints, etc.) should be described at a semantic level.

The pragmatic level The formal study of language has three aspects. Syntax deals with how the symbols are structured, semantics with what they denote, and pragmatics with how they are interpreted and used. Meaning is a combination of semantics and pragmatics.

After the syntactic and semantic level, communicating agents have to deal with pragmatics. A pragmatic interpretation of a message means that the sense given to the message depends not only on the message itself but also of the context of interpretation of this message e.g., the message place in the conversation, the fact that the message has been already sent before, the social convention, etc. For example, pragmatic makes you not answer by yes to the question 'Do you know what time it is?', but instead give the questioner the time. This aspect is for the moment totally absent for agent communication approaches.

Characteristic 14 (pragmatics)

DSGS should deal with pragmatics to ensure a context aware interpretation of messages occurring during the service conversational process.

2.3 Multi-Agents Systems

2.3.1 Concepts and definitions

Historical background. Historically, MAS emerge from distributed problem solving (mainly in US research). Afterwards, it became more and more the result of an integration of several domains of Informatics and AI (machine learning and reasoning, distributed systems, knowledge engineering, planning, etc.) (mainly in EU research). Agents and MAS were extensively studied in literature [HS98, Fer99, Jen01, BD01, Woo02]. The agent paradigm was strongly influenced by the object one, but differs from it according to three major aspects: *autonomy* i.e., the ability to act without the intervention of another entity; *intelligence* i.e., the ability to change its own state alone, without message passing; *interaction* i.e., the faculty to communicate directly and asynchronously with their environment and other agents by means of direct or indirect message passing with a communication language independent from the content of the communication and from the internals of agents. An artificial agent is generally defined as [Fer99, Jen01]:

> DEFINITION: ARTIFICIAL AGENT
> A clearly identifiable physical or virtual autonomous entity which: (i) is situated in a particular environment of which it has only a partial representation; (ii) is capable of perceiving (with sensors) and acting (with effectors) in that environment; (iii) is designed to fulfil a specific role; (iv) communicates directly with other agents; (v) possesses its own state (and controls it) and skills; (vi) offers services (in the sense of particular problem solving capabilities); (vii) may be able to reproduce itself; (viii) has a behaviour that tends to satisfy its objectives.

Agent types. Agents are reactive (able to respond in a timely fashion to changes that occur in their environment) and proactive (able to opportunistically adopt goals and take the initiative), capable and efficient (able to solve problems and reach private objectives) and adaptive (able to learn and change according to experiences). Agents have a persistent state. They are also able to use and reconcile ontologies. All these properties are very interesting for providing services as section 2.3.4 details. Traditionally, one may distinguish two main types of agents:

- *Reactive agent.* They are characterized by simple behaviours which react to stimuli coming from the environment. Their actions directly depend on perceptions. They do not have an explicit representation of the environment and they generally communicate indirectly via it. They have no large reasoning abilities. This approach is often based on the emergence of collective behaviour from several simple individual behaviours. Historically, [Bro91] shows the limits of symbolic reasoning and explains how a global behaviour should emerge from several simple behaviours. It is called subsumption architectures. For example, [Dro93] proposes the Modeling an ANThill Activity (MANTA) architecture. Reactive agents are mainly used in simulation with MAS.

- *Cognitive agent (or intelligent agent).* They have generally a more explicit (but always limited[12]) representation of their environment, some precise skills and knowledge. They answer to messages directly sent by other agents autonomously in order to reach their objectives. They are able of learning and reasoning with traditional AI methods. Their actions depend on a deliberation (action plan, motivation, objective, etc.). Pioneering cognitive agent architectures were Agent-0 [Sho93] or Belief-Desire-Intention (BDI) [RG91].

We will not give more detail in this manuscript about reactive agent architectures as we are mostly interested in the cognitive abilities of agents for service providing and using. Notice however, that an agent is seldom fully cognitive or fully reactive. There is a continuum of agent types between them called *hybrid agents*. Some of them may be distinguished, for example:

- *Software agent.* 'Agentification' of a software components (with state and functions) in order to enhance their cooperation;

- *Mobile agent.* Agent with the ability to move from an executing machine to another following data or information to process. It is sometime more interesting to migrate the program instead of the data this program has to process;

- *Assistant agent.* Agents with a specific task consisting in enhancing a human-computer interface;

- *Robot agent.* Using an agent for representing the internal architecture of a robot (physical or simulated). For example RoboCup.

Multi-agents system. The term 'societies of agents' highlights the importance of the social aspect in MAS. A MAS is not a simple set of agents put together in a common environment, but a real organization with social rules and interactions allowing cooperation and collaboration to resolve problems that centralized systems (as intelligent as they can) would not have resolved alone. These societies of agents emerge through agent communication. A MAS is often characterized as:

- Each agent has a limited amount of information or a limited set of capabilities;

- There is no global control of the MAS;

---

[12]The limited perception of an agent environment is fundamental, it guarantees the real distribution of a system, indeed, a system that could be observed, mastered, in a global manner by one single agent would not be distributed anymore.

- Data are decentralized;

- Computation is asynchronous.

MAS meets some important problems largely studied in the literature: How to define, decompose allocate problems and synthesize the results? How to make agents able to communicate and interact with each other? How to maintain coherence in different agent behaviours? How to allow individual agents to represent actions of other agents? How to manage the sharing of limited resource? In this thesis our interest focus both at the agent level and the MAS level. The STROBE model is an agent representation (agent level) and communication (MAS level) model.

2.3.2   Agent architectures

There is a large number of agent models and architectures in MAS literature. They mainly distinguish by type of knowledge representation and type of control. Defining an agent architecture means to define the different components that compose an agent, the intrinsic capability that an agent has and the interaction between these components and capabilities to specify their global organization. This description may be done at different levels: formal, conceptual, implementation, etc. Most of the time an agent architecture is decomposed in modules (e.g., interaction module, organization module, service execution module, etc.). The agent architecture strongly depends on the type of agent. [Boi01] proposes a good overview of agent architectures according to agent types (reactive, hybrid and cognitive). We may cite for example:

- In the Vowels approach [Dem95], an agent is defined following four modules: Agent (A), Environment (E), Interaction (I), Organization (O). The A module contains the internal reasoning capabilities; the E module contains capabilities for perceiving and acting in the environment; the I module contains the interacting and communicating with other agents capabilities; the O module contains capabilities and representations for structuring the agents together. An agent with the A and E modules would be called autonomous agents, with the A, E, I modules, communicative agent and with four modules, social agent.

- The main approach in agent architecture is the one of mental states architectures. The most important one is the Belief-Desire-Intention (BDI) architecture [RG91]. A BDI architecture addresses how *beliefs* (i.e., the information an agent has about its surroundings), *desires* (i.e., the things that an agent would like to see achieved) and *intentions* (i.e., things that an agent is committed to do) are represented, updated and processed. Historically, the BDI architecture comes from architectures such as Procedural Reasoning System (PRS) [GI89] or Agent Oriented Programming (AOP) [Sho93] (and the corresponding language Agent0). One of the first agent architecture with an interaction dedicated module is COZY [Had95]. We may cite also ARCHON [Wit92].

- The Advanced Decision Environment for Process Tasks (ADEPT) architecture [JFN$^+$00] proposes the ADEPT system, as an agent-based BPM system. An ADEPT agent is composed of different modules such as the Interaction Management Module, responsible for handling agent-agent negotiation an establish Service Level Agreement (SLA); the situation assessment module, responsible for evaluating and monitoring the agent's ability to meet the negotiated SLA and handling exceptions occurring during service execution; the service execution module, responsible for executing service; the communication module, responsible for communicating with other agent using shared communication language, protocol etc. All these modules utilize persistent information about the agent itself or acquaintances. The ADEPT system is an important related work as it is one of the first systems or agent architectures that really assesses the abilities of agents for providing service and for being involved in business processes. One of the new aspect is to have a set of modules dedicated to service exchange. Therefore, the ADEPT system is an important step in MAS-SOC integration even if it was not conceived within SOA standards and the Web service framework.

Other work about agents and Web services are presented in section 2.3.4.2 and about agents and high level processes of services in section 2.4.5.1.

- The SMART and *act*SMART approach [ALd05] proposes a descriptive specification of an agent architecture as well as an agent construction model, as a means to move from specification to implementation. SMART firstly considers several types of entities, rigorously specified with the Z language, such as `Object`, `Agent`, `AutonomousAgent`, `NeutralObject` and `Server-Agent`. These entities are defined in terms of attributes, actions and motivations and not in terms of how they are built or how they behave. It allows the model to be specific enough for constructing agents but general enough to support other architectures. *Act*SMART (Agent Construction Toolkit for SMART) provides a component-based description of agents. The components are of four types: information collection (sensors); information storage (infostores); decision-making (controllers) and directly effecting change in the environment (actuators). *Act*SMART specifies also the interaction and the management of these components by a simple communication language and an agent shell. Therefore, actSMART allows defining types of agents by specifying the set of components that defined them and the global execution sequence.

- Other architectures are for example, Touring machines [Fer92] or Cooperative Information Agents (CIA) [VDB97] also uses a distinct communication manager, contract manager and a service execution manager.

All these architectures are of course very interesting for MAS. However, most of them concentrate on what an agent knows, does, is able to do, wants, instead of concentrate on how an agent can learn, exchange and understand by communication with other ones. These architectures are all more or less agent centred and they neglect interaction ([Sin98] explained why mentalist architectures are limited compared to social ones). In particular, none of these architectures propose a complete[13] and completely dedicated conversation context allowing an agent to develop independently a new language for a given interlocutor or group of interlocutors. This is the reason why we propose the STROBE model. Actually, the STROBE model is presented in chapter 3 more as an agent model instead of an architecture. It is not a specification of a complete architecture, but a specification of some modules, in particular the interaction or communication module and the service execution module which, in the STROBE model, are viewed as the same thing, called a *Cognitive Environment* (CE). Integrating interaction and service execution in the same (dedicated) module is one of the innovative aspects of the STROBE model in order to implement DSG. In STROBE, there is no contract manager or negotiation module as negotiation is part of the DSG process. Everything is done at the CE level. Another strong aspect which differs with other architectures is the dedicated aspect of CE. Indeed, we should rather say that the interaction module and the service execution module are equivalent in STROBE to a set of CEs; each CE being dedicated to an interlocutor or a group of interlocutors. The agent CEs play the role of conversation contexts and allow agents to develop a different language for each interlocutor or group of interlocutors. In the rest of the thesis, everything other modules forming an agent, different from CEs will be called *brain* (section 3.3.3.2).

2.3.3 Agent communication

Simply grouping together several agents is not enough to form a MAS. It is communication[14] between these agents that makes it. Communication allows cooperation and coordination between agents [Fer99,

---

[13]We will see in chapter 3, the meaning of 'complete' in terms of levels of abstraction: data, control and interpreter.

[14]The 'official' difference between communication and interaction is the fact that interaction should have an effect on the interlocutors, while a communication is simply a transmission without effect. But in the MAS community, even a communication is supposed to have an effect on its interlocutors. We totally adhere to this idea considering that communication could not be reduced as a simple transmission in the agent paradigm. Object communication could be considered as transmission, since an object answers to a message by applying a method, without changing is own internal structure (e.g., changing the way of

Woo02]. Communication in MAS is classified in two modes, direct and indirect:

- In the *indirect mode*, agents communicate via the environment (the world where they live and evolve) e.g., the ant pheromone. This kind of communication is adopted by reactive agents and limit the potential of both coordination and cooperation;

- In the *direct mode*, communication is achieved via direct message passing between agents [Hew77]. Communication in direct mode can be *synchronous*, that means that a rendezvous between communicative agents is needed, or *asynchronous*, that means that a message can be buffered and that an agent can send a message to another one even if the second does not want to receive it or know about it. Even in synchronous mode, agents are never blocked waiting for messages.

Communication we are interested in is asynchronous direct mode communication. That means, a communication throws directly messages (that can be buffered) between agents (C11[message]).

Modelling communication has always been a problem. The traditional formal model for communication is the Shannon statistical communication theory. However, this popular model is very limited because it is basically a statistical model of information transmission. In real communicative situations such as the ones that may occur in service exchange scenarios, it is impossible to consider communication as a simple information transmission, the complete conversation or dialogue should be taken into account.

To model conversations, we have to consider important semantic

SHANNON THEORY

It considers four entities: a *sender* sends a *message* on a communication *channel* to a *receiver*. This message is written in a communication language. This simple model is the foundation of all the communication formalisms. The transmission is considered as completely reliable if:

- Messages are emitted once or not at all;

- Messages and emission/reception states are persistent and durable.

To assure this reliability, a communication mechanism should provide:

- A message repetition mechanism (coupled with management mechanism for replicas) in case of channel failures;

- A transaction management mechanism in case of failures of the sender or receiver.

(C13[semantics]) and pragmatic (C14[pragmatics]) aspects as the communication effects on the interlocutors, as well as the history of the conversations. In fact, let us assume that the first step in modelling conversations is to define and use agent communication languages.

2.3.3.1 Agent communication languages

In communicating, some questions about the meaning of things have to be taken into consideration: how the sender wants to mean something? How receivers understand the meaning of the message? These questions were firstly studied in language philosophy by Searle [Sea69] and Austin [Aus62] who explained that communication was realized by speech acts. Speech act theory considers three aspects of a message: (i) the *locutionary* act, or how it is phrased; (ii) the *illocutionary* act, or how it is meant by the sender or understood by the receiver; (iii) the *perlocutionary*, or how it influences the recipient.

---

answering messages). But agent communication could not be reduced to transmissions since the effect on the interlocutor is intrinsic to the agents. So we will now use communication and interaction with the same meaning.

Speech acts, that can be viewed as the linguistic unit of communication; they have a goal, they respect some expression rules and they reflect a certain mental or psychological state of the speaker.

Speech act theory was reused many times in agent communication. Speech acts were firstly classed by performatives by Searle and Vanderveken. A usually admit classification consists of seven categories: assertive, directive, commissive, permissive, prohibitive, declarative, expressive. Then, the work of Cohen and Perrault [CP79] on planning with speech acts, Allen and Perrault [AP80] and Cohen and Levesque [CL90] on mental states and intention, prefigured the work on architectures based on mental states representation, such as BDI, as well as of communication languages such as the Knowledge Query and Manipulation Language (KQML) [LF97] and the Foundation for Intelligent Physical Agents - Agent Communication Language (FIPA-ACL)[15] [Fip02a, Fip02b]. The former language was often criticized about the semantics of the messages [CL95, EBHM00]. The latter, its descendant, proposes therefore a strong semantics with pre and post conditions defined as agent mental states.

One important point is to distinguish in ACLs the message content and its pragmatics (i.e., metadata on the message content). Actually, the ACL message structure is composed of three levels [LF97, KP01, Eij02]: (i) *content level* which is often describe with a content language (i.e., KIF, PROLOG, Scheme, FIPA-SL), (ii) *message level* which is defined by the performative, the ontology, the language, etc. used in the message, (iii) *communication level* which allows to identify the metadata on the transmission of the message itself: the sender, the receiver, the protocol used, the conversation identifier, etc. Table 2.2 presents the parameters of a FIPA-ACL messages at each of these three levels [Fip02a]. One important parameter is `:performative`. It represents the communicative act that the agent realize e.g., `agree`, `confirm`, `propose`, `request`, etc. The number of different performatives varies between different agent communication approaches: from 2 such as in Agent-0 [Sho93] to 36 in KQML[16] and 22 in FIPA-ACL [Fip02b]. In the STROBE model, we use 6 performatives (section 3.3.4.2). We illustrate by a teacher-learner scenario in section 3.4.1.1 how agents can dynamically learn new performatives.

Table 2.2: FIPA-ACL message parameters at each level

COMMUNICATION LEVEL	MESSAGE LEVEL	CONTENT LEVEL
`:sender`	`:performative`	`:content`
`:receiver`	`:language`	
`:reply-to`	`:encoding`	
`:protocol`	`:ontology`	
`:conversation-id`		
`:reply-with`		
`:in-reply-to`		
`:reply-by`		

ACLs are interesting because they replace ad-hoc communication languages that previously were used in MAS. Using a strong ACL with a consistent semantics [Gue02, EBHM00] precisely known in advance, is a great advantage when creating heterogeneous agent systems that are designed to be easily extensible and broadly used. Agents may use all the message structure parameter to help them interpreting messages. Communication level parameters help them to deal with the conversation as it is described hereafter. Message level parameters help agents to adopt and agree on a shared semantically described vocabulary. All these parameters help agents to autonomously choose the message they want to interpret or refuse. For an introduction to semantics of agent communication see for example [Eij02]. The author makes a precise state of the art about concurrent programming from shared variable to agent communica-

---

[15]This standard is heavily influenced by Arcol developed by France Télécom in the early 90s. The Foundation for Intelligent Physical Agents (FIPA) is a nonprofit association concerned with specifying standards for heterogeneous, interoperating MAS.

[16]41, in a the first KQML version (1993).

tion. Besides, he looks at the three ACL message levels seen above (communication, message, content) and detail their semantic aspects; he also adds a top level, the MAS level, for which he also details the semantics. For a good recent overview of agent communication model and ACLs see for example [KP01].

Communication is strongly related to agent autonomy. In order to enhance agent autonomy, it is interesting to develop communication models which expect agents to communicate without being knowledgeable of the internal beliefs of their interlocutors, with whom they only handle output messages as an interface. Mental states based ACLs are often criticized for some of their requirements. One of them is the *sincerity condition* which assume that an agent 'thinks' (in term of BDI mental states) what its says. The strong semantic of FIPA-ACL give to this language this drawback. For example, the precondition of a FIPA-ACL `Inform` message states that the sender believes what it tells and that it knows that the receiver wants to know that the sender believes it. The postcondition is that the receiver can conclude that the sender believes the message content. This condition is a limit to agent autonomy. Consequently, you cannot use FIPA-ACL in settings where sincerity cannot be taken for granted such as, for instance, in negotiations of any kind. Using a strong interlocutor model and not simply a mental states architecture avoids these drawbacks. It is the choice made in this thesis. Singh [Sin98] demonstrates the insufficiency of classic approaches centred on a mental states architecture rather than on a social architecture:

> '(existing work on ACLs) appears to be repeating the past mistake of emphasizing *mental agency* – the supposition that agents should be understood primarily in terms of mental concepts, such as beliefs and intentions. It is impossible to make such a semantics work for agents that must be autonomous and heterogeneous: This approach supposes, in essence, that agents can read each other's minds.'

Singh proposes a *social agency* lying on an organization founded on group, role and commitment for each agent playing a role. He defines the commitments of a role as restrictions on how agents playing that role must act and communicate. Singh demonstrates the importance of the social aspect in MAS. Today, this aspect is specially addressed by the work on MAS organizational structure such as for example [WJK00, FGM03]. This is also assumed in this thesis. In section 2.4.5 we will show the analogy between MAS and GRID organizational structure, and in chapter 5 we will maintain this aspect in our GRID-MAS integration model.

Characteristic 15 (conversation)

> *DSGS need strong open and dynamic communication models that are able to manage conversational processes. They should not reduce agent autonomy and heterogeneity and they should allow an interpretation of messages that is as dynamical as possible.*

2.3.3.2 Conversation modelling

In agent communication, the hardest challenge lies in conversation modelling. Conversation models can be classified according to two approaches.

Interaction protocols. In the first approach, the most specified and widespread, agent conversations are modelled by *interaction protocols* or *conversation policies* [DG00, Hug03]. These protocols represent the interaction structure and specify rules that must be respected during the conversation (i.e., specification of a pattern of message exchange in a conversation). By using protocols, an agent interprets messages from a conversation one-by-one, changing at each step its own state, and following the protocol to produce the next message in the conversation. An interaction protocol has four properties [WF86, Fer99]: (i) a conversation begins with a strong performative that expresses an agent intention, (ii) for each step of the conversation there is a finite set of possible actions to perform, (iii) there are some

final states which finish the conversation, (iv) when a speech act is performed, the conversation state is changed as well as the mental states of conversational agents. The main advantage of this approach is the semantic description of the conversation (via the logical expression of pre-conditions, post-conditions and mental states). The notion of interaction protocol is strongly bound to an architecture based on mental states, such as envisioned in KQML and FIPA-ACL [Gue02].

A famous example of interaction protocol is the Contract Net protocol proposed by Smith [DS83]. However we should also mention Berthet's Introduction protocol [BDB92], Sian's Cooperative Learning protocol [Sia91] and Winograd and Flores's request for action protocol [WF86]. Among the first languages using interaction protocols we may refer COOL [BF95] and Interaction Language (IL) [Dem95]. Interaction protocols are either described by the language itself, such as in COOL, or are often represented with Finite State Machine (FSM) or with Petri Nets or Coloured Petri Nets (CPN)[17] [CCF$^+$00] or Statecharts [DDCP05]. Interaction protocols were more recently described with techniques coming from the modelling community and its adaptation to agents e.g., Agent Unified Modelling Language (AUML) [OPB00] or Communication Protocol Description Language (CPDL) [Hug01]. Figure 2.7 illustrates the request for action protocol represented with FSM (figure quoted from [MCd02]). Figure 2.8 illustrates the Contract Net interaction protocol specified by FIPA and represented in AUML [Fip02c].
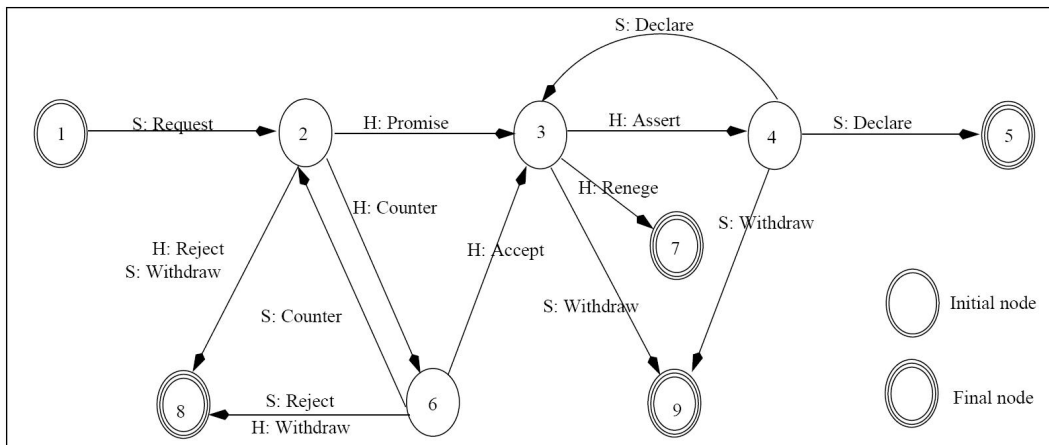
Figure 2.7: The request for action protocol

Nowadays, new approaches inspired by interaction protocols appear. For instance, *conversation plans* [BL00] or *conversation policies* [GHB00, DG00]. Conversation policies decrease the ambiguity between possible actions (and answers) in a conversation in order to facilitate the inference process about the mental states of the interlocutor. It adds constraints on the set of states and messages (syntactic constraints) rather on the set of actions and answers (semantic constraints) that an agent can perform.

Conversation policies have the advantage of abstracting from the language or representation model (e.g., CPN, AUML). They are public and shared, increasing modularity and usefulness for heterogeneous agents. Besides, some researchers still improve these models. For example, [EH00] proposes the concept of task model which takes account of the global coherence in conversation. [Hug01, HK03] propose an interaction protocol engineering: the generation of protocols after the interactions analysis. Extraction of interaction protocols may be very important in DSG, for conversation analysis and thus reasoning and learning about future conversations.

---

[17]CPN are very interesting to model multiple parallel conversations.
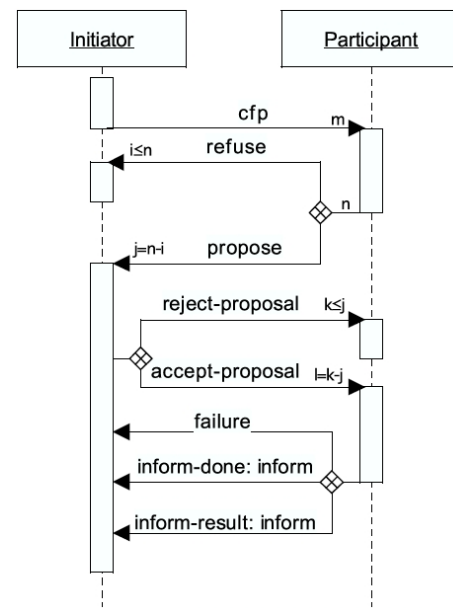
Figure 2.8: The FIPA Contract Net interaction protocol

DEFINITION: INTERACTION PROTOCOL
Interaction structure that (semantically) specifies rules that must be respected, messages that must be exchanged, and the states that agents must take, during a conversation. Interaction protocols model pre-fixed conversations.

The interaction protocol approach seems to be the main one in the MAS community, agents have to be able to deal with if they want to interact with any other heterogeneous agents (we detail the STROBE model compliance with interaction protocols in section 3.3.5). However this approach has weaknesses, especially interoperability, composition and verification of protocols which depend on a common semantics. Agents are forced to follow a policy restricting their autonomy and the dynamic interpretation of messages. Everything is specified in order to have only one way to interpret messages. The only way for an agent to consider the entire conversation is to look at the protocol, which was previously determined (before the conversation) and which cannot change dynamically. Therefore, agents are obliged to fit fixed conversations while it should be conversations which fit dynamically changing agents. As [GHB00] states:

> 'The received ACL messages and the inferred goals of the sending agent are at the root of the agent interoperability problem (. . .) Conversation policies limit the possible ACL productions that an agent can employ in response to another agent, and they limit the possible goals that an agent might have when using a particular ACL expression.'

However [RPD99] states:

> 'It may seem strange that agents which are said to be cognitive, and by definition autonomous and free in their actions, are not able to interpret messages using their own beliefs and knowledge, but have to use instead a quite reactive interaction model. The concept of interaction protocols is unsatisfactory for the same reason: interaction protocols should be an emergent property of open interactions between cognitive agents, but not a restriction on the communication capabilities of agents.'

We further think that heterogeneous agents easily communicate not by addition of constraints on conversations, but rather with an open and dynamic communication model which can fit all forms of agents and interactions. The second approach, reviewed hereafter, seems to tend towards that.

Dialogue. The inconvenient of mental states approach and interaction protocols is intrinsic to the limited vision of the underlying speech act theory which is only concerned with the interpretation of one isolated speech act. Indeed, agent dialogue modelling can not be limited to one message interpretation. A dialogue is not a set of independent messages. The second conversation modelling approach is much close to human communication where interactions are built progressively and dynamically along the conversation. Instead of modelling mental states and their dynamics in conversation, this approach deals directly with speech acts rather than with hidden intentions. Some researchers call it dialogism [BP99, RPD99], dialogic approach, or simply *dialogue* [MDCd99, Mau02, MCd02]. In dialogues, the next answer message of a conversation cannot be foreseen, and should be determined only after the interpretation of the previous incoming message. Conversational support may allow agents to handle the entire conversation dynamically and not simply interpret messages one-by-one following a previously determined structure. Modelling dialogue is widely inspired by language philosophy, cognitive science and social psychology research [MDCd99]. In dialogue modelling, the intention of a message is not pre-determined but co-constructed along the conversation. A message has a 'potential of meaning'. It is the interlocutionary logic (instead of illocutionary logic in classical speech act representations) introduced by Brassac and Trognon and reported in [BP99].

Dialogue is strongly bound to a social architecture such as the one proposed in [Sin98] and consequently to utterances [GGT94] and representation of the other (the term used in the rest of the manuscript is *interlocutor model*). Searle talks about *background* to consider social rules and implicit knowledge intervening in communication. Other researchers propose to use dialogue games, inspired by interaction protocol, as a dialogue structure. For example, [Mau02] raises the question: are mental states sufficient in order to model the conversational behaviour of a system? Dialogue modelling gives some solutions to conversation modelling, for example, the sincerity prerequisite is no longer needed with this approach since agents develop a strong interlocutor model.

> DEFINITION: DIALOGUE MODELLING
>
> Identifies the research problem of agent communication that aims to model dynamic conversations. It expects agents to consider only messages from interlocutors and allow them to handle the entire conversation dynamically and not simply interpret messages one-by-one following a previously determined structure.

The STROBE model is highly inspired from both these two conversation modelling approaches. However, one important aim of the STROBE model is to enable to conceive and implement dialogues. In chapter 4, we present a computational abstraction based on stream processing which aims to model some kind of dialogues that may occur in DSG scenarios.

Modelling dialogue, is still a research challenge in the MAS community. This challenge is analogous to the one of SOC community: to realize dynamic conversation based high level processes of services. These two challenges are part of DSG.

Characteristic 16 (dialogue)

> *DSGS conversational processes should be made of dialogues. They should be dynamic and not pre-determined by a data structure that guides the conversational processes, thus limiting agent interpretation.*

2.3.3.3 Automated negotiation

Agents are often said to have good negotiation abilities. Negotiation is the process by which two or more parties communicate in order to reach a mutually acceptable agreement on a particular matter. In [JFL$^+$01], Jennings et al. describe negotiation in terms of three broad topics: (i) *protocols*, as the set of rules that govern the interaction; (ii) *objects*, as the range of issues over which agreement must be reached; (iii) *decision making models* or *strategies*, as the way (respecting the protocol) in order to achieve private objectives. The set of agent negotiation protocols is a subset of interaction protocols. For example, the Contract Net protocol is a negotiation protocol. Game theoretic models (e.g., auctions, prisoner's dilemma, etc.) are often taken for negotiation protocols or strategies.

In the thesis negotiation is viewed as a specialization of conversation, that is why we have detailed a lot conversational aspects. We assume everything possible with negotiation is possible with conversation and even more. Indeed, negotiation may often allows service providers and users to find an agreement, however negotiation mechanisms may sometime not be applied. In DSG some requirements of negotiation models may be not satisfied. For example, negotiation intrinsically supposes the agent (e.g., the user) to have a precise and identified objective (D1). Section 2.2.1.3 explains that service negotiation is the step of the service life-cycle which leads to the contract. This contract defines the conditions under which the service will be exchanged. C10[negotiation] states that such contracts cannot be always defined with dynamically generated services. During service exchange, when a service provider agent absolutely not take into consideration the real user agent problem, negotiation may not help, because it supposes that the user exactly knows what he precisely needs and that he uses a service only for a product delivering. DSG shifts the focus of service exchange from negotiating a service to finding a solution to a problem.

2.3.4 MAS-SOC integration approaches

2.3.4.1 Agent abilities for service exchange

Agents are said to use and provide services to one another (in the sense of particular problem solving capabilities). Actually they have many interesting characteristics for service exchange:

- Reactive and proactive;

- Efficient and adaptive;

- Know about themselves (self-consciousness);

- Memory and state persistence;

- Able to have conversations (not yet dialogues) and work collaboratively;

- Able to negotiate;

- Able to learn and reason to evolve;

- Deal with semantics associated to concepts by processing ontologies.

Actually, Web services are most of the time an interface on object oriented programs (developed via J2EE or .NET) processed to produce a WSDL description and to become able to communicate with SOAP messages. Thus, services benefit from the powerful abstraction characteristic available with the object oriented paradigm, such as encapsulation, inheritance, message passing, etc. One of the crucial evolutions of DSGS concerns the substitution of the current object oriented kernel of services by an agent oriented kernel. This will have the fundamental advantage to ensure services to be proactive (autonomous), intelligent and really interactive. Viewing service providers and users as agents is for us

the best way to realize dynamically generated services and address the question of high level process of services. The persistence of conversations among service providers and users (dynamic dialogue, not just prefixed interaction protocols), as well as the run-time reasoning of them (i.e., progressive evolution of the performance) could be reached only with agents.

Characteristic 17 (agent)

> *DSGS should be agent oriented. Firstly, because of the adapted properties of agents for providing and using services. Secondly, because agents are the current best metaphor for humans in computing i.e., agents enable to use a unique model for HAs and AAs.*

2.3.4.2 Agents and Web services

The research activity about the convergence of MAS and SOC is increasingly active.[18] As [SH05] states: 'Researchers in MAS confronted the challenges of open systems early on when they attempted to develop autonomous agents that would solve problems cooperatively, or compete intelligently. Thus, ideas similar to SOAs were developed in the MAS literature'. [GMM98] prefigured the use of agents in e-commerce scenario (i.e., for service delivery scenarios). The authors made an overview of e-commerce application in 1998 and identify issues in which agent abilities may enhance these scenarios: recommender systems (product/merchant brokering), user interface approaches, and negotiation mechanisms.

For Huhns et al. [HSB$^+$05] the key MAS concepts are reflected directly in those of SOC: ontologies, process models, choreography, directories and service level agreements. Some work has already been proposed for using agents to enhance Web services or integrating the two approaches. For a detailed comparison between these two concepts see, for example, [Mor02]. [Huh02] points out some drawbacks of Web services which significantly distinguishes them from agents: they know only about themselves, and they do not possess any meta-level awareness; they are not designed to utilize or understand ontologies; and they are not capable of autonomous action, intentional communication, or deliberatively cooperative behaviour. According to us, different kind of approaches may be distinguished in agent-Web service integration:

- *Distinct view of agents and Web services.* Agents are able both to describe their services as Web services and to search/use Web services by using mappings between MAS standards and SOA standards [Mor02, LRCSN03, GC04, SHMS04]. This approach is based on a gateway or wrapper which transforms one standard into another. As the main approach in agent standardization is FIPA's, this work often only considers FIPA agents and resolves relations between SOA and FIPA standards;

- *Uniform view of agents and Web services.* Agents and Web services are the same entities. All services are Web services and they are all provided by agents (that means that the underpinning program application is an agent-based system) [IYT04, Pet05];

- *MAS-based service-oriented architecture mechanisms.* MAS to support SOA mechanisms. This approach is not directly interested in agent service-Web service interaction but rather in the use of MAS to enhance SOA mechanisms. For example, [MS03] discusses the use of agents for Web services selection according to the quality of matching criteria and ratings. This aspect is not detailed;

- *MAS-based high level process of services.* Detailed in section 2.4.5.

---

[18]See for example the 2003 and 2004 International Workshops on Web Services and Agent Based Engineering (WSABE) and the 2005 and 2006 International Workshops Service-Oriented Computing and Agent-Based Engineering (SOCABE) held in conjunction of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS).

Distinct view of agents and Web services. In this approach, [Mor02] discusses the difference between agents and Web services from an implementation perspective. Agents communicate via SOAP enabled `startpoint/endpoint` pairs that allow access and remote method invocation on distant objects. [LRCSN03] identifies two key ideas: (i) agents should be able to publish their services as Web services for the potential use of non-agent clients; (ii) agents should advertise other entities using both a SOA standard registry (e.g., UDDI registry with WSDL entries) and an agent standard registry (e.g., FIPA DF with FIPA-SD entries). This is also the case of [GC04], which proposes a Jade agent architecture, based on a Web Service Integration Gateway Service (WSIGS), that contains several components that operate on internal registries to maintain records of all registered services (both agent services and Web services). [GC04] takes some of the ideas generated by the Agentcities Web Services Working Group [DHKV03] with Web Service Agent Gateway (WSAG). A drawback of this approach lies in the fact it is limited to FIPA compliant agents.[19]

A particularly difficult factor in this approach is communication. The challenge consists of bridging the gap between asynchronous behaviour of agent communication and synchronous behaviour of Web service interactions. For example, in [BV04] a Web service or an agent plays the role of a gateway that transforms a SOAP call into an ACL message. With WSAG, when a Web service invokes a SOAP call on the WSAG, the gateway transforms this synchronous call into an asynchronous FIPA-ACL message it sends to the agent that provides the service. In the same sense, the WSAG transforms a FIPA-ACL message into a SOAP call but ignores the semantics, which cannot map in SOAP. Actually, this aspect is very important as the biggest risk in ACL - SOAP integration is to reduce high level and semantically rich agent communication to simple request/response semantically poor Web service interaction. As another example, [GC04] approach reduces WSIGS-agents interactions to FIPA `request` and `inform` message in order to map with the basic request/answer behaviour of Web services. We further think that one condition to avoid this risk, is that the SOAP - ACL transformation should be done by the agent itself and not by another entity (Web service or agent): the agent and the Web service should be the same as it is the case in the next approach.

Uniform view of agents and Web services. In this approach, [Pet05] claims that in order to integrate agent and Web service technology, components have to be designed, which map between the different mechanisms for service description, service invocation, and service discovery, in both worlds. [Pet05, IYT04] are specifically interested in mobile agents (agents which have the ability to move from one host to another with their state). They propose respectively a 'Web service engine' architecture which aims to provide bidirectional integration of both technologies and 'mobile Web services' as an integration of Web services and mobile agents in order for Web services to benefit from mobility and mobile agents to benefit from the ability to compose standardized loosely coupled distributed applications.

Notice than even if this approach avoid the problem of SOAP-ACL transformation evoked before, it is always limited by SOAP drawbacks (e.g., no semantics). This one of the strong reason for which Web services technologies are not sufficient to implement DSG.

2.4 GRID

2.4.1 Concepts and definitions

Sharing resources in virtual organizations. The initial reasons for conceiving and developing GRID technologies in the mid-90's was that we were facing a major increase of computational needs, for instance in the scientific community (for weather modelling and nuclear physics). There was an increase

---

[19]One of the force of the agent paradigm is the diversity and heterogeneity of agents. Standardization is a strong aspect for Web service interoperation but non-standardization is one of the reason for which MAS are today so different and able to address a large scope of problems. Agents and Web services need each other exactly because they come from different universes (i.e., industry for Web services and academy for MAS). See also the discussion on the page 54 sidebar.

of computational resources (machines and networks), for which many resources were proprietary and therefore not accessible directly nor on-demand. The original question was how to build a new infrastructure (the Grid) that would satisfy the computational needs in an on-demand fashion by exploiting the computational resources in a seamless way, not requiring 'physical possession' of the resource by the potential clients. The essence of GRID is nicely encapsulated by its original metaphor: the delegation to the electricity network to offer us the service of providing us enough electric power as we need it, when we need it, even if we do not know where and how that power is generated. At the end of the month, we pay a bill that corresponds to our consumption. The Grid aims to enable flexible, secure, coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organization [FKT01]. GRID provides the protocols, services and software development kits needed to enable flexible, controlled resource sharing on a large scale. This sharing is, necessarily, highly controlled, with resource providers and users defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. The Grid is naturally highly dynamic and should be able to adapt at runtime to changes in system state as resource availability may fluctuate. Such fluctuations may result from connection/disconnection of computing resources, human interaction/interruption on the computers, etc. The Grid was originally limited to computing and storage services but later extended to any kind of service that was based on the dynamic allocation of virtualized resources to an instantiated service. Actually, the Grid was originally designed to be a computational environment with a large number of networked computer systems where computing (Grid computing) and storage (data Grid) resources could be shared as needed and on demand. However, since the emergence of the *Semantic Grid* in 2001 [RJS01], the Grid is considered as a service-oriented infrastructure that includes information handling and support knowledge within the e-science process. These thesis contributions are part of a Semantic Grid perspective.

DEFINITION: GRID (THE)

Service oriented architecture based on a flexible, secure and coordinated resource sharing infrastructure allowing dynamic service exchange among members of several virtual communities/organizations. The Semantic Grid highlights the information and knowledge dimension of these service exchanges.

Projects and applications.   [FK99b] firstly identifies five major application classes for GRID: (i) distributed supercomputing; (ii) high-throughput computing; (iii) on-demand computing; (iv) data-intensive computing; (v) collaborative computing. Some famous GRID projects were NASA's Information Power Grid, the European Data Grid, the Network for Earthquake Engineering Simulation Grid (NEESgrid), etc. Today, GRID technology is used for a very large range of applications and domains: distributed data integrations [CTT05], knowledge discovery [CT03], e-learning [NGW05, ACR+05, RAC+05], resource management approach [Cao01, LL04, GCL04], provenance [CTX+05], collaborative environment [DLJC06], SOAs [KWvL02, MPD+03, FFG+04, MBB+05], semantics [RJS01, Gel04, RJS05], etc.

GRID layers.   Jeffery [Jef99], Rana and Moreau [RM00] and De Roure et al. [RJS01] identify three layers in the Grid infrastructure (figure 2.9):

- the *Computational Grid*, the lowest layer, is primarily concerned with large-scale pooling of computational and data resources; *This layer deals with the way that computational resources are allocated, scheduled and executed and the way in which data is shipped between the various processing resources*;

- the *Information Grid*, the middle layer, allows uniform access to heterogeneous information sources and provides commonly used services running on distributed resources. The granularity of the offered services can vary, from method calls to complete applications. *This layer deals with the way that information is represented, stored, accessed, shared and maintained*;

- the *Knowledge Grid*, the top layer, provides specialized services that involve an aggregation of many different types of services and the integration of data and results provided by each service. *This layer is concerned with the way that knowledge is acquired, used, retrieved, published and maintained to assist e-Scientists to achieve their particular goals and objectives.*

The last layer is concerned with what we call in this thesis high level process of services; it is the layer of DSG. The term 'Knowledge Grid' was after reused by [CT03] and [Zhu04]. Moreover, [RM00] and [RJS01] papers envision both the development of services as the key concept underpinning GRID environments and the role that agents may play in this development by providing 'Grid services'. Figure 2.9 shows different GRID key concepts at each layer. These concept symbols are introduced in chapter 5.
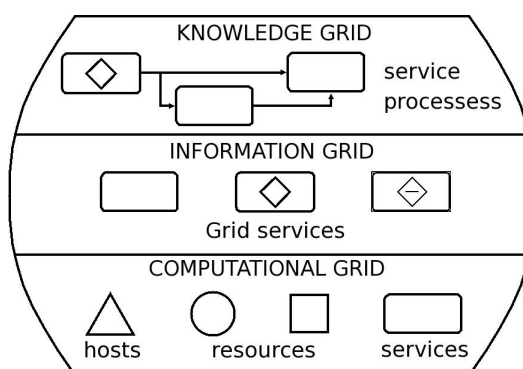
Figure 2.9: The three layers Grid infrastructure

Virtual organization. A concept introduced by GRID, fundamental for DSG, is the concept of *Virtual Organization* (VO). Grid users are members of virtual organizations (also called *communities*). A virtual organization is a dynamic collection of individuals, institutions and resources bundled together in order to share resources and services as they share common goals. It should be seen as a group of people who share common interests or participate to a common enterprise, and who assemble, collaborate, and communicate in a loose, distant, virtual way, using network communication facilities, tools and resources. A VO may be defined by the set of services that members operate and share. Examples of VOs are: members of the same company, a consortium, the organic chemistry community, etc. The social perspective of DSG is very important as services define relationships within communities (section 1.2.2). Communities are a key element of DSGS as introduced in section 1.3.2. GRID and MAS as we will see in section 2.4.5.3 propose both analogous organizational structures. They motivate and justify the integration of GRID and MAS.

Characteristic 18 (community)

DSGS should be structured in (virtual) communities. They represent the set of relationships created between service provider and service user agents during DSG. It is the social dimension of DSG.

2.4.2   The anatomy of the Grid

The Grid architecture description is given by [FKT01]. It proposes a quite complete enumeration of all required protocols and services that specify GRID. The implementation of these protocols and services is also detailed within the Globus Toolkit (GT), the main current tool that implements GRID specifications.[20]

This architecture is implemented by means of a set of components and protocols. The most relevant components are:

- The *Grid Resource Allocation and Management* (GRAM) [CFK$^+$98, FK99b] protocol which provides for the secure, reliable, transient service creation (with the GRAM gatekeeper/factory) and management (information, identity, or state) publication and monitoring with the GRAM reporter (i.e., registry and discovery features).

- The *Grid Security Infrastructure* (GSI) [FKTT98] is a set of libraries and tools that allow users and applications to share and access resources securely. In particular, it supports authentication, delegation, privacy and integrity:

    - *Authentication*: both parts of a secure exchange (e.g., resource sharing, service exchange) must be authenticated. The process of mutual authentication is an exchange of electronic certificates in X509 format. *X509 certificate* is the first important element of the GSI. It is a kind of passport which enables single sign-on and authentication. A X509 certificate is valid if it has been signed by a trusted *Certification Authority* (CA) and if it has not been altered.[21] Trusted Grid entities (i.e., X509 holders) can be either users or hosts.

    - *Delegation* is also assured by X509 certificate. A 'proxy certificate' or 'proxy credential' is a certificate not signed by a CA but by a user. This user may limit the proxy certificate period of validity. This mechanism can be assimilated to a mandate in the way that it allows the holder to act on behalf of another one.

    - *Privacy* of information is ensured with key-based cryptographic algorithms (with public/private keys[22]).

    - *Integrity* means that the recipient of data must be sure these data are conform to the original. The data could have been modified intentionally or not. In order to ensure integrity, hash functions and algorithms such as the Message Digest (MD5) prints are used.

    The other important element of the GSI is the *Community Authorization Service* (CAS). [PWF$^+$02] introduces the CAS as a solution to the problem of how to specify and enforce VO policies (cost of administering a VO, hierarchy of VO, etc.). The idea consists in allowing resource owners to grant access to blocks of resources to a VO as a whole, and let the VO itself manage fine-grained access control within that framework. The CAS is responsible for managing the policies that govern access to a VO's resources. It implements the VO's policy of service usage. A VO member may send the CAS a request for a service that will allow the member to perform a set of actions; if that request is consistent with the VO's policy, the CAS will delegate an appropriate service back to the user. The CAS reduces the complexity of VO administration by reducing the relationship

---

[20]The Globus Toolkit [FK99a] is proposed by the Globus Alliance (`www.globus.org`) and follows the specifications of the Global Grid Forum (GGF) (`www.gridforum.org`). Previous tools were Unicore, Legion, Condor, etc.

[21]A X509 certificates contains four parts: (i) information fields about the holder; (ii) the holder's public key; (iii) digital signature of the CA; (iv) a fingerprint. Any alteration of one of the three first parts of the certificate would invalidate the fingerprint.

[22]This process, known as asymmetrical algorithm, consists to use two different keys to encrypt and to decrypt the information. The owner of the information encrypts the message with its private key and provide its public key to allow the receiving peer to decrypt the message

between a user and a service from m*s (if m is the number of members of the VO and s the number of services available for this VO) to m+s: each member needs to be known, and trusted, by the CAS, but not by each service; each service needs to be known and trusted by the CAS, but not by each user.

Both X509 certificate and the CAS are defined in chapter 5 as AGIL's concepts. Their relations with other elements of GRID and MAS are also formalized.

- The *Meta Directory Service* (MDS-2) [CFFK01], which provides for publishing and accessing information about Grid resources, services, computations, etc. It becomes the *Monitoring and Discovery Service* in more recent specifications.

GRID trust environment.   In DSG, agents need a trusted environment to collaborate freely with others. One problem in remote service exchange is the fact that agents may not know each other. Collaboration and knowledge creation (D11, C4[learning]) may occur only via an environment in which they exchange their knowledge trustingly. This environment requires security to ensure privacy and reliability to ensure anytime availability. Such requirements are fulfilled by GRID thanks to the GSI.

However, trust depends not only on security, as [SH05] states:

'Trust goes beyond security, for example whereas security is about authenticating another party and authorizing actions, trust is about the given party acting in your best interest and choosing the right actions from among those that are authorized. Trust is complementary to assurance. If you trust the other party more, you need fewer assurances about its good behaviour. The existence of assurances can also engender trust. Some ingredients of trust are: capability, sincerity, helpfulness, duty, predictability, understanding.'

Characteristic 19 (trust)

DSGS need trust environments to favour real collaboration and knowledge creation through service exchange. Security plays an important role for trust.

Virtualization.   With GRID, the concept of virtualization appears. Virtualization is done at two levels:

- the *resource level*; GRID allows to virtualize resources from multiple heterogeneous platforms (after denoted hosts) within hosting environments (after denoted service containers), and re-affect (after denoted reifier) virtualized resources to Grid services. This virtualization and reification processes are formalized in AGIL in section 5.2.1.

- the *service level*; GRID allows to virtualize and encapsulate behind a common interface diverse capabilities or functionalities implemented differently. This is what we call the 'servicization' process. Within AGIL, this process is achieved by agents, placing their capabilities as Grid services at VO disposal in service containers.

The servicization process allows to virtualize high level process of services without regarding how the services being composed are implemented. Virtualization is easier if service functionalities can be expressed in a standard form, so that any implementation of a service is invoked in the same manner. For that reason, GRID adopts a SOA standards compliant set of specifications as it is described in the next section.

For the reasons exposed in section 2.2.2.2, the Web seems sufficient for storing and retrieving information, but not for real service exchange, such as it is needed in DSG. Furthermore, as [FKT01] says:

Grid architecture layers [FKT01]

1. The *Fabric* layer provides the resources to which shared access is mediated by Grid protocols (e.g., computational resources, storage systems, catalogs, network resources, and sensors). Resources should implement at least *enquiry* mechanisms that permit discovery of their structure, state, and capabilities (e.g., whether they support advance reservation) on the one hand, and *resource management* mechanisms that provide some control of delivered quality of service, on the other.

2. The *Connectivity* layer defines core communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between Fabric layer resources. Authentication protocols build on communication services to provide cryptographically secure mechanisms for verifying the identity of users and resources.

3. The *Resource* layer builds on Connectivity layer communication and authentication protocols to define protocols for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer implementations of these protocols call Fabric layer functions to access and control local resources. Information protocols are used to obtain information about the structure and state of a resource, e.g., its configuration, current load, and usage policy. Management protocols are used to negotiate access to a shared resource.

4. The *Collective* layer contains protocols and services that are not associated with any one specific resource, but rather are global in nature and capture interactions across collections of resources. This layer can implement a wide variety of sharing behaviours without placing new requirements on the resources being shared (e.g., directory services, scheduling and brokering services, monitoring and diagnostics services, data replication services, Grid-enabled programming systems, workload management systems, software discovery services, community authorization services, payment services, collaboration services).[a] These services are often said to be *persistent* (i.e.; always available – opposed to *transient*, explained next section)

5. The *Application* layer comprises the user applications that operate within a VO environment. Applications are constructed in terms of, and by calling upon, services defined at any layer. The term application identifies what we call in this thesis distributed software component based approaches e.g., CORBA, DCOM. We will see next section that application are now Web services.

---

[a]An application constructed upon a Grid-based architecture is often characterized by the set of services available at the Collective layer. For example, in [DLJC06], we proposed a 'desktop-oriented' Grid-based collaborative environment for human collaboration; Its architecture is defined by a set of six services, added by default to VOs' service containers: a notification service, a member management service, a service management service, a service activation service, a collaboration session management service, a history service. Each member can dynamically add new services (at the application layer) to address the VO collaboration needs using the service management service.

'However, while the Web technologies do an excellent job of supporting the browser-client-to-Web-server interactions that are the foundation of today's Web, they lack features required for the richer interaction models that occur in VOs.'

Without abandoning the Web (C1[web]), another infrastructure seems required. The Grid is the first distributed architecture (and infrastructure) really developed in a service-oriented perspective.

Characteristic 20 (grid)

DSGS should be GRID oriented because the Grid is the first service-oriented architecture and infrastructure addressing the question of secure and reliable resource sharing and service exchange by virtualization (resource and service level) among VOs.

For more information about GRID applications, projects, specifications, etc. the literature is very rich, for instance [FK99b, FK03, BFH03]. For an overview of GRID programming models see [LT03]. In the following, we concentrate on GRID-SOC and GRID-MAS integration approaches.

2.4.3 GRID-SOC integration approaches

2.4.3.1 Open Grid Service Architecture

In order to be widely accepted, the GRID standardization activity faced the question of convergence of GRID and SOC standards. Therefore, the GGF and the W3C decided to make GRID evolve from ad hoc solutions, and, de facto standards based on the Globus Toolkit, to SOA standards (figure 2.10). This resulted to a GRID standardization split into two sets of layered specifications:

- the Open Grid Service Architecture (OGSA) [FKNT02], on the top layer;

- the Web Services Resource Framework (WSRF) [FFG$^+$04, CFF$^+$04b], on the bottom layer.

OGSA adopts the Web service framework standards and extend services to all kind of resources (not only computing and storage). Foster et al. denote the importance of transient service instance [FKNT02]:

'VO participants typically maintain not merely a static set of persistent services that handle complex activity requests from clients. They often need to instantiate new transient service instances dynamically, which then handle the management and interactions associated with the state of particular requested activities. When the activity's state is no longer needed, the service can be destroyed.'

They define a *Grid service* as a (potentially transient) stateful service instance supporting reliable and secure invocation (when required), lifetime management, notification, policy management, credential management, and virtualization. Therefore, OGSA introduces two major characteristics in the so-called SOA by distinguishing service factory from service instance:

- *service state management*. Grid services can be either *stateful* or *stateless*;

- *service lifetime management*. Grid services can be either *transient* or *persistent*.

Key OGSA concepts (e.g., host, resources, service container, etc.) and mechanisms (e.g., virtualisation, reification, handling) are extensively described in chapter 5, within the proposed Agent-Grid Integration Language.
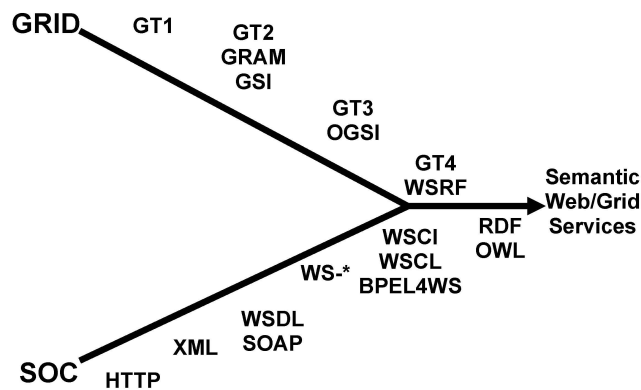
Figure 2.10: GRID and SOC standards convergence

State and lifetime management. Grid services can maintain an internal state for the lifetime of the service. The existence of state distinguishes one instance of a service from another that provides the same interface. The term *Grid service instance* is used to refer to a particular instantiation of a Grid service. A service instance is created by another service instance called a *Grid service factory*. Grid service instances can be created and destroyed dynamically.

Grid service instances are created with a specified lifetime. The initial lifetime can be extended by a specified time period by explicit request of the user or another Grid service acting on the user's behalf. If that time period expires without receiving a reaffirmation of interest from the client, the Grid service instance is terminated and the associated resources are released. We note that this approach to lifetime management provides a service with considerable autonomy. Lifetime extension requests from clients are not mandatory: the service can apply its own policies on granting such a request. A service can decide at any time to extend its lifetime, either in response to a lifetime extension user request or for any other reason. This aspect is one of the analogy detailed in section 2.4.5.2.

Characteristic 21 (stateful)

> *DSGS agents should be able to manage autonomously the state and lifetime of resources allocated for a certain period of time to the service exchanges. This state is the key element supporting interaction. The state also plays the role of history.*

A Grid service instance is identified by a *Grid Service Handle* (GSH) and a *Grid Service Reference* (GSR). The GSH is a globally unique URI which helps a user to locate a service, but which does not carry enough information for a user to communicate directly with the service instance (e.g., network address or supported protocol bindings). This is the role of the GSR, a WSDL document, that contains the information a client needs to communicate with the service. A GSH is valid for the lifetime of the Grid service instance, while a GSR might become invalid.

A Grid service instance implements one or more interfaces (corresponding to WSDL `portTypes`[23]), where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages. These interfaces and operations are summed up in table 2.3. The Grid service instance information (i.e., name, type, handle, lifetime, etc.) is encapsulated as an XML document in the `serviceData` element.

The Web service life cycle presented in figure 2.5 is enriched by a few steps more in the Grid service life cycle as figure 2.11 shows.

---

[23]A `portType` is an element in a WSDL document that comprises of a set of abstract operations each of which refers to input and output messages that are supported by the service.

DEFINITION: GRID SERVICE

A service instantiated with its own dedicated resources (state) for a certain amount of time (lifetime) available within a VO. Grid services are compliant with the Web service framework.
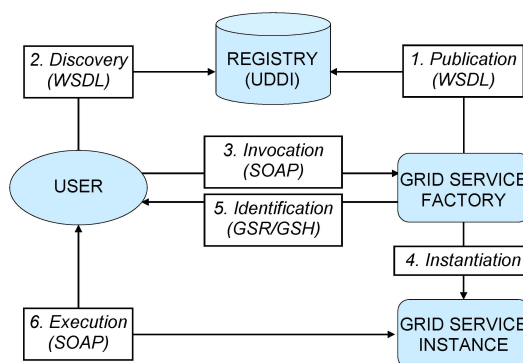

Figure 2.11: Grid service life cycle

2.4.3.2 Web Service Resource Framework

WSRF is a re-factoring and evolution of the initial specification Open Grid Services Infrastructure (OGSI) which aims at exploiting new SOA standards (in particular WS-Addressing and WS-Notifications). WSRF defines uniform mechanisms for defining, inspecting, and managing stateful resources in Web/Grid services. The motivation of WSRF comes from the fact that even if today's Web services sometimes successfully implement simple applications with state, they need to be standardized to enhance interoperability.[24] Introducing state in services is very important. Without state modelling, two service performances will always be the same. Since there is no state transitions, a stateless service does not enable interaction, adaptation, negotiation, engagement, etc. A fortiori, a stateless service cannot implement DSG. Introducing state is suggested by C17[agent], C20[grid], and C21[stateful]: DSGS merge agent state and Grid service state allowing to benefit from the advantage of both approaches (large amount of resource, security, intelligence, autonomy, reasoning, etc.). WSRF analyzes that it is important to identify and standardize the patterns by which state is represented and manipulated, so as to facilitate the construction and use of interoperable services. Therefore, WSRF describes how to implement OGSA functionalities using Web services and with respect to current SOA standards. WSRF identifies three types of services:

1. A *stateless service* implements message exchanges with no access or use of information not contained in the input message. These are represented as pure functions easy to compose;

2. A *conversational service* implements a series of operations such that the result of one operation depends on a prior operation and/or prepares for a subsequent operation. The behaviour of a given operation is based on the processing of preceding messages in the logical sequence. These are the most generic stateful services. Hard to be realized within a distributed and asynchronous context, heavy to be supported and maintained (HTTP sessions, and cookies, because HTTP does not have state);

---

[24]Actually, there are cases of stateless Web service that can manage state in an ad hoc way. For example, a timer function or a whether forecast function are considered to be stateless services, although the value of the time or the temperature may change every time the function is invoked. The state is in this case viewed as data values that persist across, and evolve because of, Web service interactions.

3. A *stateless service that acts upon stateful resources* provides access to, or manipulates a set of logical stateful resources (documents) based on messages it sends and receives.

WSRF explicitly addresses the third type. These services are modelled by an association between two entities: a stateless Web service, that do not have state, and stateful resources that do have state. Stateful resources are elements with state, including physical entities (e.g., databases, file systems, servers, distributed objects) and logical constructs (e.g., business agreements, contracts) that are persistent and evolve because of service interactions. In WSRF, a stateful resource is defined to: (i) have a specific set of state data expressible as an XML document; (ii) have a well-defined life-cycle; and (iii) be known to, and acted upon by one or more Web services. The fact of considering several services sharing the same stateful resource is one of the reason of WSRF. WSRF calls the resulting association a *WS-Resource*. Both stateful resources and stateless services can be members of several WS-Resources. However, a WS-Resource is unique and produced by a WS-Resource factory (in WSRF the GSH/GSR pair is replaced by a single WS-Addressing `EndpointReference`). When a user requests a Web service which should act upon resource, a WS-Resource is created and a endpoint is returned to the user. User's messages must be sent to a WS-Resource referred by this endpoint.

Characteristic 22 (separation)

DSGS should provide agents with the functionalities enabling several accesses and managements of stateful resources allocated to service exchange. Agents should be able to manage and dynamically change services and resources separately.

WSRF SPECIFICATIONS

WSRF is a set of five specifications [CFF$^+$04b]:

- WS-ResourceLifetime, which allows a user to specify the period during which WS-Resource definition is valid (mechanisms for WS-Resource destruction (immediately or scheduled));

- WS-ResourceProperties which defines (as a XML document) how a WS-Resource can be queried and changed using Web service technologies; it allows clients to build applications that read and update data. The WS-Resource properties document acts as a view on, or projection of, the actual state of the WS-Resource;

- WS-RenewableReferences, WS-ServiceGroup, WS-BaseFaults.

For a recent precise overview of Grid service concepts and standardization, see for example [CTT05]. We will see in section 5.4.2 how the GRID-MAS integrated model tends further towards DSG than WSRF, enabling a capability-environment couple (i.e., akin to a WS-Resource) to adapt user's specific needs both at the resource (environment) and service (capability) levels.

Table 2.3 sums up the main concepts of OGSA and makes a correspondence between OGSI and WSRF constructs (OGSI interfaces and operation are respectively in italic and in typewriter font). This table is inspired from table 2 and 3 of [CFF$^+$04a].

2.4.4 GRID-MAS integration approaches

Even though GRID and MAS are both kinds of distributed systems, their underlying motivations are different. GRID focuses on a reliable and secure resource-sharing infrastructure, whereas MAS focuses on flexible and autonomous collaboration and problem solving in uncertain and dynamic open environments. However their integration has been suggested as this section shows. Despite different original motivations, we explain in chapter 5 why these two complementary domains join with the concept of

Table 2.3: OGSA main concepts and OGSI/WSRF correspondence of constructs

CONCEPT	OGSI CONSTRUCT	WSRF CONSTRUCT
Service Representation	*Grid service*	WS-Resource
Service Property	`findServiceData`	WS-Resource Properties
	`setServiceData`	
Service Lifetime Management	`requestTerminationAfter`	WS-Resource Lifetime
	`requestTerminationBefore`	
	`Destroy`	
Service Instantiation	*Factory*	WS-Resource factory
	`createService`	
Service Identification	*HandleResolver*	WS-Renewable References
	`findByHandle`	
Service Addressing	GSH & GSR	WS-Addressing &
		`endPoint` reference
Service Notification	*NotificationSource & NotificationSink*	WS-Notification
	`subscribe`	
	`deliverNotification`	
Group Management	*ServiceGroup*	WS-Service Group
Exceptions Management	Base fault type	WS-Base Faults
Implementation	GT3	GT4

service. Actually, there is an increasing amount of research activity in GRID and MAS convergence taking place.[25] In a first time, we present here the related work at the intersection of the two domains. We detail, in particular, MAS based GRID approaches. In a second time, we explain three strong analogies between MAS and GRID concepts that have influenced our integration approach proposed in chapter 5.

The Control of Agent-Based Systems (CoABS) project [MT99], proposed in 1999 by DARPA, is the first research initiative in GRID-MAS integration. In this project, priority was given to GRID development, but the participants already envisage a combination of GRID and MAS domains. In [MT99] the authors raise several questions to characterize the 'Agent Grid' as a construct to support the rapid and dynamic configuration and creation of new functionality from existing software components and systems. The use of agents for GRID was very early suggested also by Rana and Moreau in [RM00]:

> 'By their ability to adapt to the prevailing circumstances, agents will provide services that are very dynamic and robust, and therefore suitable for a Grid environment.'

The authors specifically detail how agents can provide a useful abstraction at the Computational Grid layer (section 2.4.1) and enhance resource and service discovery, negotiation, registries, etc. MAS has also been established in 2001 as a key element of the Semantic Grid [RJS01]. In this research agenda, De Roure et al. consider a service-oriented view of the Grid where service users and providers

---

[25]See, for example, Agent-Based Cluster and Grid Computing workshops, Smart Grid Technologies workshops, the Multi-Agent and Grid Systems journal.

may be considered as agents. In particular, they insist on the important role that agent may played for implementing e-Science marketplaces i.e., the environment in which service users and providers interact one another. More recently, [RJS05] states:

> 'A pressing need is to develop standards and methods to describe the knowledge services themselves, and to facilitate the composition of services into larger aggregates and negotiated workflows. An important element of this is likely to be protocols and frameworks that emerge out of the agent community.'

In 2004, Foster et al. wrote a paper explaining why MAS and GRID need each other as brain and brawn [FJK04]; this paper is specifically detailed in next section. The most recent important paper that envision the importance of GRID in service-oriented MAS is [HSB$^+$05].

2.4.4.1   MAS and GRID need each other: brain meets brawn

Fortunately, GRID and MAS have several similarities and relate to the same domain: the development and deployment of a distributed SOA. However, GRID and MAS have developed different aspects of SOA. [FJK04] emphasizes the overlap in problems that GRID and MAS address but without sharing research progress in either area. The authors explain why GRID and MAS need each other. They describe GRID as the 'brawn' i.e., infrastructure, tools, and applications for secure resource sharing within dynamic and geographically distributed virtual organizations. However:

- GRID is rigid, inflexible and interaction-poor; GRID provides uniform mechanisms for accessing raw data, but is unable to deal with these data and their semantics to consider them as knowledge; Grid services are able to manage state, but do not have the intelligent ability to decide how and why to change state; orchestration/choreography to compose Grid services needs a high-level communication language which GRID does not provide; VOs are groups of Grid user entities (human and machine) that form according to common needs, objectives and services shared; but GRID does not provide the adequate mechanisms to create, manage, integrate into or leave a VO.

MAS is described as the 'brain', i.e., concepts, methodologies and algorithms for autonomous problem solvers that can act flexibly in uncertain and dynamic environments in order to achieve their aims and objectives. However:

- MAS are often not robust, not large scaled, and not secure; MAS need robustness, interoperation and standardization; MAS provide sophisticated internal reasoning capabilities, but offer no support for secure interaction or service discovery; MAS cooperation or collaboration algorithms produce socially optimal outcomes, but assume that agents have complete knowledge of all outcomes that any potential grouping can produce; MAS negotiation algorithms achieve optimal outcomes for the participating agents, but assume that all parties in the system are known at the outset of the negotiation and will not change during the system operation.

2.4.4.2   Status of current integration activities

Using MAS principles to improve core GRID functionalities (e.g., directory services, scheduling, brokering services, task allocation, dynamic resource allocation and load balancing) is a very active topic in the MAS community. We may distinguish two major domains of application:

MAS-based GRID for resource management. Resource management is one of the central components of wide-area distributed computing systems like GRID. It is a core functionality of GRID infrastructure (e.g., GRAM). It is not a trivial task, since the nature of the GRID environment is hardly predictable. Agents may be used for an effective management of the vast amount of resources that are made available within a GRID environment as they have, for example, excellent trading and negotiation abilities (negotiation between resource agents and allocator agent). Another example could be the use of machine learning algorithms to allow agent allocating task, and to learn each time a previous allocation turns out to be a good/bad one. We may cite some important projects that have investigated this aspect:

- The CoABS project mentioned before. The major added value of CoABS Agent Grid is coordination and seamless integration of the available distributed resources (including heterogeneous MAS, object based systems, legacy systems, etc.);

- The AgentScape project [WOvSB02] provides a multi-agent infrastructure that can be employed to integrate and coordinate distributed resources in a computational Grid environment. AgentScape is specially concerned with scalability i.e., wide-scale or Internet scale (based on the fact that peer-to-peer interaction strategies, as embraced by MAS, seems to be the promising approach for scalability);

- The Agent based Resource Management System (ARMS) project [Cao01] had started an implementation of an agent-based resource management for GRID in which an agent system bridges the gap between Grid users and resources in order to efficiently schedule applications that require Grid resources. The idea of ARMS was to consider that each agent acts as a representative for a local Grid resource. For a more recent reference see [CSJN05].

In using agent techniques for GRID resource management we can also refer: [SLGW02] proposes to use an agent oriented negotiation method (an interaction protocol, an auction model or a game theory based model) to enhance load balancing. This improves resource management by dynamically mapping user agents and resource agents. Other examples are [WBPB03, Tia05]. [GCL04] considers a system consisting of large number of heterogeneous reinforcement learning agents that share common resources for their computational needs. [LL04] provides a price-directed proportional resource allocation algorithm. [MBP05] presents an agent-based resource allocation model for GRID using three types of agents (job, resource brokering and resource monitoring agents). Using agents for resource management is a also an approach used outside GRID. For example, [HJM05] proposes a multi-agent approach for resource allocation in communication networks (allocating bandwidth) using market-based agents. An interesting aspect of this approach is the use of a back-tracking mechanism which allows alternative allocations to be made if currently reserved resource bundles cannot lead to the final destination.

MAS-based GRID for VO management. Another domain where agent abilities are used to enhance core GRID functionalities is VO management i.e., formation, operation and dissolution of VOs. The main work in this domain is the Grid-enabled Constraint-Oriented Negotiation in an Open Information Services Environment (CONOISE-G) project. It seeks to support robust and resilient VO formation and operation, and aims to provide mechanisms to assure effective operation of agent-based VOs in the face of disruptive and potentially malicious entities in dynamic, open and competitive environments [NPC+04, PTJ+05]. This project allows a VO manager to find service providers thanks to yellow pages, check service quality thanks to a QoS consultant, add/remove service provider in the VO, etc. Each of these roles are played by agents interacting one-another.

Other related work. [GHCN99] describes an approach, called the Mobile Agents Team System (MATS), to dynamic distributed parallel processing using a mobile agent-based infrastructure. In MATS, large computations are initiated under control of a coordinating agent that distributes the computation

over the available resources by sending mobile agents to these resources. [TV01] presents an architecture for monitoring services in GRID based on mobile agents. [Mor02] applies and situates its reflection on the comparison of agents and Web services to the context of bioinformatics Grid.

Our vision of integration. However, none of this work proposes a real integration of MAS and GRID. Rather, they focus on how MAS and AI techniques may enhance core GRID functionalities. Our vision of a GRID-MAS integration is not a simple interoperation of the technologies. It goes beyond a simple use of one technology to enhance the other. We aim to adopt a common approach for the integration to be able to benefit from the most relevant aspects of both GRID and MAS. This common approach is centred on the concept of service. One of the crucial explorations concerns the substitution of the current object oriented kernel of services available in GRID by an agent oriented kernel.

2.4.5 GRID-MAS analogies

Through these states of the art (SOC, MAS, MAS-SOC, GRID, GRID-SOC, GRID-MAS), we have identified a set of equivalent concepts in GRID and MAS. In particular, we detail in this section three analogies that have strongly influenced our integrated model, presented in chapter 5.

2.4.5.1 Agent communication vs. high level process of services

Message passing based communication. GRID and MAS use the same communication principles: direct message passing-based communication. MAS communication is also based on, direct or indirect, message passing. In message passing based communication, an interface (more or less standardized) hides the specific implementation detail, and private representations of the Web/Grid service or agent. The only difference (quite important) is asynchronous communication for agent and synchronous communication for Web/Grid services. The synchronicity of SOAP message exchanges forbids the dynamic creation of high level processes of services. As already explained, asynchronism is fundamental for autonomy, this is why DSGS should use asynchronous communication C11[message].

High level process of services vs. interaction protocol and agent conversation. Workflow or service orchestration (section 2.2.1.3) is analogous to interaction protocol in agent communication (section 2.3.3.2). Both terms describe a common interaction structure that specifies a set of intermediate states in the communication process as well as the transitions between these states. The applicability of MAS to workflow enactment has been noted by [SH99]. [BVV03, VBH04, BV04] explore the relation between Web services, MAS, and workflows. The authors note that, traditionally in workflow approaches, strict adherence to prescribed workflows implies that systems are largely unable to adapt effectively to unforeseen circumstances. They explain why workflow have some weaknesses and how decentralized, multi-agent workflow-enactment techniques can bridge the gap to dynamic high level process of services. Their vision is to create adaptive workflow capability through decentralized workflow enactment mechanisms that combine Web service and agent technologies. In particular, [BVV03] makes a strict comparison between workflow (in BPEL4WS[26]) and interaction protocol (as FIPA defines them):

- They are both languages for representing a series of structured communications among a set of actors. BPEL4WS uses the concept of a 'partner' which is identical to the 'roles' in interaction protocols;

---

[26]A workflow described in BPEL4WS details the flow of control and any data dependencies among a collection of Web services being composed.

- Both of them have complex flow mechanism (flow, pick, while) which support all desired iterative behaviours;

- BPEL4WS provides fault handlers. Interaction protocols do not have an explicit concept of fault handling (however, existing they could be extended to handle faults);

- BPEL4WS workflows may be seen as a new service, which could be part of other workflow. Interaction protocol can not be seen as a new agent and may not easily be part of other interaction protocols (problem of interaction protocol composition);

- Since BPEL4WS was designed for Web services, and since Web services are stateless, workflow state have to be emulated. This is in contrast to interaction protocols, which avoid the complexities of explicitly modelling state because agents are stateful;

- Partners in BPEL4WS are assumed to be completely reactive. That is, the only actions they ever take are those prescribed by the BPEL4WS description and all those actions are reactions to other actions, namely, they are triggered by arriving messages and might depend on the partner's current state. Agents are generally assumed to be pro-active. That is, agents can take actions based on their own internal 'deductions' about the world at large.

Workflows prescribe exactly what can be done by each of the participants at any moment in time. The participants, therefore, do not need to understand the whole workflow. They can be implemented as simple reactive agents. Therefore, orchestrating services is very important, but the real added value by agents is more in choreography as agents dispose of good abilities to be engaged within conversation.

Conversation or service choreography is also analogous to agent conversation. Using agent conversations to enhance service exchange is an active research topic [MML05, AGP03, HNL02]. These papers explain that the engagement of Web services in long lived conversations is necessary to provide better adapted and smart services. Conversations allow leveraging Web services from passive components to active ones. Obviously, Web services have to turn to agent, and agent communication models, as the best sophisticated approach for managing conversations:

- [AGP03] suggests using a dialogue agent conversation modelling approach. The authors do not explain how to represent Web services by agents, but propose a conversational model (speech act inspired) that is not based on a diagram of messages to be sent (i.e., interaction protocol) but rather on local operation calls: the interaction is modelled as a sequence of turns where one of the peers requires that the other peer performs an operation. The service provider has to maintain a set of context-explicit interaction contexts, corresponding to each of its users. [AGP03] ideas are very close to the ones adopted by the STROBE model.

- In [MML05, HNL02] the formalisms used to concretely express conversations are some kind of Finite State Machines and/or Petri Nets. Exactly the same formalisms that agent communication uses in the interaction protocols based conversation approach.

- With the same idea, [MML05] proposes to use conversation schema (akin to interaction protocol) represented by state charts (akin to Finite State Machine) to model conversation in order to compose Web services. They suggest a 'context aware' approach representing all the possible states of a Web service, a conversation, a composite Web service.

- BPEL4WS, WSCL, WSCI, etc. only support orchestration or choreography at a syntactic level. They are too low level (implementation focused) to support reasoning at a conceptual and semantic level. Therefore, the Semantic Web service community recently started to address high level process of services at a semantic level [DCG05, BMMS03] but unfortunately without considering yet the great ability of agents to deal with this semantics during conversation [Gue02]. In particular, [DCG05] proposes a formal definition of choreography based

95

on Finite State Machine in the Internet Reasoning Service (IRS). The IRS is a framework and platform for developing Semantic Web services which utilizes the WSMO ontology (section 2.2.3). Choreography is the component of IRS which deals with Web service interaction; it is represented by a set of events, a set of states, a set of conditions and a set of guarded transitions. The authors insist on the dynamic aspect of composite services built according to a conversation: the service may be defined at run time; i.e., designed and executed in the same time.

It is important to understand, from this little overview, that the question of modelling interaction in MAS exists also in SOC. Moreover, the challenge of having dynamic agent conversation (i.e., dialogues not described by prefixed interaction protocols), as presented in section 2.3.3.2 occurs also in SOC with the question of dynamic high level process of services. To reach this challenge is one of the objective of DSG. We further think that a dialogue approach of agent conversation (more dynamic and not previously determined before the conversation) will strongly enhance agent communication and therefore service exchanges allowing to tend towards DSG.

2.4.5.2 Agent autonomy and intelligence vs. stateful and dynamic Grid service

'Intelligent agents' means that they are able to keep their own internal states and make them evolve in a way not necessarily only dependent on the messages they receive, but for example, with machine-learning algorithms. In an analogous manner, Grid services are stateful, i.e., they own their running context, where the contextual state memory is stored. An analogy can also be made between an agent having a conversation dedicating a context (i.e., a part of its state) to the conversation and a Grid service factory which instantiates a new service instance with its own state. This idea (section 3.3.2) is fundamental in our GRID-MAS integrated model.

'Autonomous agents' means that they are able to manage these states and their own resources alone, without the intervention of another entity (agent, process, etc.). This is analogous to lifetime management which allows Grid services to be dynamic and to manage by themselves the resources allocated to them (section 2.4.3.1).

2.4.5.3 Organizational structure

As the number of entities in the system increases, as the number of problems to be tackled grows, and as the system becomes more and more distributed, different perspectives appear on how to define the system behaviour. Instead of being centred on how each of these entities should behave, one may alternatively adopt the perspective of organizations. An organization is a rule-based partitioned structure in which actions can be taken (problem solving, interaction, etc.) by people playing roles and sharing one or more goals. Both GRID and MAS choose an organizational perspective in their descriptions.

In Organization Centred MAS (OCMAS) [WJK00, FGM03], the concepts of organizations, groups, roles, or communities play an important role. They describe the social relation of agents without imposing some mental state representation or convention, but simply by expressing external rules that structures the society. In particular, [FGM03] presents the main drawbacks of agent-centred MAS and proposes a very concise and minimal OCMAS model called Agent-Group-Role (AGR). We will base our GRID-MAS integrated model on this simple but very expressive model, summarized in table 2.4.

2.4.6 GRID evolution

2.4.6.1 Semantic Grid.

Recently, the same semantic level add-on objective that took place in the Semantic Web community occurred also in GRID. This is the concept of 'Semantic Grid' [RJS01, RJS05]. The Semantic Grid is an

Table 2.4: Organizational-structure analogies between GRID and MAS

MAS	GRID
Agent	Grid user
An agent is an active, communicating entity playing roles and delegating tasks within groups. An agent may be a member of several groups, and may hold multiple roles (in different groups).	A Grid user is an active, communicating entity providing and using services within a VO. A Grid user may be a member of multiple VOs, and may provide or use several services (in different VOs).
Group	VO
A group is a set of (one or several) agents sharing some common characteristics and/or goals. A group is used as a context for a pattern of activities and for partitioning organizations. Two agents may communicate only if they are members of the same group. An agent transforms some of its capabilities into roles (abstract representation of functional positions) when it integrates into a group.	A VO is a set of (one or several) Grid users sharing some common objectives. A VO and the associated service container is used as a context for executing services and for partitioning the entire community of Grid users. Two Grid users may exchange (provide/use) services only if they are members of the same VO. A Grid user publishes some of its capabilities into services when it integrates into a VO.
Role	Service
The role is the abstract representation of a functional position of an agent in a group. A role is defined within a group structure. An agent may play several roles in several groups. Roles are local to groups, and a role must be requested by an agent. A role may be played by several agents.	The service is the abstract representation of a functional position of a Grid user in a VO. A service is accessible via the CAS service. A Grid user may provide or use several services in several VOs. Services are local to VOs (situated in the associate container), and a Grid user must be allowed to provide or use services in a VO. A service may be provided by several Grid users.

extension of the current Grid in which information and services are given well-defined meaning, better enabling computers and people to work in cooperation. The integration of the semantic layer in Grid services will enhance the dynamic execution of high level process of services by reasoning about the semantics included in service descriptions and the semantics of the messages exchanged. The GRID-MAS integration is also motivated by the Semantic Grid framework as agents have been identified in [RJS01] as a key element for using and providing services. For a good overview of the Semantic Web, the Grid and the Semantic Grid see for example [Gel04].

The Knowledge Grid.   [CT03] proposes a GRID based architecture for supporting knowledge discovery processes such as data mining, data analysis, etc. They present the Grid as the best approach to support the implementation of parallel and distributed knowledge discovery platforms supporting high-performance data intensive distributed services. This Knowledge Grid architecture is based on a set of services constructed on the top of basic core Grid services. The integration of knowledge-based systems and GRID is detailed in [Zhu04] which describes the 'Knowledge Grid' concept as an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources.

Semantic Grid services.   We can guess that, as Web services evolve towards Semantic Web services, Grid services would do the same as figure 2.12 (from [Gel04]) shows. However, both the domain of Semantic Grid and Grid services are too young to see any concrete emerging standard yet.

For example, in [Jon05], we suggest an extension of WSMO basic elements required to fit with WSRF principles. The main components of WSMO are the four main elements of Web Service Mod-
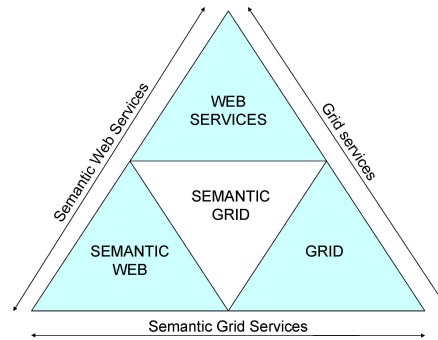
Figure 2.12: The intersection of Semantic Web, Grid and Web services

eling Framework (WSMF) [FB02]: *Goals* represent the types of objectives which users would like to achieve via a Web service. *Web service* descriptions represent the functional behaviour of an existing deployed Web service. The description also outlines how Web services communicate (choreography) and how they are composed (orchestration). *Ontologies* provide the basic glue for semantic interoperability and are used by the three other WSMO elements. *Mediators* which bypass interoperability problems by linking WSMO elements one another `oo-mediators`, `ww-mediators`, `wg-mediators` and `gg-mediators`. We concretely propose a simple extension of WSMO by two key concepts: (i) *Resources* as another WSMO main element and, (ii) *sr-mediators* to realize service-resource mediation and embodying the WS-Resource.

2.4.6.2 Learning Grid

The choice of GRID technology for learning is motivated by recent research on these topics [WW02, NGW05, ACR+05, Cer05, RAC+05] that provides a significant contribution in understanding the actual GRID potentialities related to distant collaboration, and that promotes the development of service-oriented usage (and expansion) of this technology in all contexts and especially where human collaboration is required (e-learning, e-government, e-health, e-business, e-science). This is the 'Learning Grid'. The aim of the Learning Grid is to progress effective human learning by promoting, supporting, and demonstrating a learning paradigm shift from the current information transfer paradigm, to one that focuses on knowledge construction using experiential and collaborative learning approaches in a contextualized, personalized, and ubiquitous way.

GRID for collaborative environment. GRID has many interesting aspects for service-based collaborative environments [DLJC06]: *Trusting the environment* happens through GRID reliability, robustness and security mechanisms; *synchronous and asynchronous modes of collaboration* are brought by the technical layers of OGSA, thanks to stateful and dynamic services; *Persistent memory* is brought by Grid services which allow to integrate and capitalize upon the past thanks to stateful resources; Service dynamicity is brought by transient aspect of Grid services; *Community member dynamicity* is brought by virtual organization management; *Usability* is improved by the fact that Grid services are compliant with SOA standards.

In [DLJC06] we tackle the problem of bootstrapping and supporting a collaborative environment over a GRID infrastructure. We propose the *Grid Shared Desktop* (GSD): a GRID-based, Web-accessible service that provides the members of a virtual community with a shared, online collaborative environment including protocols, services and facilities for bootstrapping and supporting their collaborative construction of shared knowledge. The GSD is developed within the European research project: European Learning Grid Infrastructure (ELeGI - `www.elegi.org`) and experimented by chemists to address the

problem of the collaborative construction of a shared ontology of organic chemistry. In the ELeGI ambitions, perhaps the most challenging one is the personalization of learning services for HAs, that will not be achieved unless a minimal formal, computational model of HA will be exploited during the generation of the corresponding services. We take this challenge by trying to develop a computable model for agents that can also serve the purpose of modelling HAs in a deliberately well-circumscribed context i.e., how to facilitate the ability of both AA and HA to dynamically generate services to one another. This aspect is not detailed in this thesis as it is just starting ongoing research [JEC05].

2.4.6.3   High level processes of Grid services

Applying high level process of services research results for GRID becomes a new research topics in the GRID community. For example, [KWvL02] addresses the challenge of modelling workflow of Grid services. Where Web service workflow approaches are centralized (with a workflow engine), Grid Service Flow Language (GSFL) proposes a peer-to-peer approach where most of the data is transferred from service to service directly.

2.5   Conclusion

The main contributions of this chapter are:

- To make a rigorous state of the art of the three domains SOC, MAS and GRID and especially of their interconnections (MAS-SOC, GRID-SOC and GRID-MAS);

- To achieve a list of DSG characteristic, inspired from this state of the art.

Chapter 1 and 2 have presented the theoretical framework of this thesis. We have proposed a reflection on the concept of service and we have tried to understand the fundamental meaning of this concept in order to be able to develop computer-mediated systems in which human and artificial entities may fully benefit from the opportunities emerging from the intrinsic meaning of the concept of service. We have introduced the concept of dynamic service generation and made a state of the art about the three domains we further think DSG is mainly related to. This state of the art includes theoretical principles and practical technologies for service exchange. It illustrates some aspects such as state, communication, dynamicity, social behaviour, etc. important for DSG, and more or less addressed by each of these three domains.

These two chapters correspond to the characterization phase of the scientific process described in the introduction. Therefore, the main result of the chapter is a list of DSG characteristics. Among these characteristics, two of them represent a concrete path to implement dynamic service generation systems. They are the agent oriented (C17[agent]) and the Grid oriented (C20[grid]) aspects of DSGS. Services have to be realized by intelligent, autonomous and interactive agents able to have conversations. These agents need a secure, robust and concrete infrastructure to host the service exchanges that the Grid may provide.

The list proposed in these two chapters (summarized at page 215) is absolutely not exhaustive. One may, of course, add characteristics if: (i) the application to service exchange is feasible; (ii) it represents an added value (enhancing today's PDS); (iii) it does not go in opposition with already identified characteristics or if it does, it can show the advantages of the substitution.

Figure 2.13 summarizes the identified DSG characteristics by including them in the sets (SOC, MAS and GRID) they are mostly related to. The figure illustrates that:

- All the SOC related characteristics are included in the GRID set. It shows us that the GRID-SOC integration may be considered quite done as it was demonstrated by the state of the art. What is not

done is the (GRID-SOC)-MAS integration (denoted after service based GRID-MAS integration). This is the objective of AGIL introduced in chapter 5;

- GRID and MAS are truly two key aspects of DSG (17/22 characteristics for MAS and 15/22 characteristics for GRID);

- 7 characteristics are included in the intersection of the three sets. Among them C10[negotiation], C12[process] and C13[semantics] are concretely addressed by research communities but they never really use the potentialities of the three domains they are related to.
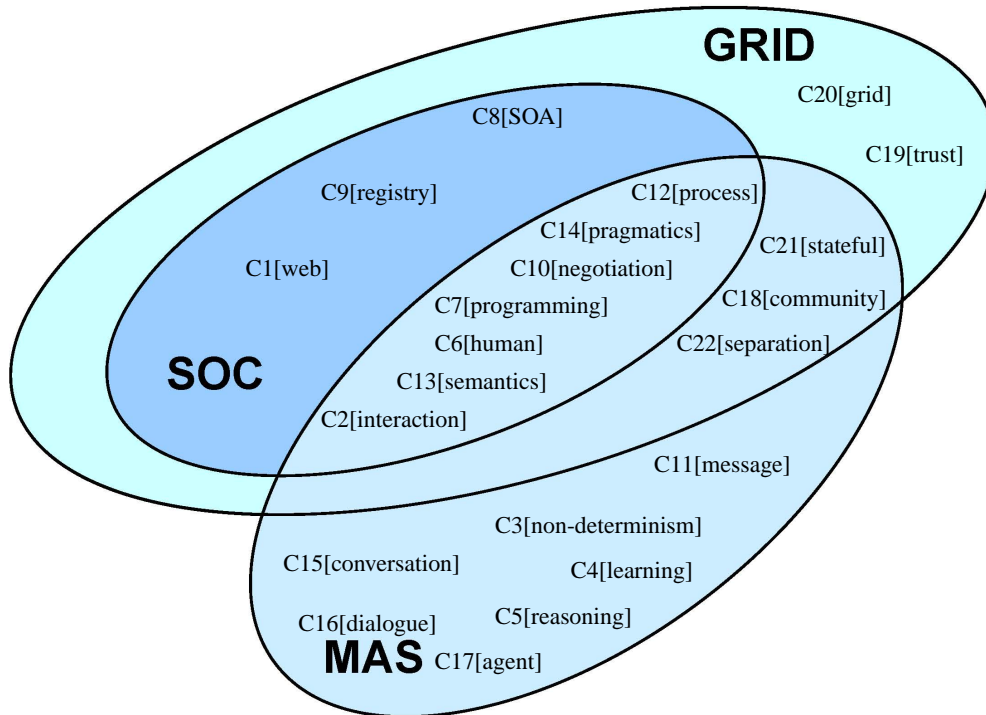

Figure 2.13: Inclusion of DSG characteristics in SOC, MAS and GRID sets

Our analysis has of course limitations. Section 1.2 could be enriched, by analyzing from sociological and psychological point of views, the way human beings exchange services one another. Informatics often takes inspiration from human beings (e.g., speech acts). Section 1.3 intentionally considers the minimal set of elements of DSG (agent, community, conversational process) but perhaps, other ones may have been considered (e.g., resource). In section 1.3.3 the comparison between DSGS and PDS should perhaps have been more technical; by taking for example Web services as PDS reference. Chapter 2 state of the art is, of course, not exhaustive and a different state of the art would have generated different research ideas and DSG characteristics.

The three next chapters present some results to achieve some of these characteristics. We could say that chapter 3 and 4 propose solutions or techniques whereas chapter 5 formalizes an integration model. Let us see what are the characteristics addressed by the next chapters. First, we choose an agent approach (C17[agent]) as it is for us the strongest characteristic. Obviously, as conversation is the DSG key element, we choose to work on agent communication (C11[message], C15[conversation], C16[dialogue]). In order to intrinsically model how agents communicate, we needed to model how to represent them. This is the reason why STROBE is both a communication and representation model for agents. The

key idea of the model is to develop a different dedicated language to each interlocutor agent by means of simple but powerful programming techniques (C7[programming]) (e.g., environments). These techniques allow us to propose solutions for addressing C22[separation] and C4[learning] and apply it to agents. I-dialogue, presented in chapter 4, addresses firstly C15[conversation] and C16[dialogue] as it is a conversation modelling approach. It also partially addresses C12[process] as this abstraction models conversations that an agent should have with other ones in order to provide a composite service (section 4.3.4). Secondly, we decided to explain how to integrate these results with the other major DSG characteristic C20[grid]. Therefore, we proposed an integration model that, according to us, addresses C8[SOA], C18[community] and C22[separation] by mapping the GRID answers for these characteristics with our previous results.

Besides highlighting the suitable aspects of GRID and agent for DSG, this thesis aims to bring the reader to reconsider some aspects of Informatics that are important in order to develop future interactive open distributed systems. The concept of service has a fundamental meaning which goes further than simply obtaining the delivery of a predetermined product. A real challenging change is needed with regards to the way services are provided and used in computer-mediated contexts. DSG enables, considering each domain challenge (i.e., modelling dialogues, dynamic high level processes of services, interactive computation, human integration, etc.) within the same perspective. It is like looking at these approaches through the same prism in order to integrate them. The quotation on the first page of the chapter reflects the need for integration. For the past twenty years, Informatics has exploded in a plethora of domains but keeping finally a quite limited number of challenges. The thesis takes the position that integration is a good direction.

Chapter 3

The STROBE Model

The whole is greater than the sum of its parts.
Aristotle

S TROBE is an agent representation and communication model. It is inspired
from constructs of applicative/functional languages and facilitates the imple-
mentation of a learning-by-being-told approach. The main idea consists in enabling
an agent to develop a dedicated language for each of its interlocutors or group of
interlocutors, by means of dedicated conversation contexts, called cognitive envi-
ronments. These conversation contexts represent the interlocutor models. They
can dynamically evolve at the data, control and interpreter levels, allowing agents
to modify at run time, for example, the way they provide a service. Some simple
concrete experimentations illustrate the potential of the model.

Contents

3.1 Introduction

This chapter presents the STROBE model, an agent representation and communication model. The approach consists of integrating selected features from agent communication, language interpretation in applicative/functional programming and e-learning/human-learning into a unique, original and simple view that privileges interaction, yet including control. The model is inscribed within a global approach, defending a shift from the classical algorithmic (control based) view to problem solving in computing to an interaction-based view of Informatics, where artificial as well as human agents operate by communicating as well as by computing.

As section 1.3.2.3 explains, the main difference between DSG and PD is the emerging *conversational process* managed by agents in the first one: DSGS need strong open and dynamic communication models able to manage these conversational processes (C15[conversation]) which may occur between agents: AA-AA, AA-HA, and also HA-HA, as services may be exchanged by any kind of agents. This is the DSG characteristic the most addressed by this chapter. Of course addressing C15[conversation] make us address also C17[agent]. STROBE is both an agent representation and communication model because real interaction centred models need to uniform these two aspects as they intrinsically depend one on another. (i) STROBE is an agent communication model because the main components, the ones structuring the agent's knowledge and capabilities, are designed for interaction; (ii) STROBE is an agent representation model because it defines a set of components that composes an agent and specifies the interaction between all these components. The aim of the model is to put at the centre of the agent architecture the conversation contexts in which it interprets messages from its interlocutors. A STROBE agent develops a communication language for each interlocutor agent. This is done with the concept of *Cognitive Environment* (CE). We assume that a language is basically a pair consisting of: (i) a language expression evaluation mechanism and (ii) a memory to store this mechanism and abstractions constructed with the language. They are called *interpreter* and *environment*[1] respectively (as in the classic tradition of programming languages).

Dedicated conversation contexts. Each STROBE agent has a set of CEs that represents its knowledge and capabilities. Each CE is dedicated to an interlocutor (or group of interlocutors). Actually, these CEs play the role of interlocutor models because they are the conversation contexts in which STROBE agents evaluate messages. The term 'conversation context' is inspired from the term 'execution context' that exists in traditional programming languages and means the context in which an expression of a language (i.e., a program) is evaluated to produce some results. In the STROBE model, agents are able to interpret messages in a given environment, with a given interpreter both dedicated to the sender of these messages. Communication between agents enables to dynamically change these environments and these interpreters in order to change their way of interpreting messages. Basically, CEs may be viewed as contexts where messages of conversation are interpreted to produce results and answer messages (C11[message]). The integration of such programming language constructs in the setting of agent communication is proved correct because they leverage the way agents communicate and adapt during conversations (C7[programming]). One of the key ideas of the STROBE model, to address the question of DSG, is to have these CEs dedicated. It means that a STROBE agent can have different representations or capabilities for each agent it communicates with.

A learning-by-being-told approach. Another important characteristic addressed by the STROBE model is the fact that DSGS should support learning and knowledge sharing (C4[learning]). Learning and knowledge communication are very important in MAS, they provide evolution and adaptation

---

[1]The term environment is here used with its programming language meaning, i.e., a dynamic data structure that binds variables and values. It does not means the world surrounding an agent as the term generally used in MAS community. This meaning is used by default in the rest of the thesis.

of these agent societies through time by automatically inferring knowledge from previous knowledge and/or from communication. Actually, the STROBE model was strongly influenced by three domains: (i) the Intelligent Tutoring Systems (ITS) or AI in Education (AIED), with its historical weight as a major source of inspirations for advances in computing and human learning; (ii) MAS particularly concerned with agent communication; (iii) applicative/functional programming as a source of powerful and simple features used for a long time in AI, especially by means of language interpretation, reflection and meta-evaluation. The goal of communication is certainly to change the interlocutor's state. This change is done after evaluating (3rd domain) new elements brought by the communication (2nd domain) in order for agents to learn (1st domain).

Compared with the three types of learning distinguished in section 1.4.3.3, the STROBE model describes extensively just an instance of the second type, learning-by-being-told, but it fits with the two others types. STROBE agents learn-by-being-told by other agents, and change their representations according to their interlocutors' information given by messages. Therefore, the interaction centred aspect of the model is inspired by the e-learning/ITS community, assuming that modelling and enabling better agent interactions implies AA-learning, which in turn helps to improve HA-learning (e-learning), and vice versa. Learning, for any kind of agents, depends on interactions [Cer94].

We also explain the potential of the model enabling several stateful services to share the same resource. Here again, some elements of answer to C22[separation] are brought by applicative/functional programming constructs in which an environment is basically a stateful context in which several functions or procedures are executed. The STROBE model integration with GRID presented in chapter 5 develops this question. Some characteristics are also indirectly addressed such as C2[interaction], C3[non-determinism], C5[reasoning], and C6[human].

> ### McCarthy inspirations and Wegner's view compliance
>
> Some concepts of the model may be recognized in McCarthy predictions described in [McC89]:
>
> - *A language should be interaction centred and there is no significant difference between interacting with AA or HA. It should be speech acts oriented and may consider the illocutionary act especially the perlocutionary act.*
>
> - *Agents are autonomous and they may change their goals during conversations. They should also take the initiative.*
>
> - *Agents do not require data structures if they can refer directly to the past. The memory model has to be improved to allow memorisation of all the values of a variable instead of simply forget them by traditional assignments.*
>
> In the same way, our approach fits with Wegner's view [Weg97, WG03, GW04] towards Interactive Computation:
>
> - *Computational Problem: performing a task or providing a service, rather than algorithmically producing an answer to a question.*
>
> - *Dynamic Streams: input and output are modelled by dynamic streams which are interleaved.*
>
> - *Environments: the world, the environment, of the computation is part of the model.*
>
> - *Concurrency: computation is concurrent and the computing agent computes in parallel with its environment and with other agents that may be in it.*
>
> - *Non-computability: the environment can not be assumed to be static or effectively computable; for example it may include humans.*

Agent communication requirements. From our analysis of agent communication questions, in section 2.3.3, we extract two requirements that an agent communication model should provide, with respect to which the STROBE model is compliant:

- To consider the communication effects on interlocutors we must consider that agents can change their goal or beliefs while communicating. Thus, they must be autonomous and should adapt during communication. This adaptation is made through language learning. Indeed, agents need a language to communicate. Thus, assuming as we do, that agents share a minimal common language, what is important is to allow language enrichment;

- To tend towards dialogue modelling, agents should only deal with messages from interlocutors without being knowledgeable of their internal beliefs. They should be able to handle the entire conversation dynamically and not simply interpret messages one-by-one following a previously determined structure.

This chapter explains how STROBE is compliant with the first requirement. The next one, presents a conversation modelling abstraction, called i-dialogue, which is a significant first step towards the implementation of the second one.

Chapter overview. The remainder of the chapter is organized as follows: Section 3.2 explains the influence of applicative/functional programming languages and in particular Scheme, on the model constructs. We present the three abstraction levels model (data, control, interpreter) also found in STROBE agents. We detail the principle of the Read-Eval-Print loop of these languages and the analogy with the STROBE agent behaviour. The basic notions about execution context and dynamic modification of an interpreter are also explained. Section 3.3 presents the STROBE model, from its first ideas, suggested in the 90s', to the formalization of the way agents are represented and communicate (sets of CEs). STROBE agent behaviour is also detailed as well as the relationships between our model and the classical other architectures found in MAS research (in particular, we explain why the STROBE model does not prevent to use a BDI architecture and interaction protocols even if it is not developed in this perspective). Section 3.4 presents some experimentations realized with STROBE agents. They are examples of scenarios and aim to show the potentiality (in particular meta-level learning and dynamic specification) of the model. Finally, section 3.5 gives some perspectives about future enhancements of the model.

3.2 The influence of applicative/functional programming languages

This section makes some technical recalls about interpretation and levels of abstraction in computing languages. We do not placed this section in appendix, because it is very important for the rest of the chapter.

The art of programming consists in writing programs whose goal is to solve problems. To solve problems, we need tools that are not more complicated to use than the problem is to solve. The Scheme language is one of these tools which explains why it is used in AI for a long time. Scheme is a Lisp like language proposed by researchers and teachers of the Massachusetts Institute of Technology at the end of the 70's. It is a functional[2] and applicative[3] language with a more academic than industrial intent. Scheme is based on lambda calculus, hence it offers a clear semantics [McC60, SJ75, KCe98]. Scheme is used all around the world as a language to teach students concepts of programming. A large number of course book exists [Que96, FF96, Cha96], however, many ideas of this thesis were generated

---

[2]A language that treats computation as the evaluation of mathematical functions. Scheme is not a pure functional language (such as Haskell for instance) as the programmer can use assignment (with the `set!` special form).

[3]A language that uses an *applicative-order* evaluation i.e., evaluates procedure call by evaluating all the arguments and then apply the procedure (opposed to *normal-order* evaluation).

by the reading and exercising of Abelson & Sussman's famous book *Structure and Interpretation of Computer Programs* [ASS96] which simply and nicely illustrates many of very powerful programming principles and techniques using Scheme. Scheme is often used in AI or in other domains to illustrate by simple examples powerful abstractions and mechanisms. For example, [MRG98] shows a set of advanced programming concepts such as first-class continuations, macros, delay and state on a quite simple Scheme example. These are the reasons why we were inspired by Scheme in the development of the STROBE model.

STReams, OBjects and Environments are the three programming constructs that support the model implementation. These are Scheme first-class primitives. Actually, Scheme is used in STROBE as a description language (such as lambda calculus or denotational semantics) as well as an implementation language (appendix C). However, the model is implementation independent. We should better say that the way of thinking of the applicative/functional community have lead our reflection for proposing the STROBE model. For example, Scheme allows representing procedures as data (*s-expression*) and vice versa, easily enabling to write programs which manipulate other programs (i.e., interpreters or compilers). This aspect facilitated our work on meta-evaluation when experimenting STROBE model's features.

3.2.1   Data, control and interpreter levels of abstraction

As already told, the bi-directional interactions we are interested in are direct messages sent from an agent to another one, and interpreted asynchronously (C11[message]). These interactions are called communications. In order to communicate, agents need a communication language understood by all the interlocutors. Therefore the problem of the common and shared language appears. We rather weaken this constraint, assuming just a kernel common communication language to be adopted and propose to enrich a very basic communication language at run time (dynamically). The STROBE model helps agents to build their own languages while communicating.

Humans learn *facts*, *rules* (or procedures or skills), and *languages* necessary to understand messages stating facts or rules, as well as necessary for generating behaviour when applying a particular rule. In the same way, in computing, we may discern the following levels:

- The *data* level consists of all the data abstractions constructed with a language and which aims to represent or store an information;

- The *control* level consists of all procedural abstractions constructed with a language and which aims to represent a computation mechanism or an algorithm;

- The *interpreter* level, also called the *meta-level*, consists of the mechanism, generally called an *interpreter* or *evaluator*,[4] that gives to an expression (related to the data and control level) a specific sense.

These three abstraction levels (or learning levels) may be found in all programming languages.[5] Every language provides an abstraction tool. For example, the `define` special form in Scheme. This abstraction tool allows to enrich the language. Abstraction at the data and control levels are usually stored in a structure storing bindings between variable names and values. This structure is called an *environment*. Abstraction at the data level consists in assigning values to already existing variables, or defining new data from already existing data. For example, in Scheme, expression such as:

---

[4]Alternatively, the reader may find in literature the terms: interpreter/evaluator and interpretation/evaluation. In this thesis we generally use evaluator/evaluation when talking about expressions, and interpreter/interpretation when talking about messages.

[5]These three levels are clearly explained in [ASS96] Building Abstractions with Procedures (chapter 1), Building Abstractions with Data (chapter 2), Metalinguistic Abstraction (chapter 4).

```
(define weekend 3)
(set! weekend '(friday saturday sunday))
```

Abstraction at the control level consists in defining new functions or procedures abstracting on the existing ones. For example, in Scheme, expression such as:

```
(define square (lambda (x) (* x x)))
(define foo (lambda ...))
```

FIRST-CLASS CONSTRUCTS

Scheme provides in particular a powerful and simple memory model via first-class environments as well as a very flexible and dynamic control model via first-class procedures and first-class continuations. The notion of first-class procedure is very important because it allows program (i.e., complex procedures) to dynamically generate other programs. first-class objects can be named by variables, can be passed as argument to procedures, can be returned as results of procedures and can be included in (and retrieved from) data structures.[a] For example:

```
(define multiply-by-n
    (lambda (n) (lambda (x) (* n x))))
(define multiply-by-5 (multiply-by-n 5))
(multiply-by-5 3) ⇒ 15
```

Here, the `multiply-by-5` function is dynamically constructed by calling `multiply-by-n`. This mechanism can be implemented with other languages such as Java but it is very heavy due to the fact that it is not intrinsic to the language.[b] The first-class view of procedures enables the programmer to consider the delivery of a procedure similar to the delivery of a datum.

---

[a]The term 'higher-order' may also be sometime used (especially for function), but it does not include the first and fourth properties.

[b]In Java, objects are considered to be first-class but procedures/methods are not (e.g., they can not be returned by other procedures or passed as arguments).

However, the abstraction tool can usually reach only the first two levels. In order to abstract at the interpreter level, what we call *meta-level abstraction* or *meta-level learning*, the interpreter itself has to be modified. This feature is necessary in order, for instance, to add some special forms to the language (e.g., in Scheme, an applicative language, the special form `if` must be implemented in the interpreter itself (i.e., at the interpreter level), it can not be defined at the control level). Changing the interpreter level must be done as well, in order to change the way of interpreting expressions, for example, in order to shift from an applicative order evaluation to a normal order evaluation, or to a lazy-evaluation (section 1.4.3.5).

The two first levels can easily be reached dynamically during execution: new data and procedures are added to the language each time an abstraction is interpreted or an assignment done. The challenge is to allow interpreter level modifications at run time, in order to generate dynamic processes. When the interpreter level is dynamically changed, the rest of the computation is interpreted with the new interpreter.

Actually, we may say that these three levels help us to define a language as a environment-interpreter pair. The interpreter is the program or mechanism that gives a sense to an expression (evaluates) in a particular context. An expression of the language is an expression recognized by the interpreter. This context is called an environment; it is used as a memory to store abstractions constructed with the language. Therefore, to enrich the language means being able to abstract (learn) at the three cited levels. The STROBE model maps these three levels schema into agents, considering that they are composed of a set of pairs of cognitive environments and cognitive interpreters. Data and control levels learning are made by changing the cognitive environment and meta-level learning by changing the cognitive interpreter.

DEFINITION: LANGUAGE ENRICHMENT/LEARNING

The process of abstracting at the data, control and in particular at the interpreter level of a given language in order to enrich both the set of recognized expressions and change the way they are evaluated. Language enrichment is said to be dynamic when it is done during the evaluation of an expression of the language.

3.2.2 Execution context

Let us remind the reader that Scheme is an interpreted language, which means that the execution machine is only composed by an interpreter (or evaluator), instead of a compiler and (virtual) machine, as in languages such as Java or C++. An interpreter is an evaluation procedure which could be [ASS96, Que96]:

`(eval e)`	Evaluation of an expression (`e`) with the substitution model (without assignment);
`(eval e r)`	Evaluation of an expression (`e`) in a specific environment (`r`);
`(eval e r k)`	Evaluation of an expression (`e`) in a specific environment (`r`) with continuation (`k`);
`(eval e r ks kf)`	Evaluation of an expression (`e`) in a specific environment (`r`) with succeed (`ks`) and failure (`kf`) continuations (non-deterministic evaluation).

As in all computer languages an expression is evaluated in an *execution context* or *state of computation*. In Scheme this context consists of the expression to evaluate itself, `e`, the environment in which this expression should be evaluated, `r` (the environment is responsible for what usually is called the memory or the state), sometimes the continuation(s), `k` (i.e., the next expression(s) to evaluate with the result of this evaluation) and of course the interpreter itself (`eval`). STROBE's idea is to model conversation contexts by interpreting conversation messages as expressions in a specific execution context. Each conversation, and thus each interlocutor or group of interlocutors have a context dedicated to the evaluation of their messages. This idea illustrates that the interpretation of an agent communication message may benefit from the same advantage of the evaluation of an expression in applicative/functional programming.

DEFINITION: EXECUTION CONTEXT

The set of elements (expression, environment, interpreter, continuation) intervening in the evaluation of an expression. The value of an expression depends on the execution context.

3.2.3 Dynamic modification of an interpreter

Scheme is very interesting in order to reach, at run time, the interpreter level as it is suggested in the previous section. Scheme offers some devices to allow the dynamic modification of an interpreter, in the following ways:

1. Using a reflective interpreter with the mechanism of *reifying procedures* such as in [JF92]. Reifying procedures is a mechanism to access to an expression execution context. The main idea is that user's programs could have the same access rights as the interpreter itself. Owing to that, procedures implementing the interpreter can be accessed and modified by user's programs in the same way as the environment and the continuation. This property makes reifying procedures the ideal

tool to dynamically modify interpreters. The problem of this approach is that reflection is still hard to be implemented and very heavy in computing resources;

2. Using first-class interpreters such as those present in [IJF92];

3. Using two levels of evaluation in order to use the evaluation function (`eval`) which is part of the Scheme language and reachable as any other procedure. In Scheme, an environment is initialized with the `eval` procedure as a primitive (since R^{5}RS [KCe98]) i.e., the `eval` procedure is included in all initialized environments such as the function +, or the other primitives of the language. It enables the programmer to call the `eval` function to dynamically interpret by default (in the same environment in which the `eval` function is included) an expression.

We experimented solution 1 and 3 in the STROBE model. Notice, that doing this kind of interpreter level modification could be very difficult with other types of languages, especially compiled ones which have an execution machine that consists of a compiler and a virtual machine instead of a simple interpreter. These mechanisms are detailed in section C.1.2.

3.3 The STROBE model

3.3.1 The STROBE model primary ideas

The STROBE model first ideas were described in [Cer96b, Cer96a, Cer99b]. These first ideas are briefly summarized in this section. In the following sections, we will see how we improve, formalize, implement and experiment this model. STROBE shows how generic conversations may be described and implemented by means of *STReams* of messages to be exchanged by agents represented as *OBjects* exerting control by means of procedures (and continuations) and interpreting messages in multiple *Environments* . The Environment aspect is addressed by this chapter. The STReam aspect is addressed by chapter 4. The OBject aspect is more explicitly detailed in STROBE implementations in appendix C. These three primitives are supposed to be first-class objects giving to the model an important dynamic behaviour. They may be detailed as:

Agents as interpreters. While communicating, an agent executes a REPL (Read - Eval - Print - Listen) loop waiting for messages (L), selecting one (R), interpreting it (E) and sending an answer (P). This loop is quite similar to the cycle of a language interpreter which evaluates an expression.[6] This principle is important because it regards agents as autonomous entities whose behaviour and interactions are controlled by a concrete interpretation procedure. Notice that the choice of interpretation – as opposed to compilation and execution – is important in order to ensure the run time dynamic behaviour of agents. The role of the interpreter is both to interpret the message and its content so the communication language and the content language can be the same one. We call `evaluate`, the procedure which emphasizes the agent and interprets the message contents, and `evaluate-strobemsg`, the sub-procedure able to interpret messages. To sum up, the first primary idea of the STROBE model is to consider agents as (Scheme) interpreters.

Objects as procedures. Object-oriented programming is often adopted to program agents. Many agent's characteristics historically come from the object-oriented paradigm (message passing communication, encapsulation, etc.). Scheme object-oriented programming was explained for instance by Normark [Nor91]. He explains how to create object using procedures. Implementing objects as procedures

---

[6]The traditional cycle of interpreter is REP, the step L has not real sense as the interpreter does not do anything while it waits for expressions. On the other hand agents, when they do not communicate, use their time to accomplish what they are conceived for, so the step L has got a sense.

allows, in language such as Scheme, to manipulate objects as first-class entities. Therefore, another primary idea of the STROBE model is to represent agents by objects, and objects by procedure, enabling to consider agents as the result of another procedure, eventually dynamically evaluated.

Cognitive Environments.   The STROBE model was initially thought to address the question of representing agent's multi-points of views. How to address the fact that a concept has been defined in different ways by one and another interlocutor? The model proposes a solution based on the reconstruction of the interlocutor model using a dedicated structure for conversations: the *Cognitive Environment* (CE). The concept of cognitive environment was firstly presented in [Cer96b] and [Cer96a]. It explains that multiple environments are simultaneously available within the same agent. These CEs represent the agent's knowledge – for example, the value of a variable, the definition of a procedure – they embody the agent's KB, as they represent a part of the different languages known by an agent (data and control levels). Then CEs address the question of representation of each interlocutor or group of interlocutors. An interlocutor model enables an agent to reconstruct as much as possible the interlocutor's internal state. Interlocutor models allow to free an agent communication model of the assumption of mental state ACLs that mutual beliefs of agents are correct. The cognitive environment plays this role. Therefore, a third primary idea of the STROBE model is to endow agents with several first-class Scheme environments to interprets messages.

Messages as streams.   The flow of messages (inputs and outputs) of STROBE agents are represented by streams.[7]   As [ASS96] explains, streams allow to model systems and changes on them in terms of sequences that represent the time histories of the systems being modelled. They are an alternative approach to modelling state. Streams are a smart data structure that can be used to model agent conversations, as they allow to define recursive data structures. The latter can be used to represent sequences that are undetermined and infinitely long, such as conversation inputs and outputs. This feature of applicative/functional programming seems particularly adapted for modelling agent conversations as lazy evaluation is relevant to express the natural retarded aspect of interactions: an agent may delay the production of the next message until it interprets its interlocutor's reaction to its previous message. It allows to remove the necessity of global conversation planning (i.e., the classic interaction protocol) substituting it by history memorization and one-step, opportunistic planning. Therefore, a fourth primary idea of the STROBE model is to consider flows of messages as streams. Chapter 4 proposes an approach for conversation modelling in the STROBE model based on streams.

DEFINITION: STROBE MODEL

An agent representation and communication model inspired by constructs of functional/programing languages. STROBE agents behave like interpreters that evaluate streams of messages in multiple dedicated cognitive environments.

3.3.2   Cognitive environments as interlocutor models

[SH05] summarized perfectly our ideas when facing the problem of interlocutor model:

> 'How should one agent represent another? The agent should presume that unknown agents are like itself, and it should choose to represent them as it does itself (1). As it learns about them, it has only to encode any differences it discovers (2) (...) Here are some other advantages: An agent has a head start in constructing a model for a unknown or just-encountered agent (3); An agent has to manage only one kind of model and one kind of

---

[7]The concept of stream is detailed in chapter 4.

representation (4); The same inference mechanism used to reason about its own behaviour can reason about the behaviour of other agents (5).'

(4) is the case with STROBE agents which use CE to represent both their own knowledge and generic capabilities (in global CE) and their interlocutor models (local CEs). This aspect is specifically detailed in the next section. Section 3.3.4.3 explains how an agent instantiates a new CE, by copying or sharing another one, when it meets a new interlocutor for the first time. As a matter of fact, because of is instantiation mechanism a STROBE agent represents the new interlocutor as itself or as another interlocutor (1 and 3). (2) is achieved with communication, when an agent modifies the dedicated CE according to messages sent and received with the interlocutor this CE is dedicated to. (5) is detailed in section 3.3.4.4.

Cognitive interpreters in cognitive environments.   By analogy with execution contexts presented before, we significantly augment the concept of cognitive environments, by including in them *Cognitive Interpreters* (CI) allowing agents to have a complete language dedicated to interlocutors. These CI (i.e., `evaluate` procedures) are stored in the CE as any other procedures. The use of the terms 'cognitive' has to be explained:

- For cognitive environments, it means that the data and control levels can be modified by communication;

- For cognitive interpreters, it means that the interpreter level can be also modified by communication according to the techniques presented in section 3.2.3.

Enabling these cognitive structures to evolve with message interpretation is the STROBE implementation of the learning-by-being-told paradigm (section 1.4.3.3). This tends towards an achievement of a part of C4[learning].

STROBE agents interpret communication messages in a given CE that includes a given CI, both dedicated to the current conversation. We illustrate in section 3.4.1 how communication enables dynamic changes of values in a CE and especially how these CIs can dynamically adapt their way of interpreting messages (meta-level learning by communication). Allowing learning at the three levels enables STROBE agents to dynamically realize the language enrichment of their dedicated languages. The concept of CE[8] may be seen as a dedicated conversation context (by analogy with execution context) an agent maintains while communicating with a specific interlocutor or group of interlocutors.

The advantage of seeing STROBE agents as interpreters is that they can process programs (i.e., services) for other agents, with a complete dedicated execution context. Moreover, they can exchange their interpreters as any other simple procedures. We may refer here to an original idea expressed in [ASS96], which becomes possible for STROBE agents:

> 'If we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications.'

Some of the inspirations of the STROBE model come from the actor model [Agh86]. Actors are objects exchanging asynchronous messages and able to transform dynamically their own behaviour. In order to address the question of having two different behaviours, actors are able to split into two. Similarly, STROBE agents instantiate a new CE to address the new behaviour.

---

[8]In the rest of the manuscript, we generally simply use the term CE to refer to the CE-CI pair (because CI are included in CE).

DEFINITION: COGNITIVE ENVIRONMENT

A structure which binds variables and values together. It includes a cognitive interpreter. The pair represents an agent's interlocutor model. STROBE agents use dedicated cognitive environments as conversation contexts to interpret messages (execution contexts) and to develop a different language for each of their interlocutors.

Two types of cognitive environments. Actually, a STROBE agent has two types of CEs:

- One *global CE*, unique and private, which represents the agent own beliefs and contains and executes generic capabilities;

- Several *local CEs*, dedicated to a specific interlocutor or group of interlocutors. They represent both the model of the interlocutor (i.e., others beliefs and dedicated capabilities) and conversation contexts (i.e., a part of the agent state dedicated to others) as explained above. Each time an agent receives a message, it selects the unique corresponding CE dedicated to the message sender in order to interpret the message.

3.3.3 Structure of the STROBE model elements

The structure of the STROBE model elements is given by figures 3.1, 3.3 and 3.4.

3.3.3.1 STROBE agent's structure

Figure 3.1 shows a STROBE agent representation. Generally, an agent possesses both intelligent and functional abilities. These are represented respectively by the agent *head* and *body*. STROBE agents are represented by a skittle as it is often done in the MAS community. The cylinder represents the body of the agent while the sphere represents the head. STROBE agents' head contains a *brain* and STROBE agents' body contains a set of CEs as dedicated conversation contexts, as presented before. Let A be the set of agents (a), B, the set of brains (b), and E, the set of cognitive environments (e). Then, an agent is composed of a brain and a non empty set of cognitive environments:[9]

$$A = \{(b, E_a), b \in B, E_a \in \mathcal{P}(E) - \{\emptyset\}\}$$

On figure 3.1 an agent, has a set of cognitive environments, represented by folders, and a brain, represented by a shapeless form. The set of CEs is not empty, because an agent unavoidably has its own global CE, and eventually other local CEs corresponding to the conversation contexts dedicated to its interlocutors.
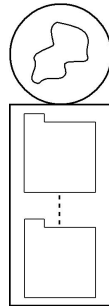

Figure 3.1: STROBE agent's structure

NOTATIONS

A local CE is said to be *totally* dedicated if the set of agents it is dedicated to is a singleton. If not, a local CE is said to be *partially* dedicated. A convention is adopted for this CE notation (table 3.1).[a]

Table 3.1: Convention adopted for CE notations

A's global CE	E_A^A
A's local CE totally dedicated to B	E_B^A
A's local E partially dedicated to C, D and E	$E_{C,D,E}^A$

Figure 3.2 shows three different STROBE agents A, B and C, with different kinds of CEs. Each agent has a global CE. A and C have a local CE dedicated to each interlocutor, whereas B has only one local CE dedicated to both A and C.
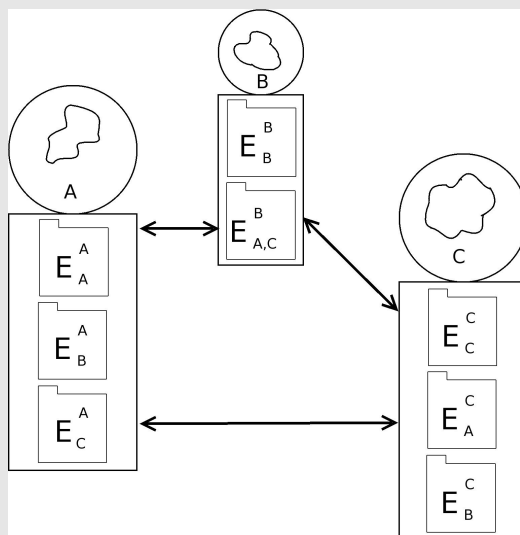

Figure 3.2: Three STROBE agents communicating one another

[a]Notice this convention stays the same with other dedicated elements. Notice also that this convention is not used in the rule expressions.

The *owning* relation formalizes the relation between agents, CEs and brains. Any CE and any brain are owned by exactly one agent. Any agent owns exactly one brain and at least one CE (the global CE).

$$owning : E \cup B \to A \text{ (surjection)}$$

The *dedicating* relation formalizes also the relation between agents and CEs. Any CE is dedicated to exactly one subset (maybe singleton but non empty) of agents. Any subset of agents dedicates zero, one or several CEs.

$$dedicating : E \to \mathcal{P}(A) - \{\emptyset\} \text{ (application)}$$

Rule 1 *An agent owns all the CEs and the brain that form it.*
$$\forall a = (b, E_a) \in A, \forall e \in E_a, owning(e) = a \wedge owning(b) = a$$

Rule 2 *An agent owns a unique brain.*
$$\forall a = (b, E_a) \in A, \forall b' \in B, owning(b') = a \Rightarrow b = b'$$

Rule 3 *An agent owns at least one CE, the global one. This global CE is unique and totally dedicated to this agent.*
$$\forall a = (b, E_a) \in A, \exists! e \in E_a, owning(e) = a \wedge dedicating(e) = \{a\}$$

The STROBE model presupposes an agent owns one unique local CE for a given interlocutor, i.e., interlocutor messages are always interpreted in the same CE (partially or totally dedicated). It is preferable for DSG to own one and only one conversation context for an interlocutor. This is expressed by rule 4.

Rule 4 *The intersection of the set of agents for which two CEs, owning to the same agent, are dedicated to, is empty.*
$$\forall e_1, e_2 \in E, owning(e_1) = owning(e_2) \Rightarrow dedicating(e_1) \cap dedicating(e_2) = \{\emptyset\}$$

3.3.3.2 Brain's structure

The brain contains a set of rules and algorithms (e.g., machine learning) that gives to the agent learning and reasoning skills. It also contains the rest of the agent's knowledge, its objectives, and eventually mental states. STROBE agent's brain is composed of modules. These modules depends on the architecture of the agent (BDI, etc.) as explained in section 2.3.2. It could be composed for example of a *message box* module, which is used to store messages before interpreting them, a *dynamic scheduler* module, which corresponds to the policy of choice of messages in the message box,[10] a *behaviour* module, corresponding to the role of the agent i.e., its intrinsic purpose, a *learning rule* module to change environments in a global way (section 3.3.4.4), an *i-dialogue* module, to model conversation with streams (chapter 4), etc. As brains of agents are never the same, we graphically represent them by a shapeless form as illustrated in figure 3.3.

From a certain point of view, the brain modules allow us to keep STROBE agents open to other initiatives; they represent everything agents are able to do (cf. the MAS literature) but we do not directly address with the STROBE model.

---

[9]In set theory, $\mathcal{P}(S)$ represents the power set of S i.e., the set of all subsets of S.

[10]This idea proposed originally by [GB99] and [Cer99b] allows an agent to choose the next message to process without interpreting it, only using message metadata (i.e., communication level and message level).
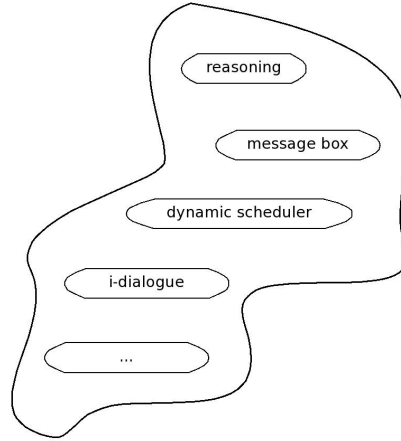
Figure 3.3: STROBE agent's brain structure

3.3.3.3 Cognitive environment's structure

A cognitive environment is a set of bindings. As in the pure programming language approach, a binding is an association between a variable and a value. Variables of CE are not statically typed,[11] and value can be of any types: (i) simple ones e.g., string, integer, boolean, pair, list; (ii) complex ones e.g., primitive, procedure, stream, thunk,[12] etc. We adopt the same writing convention for denoting CE elements than the one adopted for CE. In particular, each CE, E_Y^X, contains at least:

- a *cognitive interpreter*, INT_Y^X, which is a procedure type binding:

$$\lambda : EXP \rightarrow \texttt{evaluate}_Y^X(EXP, E_Y^X)$$

  It evaluates an expression EXP, in the corresponding environment, E_Y^X. EXP is an expression of a language formed by the pair (E_Y^X, INT_Y^X), i.e., a message or the content of a message.

- an *input stream*, I_Y^X, which is a stream type binding corresponding to the stream of messages send by Y to X;

- an *output stream*, O_Y^X, which is a stream type binding corresponding to the stream of messages to be sent by X to Y;

The rest of the bindings is sometimes denoted $BIND_Y^X$. It is not represented in figure 3.4 as we use this notation only to distinguish what is not INT_Y^X, O_Y^X, I_Y^X. Bindings in a CE are represented by rectangles with a fold back upper right corner.

In particular, we call *capabilities* procedure type bindings. They correspond to the agent's capacities or abilities to do something, i.e., to perform some task. The will represent the ability of an agent to provide a service. STROBE agent's capabilities are said to be executed in a specific dedicated CE i.e., an agent has specific capabilities for each agent it communicates with. The fact of having dedicated CE and thus dedicated capabilities, is for us the key element to implement DSG. Let Λ be the set of capabilities (λ). The *executing* relation formalizes the relation between capabilities and CEs. Any capability is executed in exactly one CE. Any CE executes at least one capability (the CI included).

---

[11] Actually, this is not a final choice. It depends of the communication language (section 3.3.4.2). We used Scheme, a dynamic typing based language, for the experimentations presented after. Advantages of dynamic typing was briefly described in section 1.4.3.5.

[12] A 'thunk' is an expression non evaluated yet. It contains the information required to produce the value of the expression when it is needed. Thunks are for example used with lazy evaluation.
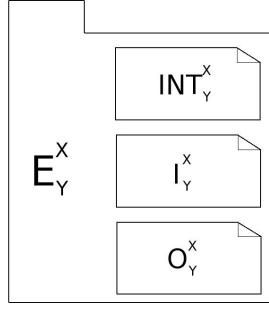
Figure 3.4: STROBE agent's cognitive environment structure

$$executing : \Lambda \to E \text{ (surjection)}$$

Each CE also contains a stream of inputs and a stream of outputs as data structures which store received and sent messages. Each CI consumes and produces messages in the streams of the same CE it is included in. An agent may have a *message box* module, as a dispatch function adding each new message an agent receives in the corresponding input stream. It may also be provided by send-messages function which role is to consume each output stream and send the message to the right interlocutor.

We call `produce(S,M)` the procedure that returns a new stream by adding to the stream S the message M (akin to `push`). We call `consume(S,M)` the procedure that returns a message by removing it from the stream S (akin to `pop`).[13] An agent can only consume its input streams and produce its output streams.

Figure 3.5 illustrates a possible configuration for figure 3.2 agent A's body. We can note bindings of the different CEs are not the same.
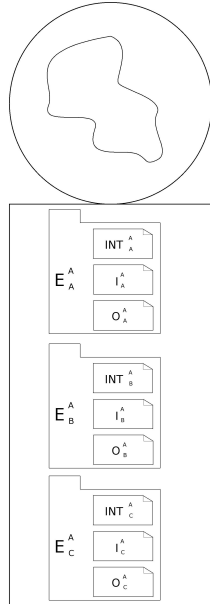
$$E_A^A = \begin{cases} INT_A^A = \lambda : EXP \to evaluate_A^A(EXP, E_A^A) \\ \\ BIND_A^A = \{(u,3)(v,5)(square, \lambda : x \to x*x)\ldots\} \end{cases}$$

$$E_B^A = \begin{cases} INT_B^A = \lambda : EXP \to evaluate_B^A(EXP, E_B^A) \\ \\ BIND_B^A = \{(u,3)(v,5)(carre, \lambda : x \to x*x)\ldots\} \end{cases}$$

$$E_C^A = \begin{cases} INT_C^A = \lambda : EXP \to evaluate_C^A(EXP, E_C^A) \\ \\ BIND_C^A = \{(u,6)(v,2)(id, \lambda : x \to x)\ldots\} \end{cases}$$

Figure 3.5: Example of the STROBE agent A's body

---

[13]These procedures are not simply `push` and `pop` procedures because streams are more complex than FIFO (First In First Out) or FILO (First In Last Out) data structures.

3.3.3.4 Human agent metaphor

This agent representation unifies the artificial agent and human agent representations. HAs are, of course, autonomous, intelligent and interactive; we can easily consider that they have a set of capabilities as well as dedicated contexts for conversations. Each HA has a set of interlocutors and different representations of these ones. He/she develops different languages for each of them. Other HAs do not know what he/she really thinks, they can just deduce facts from messages they received. Each time a HA interacts with another one he/she changes his/her representations and mental states. HAs are able to infer, or deduce knowledge from what they know or learn. Of course, this is a simple (and restrictive) HAs representation. Considering that agents should interact with other agents (AA or HA) following the same principles, is an important aspect in modelling interaction [McC89]. The agent communication model should be generic with respect to the types of the communicating agents. This one of the purpose of the STROBE model: what is important is the agent representation of each of its interlocutors i.e., the interlocutor model.

3.3.4 STROBE agent behaviour

3.3.4.1 REPL loop of interaction

We can now illustrate the STROBE model REPL loop more precisely as figure 3.6 shows. Each time an agent *read*s a message, it chooses both the dedicated CE and CI to *eval* it (i.e., applies the selected interpreter on the message in the selected environment). Then it *print*s the corresponding answer (i.e., sends the answer message(s) and/or processes the evaluation result), and *listen*s for next messages.

Agent X REPL loop

> *LISTEN*
> > - Try to realize its objectives;
> > - Reason on its knowledge base;
> > - Choose the next conversation to follow (e.g., with Y);
>
> *READ*
> > **if E_Y^X *does not exist* then**
> > > - Instantiate a dedicated CE, E_Y^X;
> >
> > - Select the next message to interpret: $M_r = \texttt{consume}(IN_Y^X)$;
> > - Choose the CE (E_Y^X) and CI (INT_Y^X) dedicated to Y;
>
> *EVAL*
> > - Interpret the message: $M_s = \texttt{evaluate}_Y^X(M, E_Y^X)$;
>
> *PRINT*
> > - Send to Y the answer message(s): $\texttt{produce}(OUT_Y^X, M_s)$;
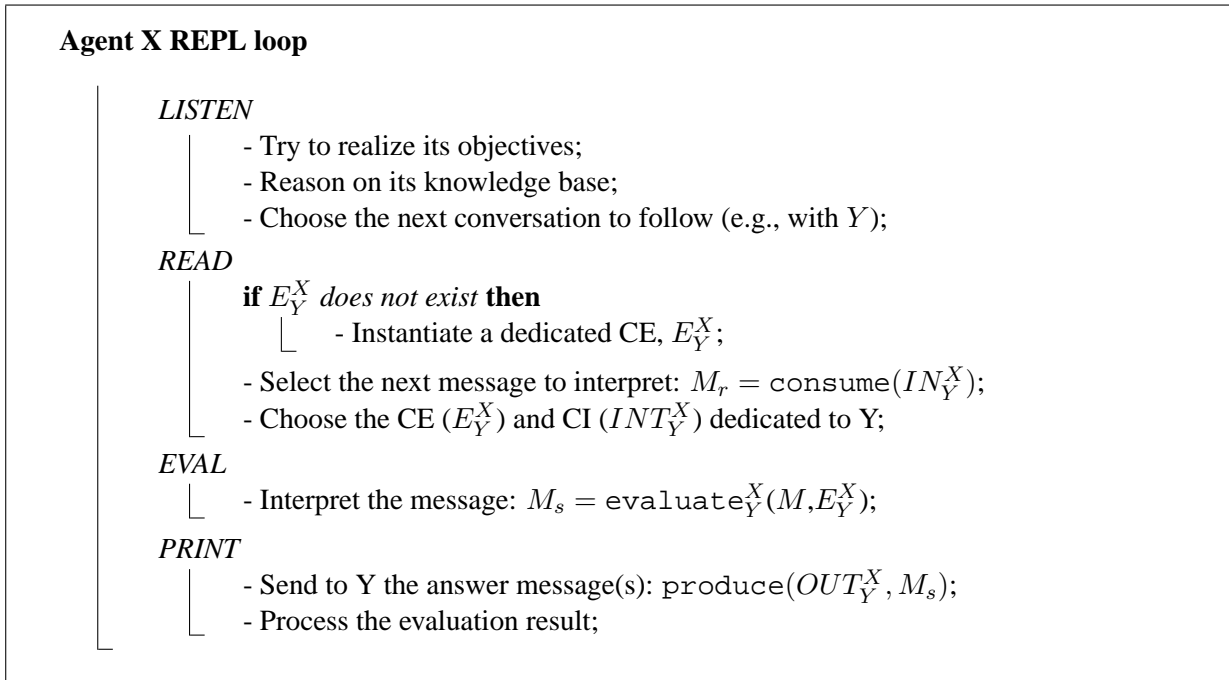> > - Process the evaluation result;

Figure 3.6: STROBE model's REPL loop

Therefore, as figure 3.6 shows, a STROBE agent models a conversation by recursively applying:

$$\texttt{produce}(O_Y^X, \texttt{evaluate}_Y^X(\texttt{consume}(I_Y^X), E_Y^X))$$

A conversation depends on three parameters (an input stream, an output stream and a CE including a CI). Chapter 4 introduces a recursive function, called *i-dialogue*, which models, for a given agent, intertwined-dialogues by taking in parameters a set of input streams, set of output streams, and a set of CEs.

When two agents send and receive messages one another, we say that they communicate or interact. The *interacting* relation formalizes the relation of interaction between agents. Any agent interacts with zero, one or several agents.

$$interacting : A \rightarrow A \text{ (relation)}$$

In the following, $\forall a \in A$, we denote $interacting(a)$ the set of agents which interact with a.

Rule 5 *The interacting relation is symmetric.*

$$\forall a_1, a_2 \in A, a_1 \in interacting(a_2) \Leftrightarrow a_2 \in interacting(a_1)$$

Rule 6 *The interacting relation is irreflexive i.e., an agent does not interact with itself.*

$$\forall a \in A, a \notin interacting(a)$$

Rule 7 *Two interacting agents have each one a unique CE dedicated (partially or totally) to the other.*

$$\forall a_1 = (b_1, E_1), a_2 = (b_2, E_2) \in A, a_2 \in interacting(a_1) \Leftrightarrow \exists! e_1 \in E_1, \exists! e_2 \in E_2, a_2 \in dedicating(e_1) \wedge a_1 \in dedicating(e_2)$$

Chapter 5 specializes this relation in order to formalize the special case of service exchange (section 5.3.1).

We can now detail the procedure `evaluate` (figure 3.7) which embodies an agent's CI. If the expression to interpret is a message then it calls `evaluate-strobemsg` (detailed after), else it calls `evaluate-exp` procedure which is a classical evaluation procedure (such as `eval` from [ASS96], or `evaluate` from [Que96] or [JF92]) which dispatches on the expression type (definition, quotation, condition, combination, etc.) to evaluate it. See also section C.1 for details about evaluation procedures.

Data: EXP, E_Y^X
Result: value or message(s)
if EXP *is not a message* **then**
     `evaluate-exp`$_Y^X(EXP, E_Y^X)$;
else
     `evaluate-strobemsg`$_Y^X(EXP, E_Y^X)$;
end

Figure 3.7: Agent X's `evaluate`$_Y^X(EXP, E_Y^X)$

3.3.4.2 Communication language

In agent communication, it is common practice to separate the intention of a message from its content, by both abstracting and classifying communicative acts (speech act theory) by performatives and structuring messages with parameters (section 2.3.3.1). This section details the communication language used by

STROBE agents (i.e., message structure and performatives used). It is considered minimal because we made the choice, in the STROBE model, to give priority to learning methods rather than presupposed common knowledge. The STROBE message structure is inspired by the KQML and FIPA-ACL message structure but do not use all the parameters presented in table 2.2. The communication level is limited to `:sender` and `:receiver`; the message level is limited to `:performative`, and the content level to `:content`. The performative parameter is very important as it brings the main part of the message semantics. We do not consider other FIPA-ACL parameters (e.g., `:ontology`, `:in-reply-to`, `:reply-with`, `:protocol`, etc.) for four reasons:

- In a sake of simplicity;

- The `:language`, `:encoding`, `:ontology`, `:parameters` may be considered as a part of bindings of the CE in which messages are interpreted. Labelling a message with the explicit language in which the message content is expressed, is equivalent in the STROBE model, to make the agent interprets the message by using an environment for the interpretation in which a frame binds the language symbols to their specific meaning. In the same nature, an ontology may be viewed as a specific frame of an environment binding the concept, their relations, and the rules with their specific meaning even if it is less trivial (ontologies are more than simple variable-value associations). This frame needs to be common to both interlocutor agents. Therefore, CEs play in STROBE a role similar to ontology in ACLs messages: they are contexts in which expression receive meanings;

- The STROBE model first objective is not to model agent conversations with the interaction protocol approach (for which `:reply-to`, `:protocol`, `:reply-with`, etc. parameters are needed);

- Other parameters may eventually be added on the fly as the `evaluate-strobemsg` procedure is part of the interpreter (meta-level learning such as it is presented in section 3.4.1).

Therefore, the STROBE message structure is:

$$MSG = \{AGENT_S, AGENT_R, PERFORM, CONTENT\}$$

$$\text{with } PERFORM = \{assertion, ack, request, answer, order, executed\}$$

The sender ($AGENT_S$) and the receiver ($AGENT_R$), are the two agents concerned by the message. The $CONTENT$ of the message is an expression written in a content language, for example the Scheme expression `(+ 2 (* 3 5) 4)`. STROBE uses the same language both for the message and its content (i.e., Scheme) as agents can use the same interpreter to interpret the message and evaluate its content.[14] In order to construct a minimal language, STROBE agents process six performatives by default. A large number of performatives is not necessary needed to have a rich communication language. Generally, the minimal set of performatives is {`inform`, `request`} such as in [Sho93] and [CP79]. For example, FIPA-ACL considers four primitive performatives (`inform`, `request`, `confirm`, `disconfirm`). All other performative formal models can be expressed by these four ones [Fip02b]. These six performatives (three for questioning and three for answering) are:

assertion messages modify the interlocutor's behaviour or some of its representations (i.e., bindings in the dedicated CE). Answers are **ack** (acknowledgement) messages reporting a success or an error. For example, in Scheme, the content of assertion messages are `define` or `set!` expressions. These messages are used to perform classical data/control levels communication learning

---

[14]Note that the choice of Scheme allows to consider also a minimal content language as Scheme define a very few fundamental primitives (`define`, `set!`, `lambda`, `quote`, `if`) which are sufficient to re-implement nearly all the language.

request messages ask for one interlocutor's representation, such as the value of a variable or the closure of a function. Their **answer**s return a value or an error. In Scheme, request messages are symbol names `x, y, square`

order messages require the interlocutor to apply a procedure (i.e., execute a capability). This interlocutor sends the result as the content of an **executed** message. In Scheme, it would correspond to a procedure call, `(foo ...)`. These messages are used when an agent achieves a task for another one.

The performatives were also chosen following the STROBE communication philosophy, inspired by Scheme [Cer99b]. They correspond to the different semantics of Scheme expression (request, assertion and order). Using a Scheme interpreter we do not need yet other performatives. For example, these three main performatives are also chosen as the communication language of the components that specify an agent in the *act*SMART approach [ALd05] (`inform`, `request` and `execute`). Table 3.2 illustrates a set of three message exchanges with the six kinds of performatives.

Table 3.2: Examples of message between a teacher agent and a learner agent

TEACHER (A_T)	LEARNER (A_L)
$\{A_T, A_L, request, $ `square` $\}$	$\{A_L, A_T, answer, $ `undefined` $\}$
$\{A_T, A_L, assertion, $ `(define (square x) (* x x))` $\}$	$\{A_L, A_T, ack, $ `(*.*)` $\}$
$\{A_T, A_L, order, $ `(square 3)` $\}$	$\{A_L, A_T, executed, $ `9` $\}$

As we see in the experimentation, section 3.4.1, the list of known performatives is not closed. Surely, due to the fact that performatives are processed by `evaluate-strobemsg` procedure, which is a sub-procedure of `evaluate`, it is possible to dynamically modify it to process new performatives. For example, the broadcast performative, as illustrated in the experimentation.

broadcast messages consist in sending a message with a pair as content (`perform, content`) its semantics is that the interlocutor must send a message with the performative `perform` and with the content `content` to all its current interlocutors. There is no answer defined for the broadcast messages.

We now can detail the `evaluate-strobemsg` procedure (figure 3.8).

Note that for each performative the interpretation of the content is performed. Actually, it could seem useless to add a performative to STROBE messages because in all cases the message content is often evaluated and the evaluation result becomes the answer message content. But it is not. The information brought by the performative seems redundant: (i) an expression without brackets is always a request; (ii) a `define` or `set!` expression is always an assertion; (iii) all other expressions are orders. It is the same information, but not expressed at the same level. The performative is a meta-information about the message given to the agent before it evaluates the message content. Different performatives may be used to distinguish pragmatic effect of messages, for instance, to distinguish if an agent is going to learn at the data/control levels (assertion message) or simply answer a question (request message) or perform a task for an interlocutor (order message). This is an important aspect of ACLs [LF97, KP01, Eij02]. The STROBE model respects this aspect, even if it is true that in the presented experimentations, this meta-information is not really used yet.

Data: MSG=$(A_Y, A_X, PERFORM, CONTENT), E_Y^X$
Result: value or message(s)
switch *PERFORM* **do**

    case *assertion*
        | `produce`$(O_Y^X, A_X, A_Y, ack,$ `evaluate`$_Y^X(CONTENT, E_Y^X))$;

    case *request*
        | `produce`$(O_Y^X, A_X, A_Y, answer,$ `evaluate`$_Y^X(CONTENT, E_Y^X))$;

    case *order*
        | `produce`$(O_Y^X, A_X, A_Y, executed,$ `evaluate`$_Y^X(CONTENT, E_Y^X))$;

    otherwise
        | `analyse-answer`$_Y^X(CONTENT)$;

end

Figure 3.8: Agent X's `evaluate-strobemsg`$_Y^X(EXP, E_Y^X)$

3.3.4.3 New cognitive environment instantiation

In MAS, when an agent has a conversation, it can dedicate a part of its state to this conversation. It is called the conversation state or context. In the STROBE model, this is played by the CE. When an agent receives a message from another agent for the first time, it instantiates a new local CE dedicated to this agent following three different policies.

1. by *copying its own global CE*, e.g., when two agents meet together for the very first time, and when they do not have any idea of the type of the new interlocutor (e.g., HA, AA, service provider, etc.);

2. *by copying one of its local CEs*, e.g., when another agent finishes a conversation that a first one has begun or when an agent is mandated by another one;

3. by *sharing one of its local CEs*, e.g., when an identical agent, or an agent playing the same role, or belonging to the same group, replaces another one. The CE, dedicated to a group of interlocutors, is used for modelling multi-party conversations. All messages coming from this group of interlocutors are interpreted in this CE.[15]

Figure 3.9 illustrates these three cases. It represents the STROBE agent A's body when it meets agent C for the first time. The new STROBE agent's body is illustrated according to the three cases enumerated just before.

In AGIL, the instantiation relation between CEs is coupled with the service instantiation mechanism of GRID. It is described in section 5.3.2.

3.3.4.4 Learning and reasoning on cognitive environments

A KB is a set of sentences expressed in a knowledge representation language. Learning and reasoning are the processes of constructing new sentences from existing ones in an agent's KB (machine learning

---

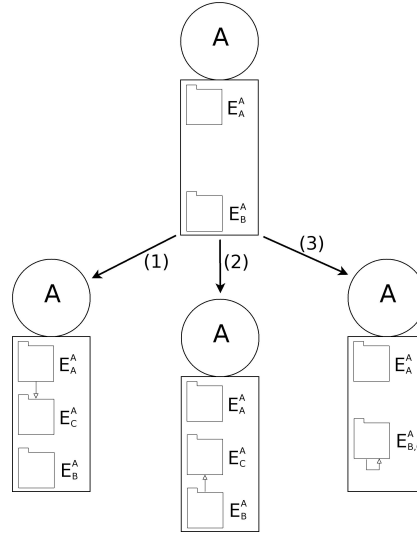[15]In this case, none new CE instance is created.

Figure 3.9: Cognitive environment instantiation policies

in section 1.4.3.3). STROBE agent's KB is composed of two parts: the brain part and the body part. The brain part depends on the modules the STROBE agent implements. The body part is composed of the set of CEs that belongs to an agent. Even if the STROBE model is mainly influenced by the learning-by-being-told paradigm, i.e., CE changes (learning) is done according to messages 'told' by interlocutors, other types of learning may also be realized. This is important in order to address as much as possible C4[learning].

Notice that the global CE is private and changes only for precise reasons. However, the reader could have noticed in figure 3.5, that E_C^A, E_B^A and E_A^A contain different bindings. Which value $AGENT_A$ must use for u or v in order to make E_A^A evolves? How to infer that the square procedure of $BIND_A^A$ and carre of $BIND_B^A$ are the same? Actually, On its KB, a STROBE agent may have two kinds of learning/reasoning processes:

- *Local reasoning processes* consist of learning on a particular CE (local or global). That means for example deducing a new proposition from other propositions, such as in the *modus ponens*, if $A \rightarrow B$, and A, then B. Or detecting if several bindings of a CE are the same. These kinds of machine learning processes can be applied on a data structure which binds variables with values, such as the one used for CEs. For example, in Scheme, algorithms can detect if two functions produce the same result, or if they have the same body (e.g., detect that `(square (lambda (x) (* x x)))` and `(carre (lambda (y) (* y y)))` are the same functions on figure 3.5).

- *Global reasoning processes* consist of learning on all CE in order to change the global one. Indeed, STROBE agents may transfer knowledge they learned in the different local CEs to the global one. They have to do that in order to take into account what they have learned in a global way. It is part of their evolution. They also have to do that if they want to reuse, with other agents, knowledge learned with one of them. Then an agent should have a global CE *evolution policy* (or a *learning rule* module), which emphasizes its autonomy, desire and intentions. For example, such a policy may be: 95% of my interlocutors teach me that x=5 and 5% teach me that x=3, then I decide that for me (i.e., in my global CE) x will be 5. Or, as another example, 95% of my interlocutors teach me that x=5 but my boss teaches me that x=3, then I decide that for me x=3. The second example shows that an agent global CE evolution policy may use metadata on agent interlocutors. Notice that global reasoning processes can be similar to local reasoning processes e.g., detecting that two functions in two different local CEs are the same.

These kinds of reasoning processes may be implemented with Lisp based knowledge modelling language such as for example Operational Conceptual Modelling Language (OCML) [Mot99].

3.3.4.5 Agent reproduction

One of the hypothesis of the STROBE model, in order to enable customized and adapted services, is to consider that an agent has one unique local CE totally or partially dedicated to another agent it is interacting with. This is expressed by rule 4. To respect this hypothesis agents must reproduce themselves if they want to dedicate more than one local CE to an interlocutor. For example, if a service user agent wants two instances of the same service, the service provider agent must reproduce itself. The *reproducing* relation formalizes this relation between agents. Any agent is reproduced by at most one parent agent (i.e., not all agents are result of reproduction). Any agent reproduces in zero, one or several children agents.[16]

$$reproducing : A \rightarrow A \ \text{(function)}$$

Rule 8 *An agent can have a unique parent.*

$$\forall a_1, a_2, a_3 \in A, reproducing(a_1) = a_2 \wedge reproducing(a_1) = a_3 \Rightarrow a_2 = a_3$$

With this rule an agent can have zero or one parent. Indeed, some agents are not the result of the *reproducing* relation. They are created directly by the designer of a GRID-MAS integrated system.

Rule 9 *The reproducing relation is irreflexive i.e., an agent is not reproduced by itself.*

$$\forall a \in A, reproducing(a) \neq a$$

Rule 10 *The reproducing relation is asymmetric i.e., an agent that reproduces another one, cannot be reproduced by this one.*

$$\forall a_1, a_2 \in A, reproducing(a_1) = a_2 \Rightarrow reproducing(a_2) \neq a_1$$

When an agent reproduces, it produces a new agent that is exactly the same than itself except considering the set of local CEs (same brain and same global CE). The child agent is created with a local CE dedicated to the interlocutor of the father agent which provokes the reproduction; none of the father local CEs are inherited. Local CEs represent the model of an interlocutor (or group of interlocutor) an agent has. They are the property of agents and cannot be shared. By opposition, the global CE represent the generic knowledge of an agent. Therefore, it seems logic than the global CE should be copied in a child agent, but not the local CEs. The new agent will develop its own local CEs, thanks to his global CE, according to the interaction it has in the future. Indeed, since the child agent is created, it starts evolving alone autonomously: changes its brain, changes its global CE, exchange some services, etc. The father agent cannot instantiate CE in the child body anymore. This situation is illustrated in figure 3.10.

The reproduction of STROBE agents is also detailed in section 5.3.3 at the light of the GRID-MAS integration. Chapter 5 fixes a rigourous graphical notations for each of these concepts and relations.

---

[16]The relations help us to represent a situation for a time 't'. Therefore, when an agent has reproduced in another one, we consider that the two agents are definitively in relation. This is not the caee for all relations. The use of the forms 'ing' and 'ed' is just used to express a relation and the reciprocal.
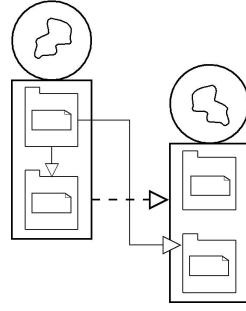
Figure 3.10: Example of agent reproduction

3.3.5   Related work on agent representation and agent communication

The same concept of putting the communication contexts at the centre of the agent architecture in which it interprets messages appears also in [AGP03], which assumes that each agent in service exchange may separately maintain its own internal conversation state. Having a distinct component of the structure devoted to interaction was suggested by [Had95, Wit92, Fer92, VDB97]. As we explained in section 2.3.2 these architectures were more often based on mental state representations. Mentalist architectures are strongly bounded to ACLs and interaction protocols.

The STROBE model does not prevent the use of mentalist approaches and interaction protocols in agent communication. For instance, concerning conversation modelling: STROBE messages interpretation is done with a dedicated procedure, `evaluate-strobemsg`. An agent may be constructed with (or may learn) a `evaluate-strobemsg` which takes into account some interaction protocols or conversation policies. Thus, the `evaluate-strobemsg` procedure becomes as the figure 3.11 shows and the messages structure becomes:

$$MSG = \{AGENT_S, AGENT_R, PERFORM, PROTOCOL, CONTENT\}$$

$$\text{with } PROTOCOL = \{contract - net, introduction, coop - learning, \ldots\}$$

A STROBE agent can adapt (some of) its CEs to fit these conversation modelling approaches. However, these approaches are strongly bounded to mental states architectures [DG00]. For example, agents that envision to use FIPA-ACL in a semantically coherent way are required to adhere to a BDI-style architecture. STROBE should also be able to deal with this kind of architecture. Subsequently, data structures as environments allow doing it. Booleans true and false, and basis modal operators are part of the content language. Moreover, other modal operators such as B, D, I (Belief, Desire, Intention) from [RG91] or B, U, C (Belief, Uncertainty, Choice) from [CL90] (e.g., $B_I p = $ 'agent I believes that p is true') can be added. Processing on these logic expressions, is of course also possible, it is the domain of logic programming [Rob83] and it can be done with languages such as PROLOG or Scheme as presented in chapter 4 of [ASS96]. This last example shows how to write a query system (i.e., a logic expression evaluator). STROBE agent's interpreters can become such query systems.

Concerning the sincerity condition evoked in section 2.3.3.1, we should say that, a STROBE agent does not know what another agent thinks, it just knows what the other agent knows of what it knows as far as knowledge has been communicated.

3.4   Experimentations: examples of scenarios

In order to express the potentiality of the model, this section suggests two examples of scenario experimented with the STROBE model. They illustrate: (i) *meta-level learning* by communicating i.e., agents which modify their interpreters while communicating; (ii) the *dynamic specification* of a problem by

Data: MSG=$(A_Y, A_X, PERFORM, PROTOCOL, CONTENT), E_Y^X$
Result: value or messages(s)
switch *PROTOCOL* **do**
    case *contract-net*
        switch *PERFORM* **do**
            case *call*
                $\ldots$
            case *proposal*
                $\ldots$
            case *decision*
                $\ldots$
            case *contract*
                $\ldots$
        end
    case *introduction*
        $\ldots$
    case *coop-learning*
        $\ldots$
    otherwise
        switch *PERFORM* **do**
            case *assertion*
                $\texttt{produce}(O_Y^X, A_X, A_Y, ack, \texttt{evaluate}_Y^X(CONTENT, E_Y^X));$
            case $\ldots$
                $\ldots$
        end
end

Figure 3.11: Agent X's $\texttt{evaluate-strobemsg}_Y^X$(MSG,$E_Y^X$) with interaction protocols

means of communication. Even if these experimentations are 'toy examples', they are significant as they represent solutions for large classes of problems.

3.4.1 Meta-level learning by communicating

3.4.1.1 Learning how to process a new performative

The first experimentation shows how an agent can learn-by-being-told a new performative, thus dynamically modifies its message interpretation procedure (i.e., `evaluate-strobemsg` which is part of the interpreter). It is a standard 'teacher-learner' dialogue. An agent teacher (A_T) asks to an agent learner (A_L) to broadcast a message to all its interlocutors using a special performative, `broadcast`. However, A_L does not initially know the performative used by A_T. So, A_T transmits three messages (`assertion` to define it the way of interpreting broadcast messages and `order` to invite A_L to change its interpreter) clarifying to A_L the way of processing this new performative. Finally, A_T formulates once more its request to A_L and obtains satisfaction. After the last message process, `evaluate-strobemsg`$_T^S$ procedure is modified. Thus a part of its cognitive interpreter (INT_T^S) was dynamically changed and the corresponding definition in its dedicated CE (E_T^S) is transformed. It is thus a meta-level learning. Meta-level learning modifies the way a STROBE agent sees things by changing its own interpreter during communicating. At the end of the dialogue, A_L can process `broadcast` messages coming from A_T. Table 3.3 describes the set of messages exchanges occurring in the experimentation (using the third mechanism of section 3.2.3. We invite the reader to refer to [JC03] and specifically to [Jon03] for more details about this experimentation using reifying procedures.

    Remark – Note that the new `evaluate-strobemsg` procedure integrates the new code for evaluating broadcast messages, but the previous code is not modified as it is saved and called by `old-eval-strobemsg`. The term integration is important, because A_L does not simply replace its procedure by the A_T one, it integrates this code to its previous procedure yet conserving as valid the previous modifications on it. This is a constructivist method.

3.4.1.2 Learning how to process a logic operator

Another important task in learning is to test learner knowledge, asking good questions to learner in order to determine its knowledge or capability of doing a task. This 'toy-example' illustrates this point by testing the learner meta-level knowledge. The experimentation still concerns a teacher agent and a learner agent. A_T wants to test the way A_L processes the `and` logic operator. Logic operators `and` and `or` have to be processed in a particular way. Indeed, there is no need to evaluate the other clauses if: the first clause of an `and` is false or if the first clause of an `or` is true. The interpreter can immediately return false for the `and` expression in the first case, and true for the `or` expression in the second case. `and` and `or` expressions are said to be 'lazily evaluated' as for example the `if` special form. In Scheme, to process these operators in a lazy way, they have to be defined as special forms, and not as classical procedures.[17] It means that these operators are defined at the interpreter level, thus to add these special forms to a language, the interpreter of the language itself has to be modified . In order to test A_L way of processing `and`, A_T uses a global variable, `counter`, incremented each time the interpreter evaluates an expression such as `(begin (set!  counter (+ 1 counter))` *predicate*`)` which returns the value (true or false) of *predicate*. Table 3.4 describes the set of message exchanges occurring in the experimentation.

    Remark – In this experimentation it is the `evaluate-exp` procedure, sub-procedure of `evaluate` which has been modified. So comparing to the first experimentation, it is another part of the interpreter but always a meta-level learning.

---

[17]You can also do it using continuation passing style programming.

Table 3.3: Teacher-learner dialogue for `broadcast` performative learning

TEACHER (A_T)	LEARNER (A_L)
Here is the definition of the square procedure: $\{A_T, A_L, assertion, $`(define (square x) (* x x))`$\}$	*Ok, I know now this procedure:* $\{A_L, A_T, ack, $`(*.*)`$\}$
Broadcast to all your current interlocutors: $\{A_T, A_L, broadcast, $`'(order (square 3))`$\}$	*Sorry, I don't know this performative:* $\{A_L, A_T, answer,$ `'(unknown broadcast)`$\}$

Save your `evaluate-strobemsg` *procedure:*
$\{A_T, A_L, assertion,$
   `(define old-eval-strobemsg`
                  `evaluate-strobemsg)`$\}$
 	
Here is the way to process broadcast:
$\{A_T, A_L, assertion,$
   `(define (evaluate-broadcast msg) ...)`$\}$
 	
Then, you can change your `evaluate-strobemsg` *procedure:*
$\{A_T, A_L, order, $`(set!  evaluate-strobemsg`
   `(lambda (msg)`
     `(if (eq?  (performative?  msg) 'broacast)`
       `(evaluate-broadcast msg)`
       `(old-eval-stobemsg msg))))`$\}$
	Ok, I have added this new binding:
	$\{A_L, A_T, ack, $`(*.*)`$\}$
	
	Ok, I have added this new binding:
	$\{A_L, A_T, ack, $`(*.*)`$\}$
	
	Ok, I have changed my procedure:
	$\{A_L, A_T, executed, $`(*.*)`$\}$

Broadcast to all your current correspondents: $\{A_T, A_L, broadcast, $`'(order (square 3))`$\}$	*Ok, I broadcast* $\{A_L, ..., order, $`(square 3)`$\}$

The meaning of the '...' corresponds to the body of the `evaluate-broadcast` *function. Basically, the algorithm consists to produce a new message for each of the interlocutor of the agent receiving the broadcast message. In the example this new message use the* order *performative and has* `(square 3)` *for content.*

3.4.2 Enabling dynamic specification by communication

This section illustrates another aspect of the model: non-deterministic interpretation to enable the dynamic specification of a capability during a conversation between two STROBE agents. As suggested before (section 3.3.2) this aspect will be implemented by using a different type of interpreter: non-deterministic one.

Non-deterministic interpreter. The key idea is that expressions, in a non-deterministic language, can have more than one possible value. With non-deterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Actually, programs have different possible execution histories and the non-deterministic interpreter frees the programmer from the details of how choices are made. Non-deterministic interpretation is very useful for constraint satisfaction programming, but moreover AI in general relies heavily on non-determinism. For this experimentation, we use a very simple Scheme non-deterministic interpreter inspired from the one proposed in chapter 4 of [ASS96]. This interpreter has the signature (`eval e r ks kf`), as presented in section 3.2.2. It allows the simplest (but least efficient in complexity) method for implementing non-deterministic

Table 3.4: Teacher-learner dialogue for lazy and operator learning

TEACHER (A_T)	LEARNER (A_L)
Here is the definition of the counter variable: $\{A_T, A_L, assertion, $ `(define counter 0)` $\}$	*Ok, I know now this variable:* $\{A_L, A_T, ack, $ `(*.*)` $\}$

Please, evaluate the above expression:
 $\{A_T, A_L, order,$
 `(and (begin (set! counter (+ 1 counter))`
 `        (> 2 3))`
 `    (begin (set! counter (+ 1 counter))`
 `        (= 0 0)))`$\}$
	Ok, I have executed, here is the result:
	 $\{A_L, A_T, executed, $ `#f` $\}$

What is the value of `counter`*?:* $\{A_T, A_L, request, $ `counter` $\}$	*Ok, here is the value of* `counter`*:* $\{A_L, A_T, answer, 2\}$

Save your `evaluate` *procedure:*
 $\{A_T, A_L, assertion, $ `(define old-eval evaluate-exp)` $\}$
 	
 Here is the way to process and *in a lazy way:*
 $\{A_T, A_L, assertion, $ `(define (evaluate-and exp) ...)` $\}$
 	
 Then, you can change your `evaluate` *procedure:*
 $\{A_T, A_L, order, $ `(set! evaluate-exp`
 `  (lambda (exp)`
 `    (if (and-exp? exp) (evaluate-and exp)`
 `       (old-eval exp))))`$\}$
	Ok, I have added this new binding:
	 $\{A_L, A_T, ack, $ `(*.*)` $\}$
	
	 Ok, I have added this new binding:
	 $\{A_L, A_T, ack, $ `(*.*)` $\}$
	
	 Ok, I have changed my `evaluate` *procedure:*
	 $\{A_L, A_T, executed, $ `(*.*)` $\}$

Please, evaluate the above expression:
 $\{A_T, A_L, order,$
 `(and (begin (set! counter (+ 1 counter))`
 `        (> 2 3))`
 `    (begin (set! counter (+ 1 counter))`
 `        (= 0 0)))`$\}$
	Ok, I have executed, here is the result:
	 $\{A_L, A_T, executed, $ `#f` $\}$

What is the value of `counter`*?:* $\{A_T, A_L, request, $ `counter` $\}$	*Ok, Here is the value of* `counter`*:* $\{A_L, A_T, answer, 3\}$

interpreters: backtracking.[18] Some details about non-deterministic evaluation with Scheme are given in section C.1.3.

STROBE agent's CIs could be non-deterministic interpreters. It is interesting for STROBE agents for instance to solve constraint based programs. But above all, the most interesting thing is that agents can progressively build such programs by adding or removing constraints while communicating with other agents and then apply these programs to generate a solution for another agent. The constraints, defining a non-deterministic program, can be determined progressively with the conversation using STROBE's features to dynamically change procedures and the way of interpreting them. Actually, CSP formalizes a problem with: X the set of variables, D the restricted domains of these variables, and C the set of constraints on these variables [Dec03]. In the following we consider that the elements X and D are determined by the service provider and let the service user express C.

An e-commerce scenario. Let us consider, a standard e-commerce dialogue for a train ticket booking. A ticket is characterized by a departure city, a destination city, a price, and a date. An SNCF[19] agent dialogues with a customer agent which tries to book a ticket. A realistic conversation could be:

a. Customer: *I want a ticket from Montpellier to Paris*

b. SNCF: *Ok, what are your conditions?*

c. Customer: *Tomorrow before 10AM. Please, give me a proposition for a ticket!*

d. SNCF: *Ok, train 34, departure tomorrow 9.30AM, from Montpellier to Paris, 110€*

e. Customer: *Is it possible to pay less than 100€?*

f. SNCF: *Ok, train 31, departure tomorrow 8.40AM, from Montpellier to Paris, 95€*

g. Customer: *Another proposition please?*

h. SNCF *Ok, train 32, departure tomorrow 9.15AM, from Montpellier to Paris, 98€*

i. Customer: *Ok, I accept this one*

Notice that the messages **a**, **b**, **c** and **e** deal with the constraints on ticket selection. The messages **d**, **f** and **h** are the result of application of a ticket search procedure, with various constraints. The message **g** corresponds to a request of the customer to get another answer, that means to explore another branch of the solution tree. Table 3.5 illustrates how this dialogue can be represented into Scheme expressions in order to be realized by our agents.

In this experimentation, STROBE agent's CIs (`evaluate` procedures) are able to recognize the expressions `amb`, `require` and `try-again` introduced in section C.1.3. Notice that the modification of the REPL loop of STROBE agents is not necessary: the `try-again` symbol is replaced by the `(try-again)` message, which provokes another interpretation of the previous message. Before the messages of table 3.5, A_S send to A_C a set of variables with their domains (noted `*variable-set*` in table 3.5). In the rest of the experimentation A_C does not express values for these variables but expresses constraints (with the `require` function) that take in parameter these variables. Each time A_S receives a new constraint, it integrates it in its `find-ticket` function, changing the set of possible values returned by the function.

The idea of this experimentation is interesting because it is the conversation that builds the service to be carried out and not the opposite. It is a typical scenario that could be found in many e-commerce applications or other ones of the same kind where agents must build a program to find a solution together. The classical approach of program construction (which can be found in traditional software engineering)

---

[18]Nowadays, CSP algorithms are not based on retrospective methods (e.g., backtracks, backjumping) anymore but on prospective methods (e.g., look-ahead, arc-consistence, propagation) much more efficient in term of complexity.

[19]The acronym of the French railway company: Société Nationale des Chemins de Fer.

Table 3.5: Scheme expressions in dialogue between the SNCF agent and the customer agent

CUSTOMER (A_C)	SNCF (A_S)

I want a ticket from Montpellier to Paris
`(require`
`  (eq?  depart 'montpellier))`
`(require`
`  (eq?  dest 'paris))`
	Definition of a new `find-ticket` function:
	`(define (find-ticket)`
	`  (let ((depart (amb *city-set*))`
	`        (dest (amb *city-set*))`
	`        (price (amb *price-set*))`
	`        (date (amb *date-set*)))`
	`    (require (not (eq?  depart dest)))`
	`    (require (eq?  depart 'montpellier))`
	`    (require (eq?  dest 'paris))`
	`    (list (list 'depart depart)`
	`          (list 'dest dest)`
	`          (list 'price price)`
	`          (list 'date date))))`
	
	Ok, what are your conditions?

Tomorrow before 10AM
 	
`(require (< date`
`*tomorrow10AM*))`
Please, give me a proposition for a ticket!
 	
`(find-ticket)`
	`find-ticket` fonction modification adding a new constraint. Then procedure execution:
	
	Ok,train 34, departure tomorrow 9.30AM, from Montpellier to Paris, 110€
	`((depart montpellier) (dest paris) (price 110) (date *tomorrow9.30AM*))`

Is it possible to pay less than 100€?
`(require (< prix 100))`
 	
`(find-ticket)`
	idem
	Ok, train 31, departure tomorrow 8.40AM, from Montpellier to Paris, 95€
	`((depart montpellier) (dest paris) (price 95) (date *tomorrow8.41AM*))`

Another proposition please? `(try-again)`	`find-ticket` function execution: *Ok, train 32, departure tomorrow 9.15AM, from Montpellier to Paris, 98€* `((depart montpellier) (dest paris) (price 98) (date *tomorrow9.15AM*))`
Ok, I accept this one	

The customer agent (A_C), which represents the service user, transmits its requests with `require` *and* `try-again` *expressions. At the beginning of the conversation, the SNCF agent (A_S), which represents the service provider, starts a new* `find-ticket` *procedure construction which is dynamically modified and built progressively during the conversation. These modifications consist in changing a value in the SNCF agent's local CE dedicated to the customer agent, E_C^S.*

that specifies the problem before coding it, is changed into a *dynamic specification* approach during coding. That is to say, specification and realization are achieved at the same time.

Constraints on figure 3.5 are pretty simple, but with such a system, more complex constraints are possible (tested during experimentation). For instance, conjunction and disjunction of predicates as: *'I want to leave between the 10th and the 12th of this month'* can be expressed by the following Scheme expression:

```
(require (and (<= *10-of-this-month* date)
              (>= *12-of-this-month* date)))
```

Other dependencies among ticket attributes such as: *'I want to leave from Montpellier if the price is lower than 15€, otherwise I want to leave from Nîmes'*, which can be expressed by the following Scheme expression:

```
(if (not (and (< price 15) (eq? depart 'montpellier)))
    (require (eq? depart 'nimes)))
```

Therefore, this approach enables the user to explore a large possibility in its own constraint definition. In fact, in this experimentation A_S does not want to know values of variables proprieties (i.e., departure city, price, etc.), it simply exposes to A_C the variables and domains on which it can express constraints, and lets it communicate its restrictions. The travel agency example is often used in the Web research community, but seldom associated to a dynamic interaction with the user as our approach presents. This experimentation is a first step towards DSG because instead of asking values to user, it asks constraints on values, which is substantially different from the user's point of view.

This non-deterministic execution of agents' capabilities tends towards C3[non-determinism]. However, for example, we should say that dynamic specification may also bring a solution approach for C10[negotiation]. The contract defining the performance of A_S for A_C may be dynamically established during the service performance.

Experimentation comments. One difference of DSGS is that the user does not exactly and necessarily know what he wants (D1). This statement supports the fact that DSG support learning (C4[learning]). At the end of the process the user has certainly learned something: in the e-commerce experimentation, the customer, communicating with the SNCF agent, has simply found a solution to a ticket problem (he learned what is the train that he is going to take). Even if the e-commerce scenario is a very simple one, it may be easily extended to complex ones, and especially 'learning oriented' scenarios with complex knowledge creation.

D6 in section 1.3.3 mentions that DSG subsumes PD. In this experimentation, asking to the user constraints on variables 'includes' asking values on these variables, but not the opposite. For example, constraints expressed by the customer agent can be equalities (i.e., values) instead of inequalities for instance: `(require (= price 95))` or `(require (= date *tomorrow10AM*))`. In this case, DSG is downgraded to PD. It becomes equivalent to fill a Web form, what is a current practice in many Web sites.

The first experimentation shows how to add a new performative to the ones already known by an agent, thus how to modify an agent messages interpretation procedure (i.e., `evaluate-strobemsg`). The same principles can be used to modify any part of an agent interpreter. For instance, to add a special form to the recognized set of expressions. We can imagine a prelude to the second experimentation with a scenario where an agent teaches another one how to transform its interpreter into non-deterministic one.

Dynamic specification, meta-level learning and dedicated capabilities are for us the key features of the STROBE model to go towards on implementation of DSGS agents. These experimentations show that such a model is realizable (we experiment the first and the third one) and has an important potential.

3.5 The STROBE model tomorrow

In this section we presents some perspectives concerning the STROBE model.

3.5.1 Increasing control with dedicated continuations

Continuation is a fundamental notion in programming languages. When a program is interpreted, the continuation is the following move of the current process i.e., the next expression to evaluate with outputs from the current evaluation. Continuation passing style programming allows to dynamically manage the evaluation progress of programs and capture or change (by escape or resume) its future. In this sense, continuations should be very useful for DSG as they may enable an agent to operate on the following of a service generation according to results of the previous moves returned by the process. This would significantly contribute to the dynamicity of the process. In Scheme there are two approaches for doing continuation passing style programming:

- Use the `call/cc` primitive witch allows to reach the continuation. For example, `call/cc` allows writing a product function with escape when a multiplication by zero occurs;

- Use the continuation as part of the execution context itself (section 3.2.2). For example a Scheme non-deterministic interpreter uses two continuations: a succeed one (`ks`) and a failure one (`kf`).

Our interest concerns only the second way for the moment. Using continuation in the execution context means to give to the evaluator the next expression to evaluate with this result. That is why continuations are functions (then first-class object in Scheme) with an expression as parameter which corresponds to the result of evaluation of the precedent expression. For example, if we consider the environment `R` with the binding `(x 3)` then:

```
:(evaluate '(+ 1 x) *R* (lambda (v) (* v 11)))
> 44
```

In order to increase program execution dynamicity, it is interesting to have programs that can access and change the continuation of the current evaluation, in order to dynamically generate processes. Reaching the continuation in the execution context, can be done with the same approaches that the ones to reach interpreters, presented in section 3.2.3. Therefore, the STROBE model should regard CE as the embodiment of the complete execution context of a message and then integrate a *Cognitive Continuation* (CC).[20] As interpreter and other bindings this CC can also be dynamically modified, improving a STROBE agent way of adjusting its interlocutor models and as consequence increase the dynamicity of the generated processes. For example, we can imagine an agent global CE evolution policy which consists in interpreting messages of a special interlocutor in both its dedicated CE and in the global CE (e.g., if the interlocutor is the agent programmer). It can be done via the dedicated continuation coding that any message must be interpreted in the global CE after being interpreted in the local CE.

This perspective for the STROBE model gives to agents access to all the expression execution context. Then, a communication message would be interpreted by a dedicated interpreter in a dedicated environment, with a dedicated continuation. Therefore, the interpretation process would be totally controlled and dynamic.

---

[20]In fact, it is one or several continuations following the interpreter: `(evaluate e r k)` or `(evaluate e r ks kf)`. In the following, we simply consider one continuation.

3.5.2 Other perspectives

Streams to emulate state.   One idea that we would like to explore is the representation of environment bindings not by pair (such as `(var val)`) but by stream (such as `(var val1 val2 ...valn ...)`). As [ASS96] states, streams are an alternative approach to model state. Representing bindings with streams will allow us to keep all the history of values taken by a variable and thus realize McCarthy's prediction: 'Agents do not require data structures if they can refer directly to the past'. Assignment becomes a simple production of a new element of the stream. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment (side effects). For example, variable values often follow a sequential evolution as it is the case for a counter, which can be very easily expressed with a stream.

Learning rules.   To develop learning/reasoning rules such as the one mentioned in section 3.3.4.4 is an interesting perspective that could be very interesting to follow. To integrate in the STROBE model, under the form of brain modules, different agent learning/reasoning approaches is another one.

3.6   Conclusion

The main contributions of this chapter are:

- The proposal of a new agent representation and communication model;

- The specification of the concept of CE as a conversation context private to an agent and dedicated to an interlocutor (or group of interlocutors);

- The validation of the model with experimentations illustrating its adequateness and the feasibility of its prototypical applications.

We have thoroughly adopted the first requirement of the introduction: to allow language enrichment by enabling agents to learn at the data, control and interpreter levels at run time. STROBE agents are able to interpret communication messages in a given environment, including an interpreter, dedicated to the current conversation. We have shown how communication enables to dynamically change values in an environment and especially how these interpreters can dynamically adapt their way of interpreting messages. Then, we have shown how a STROBE agent can build, dynamically while communicating, a different language dedicated to each of its interlocutors.

As previously expressed, the STROBE model is the result of research in different domains and therefore has the advantage of gluing them together in a coherent way. Integration stays an important concern behind this thesis work. Re-usability of concrete applicative/functional programming solutions and their integration within agent and MAS approaches were the choices made. We have seen in this chapter that constructs of applicative/functional languages may importantly enhance existing agent architectures. In this sense, this chapter is important because it connects very high level theoretical considerations for future service exchange in Informatics with very low programming mechanisms known for a long time in Computer Science. The philosophy expressed through the STROBE model ideas is that it is not mandatory to use very complex mechanisms to realize complex behaviours. Assuming that the primitives as well as the rules of composition are simple and powerful. We have seen in this chapter that the model allows us to address some important DSG characteristics. Even if DSG seems today unrealizable completely, some characteristics may be already addressed by elegant existing mechanisms. The following chapter presents another aspect of our work on agent communication. We propose a computing abstraction likewise inspired by dynamic concepts of programming, to model conversations within the STROBE model. Chapter 5 presents our GRID-MAS integrated model and explains how STROBE particularly fits this integration without being indispensable.

Chapter 4

I-dialogue

Intertwined. adjective. *To be twined or twisted together;*
to become mutually involved or enfolded.

T HIS CHAPTER defines and exemplifies a new computational abstraction, called
i-dialogue, which aims to model communicative situations such as those where
an agent conducts multiple concurrent conversations with other agents. It takes
the form of a recursive function, producing and consuming streams of messages,
run by each agent in a conversation. The i-dialogue abstraction is inspired by the
dialogue abstraction proposed by [O'D85] to model process interactions and by
[Cer96b] which suggests to model agent communication by means of streams. In
this chapter, we extend the dialogue abstraction to more than two communicating
processes and consider it for agent communication. I-dialogue (intertwined dia-
logues) models conversations among agents by means of fundamental constructs
of applicative/functional languages. (i.e., streams, lazy evaluation and first-class
functions). Moreover, the i-dialogue abstraction fits perfectly the STROBE model.
The i-dialogue abstraction is adequate for representing intertwined dialogues, exe-
cuted simultaneously and for which inputs and outputs depend on each other, such
as those that can occur in service composition.

Contents

4.1 Introduction

Current computing systems are more and more distributed. As they are by definition dispersed, these systems often consist of interacting entities or processes. For that reason interaction modelling has always be an important concern, for simple processes in the 80's, as well as for today's complex MAS. In particular we have seen in section 2.3.3.2 that agent conversations are more often modelled by means of interaction protocols which represent a pattern of message exchange in a conversation. Using protocols, an agent interprets messages from a conversation one-by-one, changing at each step its own state, following the protocol to produce the next message in the conversation. One shortcoming that we have shown in this approach is that they force an agent to follow a pre-fixed structure and thus to have a predetermined behaviour. The only way for an agent to consider the entire conversation is to look at the protocol, which was previously determined (before the conversation) and which cannot change dynamically. Therefore, agents are obliged to fit fixed conversations while it should be conversations which fit dynamically changing agents. By contrast, dialogue modelling tries to explain that the next answer message of a conversation can not be foreseen, and should be determined only after the interpretation of the previous incoming message. The i-dialogue abstraction is inscribed in this second idea by providing a simple, yet powerful abstraction to model message exchanges in agent communication.

In this chapter we describe *i-dialogue*, an abstraction of the interaction between several processes inspired from O'Donnell's *dialogue* [O'D85]. The dialogue and i-dialogue combinators are first-class functions that structure two-agent and n-agent conversations as networks of communicating processes. These abstractions deal directly with different sequences of inputs, different sequences of outputs, and state of the agents concerned. These abstractions are useful to describe interaction between several agents, whatever the type of the agents, by dealing merely with the flow of exchanged messages. The kernel of the model is based on the assumption that messages are elements of input and output 'streams'.

An i-dialogue (intertwined dialogues) is an interactive session between n agents which take turns sending messages to each other. It is an n-agent conversation. Each agent has a *state* at a given time that contains personal information (internals) and information about the history of the conversation. Each agent models the conversation with the interlocutor(s) by executing an i-dialogue function. It means that for a n-agent conversation, n i-dialogue functions are run: one by each agent. During the conversation, each agent computes a new state and a new *output* from its previous state and the last *input* it received from other agents, using its *transition function*.

NOTATIONS

In order to express the dialogue and i-dialogue abstractions, we use a Daisy language [Joh89] inspired syntax to ensure continuity with [O'D85]. This syntax is briefly repeated in sidebar page 151. The rest of the notations used are:

- X, Y (uppercase letters) represent agents;

- x, y (lowercase letters) represent elements of sequences (i.e., messages);

- x_{Yi} is the ith message from X to Y;

- ξ, ψ (Greek letters) represent states of agents X, Y;

- f_Y^X is X's transition function dedicated to Y;

- R_X is X's function producing a result value from a state;

- I_Y^X is X's input sequence of messages from Y;

- O_Y^X is X's output sequence of messages to Y;

- then $O_Y^X = (x_{Y1}, x_{Y2}, \ldots, x_{Yn}) = I_X^Y$;

- and $O_X^Y = (y_{X1}, y_{X2}, \ldots, y_{Xn}) = I_Y^X$.

More specifically, [O'D85] shows how programming environments can be defined using dialogues. His popular 1985 paper gives an application, called RIE (Recursive Interactive Environment), in which the traditional REP (Read-Eval-Print) interaction loop of applicative/functional languages is implemented by the dialogue abstraction. The aim of O'Donnell's paper was to help programmers to develop useful and smart programming environments. The aim of this chapter is to help to model agent conversations with a simple, powerful and elegant abstraction: i-dialogue.

Part of the chapter is used to present a general implementation of the i-dialogue abstraction for functional/applicative languages such as Lisp or Scheme. This implementation is based on simple features of these languages: first-class functions (specifically function as argument), streams (as sequences of unevaluated elements that are potentially infinitely long) and lazy evaluation (to implement streams). We show how streams are a smart data structure to model agent conversations because they allow defining recursive data structures that can be used to represent sequences that are infinitely long such as the inputs and outputs of the dialogue and i-dialogue abstractions.

The chapter also connects this new abstraction with the STROBE model presented in the previous chapter, which suggests to model agent conversations by streams. In particular, the i-dialogue abstraction is particularly synergic with the STROBE model: the cognitive interpreters are the transitions functions which produce an answer message in the dedicated output stream according to both an incoming message in the dedicated input stream and CE bindings (section 4.3.3).

The i-dialogue abstraction has two main advantages: (i) it does not presuppose anything about the internals of the interlocutor agent and only deals with output streams of messages; (ii) it deals with the entire conversation (expressed by potentially infinite and not predetermined streams of messages) and not simply with a message alone. The i-dialogue abstraction addresses the second requirement identified in the introduction of chapter 3.

Therefore, the DSG characteristics addressed by this chapter are at a first time C15[conversation] and C16[dialogue], as the i-dialogue abstraction aims to model dynamic dialogue that must occur with several agents. In particular C16[dialogue] is a quite important characteristic as explained in section 2.3.3.2. Using streams in this perspective is the new aspect. In a second time, i-dialogue addresses also partially C12[process] as this abstraction models conversations an agent should have with other ones in order to provide a composite service, as it is illustrated in section 4.3.4 on a travel agency example.

A Scheme implementation of the i-dialogue abstraction has been developed and is detailed in section C.3.

Chapter overview. The remainder of the chapter is organized as follows: Section 4.2 recalls the principles of O'Donnell's dialogue abstraction and explains the concepts that are going to be extended further in the chapter. This part of the chapter is highly inspired from O'Donnell's paper. Before presenting the concepts of a general implementation of dialogue and i-dialogue, this section makes a little state of the art of the notions of streams and lazy evaluation. Then, the extension of dialogue, in section 4.3, presents the i-dialogue abstraction by generalizing it to the three agents case. Section 4.3.3 shows how to use i-dialogue in the STROBE model. We also show how the i-dialogue abstraction suits for a type of high level process of services and we give an example in section 4.3.4.

4.2 The dialogue abstraction

4.2.1 Description of dialogue

A dialogue is an interactive session between two agents, A and B, which take turns sending messages. The sequence of output messages from A to B is equal to the sequence of input from B to A ($O_B^A = I_A^B$) and vice and versa as figure 4.1 shows.

Each agent has a state that contains personal information (internals) and information about the history of the conversation (α, β). One of the agents (A) begins the dialogue by sending the first message to the
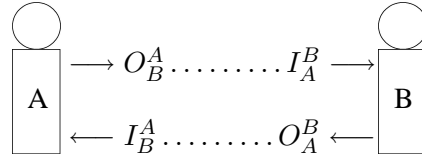
Figure 4.1: Streams in dialogue between two agents

other (B). Initially A is in state α_j, $j \geq 0$.[1] Each agent models the conversation with the interlocutor with the dialogue abstraction i.e., by running a dialogue function. During the conversation, each agent computes a new state and a new output from its previous state and the last input it received from the other agent, using its transition function (f_B^A for agent A and f_A^B for agent B):

$$f_B^A : \left[\alpha_{j+k}\ I_B^A \right] \rightarrow \left[\alpha_{j+k+1}\ O_B^A \right]$$

$$f_A^B : \left[\beta_k\ I_A^B \right] \rightarrow \left[\beta_{k+1}\ O_A^B \right]$$

Thus the transition functions f_B^A and f_A^B and the initial state α_j and β_0 define, for each agent, a sequence of states and a sequence of outputs. The four-tuple of following sequences specifies a complete history of the dialogue as figure 4.2 shows:

$$A: \quad (\alpha_j\ \alpha_{j+1}\ \ldots)\quad O_B^A = (a_{B1}\ a_{B2}\ \ldots a_{Bn}\ \ldots) = I_A^B$$

$$B: \quad (\beta_0\ \beta_1\ \ldots)\qquad O_A^B = (b_{A1}\ b_{A2}\ \ldots b_{An}\ \ldots) = I_B^A$$

Furthermore, each agent has a 'result function' (R_A and R_B) which it is used, if the agent decides to terminate the dialogue, to produce a final dialogue value according to their last state.

$$R_A : [\alpha_{j+k}] \rightarrow val \text{ and } R_B : [\beta_k] \rightarrow val$$

A dialogue between A and B, initiated by A, is defined to be a four-tuple of sequences. Advantages of this representation are quoted from [O'D85]:

> 'The advantage of this definition is that it doesn't rely on an explicit notion of time, input/output or side effects. The ordering of sequence elements corresponds to their relative order in time. Similarly, the definition of each sequence element as the value of a function applied to other sequence elements (except for the initially defined elements) replaces the notion of sending messages with explicit input/output. Finally, it is not necessary to view the state of a participant as a variable which must be changed through a side effect upon each communication. Instead of destroying old state values, we view state as a sequence of values.'

4.2.2 Dialogues model conversational processes

The dialogue abstraction is a possible model for conversational processes as they are presented in section 1.3.2.3. Indeed, a conversational process was presented as a sequence of message exchanges. Each answer message is produced by the interpretation of a received message in a context determined by the agent state. A message interpretation produces a new state and some elements. In dialogue, functions are the message interpretation procedures and the answer messages are the elements. Moreover, the value

---

[1]If agent A has just been created then $j = 0$, but if A has already had a dialogue with some other agent then $j \geq 0$. As agent B does not start the dialogue, we consider it as 'new', thus its beginning state is β_0.

Agent A   Agent B

$(initial)\alpha_j$ $\beta_0(initial)$
$(\text{send to B})a_{Bj} \Rightarrow$

$[\beta_1, b_{A0}] = f_A^B : [\beta_0, a_{Bj}]$
$\Leftarrow b_{A0}(\text{send to A})$

$[\alpha_{j+1}, a_{Bj+1}] = f_B^A : [\alpha_j, b_{A0}]$
$(\text{send to B})a_{Bj+1} \Rightarrow$

$\vdots$

$[\alpha_{j+k}, a_{Bj+k}] = f_B^A : [\alpha_{j+k-1}, b_{Ak-1}]$
$(\text{send to B})a_{Bj+k} \Rightarrow$

$[\beta_{k+1}, b_{Ak}] = f_A^B : [\beta_k, a_{Bj+k}]$
$\Leftarrow b_{Ak}(\text{send to A})$

Figure 4.2: Dialogue between two agents A and B

returned by the result function may represent what we call 'final result' in conversation processes. In DSG, agents are brought to have multi-party conversations, as for example when a service provider may use another service in order to provide its own service. The i-dialogue abstraction extends the dialogue abstraction, to modelling conversational processes as they may occur in such situations.

Until now we used the term 'sequence' to define inputs and outputs. The dialogue abstraction implementation must be able to operate on sequences that represent state and communications in which the elements are not evaluated before they are really needed. Indeed, an agent run a dialogue function when it starts a conversation. However, the parameters of this function (i.e., inputs and outputs sequences) are not produced yet since the dialogue is just starting. Therefore, agent runs a function that may not evaluate completely these sequence arguments. Those sequences are potentially infinite lists called streams.

Features of applicative/functional programming seem particularly adapted for modelling agent conversation. For example, lazy evaluation is relevant to express the natural retarded aspect of interactions: an agent may delay the production of the next message until it interprets its interlocutor's reaction to its previous message. As streams allow to define partially evaluated sequences of elements, we can use them to represent sequences of message that may be infinitely long such as the inputs and outputs of the dialogue and i-dialogue abstractions. Wegner et al. [WG99] explain that a computation is no longer executed in a closed world but in an open system where many inputs and outputs may be interleaved with one another without being all defined before the computation begins. This is exactly the case using streams in the dialogue and i-dialogue abstraction. Streams used in dialogue and especially in i-dialogue are even lazy lists as it is explained later.

DEFINITION: STREAM OF MESSAGES
A data structure which represents a potentially infinite sequence of messages consumed and produced by agents within a dialogue.

4.2.3 Implementation of the dialogue function

A two-agent conversation is modelled differently by each of the two agents, using the dialogue abstraction and local parameters. The dialogue abstraction can be implemented efficiently by a simple recursive function with the techniques of applicative/functional programming. This function has vocation to be

HISTORY OF STREAMS AND LAZY EVALUATION

- For [ASS96] a stream are sequences as lists are. The difference is the time at which the elements of the sequence are evaluated: list elements are evaluated when the list is constructed where as stream elements are evaluated only when they are accessed. The basic idea consists in constructing a stream only partially, and to pass the partial construction to the program that consumes the stream. Stream processing allows to model stateful systems without ever using assignment or mutable data. It is often easier to consider the sequence of values taken on by a variable in a program as a structure that can be manipulated, rather than considering the mechanisms that use, test, and change the variable. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment (side effects). The stream formulation is particularly elegant and convenient because the entire sequence of states is available as a data structure that can be manipulated with a uniform set of operations.

- For Burge [Bur75], a stream is a functional analog of a coroutine and may be considered to be a particular method of representing a list in which the creation of each list element is delayed until it is actually needed. He discusses the use of streams as a method for structured programming and introduces a set of functional stream primitives for this purpose.

- In [FFJ90] mechanism for the maintenance of streams based on continuations is presented.

- Stream processing is also presented in [IT83] where streams are represented by pair of functions (enumerator and selector).

Stream implementation. For the applicative/functional language community a stream could be seen as a list which is built using a non-strict constructor. Peter J. Landin first originated the idea of a non-strict data structure. Most applicative/functional languages as Scheme [ASS96] or Lisp are applicative order languages, that means that all the arguments of functions are evaluated when the function is applied. In contrast, normal order languages (such as Daisy [Joh89]) delay evaluation of function arguments until the actual argument values are needed. Delaying evaluation of function arguments until the last possible moment (e.g., until they are required by a primitive, or returned answer) is called lazy evaluation. Lazy evaluation, which is the key to making streams practical, comes from Algol 60, and was used to implement streams firstly in [Lan65]. Lazy evaluation for streams was introduced into Lisp by [FW76], which shows that with this mechanism, streams and lists can be identical. A lazy evaluator only computes values when they are really required avoiding computing values that are not really needed.

- Scheme provides a special form (`delay`) which enables to delay the evaluation of an expression. The use of `delay` and the macro tool (`define-syntax`) allows to implement streams in Scheme. This is detailed in section C.3.3.

To complete this brief overview of streams, we should make a little difference between Abelson and Sussman's stream definition and Burge's one. The first one defines streams as a pair which head part (`car`) is evaluated and tail part (`cdr`) is delayed. The second one defines them as list in which the creation of each list element (even the head) is delayed. The second definition is called 'lazy list'. It permits to create delayed versions of more general kinds of list structures, not just sequences, but for example trees [Hug89].

run, by each agent involved in a two-agent conversation, in an execution context private and local to the agent. Figure 4.3 shows an implementation of dialogue. The dialogue function takes a list of four parameters:

1. *inputs*. It is a stream of input messages (I_B^A for agent A, I_A^B for agent B);

2. *initial-state*. It is the initial state of the agent running a dialogue function (α_j for agent A, β_0 for agent B);

3. *step-fcn*. It is the transition function that defines the actions of the agent (f_B^A for agent A, f_A^B for agent B). The *step-fcn* takes a list of two parameters (i.e., inputs and state) and must return a stream of four elements:

   (a) A stream of unused inputs, which is usually the tail of *inputs*. That value will be used in the next step of the dialogue unless the dialogue terminates;

   (b) A stream of outputs messages sent to the interlocutor;

   (c) A new state;

   (d) A boolean value that is true if the agent wishes to terminate the dialogue and false otherwise.

   The dialogue function repeatedly applies *step-fcn* to the current values of *inputs* and *state* in order to find the new *outputs'* and *state'*;

4. *result-fcn*. It is a function which the agent uses to produce a 'dialogue value' with its last state (R_A and R_B).

The dialogue function returns a stream[2] of three elements:

1. *unused-inputs*. The agent removes the input elements that it needs from *inputs*, and returns the remainder;

2. *outputs*. It is a stream of output messages (O_B^A and O_A^B);

3. *result*. A value computed by the agent by applying its *result-fcn* to its final state.

Then, in a case such as the one illustrated in figure 4.1, the conversation between A and B is modelled locally to each agent by executing a different call to dialogue:

for agent A: *dialogue :* $\langle I_B^A \ \alpha_j \ f_B^A \ R_A \rangle \rightarrow (I_B^A \ O_B^A \ val)$

for agent B: *dialogue :* $\langle I_A^B \ \beta_0 \ f_A^B \ R_B \rangle \rightarrow (I_A^B \ O_A^B \ val)$

DEFINITION: DIALOGUE ABSTRACTION

A recursive function run by an agent to model a conversation with another one. The dialogue abstraction produces an output stream of messages and a new agent state by consuming an input stream of messages and an old state.

Remark – In dialogue the *step-fcn* takes an input stream as argument, but nothing implies it should begin by processing the first element of this stream (i.e., the first message send by the interlocutor agent); it may decide to process any element of the input stream at any time. This is the reason of the existence on *unused-inputs*.

---

[2]This sequence must be a stream because the *unused-inputs*, which can include future inputs from a future dialogue, has not to be evaluated since the messages inside have not been produced yet.

dialogue : ⟨*inputs initial-state step-fcn result-fcn*⟩ ≡
   letrec
      run ≡ λ ⟨*inputs state*⟩ *.*
         let
            (inputs' outputs' state' done') ≡ *stef-fcn :* ⟨*inputs state*⟩
         in
            if *done'*
               then *(inputs' outputs' result-fcn:state')*
               else
                  let
                     (inputs" future-outputs" result") ≡ *run :* ⟨*inputs' state'*⟩
                  in
                     (inputs" append-ll : ⟨*outputs' future-outputs"*⟩ *result")*
   in
      run : ⟨*inputs initial-state*⟩

Figure 4.3: Definition of the function *dialogue*

4.2.4 The dialogue abstraction limits

The dialogue abstraction was adequate for the interaction of two processes but today's distributed system requirements oblige to consider interaction between more than two processes. Therefore, the limit of the dialogue abstraction is intrinsic to it: it does not model more than two-agent conversations. Indeed, several uses of dialogue (executed serially or in parallel) do not model conversations among several agents but several different dialogues that an agent has with each interlocutor at the same time. For example, two dialogues executed serially by a same agent do not model a three-agent conversation as the first one must terminate before the second one starts. In the same way, two dialogues executed in parallel do not model a three-agent conversation as the different inputs and outputs are not intertwined. The processing of inputs of the first dialogue always produces outputs of the first dialogue and same thing respectively for the second dialogue.[3] We need to extend the dialogue abstraction to model conversations where processing of one agent inputs possibly produces not the outputs for this agent but another outputs intended to another agent. It is the aims of the i-dialogue abstraction.

4.3 The i-dialogue abstraction

This section extends the dialogue abstraction from two agents to n agents. In a first time, for the reader comprehension, we explain the '*trialogue*' abstraction which deals with three agents. Then, we generalize to n agents.

4.3.1 The trialogue abstraction

Trialogue aims to model an interactive session between three agents, A, B and C, where A and C send messages to B and B to A and C, as figure 4.4 shows. For agent B, the difference between two dialogues and trialogue is that the transition functions of B, f_A^B and f_C^B, do not produce respectively an output stream for A and B but the opposite, as shown in figure 4.5.

---

[3] Even if the agent is multi-processes enabled, it would need another function enabling the agent to pause the first dialogue process, to access the process structures in order to run another dialogue process and resume the first process after pausing the second one (and re-iterate).
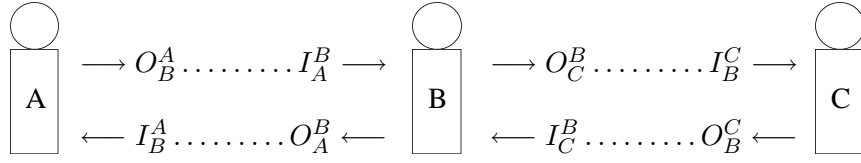
$$\begin{array}{c}
\text{A} \quad
\begin{array}{l}
\longrightarrow O_B^A \ldots\ldots\ldots I_A^B \longrightarrow \\
\longleftarrow I_B^A \ldots\ldots\ldots O_A^B \longleftarrow
\end{array}
\quad \text{B} \quad
\begin{array}{l}
\longrightarrow O_C^B \ldots\ldots\ldots I_B^C \longrightarrow \\
\longleftarrow I_C^B \ldots\ldots\ldots O_B^C \longleftarrow
\end{array}
\quad \text{C}
\end{array}$$

Figure 4.4: Streams in trialogue between agents A, B and C

$$(initial)\alpha_j$$
$$(\text{send to B})a_{Bj} \Rightarrow$$

$$\beta_0(initial)$$

$$\gamma_0(initial)$$

$$[\beta_1, b_{C0}] = f_A^B : [\beta_0, a_{Bj}]$$
$$(\text{send to C})b_{C0} \Rightarrow$$

$$[\gamma_1, c_{B0}] = f_B^C : [\gamma_0, b_{C0}]$$
$$\Leftarrow c_{B0}(\text{send to B})$$

$$[\beta_2, b_{A0}] = f_C^B : [\beta_1, c_{B0}]$$
$$\Leftarrow b_{A0}(\text{send to A})$$

$$[\alpha_{j+1}, a_{Bj+1}] = f_B^A : [\alpha_j, b_{A0}]$$
$$(\text{send to B})a_{Bj+1} \Rightarrow$$

$$\vdots \qquad\qquad\qquad \vdots$$

$$[\alpha_{j+k}, a_{Bj+k}] = f_B^A : [\alpha_{j+k-1}, b_{Ak-1}]$$
$$(\text{send to B})a_{Bj+k} \Rightarrow$$

$$[\beta_{2k+1}, b_{Ck}] = f_A^B : [\beta_{2k}, a_{Bj+k}]$$
$$(\text{send to C})b_{Ck} \Rightarrow$$

$$[\gamma_{k+1}, c_{Bk}] = f_B^C : [\gamma_k, b_{Ck}]$$
$$\Leftarrow c_{Bk}(\text{send to B})$$

$$[\beta_{2k+2}, b_{Ak}] = f_C^B : [\beta_{2k+1}, c_{Bk}]$$
$$\Leftarrow b_{Ak}(\text{send to A})$$
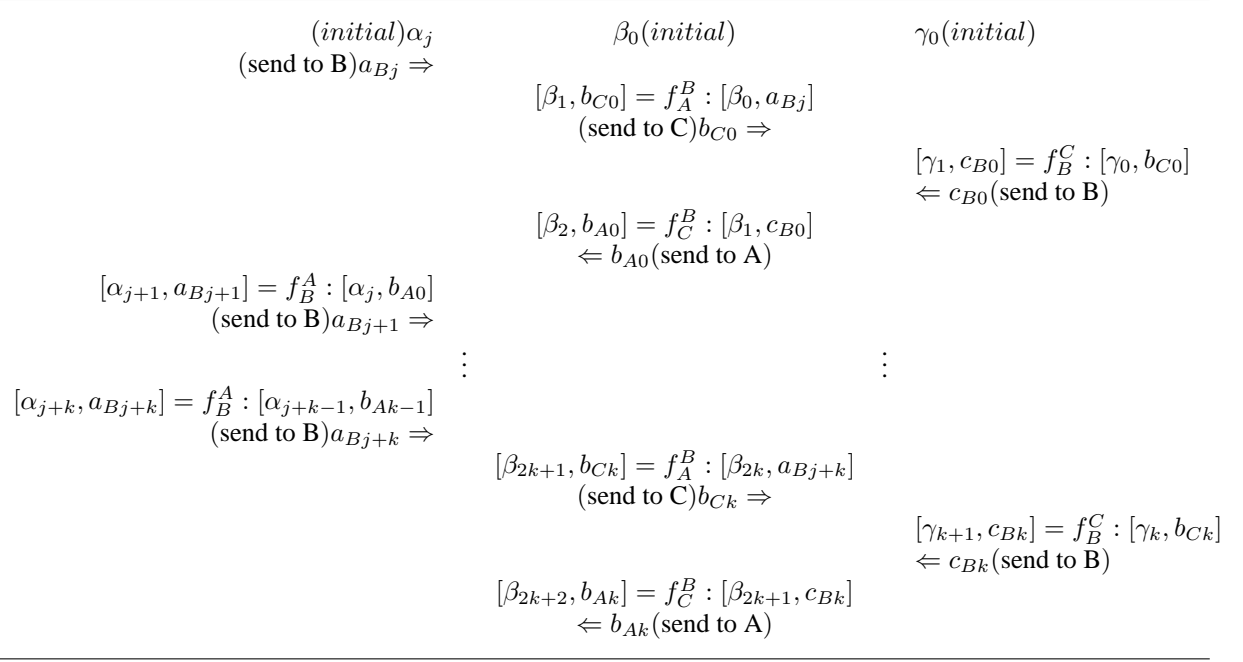
Figure 4.5: Trialogue between three agents A, B and C

There are two ways for implementing trialogue. The first one is defined in figure 4.6. The trialogue function take a list of six parameters: *inputs$_A$*, *inputs$_C$*, *step-fcn$_A$*, *step-fcn$_C$* and the original *initial-state* and *result-fcn*. An agent has a transition function dedicated to each agent with which it communicates. It gets the message from *inputs$_A$* and produces the answer message in *outputs$'_C$* before getting the message from *inputs$_C$* which is only produced after C has interpreted the message B has just sent. If *step-fcn$_A$* incites to terminate the trialogue (by returning a true boolean), then *step-fcn$_C$* is not applied. The trialogue function returns a stream of five elements: *unused-inputs$_A$*, *unused-inputs$_C$*, *outputs$_A$*, *outputs$_C$*, *result*.

So, in a case such as the one illustrated in figure 4.4, A and C run a dialogue as in previous section, and B runs a trialogue:

for agent A: *dialogue :* $\langle I_B^A \; \alpha_j \; f_B^A \; R_A \rangle \rightarrow (I_B^A \; O_B^A \; val)$

for agent B: *trialogue :* $\langle I_A^B \; I_C^B \; \beta_0 \; f_A^B \; f_C^B \; R_B \rangle \rightarrow (I_A^B \; I_C^B \; O_A^B \; O_C^B \; val)$

for agent C: *dialogue :* $\langle I_B^C \; \gamma_0 \; f_B^C \; R_C \rangle \rightarrow (I_B^C \; O_B^C \; val)$

The second way of implementing trialogue consists in giving as parameters two input streams but a single *step-fcn* processing the two streams. Then this *step-fcn* must return five elements (two unused-streams, two output streams and a result). The main drawbacks of this second way are: (i) the fact that it is more complicated to program, (ii) that with a *step-fcn* processing two streams, all the language and all

trialogue : $\langle inputs_A \; inputs_C \; initial\text{-}state \; step\text{-}fcn_A \; step\text{-}fcn_C \; result\text{-}fcn \rangle \equiv$

  letrec

    $run \equiv \lambda \; \langle inputs_A \; inputs_C \; state \rangle \; .$

      let

        $(inputs'_A \; outputs'_C \; state' \; done') \equiv s\text{-}fcn_A : \langle inputs_A \; state \rangle$

      in

        if *done'*

          then $(inputs'_A \; inputs_C \; null \; outputs'_C \; result\text{-}fcn{:}state')$

          else

            let

              $(inputs'_C \; outputs'_A \; state'' \; done'') \equiv step\text{-}fcn_C : \langle inputs_C \; state' \rangle$

            in

              if *done''*

                then $(inputs'_A \; inputs'_C \; outputs'_A \; outputs'_C \; result\text{-}fcn{:}state'')$

                else

                  let

                    $(inputs''_A \; inputs''_C \; f\text{-}outputs''_A \; f\text{-}outputs''_C \; result'') \equiv run : \langle inputs'_A \; inputs'_C \; state'' \rangle$

                  in

                  $(inputs''_A \; inputs''_C \; append\text{-}ll{:}\langle outputs'_A \; f\text{-}outputs''_A \rangle$

                              $append\text{-}ll{:}\langle outputs'_C \; f\text{-}outputs''_C \rangle \; result'')$

  in

    $run : \langle inputs_A \; inputs_C \; initial\text{-}state \rangle$

Figure 4.6: Definition of the function *trialogue*

applications of functions must be lazy (not simply stream processing functions – section C.3.3). Indeed, when the *step-fcn* is applied the value of the first stream could be evaluated, but not the value of the second one: its evaluation must be delayed (because the messages inside have not been produced yet). So, in a traditional applicative language where all arguments are evaluated before being applied, the first way should be used while in a lazy language the two ways are suitable. Moreover, this second way does not fit with the STROBE model as it is explained in section 4.3.3 compared to the first one. This way of implementing trialogue is not detailed.

    Remark – Note that in trialogue the *unused-inputs* stream produced by the first *step-fcn* is accessible while the second *step-fcn* is applied. In figure 4.6, when *step-fcn$_C$* is applied $inputs'_A$ is accessible. Then, if *step-fcn$_C$* was written in order to process it, B can access messages previously sent by A but not interpreted during *step-fcn$_A$* application.

4.3.2   Generalization: the i-dialogue abstraction

DEFINITION: I-DIALOGUE ABSTRACTION

A generalization of the dialogue abstraction applied in the STROBE model for modelling multi-party intertwined agent conversation. I-dialogue models conversations that must occur when an agent provides a composite service.

    The term i-dialogue is the abbreviation of *intertwined-dialogues*. An i-dialogue aims to model conversations between an agent and a group of agents. These conversations are dialogues intertwined to-

gether and executed in the same time as their inputs and outputs depend on each other. These situations (described in figure 4.7) can not be expressed by the dialogue abstraction. Concretely, the i-dialogue abstraction is a generalization of trialogue. The main idea consists in processing several inputs coming from several agents in a special order and being able to process each input stream to process the next output one.
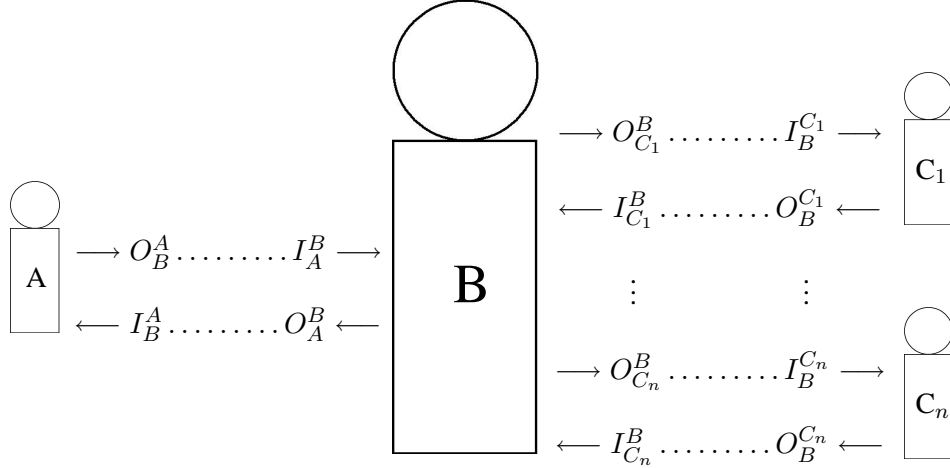
Figure 4.7: Streams in i-dialogue between agents A, B and $C_1 \ldots C_n$

The implementation of i-dialogue follows the same principles as dialogue and trialogue (first way). In order to process several *inputs$_i$* and several *step-fcn$_i$*, i-dialogue takes as parameter a list of input streams, *l-inputs*, and a list of transition functions, *l-step-fcn*. Therefore, the implementation of i-dialogue consists just in a classic list recursion! (figure 4.8). Notice that i-dialogue can realize all the calls of dialogue (or trialogue). If both the *l-inputs* and *l-step-fcn* have only one element then i-dialogue is equivalent to dialogue.

In a case such as the one illustrated in figure 4.7, calls to i-dialogue are, and produce:

for agent A: *i-dialogue* : $\langle \langle I_B^A \rangle \, \alpha_j \, \langle f_B^A \rangle \, R_A \rangle \to (\langle I_B^A \rangle \, \langle O_B^A \rangle \, val)$

for agent B: *i-dialogue* : $\langle \langle I_A^B I_{C_1}^B \ldots I_{C_n}^B \rangle \, \beta_0 \, \langle f_A^B f_{C_1}^B \ldots f_{C_n}^B \rangle \, R_B \rangle$
$$\to (\langle I_A^B I_{C_1}^B \ldots I_{C_n}^B \rangle \, \langle O_A^B O_{C_1}^B \ldots O_{C_n}^B \rangle \, val)$$

for agent C_1:   *i-dialogue* : $\langle \langle I_B^{C_1} \rangle \, \gamma_{10} \, \langle f_B^{C_1} \rangle \, R_{C_1} \rangle \to (\langle I_B^{C_1} \rangle \, \langle O_B^{C_1} \rangle \, val)$

             ⋮   ⋮

for agent C_n: *i-dialogue* : $\langle \langle I_B^{C_n} \rangle \, \gamma_{n0} \, \langle f_B^{C_n} \rangle \, R_{C_n} \rangle \to (\langle I_B^{C_n} \rangle \, \langle O_B^{C_n} \rangle \, val)$

In the i-dialogue function, the ordering of the elements of the lists (*l-inputs* and *l-step-fcn*) is important. Indeed, the order in which the inputs are processed by the agent running the i-dialogue corresponds to the semantics of the i-dialogue. This semantics may be, for example, determined by the agent before the starting of the i-dialogue. Thus, if an agent runs an i-dialogue for computing the sum of squares of numbers by interacting with an agent computing sums (A-sums) and an agent computing squares (A-squares) then it must send the numbers firstly to A-squares and then send the result to A-sums, because the two operations are not commutative.

However, if the second way for implementing trialogue is extended to i-dialogue (that means instead of using a list of *step-fcn$_i$*, using only one *step-fcn* processing all the streams) then the semantics of the

---

i-dialogue : ⟨l-inputs initial-state l-step-fcn result-fcn⟩ ≡
  letrec

    iter ≡ λ ⟨listi listf listui listo state⟩ .
      if *null?:listi*
        then *(listui listo state #f)*
        else
          let
            (inputs' outputs' state' done') ≡ car:listf : ⟨car:listi state⟩
          in
            if *done'*
              then *(append : ⟨listui cons:⟨inputs' cdr:listi⟩ ⟩*
                    append : ⟨listo   cons:⟨outputs' map:⟨λ x . null cdr:listi⟩ ⟩ ⟩
                    state'
                    #t)
              else
                iter : ⟨cdr:listi cdr:listf append:⟨listui ⟨inputs'⟩ ⟩ append:⟨listo ⟨outputs'⟩ ⟩ state'⟩

    run ≡ λ ⟨l-inputs state⟩ .
      let
        (l-inputs' l-outputs' state' done') ≡ iter : ⟨l-inputs l-step-fcn null null state⟩
      in
        if *done'*
          then *(l-inputs' l-outputs' result-fcn:state')*
          else
            let
              (l-inputs'' future-l-outputs'' result'') ≡ run : ⟨l-inputs' state'⟩
            in
              (l-inputs'' append-ll-with-list:⟨l-outputs' future-l-outputs''⟩ result'')

  in
    run : ⟨l-inputs initial-state⟩

Figure 4.8: Definition of the function *i-dialogue*

i-dialogue is not anymore determined by the order as before, but directly by the *step-fcn*.

Remark – Note that streams used in i-dialogue are even lazy lists because when agent B runs the i-dialogue function with $I_A^B, I_{C_1}^B, \ldots, I_{C_n}^B$, the first elements of $I_{C_1}^B, \ldots, I_{C_n}^B$ cannot be determined because the first elements of $O_{C_1}^B, \ldots, O_{C_n}^B$ needed for $C_1, \ldots, C_n$ to produce an answer, have not been produced yet.

Remark – The remark of section 4.3.1 may be generalized in i-dialogue. Each *step-fcn* in the *step-fcn*s list can access when applied the *unused-inputs* streams produced by the previous *step-fcn*s in the list. For example, when B processes the information coming from C_3, it can also use an information previously send by A, but not used before.

4.3.3 I-dialogue in the STROBE model

The i-dialogue abstraction is particularly synergic with the STROBE model. The cognitive interpreters are the same thing that the *step-fcn*s of the i-dialogue abstraction: they are functions producing an answer message (i.e., output) according to an incoming message (i.e., input) and an environment (i.e., state). Considering, for example the situation of figure 4.4, the analogy with the STROBE's notation is represented by table 4.1. Considering the STROBE model and i-dialogue, we can give more information concerning the agent local execution context in which the i-dialogue function is executed. Actually, we may distinguish two situations according if the modelled conversation is a two-agent or a n-agent one:

1. For a two-agent conversation between two STROBE agents A and B (figure 4.1), B can execute the i-dialogue function (or dialogue as they are equivalent for two agents) in E_A^B, because I_A^B, O_A^B and INT_A^B are available in this CE; A can execute the i-dialogue function in E_B^A, because I_B^A, O_B^A and INT_B^A are available in this CE. In this case, the i-dialogue function is therefore a capability of the agent as any other binding;

2. For a three-agent conversation with A and C (figure 4.4), a STROBE agent B must execute the i-dialogue function in a module separated of CEs because the function needs to call variables of several CEs. For example, in the situation of figure 4.4 the i-dialogue function run by B must access $I_A^B, I_C^B, O_A^B, O_C^B, INT_A^B, INT_C^B$. This is also the case for n-agent conversations.

The second situation is the reason why i-dialogue abstraction must be implemented, in the STROBE model, as a brain module i.e., a function with its own context, which can access variables in CEs. Notice however the special case in which a CE is dedicated to a group of interlocutors. From a STROBE agent's point of view, all the agents of a group are seen as a same entity; a dialogue abstraction may model the conversation. If for a specific reason, the STROBE agent wants to see differently some interlocutors and uses the same CE for them, it may specialize the CI included in the CE in order to process differently messages according to their sender. In this case, the second way of implementing i-dialogue (mentioned in section 4.3.1) may be used, as a single CI implements all the *step-fcn*s.

Table 4.1: Analogies between STROBE's elements and i-dialogue

state	CE
α	E_B^A
β	E_A^B & E_C^B
γ	E_B^C

step-fcn	CI
f_B^A	INT_B^A
f_A^B & f_C^B	INT_A^B & INT_C^B
f_B^C	INT_B^C

What is really interesting, in the STROBE model, is the fact that the CIs could evolve dynamically, while communicating (meta-level learning) so, by analogy, agents should dynamically change *step-fcn*s during the execution of the i-dialogue function. Such a feature seems very attractive for the dynamicity of conversations being modelled by the i-dialogue abstraction.

The dynamicity of the STROBE model combined with the i-dialogue abstraction is a good means to model conversational processes as they are needed in DSG. An agent executing an i-dialogue function provides a service realized by the interpretation of the message done by the *stef-fcn*s. Without a doubt, in figure 4.7 the agent A may be a service user (AA or HA), the agent B may be the service provider and the group of agents $C_1 \ldots C_n$ may represent the other members of the community. Agent B may interact with them in order to provide A with its service. If we consider that each C_i agent provides B with a service, i-dialogue models the composition of all these services. By enabling each agent to change dynamically the way it provides a service (i.e., change the *step-fcn*) we substantively make a step towards DSG by partially addressing C12[process].

4.3.4   Example: the travel agency

In this section we illustrate an example using i-dialogue on a classical scenario in service composition: the travel planning. A travel agency agent (A) provides a user agent (U), both with an airplane ticket booking service and with an hotel reservation service coordinated together, by composing the service provided by an airplane ticket agent (T) and the service provided by a hotel reservation agent (R), as illustrated in figure 4.9.
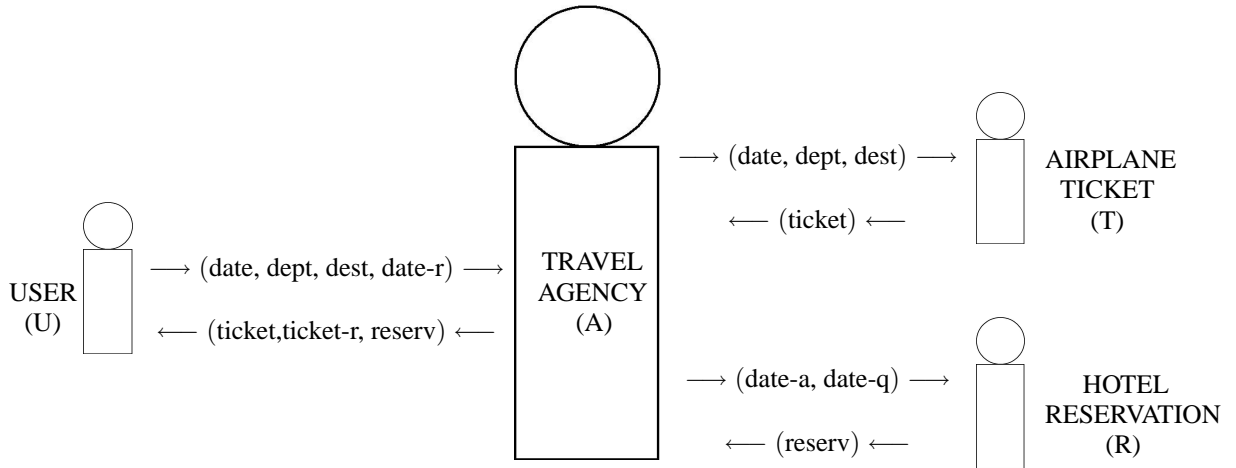
Figure 4.9: Streams in the travel agency scenario

The *step-fcn*s are detailed below (with their side effects):

$$f_U^A : \langle (\text{date dept dest date-r}) \; \alpha \rangle \longrightarrow \langle ()(\text{date dept dest date-r dept dest}) \; \alpha \; \text{b} \rangle$$

f_U^A: The agent U produces a stream of parameters which contains variables of the trip, i.e., departure date, departure city, destination city and return date. The agent A consumes all this stream and produces a stream for T composed of two sequences defining the two tickets the user requests, i.e., date, departure city, destination city. The state, α, of agent A is not changed by f_U^A.

$$f_A^T : \langle (\text{date dept dest}) \; \tau \rangle \to \langle ()(\text{ticket}) \; \tau' \; \text{b} \rangle$$

- Change the database of T

f_A^T: The agent T consumes the stream produced by A and for each triplet date, departure city, destination city, it produces a one-way ticket. The state, τ, of agent T is changed by producing ticket(s) (the database of T is updated to consider the new ticket(s) booked). The agent T produces a stream containing the ticket(s).

$$f_T^A : \langle(\text{ticket ticket-r}) \, \alpha\rangle \rightarrow \langle()(\text{date date-r}) \, \alpha' \, \text{b}\rangle$$

- Remember ticket and ticket-r

- Extract date and date-r from ticket and ticket-r

f_T^A: The agent A consumes the stream produced by T and for each ticket, it extracts the date of the ticket, in order to ask to R an hotel reservation. Note that the date extracted from the ticket are not necessarily the same that the date firstly produced by U. The agent A produces a stream of two dates, an arrival date and a departure date. It also changes its state, α, by memorizing the two tickets identifiers returned by T for the final answer to U.

$$f_A^R : \langle(\text{date-a date-q}) \, \rho\rangle \rightarrow \langle()(\text{reserv}) \, \rho' \, \text{b}\rangle$$

- Change the database of R

f_A^R: The agent R consumes the stream produced by A and for each pair of dates produces a hotel reservation. The state, ρ, of agent R is changed by producing reservation(s) (the database of R is updated to consider the new reservation(s)). The agent R produces a stream containing the reservation number.

$$f_R^A : \langle(\text{reserv}) \, \alpha\rangle \rightarrow \langle()(\text{ticket ticket-r reserv}) \, \alpha' \, \text{b}\rangle$$

- Extract ticket and ticket-r from state

f_R^A: The agent A consumes the stream produced by R and produces a stream directed to U with the reservation number and the two tickets identifiers previously memorized. A new state is thus computed.

$$f_A^U : \langle(\text{ticket ticket-r reserv}) \, \upsilon\rangle \rightarrow \langle()(\text{date dept dest date-r}) \, \upsilon' \, \text{b}\rangle$$

- Decide if the proposed solution is convenient

f_A^U: Finally, the agent U consumes the stream produced by A and according to its satisfaction (if $b = true$) it produces a new stream for A which recursively restart the dialogue.

Remark – In this scenario, in a sake of simplicity, messages (i.e., elements of streams) consist just of variables.

In this scenario the STROBE agents can change dynamically the *step-fcn*s, for example, f_R^A and f_A^R. The former could be changed in order to produce a different stream, including for example customer services requested by the user, and the latter could be changed to process this new stream. If these changes depends on A's initiative, then f_R^A changes are internal, but f_A^R changes are externals (do not concern agent A) and should be accomplished by communicating. These kinds of changes on the way messages are interpreted are considered as meta-level learning by communicating and are typical of STROBE agents.

4.4 Conclusion

We have presented in this chapter the i-dialogue abstraction, a model of multiple concurrent conversations in MAS. Each agent can use i-dialogue locally to model a conversation. This abstraction is based on simple constructs of applicative/functional languages (i.e., streams, lazy evaluation and first-class functions). It has two main advantages that respect the second requirement identified in the introduction of chapter 3: (i) it does not presuppose anything about the internals of the interlocutor agent and only deals with output streams of messages; (ii) it deals with the entire conversation (expressed by potentially infinite and not predetermined streams of messages) and not simply with a message alone. The main contributions of this chapter are:

- To investigate and deepen the elegant model of O'Donnell to more than two-process interaction;

- To consider this quite new model for MAS within the STROBE model;

- To propose a new approach (streams based) in addressing the question of high level process of services and in particular composition of services as it can occur in DSG.

Working on the i-dialogue abstraction and its related aspects pointed up some important consequences:

- It shows us an elegant method using simple but powerful techniques in order to model conversation. In particular, it revealed the importance that streams or lazy-evaluation should have to address agent conversation problems;

- It demonstrates the importance of separating the conversation modelling itself from the interpretation of communication message. I-dialogue shows that the former (i.e., the i-dialogue function) can be easily expressed where as the latter (i.e., the *step-fcn*s) can be very complex. Using i-dialogue, a programmer can concentrate on what he/she wants an agent to do when receiving messages without taking care of how the conversation is modelled;

- The use of streams in a Web context could be very interesting, especially to address the question of state. Indeed, streams are an alternative way for representing state. It is a possible perspective, as it was suggested early by Cerri [Cer99a];

- The contributions have partially addressed some characteristics of DSG as it was previously explained. C7[programming], C15[conversation], C12[process] and C16[dialogue] are directly addressed by i-dialogue. C2[interaction], C11[message] and C17[agent] are also indirectly addressed;

- We can note the importance of integration in our work. Conclusion of chapter 2 has claimed the need of looking sometime back in order to find today's and tomorrow's problem answers. The dialogue abstraction was proposed by O'Donnell in 1985. He was already concerned with making autonomous processes communicate one another. More than twenty years after, the problem is the same in MAS. We particularly like the dialogue abstraction, because it follows the same philosophy of using simple concepts or tools to address very complex problems. Basically, the dialogue and i-dialogue abstractions are simple recursive functions that manipulate streams.

Of course this work has some limits and weaknesses. I-dialogue does not directly model multiple conversations where each agent can send messages to any other agents or broadcast messages to all, as it can occur in a 'human meeting'. Using i-dialogue an agent can decide which part of an input stream it wants to process, but cannot for the moment change dynamically the order in which it processes all the input streams together (This order is, for the moment, determined when the i-dialogue is run). Or, add dynamically one or several new stream(s) to process. The ordering is not dynamic yet. The *step-fcn*s may change (as it is explain just before) but also the ordering of the elements of *l-inputs* and *l-step-fcn*. As a perspective for future work, and in the same logic than i-dialogue, we may suggest to add to the i-dialogue function a *select* parameter. This parameter should be a function that takes a list in parameter and returns one of the element of the list according to a specific algorithm. In the body of the i-dialogue function (figure 4.8), the *select* function should replace the call to *car* in the expression *car:listf*.[4]

Notice the proposed scenario addresses C12[process] but does not respect all DSG requirements. For example, D1 is not respected as the agent is supposed to directly send to the agency agent, A, the set of values it wants for its trip (i.e., dates and city). Remember that it was not the case in the dynamic

---

[4]Notice the *cdr* function need also to be changed by a `rest` function.

specification travel scenario (section 3.4.2) where an agent expressed constraints on the set of variables it may play on.[5] It shows us that the C12[process] question often addressed by SOC and MAS communities does not give an answer to DSG. DSG is more than enabling high level process of services.

<div style="background:#eaeaea;padding:1em">

<div align="center">DAISY LANGUAGE INSPIRED SYNTAX</div>

A part of this sidebar is quoted from [O'D85]. Some changes have been done specially to make the distinction between 'evaluated list' and 'lazy list'. In the descriptions below, ϵ may be any expression and η may be any identifier list (evaluated or lazy); an identifier list is an identifier or a list of identifier lists.

1. $\langle\ \epsilon_0\ \epsilon_1 \ldots \epsilon_n\ \rangle$
   An 'evaluated list' is a list of expressions enclosed in angle brackets. Lists are constructed with **cons**, **append** and **map**, and accessed with **car** and **cdr**. The predicate **null?** taste if a list is **null**.

2. $(\epsilon_0\ \epsilon_1 \ldots \epsilon_n)$
   A 'lazy list' or a stream (sidebar page 140) is a list of expressions enclosed in brackets. The function **append-ll** is equivalent to **append** but for lazy list. The function **append-ll-with-list** apply **append-ll** to the elements of the list in arguments.

3. $\lambda\,\eta\,.\,\epsilon$
   Function definitions. A lambda expression evaluates to a closure, as in Scheme. η represents the parameter(s) and ϵ is the body of the function.

4. $\epsilon_0 : \epsilon_1$
   Function applications are written with an infix apply operator ':'. The expression to the left of the ':' should evaluate to a closure. In this syntax functions take exactly one parameter, but that parameter may be an evaluated list. For example, inc : 3 and mpy : $\langle 3\ 4\rangle$ compute 3+1 and 3*4 respectively.

5. **if** ϵ_0 **then** ϵ_1 **else** ϵ_2
   The if expression. ϵ_1 is evaluated if ϵ_0 is true (#t) or ϵ_2 is evaluated if ϵ_0 is false (#f).

6. **let** $\eta_0 \equiv \epsilon_0\ \eta_1 \equiv \epsilon_1 \ldots$ **in** ϵ
   The **let** expression evaluates the right-hand sides of the equations in the existing environment, binds the resulting values to the left-hand sides, and evaluates ϵ in the new environment.

7. **letrec** $\eta_0 \equiv \epsilon_0\ \eta_1 \equiv \epsilon_1 \ldots$ **in** ϵ
   The **letrec** expression is like **let**, except that the right-hand sideq of the equations are evaluated in the extended environment, rather than the original environment. This makes it possible to define sets of recursive functions or recursive data structures in a **letrec**.

</div>

---

[5] Dynamic specification was allowed thank to a non-deterministic interpreter. However, the i-dialogue function is deterministic. Thus, to address C3[non-determinism] within i-dialogue we should notice two solutions: (i) the *step-fcns* should be non-deterministic; (ii) the ordering can also be non-deterministic if the previously suggested *select* function is also non-deterministic.

Chapter 5

Agent-Grid Integration Language

*The convergence of agent and Grid concepts and technologies will be
accelerated if we can define an integrated service architecture
providing a robust foundation for autonomous behaviors.*
Foster et al. [FJK04]

THE GRID and MAS communities believe in the potential of GRID and MAS to enhance each other as these models have developed significant complementarities. Thus, both communities agree on the 'what' to do: promote an integration of GRID and MAS models. However, while the 'why' to do it has been stated and assessed as we have seen in chapter 2, the 'how' to do it remains a research problem. This chapter addresses this problem by means of a service-oriented approach. Services are exchanged (i.e., provided and used) by *agents* through the *Grid* mechanisms and infrastructure. It proposes a model for GRID-MAS integrated systems. Concepts, relations between them and rules of these systems are semantically described by a set-theory formalization and a common graphical description language, called Agent-Grid Integration Language (AGIL).

Contents

5.1   Introduction

The analysis about the concept of service done in chapters 1 and 2 leads us to realize the importance of MAS and GRID in today SOAs. Section 2.4.3 has confirmed that the GRID-SOC integration approaches are today effective. When talking about Grid service the SOC and GRID communities talk about the concept resulting from this integration. Therefore, we may consider that achieving GRID-MAS integration (section 2.4.4) includes achieving MAS-SOC integration (section 2.3.4) as it is confirmed by, for example, the first analogy (section 2.4.5.1). This is what we call in this chapter, the service-based integration of GRID and MAS. Even if using agents for GRID was very early suggested [MT99, RM00, RJS01], Foster et al. [FJK04] propose the real first step in GRID-MAS integration as they examine work in these two domains, first to communicate to each community what has been done by the other, and second to identify opportunities for cross fertilization as they explained how GRID and MAS developed significant complementarities (section 2.4.4.1). This chapter suggests a second step by proposing a GRID-MAS integrated model described by a set-theory formalization and a common graphical description language.

As chapter 2 explains, the concept of service seems a good candidate for supporting this integration. The GRID is said to be the first distributed architecture (and infrastructure) really developed in a service-oriented perspective: Grid services are compliant Web services, based on the dynamic allocation of virtualized resources to an instantiated service [FKNT02]. Quite recently, GRID acquired major importance in SOA by augmenting the basic notion of Web Service with two significant features: service state and service lifetime management. On the other hand, agents are said to be autonomous, intelligent and interactive entities who may use and offer services (in the sense of particular problem-solving capabilities) [Fer99, Jen01]. MAS have also followed naturally the path towards SOC as interest turns to providing dynamic high level processes of services, semantic services, etc. The SOC community is also turning to MAS considering the important capacities agents have for using and providing services to one another. One may think MAS has found in SOC the possible killer-application it needs for years to prove the large set of feasibilities of the MAS approach. The concept of service is clearly at the intersection of the GRID and MAS domains as figure 5.1 illustrates.
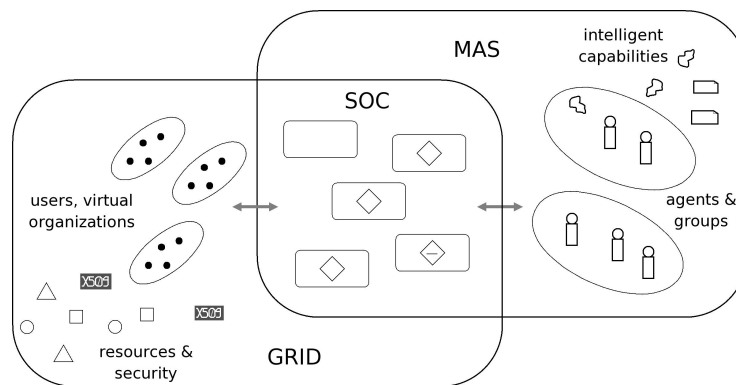
Figure 5.1: Intersection of GRID and MAS

The service-based integration of GRID and MAS models maybe summarized in two keys idea that motivate it:

- GRID and MAS have each developed a service-oriented behaviour, therefore the concept of service may represent a common integration;

- New needs in service exchange scenarios are clearly highlighted as DSG shows. An integration of GRID and MAS may address these needs.

As explained in section 2.4.4.2 our vision of a GRID-MAS integration is not a simple interoperation of the technologies. Integration refers to the idea of putting diverse concepts together to create an integrated whole. This contrasts with interoperation, which refers to making services work together by sharing the appropriate messages and using narrow, agreed-upon, interfaces, but without any single conceptual integration [SH05]. Integration is a very important concept in this thesis. This chapter combines all the propositions introduced before into a theoretical model that represents our view of the integration of SOC, MAS and GRID. AGIL is thus a language, but also a model of integration of GRID and MAS systems.[1]

The GRID-MAS integrated model addresses some characteristics of DSG that were not addressed before, such as C8[SOA], C20[grid] and C18[community]. It also shows how to map the answer to C22[separation] brought by CEs as suggested in chapter 3 with the last set of GRID specifications (i.e. WSRF). The relation with the STROBE model and i-dialogue presented before gives to AGIL an important number of aspects useful for DSG. However, AGIL may be considered independently from the STROBE model and the DSG questions in order to describe and represent future GRID-MAS integrated systems.

GRID-MAS integration. In order to propose the AGIL's integrated model, we extracted from GRID and MAS related work some key concepts. These concepts are directly influenced by OGSA (Open Grid Service Architecture) and the STROBE model. These concepts are step-by-step defined in section 5.2 and 5.3; we sum up here the two main underlying ideas:

- The representation of agent's capabilities as Grid services in a service container, i.e., viewing a Grid service as an 'allocated interface' of an agent's capability by substituting the object oriented kernel of Web/Grid services with and agent oriented one;

- The assimilation of the service instantiation mechanism – fundamental in GRID as it allows Grid services to be stateful and dynamic – with the dedicated cognitive environment instantiation mechanism – fundamental in STROBE as it allows an agent to dedicate to another one a conversation context.

Agent-Grid integration language. Describing simply and clearly key GRID and MAS concepts and their integration is a real need for both communities. GRID and MAS should appreciate a common description language which:

- defines the respective domain ontologies. For instance, GRID needs a description language that summarizes and rigorously explains GRID concepts. Without considering the agent side, AGIL may play also this role;

- uses the same terms and representations for an identical concept. For example, virtual organization and group; choreography of service and agent conversation, role and service, etc. The analogies in GRID and MAS concepts were precisely described in section 2.4.5;

- rigorously fixes the integration rules e.g., a service instance is instantiated by a unique service factory or any agent member of a VO holds a X509 certificate;

- may help researchers of GRID, MAS or SOC communities to specify and model their GRID-MAS integrated applications and systems. A kind of UML, applicable for GRID-MAS integrated systems;

- would promulgate the development of GRID-MAS integrated systems by proposing a uniform way of describing GRID and MAS together.

---

[1]Therefore the reader will find expression such as 'In AGIL, an agent can...'

AGIL is such a language. Modelling languages must [SH05]: (i) provide explicit representations for concepts; (ii) provide means for representing relationships among the concepts; (iii) include constraints on the relationships that can occur among the concepts. This is exactly what AGIL does by taking a specific graphical representation for each concept and their relations. Moreover, the rules of the integration are all expressed by a set-theory formalization which rigourously defines constraints on these concepts and relations.

Chapter overview.   The remainder of the chapter is organized as follows: Section 5.2 presents AGIL's main concepts, relations and rules. Each time a concept is introduced, the corresponding graphical representation is illustrated and the relation(s) between this concept and the other ones is formalized. Related rules are also specified. Whereas this section rather presents the 'static' concepts and relations of AGIL, section 5.3 details the dynamics of such a model: agent interaction, service-capability instantiation, agent reproduction, etc. Some aspects of the behaviour of STROBE agents introduced in section 3.3.4 are completed at the light of the integration. Section 5.4 sums up the integrated model with AGIL. A simple example is illustrated and a comparison with WSRF done. Advantages for SOC, MAS and GRID is also discussed. In section 5.5, an example of application scenario is taken to illustrate the utility of such a language (looking for a job scenario). Finally, section 5.6 concludes the chapter and gives some perspectives.

5.2   AGIL, a GRID-MAS integrated model

This section defines progressively each AGIL's concept, relations between these concepts and rules. Appendix B summarizes both AGIL's concepts and relations (some short recalls about relations in set theory are also given). Notice that key GRID concepts presented in this section have been established by the OGSA or WSRF specifications. Similarly, key MAS concepts have been established by different approaches in the MAS literature, especially the STROBE model. As we are focussing on concepts, we adopt the most convenient terminology from these sets of specifications. Besides, some concepts and relations (related to MAS) were previously presented when STROBE was introduced in chapter 3.

DEFINITION: AGENT-GRID INTEGRATION LANGUAGE

GRID-MAS integrated systems description language which formalizes both key GRID and MAS concepts, their relations and the rules of their integration with graphical representations and a set-theory formalization. AGIL represents also a GRID-MAS integrated model which formalizes the service exchange interactions of STROBE agents through the Grid infrastructure.

5.2.1   Grid resource representations

The Grid is a resource-sharing system. The two elementary physical resource types shared in the Grid are *computing resources* and *storage resources*. These are classes of resources. Example of instances of these classes of resources are clusters, simple PCs, processors farms, network storage, data bases, servers, etc.[2] Let Ω be the set of computing resources (ω) and let Θ be the set of storage resources (θ). Grid resources are represented in AGIL by circles and squares as figures 5.2 and 5.3 show.

---

[2]Grid users are sometime considered as resources. We address that later in the chapter.

Figure 5.2: Computing resource representation

Figure 5.3: Storage resource representation

However, the Grid does not directly deals with these elementary elements but with association of different types of elementary resources called *hosts*. A host is an abstraction used by GRID to see a Grid resource in a unique and standard way. To simplify this abstraction we can consider a host as either a physical association (resource-coupling) between a computing resource and a storage resource, called a *single host* or an association of several hosts (host-coupling) coupled together, called a *coupled host* (notice a coupled host can be part of another coupled host.). Let H be the set of hosts (h); a single host is a pair (θ, ω), $\theta \in \Theta, \omega \in \Omega$. Hosts are represented in AGIL by triangles as figure 5.4 shows.

Figure 5.4: Host representation

The *resource-coupling* relation formalizes the relation between pairs of resources and hosts. The *resource-coupling* relation is represented in AGIL as figure 5.5 shows. Any[3] pair of storage and computing resource is coupled by zero or exactly one single host. Any host couples at most one pair of resource.

$$resource - coupling : \Theta \times \Omega \to H \text{ (function)}$$

Figure 5.5: *Resource-coupling* relation representation (single host)

Rule 11 *A pair of storage and computing resource is coupled by a unique host.*

$\forall \theta \in \Theta, \forall \omega \in \Omega, \forall h_1, h_2 \in H, resource - coupling(\theta, \omega) = h_1 \wedge resource - coupling(\theta, \omega) = h_2 \Rightarrow h_1 = h_2$

Rule 12 *A host couples one unique pair of storage and computing resource.*

$\forall \theta_1, \theta_2 \in \Theta, \forall \omega_1, \omega_2 \in \Omega, \forall h \in H, resource - coupling(\theta_1, \omega_1) = h \wedge resource - coupling(\theta_2, \omega_2) = h \Rightarrow \theta_1 = \theta_2 \wedge \omega_1 = \omega_2$

---

[3]In order to precise AGIL's relation cardinalities, we will always use the article 'any' to identify all the elements of a set, and we will specified if an element is in relation with zero, (at least, at most, exactly) one or several elements of the other set. We add a rule each time the type of relation (function, application, surjection, etc. doest not includes the restriction.

The *host-coupling* relation formalizes the relation between hosts. The *host-coupling* relation is represented in AGIL as figure 5.6 shows. Any host is coupled by zero or exactly one coupled host. Any host couples either zero or at least two hosts.

$$host - coupling : H \rightarrow H \text{ (function)}$$

Figure 5.6: *Host-coupling* relation representation (coupled host)

Rule 13 *A host is coupled by a unique coupled host.*
$\forall h_1, h_2, h_3 \in H, host - coupling(h_1) = h_2 \land host - coupling(h_1) = h_3 \Rightarrow h_2 = h_3$

Rule 14 *A coupled host couples at least two hosts.*
$\forall h_1, h_2 \in H, host - coupling(h_1) = h_2 \Rightarrow \exists h_3 \in H, host - coupling(h_3) = h_2 \land h_1 \neq h_3$

Rule 15 *The host-coupling relation is irreflexive i.e., a coupled host does not couples itself.*
$\forall h_1, h_2 \in H, host - coupling(h_1) = h_2 \Rightarrow h_1 \neq h_2$

Rule 16 *The host-coupling relation is asymmetric i.e., a coupled host that couples a host, cannot be coupled by this host.*
$\forall h_1, h_2, h_3 \in H, host - coupling(h_1) = h_2 \land host - coupling(h_2) = h_3 \Rightarrow h_1 \neq h_3$

Remark – Some questions about hosts and coupling are still open. For example, should we consider the host-coupling relation a transitive one? We do not answer yet this question, however it could be later added as an AGIL rule.

5.2.1.1 Virtualization of resources

The sharing of these resources consists of a two steps process: virtualization and reification. The virtualization consists to accumulate the available resources provided by hosts. The result of the virtualization process is a set, R, of virtualized resources (r). Virtualized resources are not directly represented in AGIL, but a subset of R is represented by a trapeze as figure 5.7 shows.
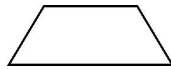
Figure 5.7: Set of virtualized resources representation

The *virtualizing* relation formalizes the relation between hosts and virtualized resources. The *virtualizing* relation is represented in AGIL as figure 5.8 shows. Any host virtualizes its resources in exactly one subset of virtualized resources (maybe empty). Any subset of virtualized resources is virtualized by zero, one or several hosts.[4]

---

[4] A good metaphor to understand virtualization is currency conversion: each host uses a different currency to measure its amount of resources. To virtualize means to transform (following a rate of conversion) a part of its amount in a common currency.

$$virtualizing : H \to \mathcal{P}(R) \text{ (application)}$$
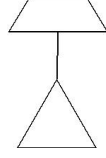
Figure 5.8: *Virtualizing* relation representation

5.2.1.2 Reification of resources

The second step in the sharing of resource is the reification. Virtualized resources are reified in (i.e., re-allocated to) *service containers* (described in section 5.2.2.3). Let U be the set of service containers (u). The *reifying* relation formalizes the relation between virtualized resources and service containers. The *reifying* relation is represented in AGIL as figure 5.9 shows. Any service container reifies exactly one subset of virtualized resources (non empty). Any subset of virtualized resources is reified in zero, one or several service containers.

$$reifying : U \to \mathcal{P}(R) - \{\emptyset\} \text{ (application)}$$
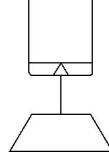
Figure 5.9: *Reifying* relation representation

Notice that resource virtualization and reification is done at the core GRID level (middleware). The rest of core GRID functionalities (e.g., container, CAS, etc.) described in AGIL is itself described by a single unit: the Grid service.

Formalization of mechanisms. Rigorously formalizing GRID and MAS concepts and their relations, allow us to rigourously express mechanisms occurring in GRID-MAS integrated systems. For example:

Let V_h be the value (e.g., in GBytes and/or GFlops) of the amount of resource of the host h. Let V_r be the value of the amount of resource of a virtualized resource r. Therefore, the amount of resource of a virtualized resource may be expressed as a weighted sum of amount of resources of hosts that virtualize their resources:

$$\forall h_i \in H, \forall r \in R, r \in virtualizing(h_i) \Leftrightarrow \exists \alpha_i \in [0,1] \ V_r = \sum_i \alpha_i.V_{h_i}$$

The set of coefficients $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ may be called the *virtualization vector*. For a given subset of virtualized resources, the set of virtualisation vectors form the *virtualization matrix*. This matrix models at a given time all the sharing of resources (who share and how much).

Let V_u be the value of the amount of virtualized resource of the service container u. The amount of virtualized resource of a service container may be expressed as a weighted sum of amount of resources of reified virtualized resources:

$$\forall r_i \in R, \forall u \in U, r_i \in reifying(u) \Leftrightarrow \exists \beta_i \in [0,1] \ V_u = \sum_i \beta_i . V_{r_i}$$

The set of coefficients $beta = (\beta_1, \beta_2, \ldots, \beta_n)$ may be called the *reification vector*. For a given subset of virtualized resources, the set of reification vectors form the *reification matrix*. This matrix models at a given time all the allocation of resources (who consumes and how much). At a given time the amount of virtualized resource reified for a service container can be expressed as a function of the amounts of resource of hosts (not demonstrated here):

$$\forall h_j \in H, \forall r_i \in R, \forall u \in U, r_i \in virtualizing(h_j) \wedge r_i \in reifying(u) \Leftrightarrow$$

$$\exists \alpha_{j_i}, \beta_i \in [0,1] \ V_u = \sum_j \left(\sum_i \beta_i \alpha_{j_i} \right) . V_{h_j}$$

5.2.2 Service instance representations

For SOC an GRID communities, a *service* is an interface of a functionality (or capability) compliant with SOA standards. Let Σ be the set of services (σ). As we will see later, community authorization services (described in section 5.2.2.2 and introduced in 2.4.2) and service containers are themselves service instances. In order to distinguish them, we call *normal services*, services that are not CAS or service container. Let S be the set of normal services (s), and let C be the set of CAS (c). Then, $\Sigma = U \cup C \cup S$.

5.2.2.1 Normal service representations

We may distinguish three kinds of normal services:[5]

- stateless services;

- transient stateful services;

- persistent stateful services.

Services are said to be stateless if they delegate responsibility for the management of the state to another component. Statelessness is desirable because it can enhance reliability and scalability. A stateless service can be restarted following failure without concern for its history of prior interactions, and new copies (instances) of a stateless service can be created (and subsequently destroyed). Stateless service instances are quite restrictive: they are synchronous (i.e., messages can not be buffered and do block the sender or receiver), point-to-point (i.e., used by only one user) and interact via simple one-shot interaction (i.e., request/answer). A stateless service does not establish a conversation. Instead, it returns a result from an invocation, akin to a function.

Stateful services require additional consideration: they have their own running context where is kept the contextual state memory. This state may evolve not simply with external messages (as simple objects) but also according to autonomous rules. A stateful service has an internal state that persists over multiple interactions. They can be persistent or transient. Transient services are instantiated by a service factory, whereas persistent services are generally created by out-of-band mechanisms such as the initialization of a new service container. Stateful service instances may be multipoint (i.e., used by several users) and may interact by simple one-shot interaction or long-lived conversation. Stateful services may be synchronous or asynchronous.

---

[5]Some other comparison criteria were introduced in the sidebar page 62.

In AGIL, normal services are represented with a rectangle with round angles as figure 5.10 shows. If the service is stateful, a lozenge represents its context; if it is transient, a short line represents its lifetime.

stateless   stateful   transient stateful
service instance   service instance   service instance

Figure 5.10: Normal service representations

5.2.2.2 Community authorization service representation

The relation between a VO (described in section 5.2.3.1 and introduced in section 2.4.1) and a service container is realized by a *Community Authorization Service* (CAS) which formalizes the VO-dedicated policies of service by members. The CAS is the first important element of the Grid security infrastructure represented in AGIL. A CAS is itself a service. The CAS may be viewed as a MxS matrix, where M corresponds to the number of members of the VO, S to the number of currently active services available for the VO, and the matrix nodes are deontic rules. These rules permit the accurate specification of the right level for a member on a service (e.g., permissions, interdictions, restrictions, etc.). This matrix contains at least one column (a VO exists only with at least one user) and one row (the CAS has itself an entry in the CAS). CASs are represented in AGIL as figure 5.11 shows.
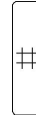
Figure 5.11: CAS representation

Let O be the set of virtual organizations (o). The *authorizing* relation formalizes the relation between VOs and CASs. The *authorizing* relation is represented in AGIL by a curve thin arrow as figure 5.12 shows. Any VO is authorized by exactly one CAS (each agent of the VO has an entry as a column in this CAS matrix). Any CAS authorizes exactly one VO.

$$authorizing : O \rightarrow C \text{ (bijection)}$$

Figure 5.12: *Authorizing* relation representation

Each service is identified by a *handle*. A running service can be retrieved with its handle. A Grid service handle, as specified in OGSA, or a endPoint reference, as specified in WSRF, provides a unique pointer that is a URI, to a given service. The *handling* relation formalizes the relation between services and CASs. The *handling* relation is represented in AGIL by a straight thin arrow as figure 5.13 shows. Any CAS and any normal service are handled by exactly one CAS (each service as a row entry in this CAS matrix). Any CAS handles at least one service (itself).

$$handling : S \cup C \rightarrow C \text{ (surjection)}$$
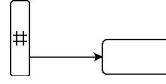
Figure 5.13: *Handling* relation representation

Rule 17 *The handling relation restricted to C is reflexive i.e., a CAS handles itself.*
$\forall c \in C, handling(c) = c$

5.2.2.3 Service container representation

Services need a hosting environment to exist and to evolve with their own private contexts (i.e., set of resources). This is the role of the *service container* which implements the reification of a portion of the virtualized resources available in a secure and reliable manner. A service container is defined as a triplet composed of a non empty set of virtualized resources, a CAS and a set (maybe empty) of normal services: $U = \{(R_u, c, S_u), R_u \in \mathcal{P}(R) - \{\emptyset\}, c \in C, S_u \in \mathcal{P}(S)\}$. Since a container is a particular kind of service, it is created either through the use of a service factory or by a direct core GRID functionality. Service containers are represented in AGIL as figure 5.14 shows.

Figure 5.14: Service container representation

A service container includes several types of services. The *including* relation formalizes the relation between services and service containers. The *including* relation is represented in AGIL as figure 5.15 shows. Any normal service and any CAS are included in exactly one service container. Any service container includes at least one services (one CAS and zero, one or several normal services).

$$including : S \cup C \rightarrow U \text{ (surjection)}$$
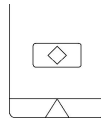
Figure 5.15: *Including* relation representation

Rule 18 *The subset of virtualized resources that partially composes a service container is reified for it.*
$\forall u = (R_u, c, S_u) \in U, \forall r \in R_u, r \in reifying(u)$

Rule 19 *All the services (CAS and normal services) that partially compose a service container are included in this service container.*
$\forall u = (R_u, c, S_u) \in U, \forall s \in S_u, including(c) = u \wedge including(s) = u$

Rule 20 *A service is handled by a CAS if and only if they are both included in the same service container.*
$\forall \sigma \in S \cup C, \forall c \in C, handling(\sigma) = c \Leftrightarrow including(\sigma) = including(c)$

Remark – Some questions about service container are still open. For example, does a service container has an entry in the matrix of the CAS it includes?

5.2.3 Agent representation

In AGIL, the term *agent* is used to uniformly denote artificial agent, human agent and Grid user. In particular, by viewing Grid users as agents, we may consider them as potential artificial entities. As introduced in section 3.3.3 an agent is represented in AGIL by a skittle as figure 5.16 shows. The *head* contains the *brain* and the *body* contains the agent's capabilities.

Figure 5.16: Agent representation

AGIL is limited to the MAS concepts described in this thesis. For example, there is no representation of the environment (i.e., the world surrounding the agents) or objects present in this environment [Fer99].

5.2.3.1 Virtual organization

In AGIL, agents are *members* of *virtual organizations*. The term VO unifies the concept of organization or community in GRID and the concept of group in MAS. Thus we can now talk about 'VO of agents'. A service container is allocated to (and created for) one and only one VO as it is formalized by the *authorizing* relation. A VO is a subset (non empty) of agents: $O \in \mathcal{P}(A) - \{\emptyset\}$. VOs are represented in AGIL by large ellipses as figure 5.17 shows.

Figure 5.17: VO representation

The *membershipping* relation formalizes the relation between agents and VOs. The *membershipping* relation is represented in AGIL by a full dot, and a simple line binding skittle and full dot as figure 5.18 shows. Any agent is a member of zero, one or several VOs. Any VO contains at least one agent.

$$membershipping : A \rightarrow O \text{ (relation)}$$
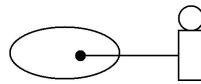
Figure 5.18: *Membershipping* relation representation

$\forall a \in A$, we denote $membershipping(a)$ the set of VOs a is a member of. Notice that as a VO is a subset of agents, the *membershipping* relation is represented in the set-theory by the inclusion ($\in$) relation i.e., $\forall o \in O, \forall a \in A, a \in o \Leftrightarrow o \in membershipping(a)$. In the following, we will prefer the inclusion symbol.

Rule 21 *A VO contains of at least one member.*

$\forall o \in O, \exists a \in A, a \in o$

5.2.3.2 X509 certificate

The *X509 certificate* is the second important element of the Grid security infrastructure represented in AGIL (section 2.4.2). It allows mutual authentication and delegation. A X509 certificate is valid if it has been signed by a trusted Certification Authority (CA). There are four kinds of X509 certificates: Grid user certificate, Grid host certificate, Grid proxy certificate, CA certificate. Let X be the set of X509 certificates (x). X includes all the types of certificates, except CA certificates (not formalized yet). X509 certificates are represented in AGIL as figure 5.19 shows.

Figure 5.19: X509 certificate representation

The *holding* relation formalizes the relation between agents, hosts and X509 certificates. The *holding* relation is represented in AGIL as figure 5.20 shows. Any X509 certificate is held by either exactly one agent or exactly one host. Any agent holds zero, one or several X509 certificates (i.e., type Grid user and proxy certificate). Any hosts holds at most one X509 certificate (i.e., type Grid host certificate).

$$holding : X \rightarrow A \cup H \text{ (application)}$$
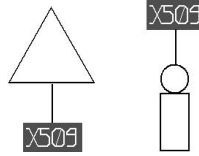
Figure 5.20: *Holding* relation representation

Rule 22 *A host can hold at most one X509 certificate (type Grid host certificate)*

$$\forall h \in H, \forall x_1, x_2 \in X, holding(x_1) = h \land holding(x_2) = h \Rightarrow x_1 = x_2$$

Rule 23 *All agents members of a VO hold a X509 certificate.*

$$\forall a \in A, \forall o \in O, a \in o \Rightarrow \exists x \in X, holding(x) = a$$

Rule 24 *All hosts that virtualize their resources in the Grid hold a X509 certificate.*

$$\forall h \in H, \forall r \in R, r \in virtualizing(h) \Rightarrow \exists x \in X, holding(x) = h$$

Remark – We do not formalize yet if a coupled host that couples n hosts should hold 1, n or n+1 X509 certificates. We do not formalize yet CAs and CA certificates (notice that viewing CAs as agents would not change the holding relation). Besides, entities with a X509 certificate are sometimes called Grid *nodes*. These set of nodes and authorities form Grid *trusts*. Grid nodes and Grid trusts are not represented in AGIL yet.

5.2.3.3 Capability representation

The body of an agent is composed of a set of *capabilities* which correspond to the agent capacity or ability to do something. Capabilities are represented in AGIL by rectangles with a fold back upper right corner as figure 5.21 shows (like bindings in the STROBE model). Let Λ be the set of capabilities (λ).

Figure 5.21: Capability representation

A strong element of the GRID-MAS integration consists in viewing services as allocated interfaces of agents' capabilities. A Grid service is seen as the interface of a capability published in a service container and with allocated resources. An agent has a set of capabilities it may transform into Grid services available in the different VOs it is a member of. The process of 'transforming' or 'publishing' a capability into a service is called the *servicization process*.[6] When a capability is servicized, it means:

- the interfacing of this capability with SOA standards i.e., mainly WSDL;

- the addition (possibly by using an add-service service) of this service to the VO's service container by assigning it a handle and by allocating it private resources;

- the requesting of the VO's CAS service to add an entry for this service (the agent has to decide the users' right levels);

- the publishing of the service description in the VO's registry, if it exists;

- the notification to the VO's members of the VO that a new service is available;

- etc., according to VO or service container local rules.

When an agent servicizes one of its capability into a service available for a VO it uses a set of services of this VO. Each of the previous step of the servicization process is achieved using a specific VO local service (e.g., interfacing, adding, notifications services). This servicization process is not discrete but continuous. Service and capability keep aligned one another. For example, if the capability of the agent changes, then the service changes at the same time. With this viewpoint, an agent can provide different VOs with different services. Notice also that a service is agent-specific, that means that only one agent executes (i.e., provides (section 5.3.1)) the service in a container. However, it does not prevent another agent of the VO from providing the same type of service. What is important in this servicization process is that it remains the same irrespective of the kind of agent involved. Both AAs and HAs transform their capabilities in the VO's service container modulo different (graphical) interfaces. For example, an AA may servicize its capability to compute square roots (i.e., a function that receives an integer as a parameter and returns a float as result), and a HA may servicize its pattern-recognition capability (i.e., a function that receives an image as a parameter and returns a concept as result (described in an ontology)). Notice that the service and the capability lifetimes are not necessarily the same. Even if a service is transient in a service container the corresponding capability maybe persistent in the agent's body according to its initiative.

Notice also that some sets of services, servicized and available within a VO, may correspond to roles and should interface the capabilities of that roles. When agents are designed with a specification of their roles, a set of capabilities implements these roles.

---

[6]We can say that as GRID virtualizes resources and reifies them in a service container, an agent virtualizes capabilities and reifies them in a service container. This realizes the service level virtualization mentioned in section 2.4.2.

Remark – In order to avoid the problem of mapping between SOAP and ACLs (section 2.3.4.2) we do not consider that there is a message transformation (or any interaction) between the service and the agent's capability; they are the same thing represented differently. Agents are supposed to be able to interpret directly SOAP messages corresponding to the capabilities they servicized in WSDL. For example, the question of semantics is not resolved by the servicization process.

> DEFINITION: SERVICIZATION PROCESS
> The virtualization process of an agent's capability into a Grid service within a service container. It is the key element of AGIL to represent a service as an interface of a capability executed with Grid resources but managed by an intelligent, autonomous and interactive agent.

The *interfacing* relation formalizes the relation between capabilities and services. The *interfacing* relation is represented in AGIL by a dotted-line as figure 5.22 shows.[7] Any service interfaces exactly one capability. Any capability is interfaced by at most one service.

$$interfacing : \Sigma \rightarrow \Lambda \text{ (injection)}$$
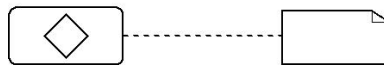
Figure 5.22: *Interfacing* relation representation

Remark – Grid resources are available for services (i.e., servicized capabilities) execution. The agent itself is still executed autonomously with its own resources and process (e.g., on an agent platform (JADE, MadKit) or somewhere else for mobile agents).[8]

5.2.4 Conversation context representation

Cognitive environment. In conjunction with the STROBE model, agents' capabilities may be executed in a particular conversation context called a *cognitive environment* (section 3.3.3.3). CEs are represented in AGIL by folders as figure 5.23 shows.

Figure 5.23: CE representation

The *executing* relation, introduced in section 3.3.3.3, formalizes the relation between capabilities and CEs. The *executing* relation is represented in AGIL as figure 5.24 shows.

Figure 5.24: *Executing* relation representation

---

[7]Notice that figure 5.22 illustrates the *interfacing* relation with a normal service. The same representation is used with CAS and service container.

[8]Except if the agent platform, or the agent use an 'allocating resources Grid service' for its private execution! The agent platform may also be deployed on the Grid.

The *executing* relation allows to make an analogy with WSRF. In STROBE, we have an association between a capability and its execution context. In one hand, the execution context may be viewed as a stateful resource, because an environment is a structure that stores a set of variables i.e., that represents a state. On the other hand, a capability, is a syntactic definition of a procedure or function, a stateless abstraction, which can only be applied in an execution context, stateful, representing the state of the world when the capability is applied (e.g., an environment that binds its parameters, primitives, etc.). In AGIL, we have explained how these capabilities are interfaced as Grid services in a service container. Therefore, the capability represents the stateless service and the CE represents the stateful resource, separated from the service but accessed by several ones. It is exactly the kind of association done by WSRF, which considers a WS-Resource as an association between a stateful resource and a stateless service. This analogy is very important, it gives us a model to address C22[separation]. This vision fits with the solution proposed by WSRF to get out from the service, the stateful resource manipulated by this service.

The *owning* and *dedicating* relations, introduced in section 3.3.3.3, formalize the relation between agents and CEs. The *owning* relation is represented in AGIL as figure 5.25 shows. The *dedicating* relation is not graphically represented.

Figure 5.25: *Owning* relation representation

Remark – Generic capabilities contained in an agent's global CE may be servicized, but not adapted for a specific interlocutor or group of interlocutors, because the global CE is not dedicated. Using a service that interfaces a global CE capability is analogous to ask to a Grid service factory the creation of a new service instance. In other words: a service instance creation demand addressed to a service factory interfacing a global CE capability is not interpreted in a dedicated context. Notice that the service instance created may itself be a service factory. In this case, this factory may be adapted because it interfaces a local CE dedicated capability.

Advantage of the STROBE model. On one hand, using OGSA specifications in a GRID-MAS integration is natural: OGSA is the unique specification of GRID and Grid services. On the other hand, the MAS community proposes many different approaches and models. This section justifies why STROBE is convenient on the basis of three main reasons:

- CEs represent dedicated conversation contexts (i.e., stateful resources) and execute dedicated capabilities-services;

- the STROBE's instantiation of CE mechanism map perfectly Grid service instantiation mechanism;

- STROBE's advantages such as meta-level learning or dynamic-specification or i-dialogue are available.

STROBE agent's capabilities are located in CEs i.e., an agent has specific capabilities for each agent it communicates with. Capabilities are said to by dedicated. The fact of having dedicated capabilities is the basic feature of the STROBE model to implement DSG. The aim of the model is to put at the centre of the agent architecture the conversation contexts in which it interprets messages from its interlocutors (i.e., in which services are executed).

In a sake of openness to other approaches in agent modelling, AGIL is not restricted to the STROBE model. In other agent architectures, CE may simply be viewed as a conversation context. Without doing any supposition on agent architecture, we should simply say that an agent's capability is executed in the agent body.

5.3 AGIL dynamics

We have just presented in section 5.2 presented AGIL's static concepts. This section deals more with AGIL's dynamics i.e., the specification of the dynamic relation of the systems AGIL aims to describe. In particular, the dynamics of service exchanges. Notice that we do not describe in this section the processes, but the realtions and the rules they must satisfy at a given time.

5.3.1 Agent interaction

A very important aspect of MAS is interaction. In AGIL, agent-agent interactions include all other kinds of interactions that may occur in GRID or MAS (Grid user-Grid service, Grid service-Grid service, agent-agent, etc.). This interactions are realized by means of asynchronous message passing between agents. The *interacting* relation formalizes this relation between agents (section 3.3.4.1). The *interacting* relation is represented in AGIL by a double thin arrow as figure 5.26 shows. There is two kind of interactions:

1. *Direct agent-agent interaction.* Messages are exchanged directly from agent to agent. These are interactions in a general sense, i.e., any interaction, standardized or ad hoc, protocol guided or not, semantically described or not, long-lived or one-shot, etc. These interactions may occur within a VO, but also outside it;

2. *Through-service agent-agent interaction.* They occur during service exchange. Messages are exchanged from agent to agent through a service. These are interactions that an agent may have with another agent, without directly communicating with the other agent but instead via the service interface this second agent offers in the VO's service container. These 'through-service interactions' occur only within a VO.
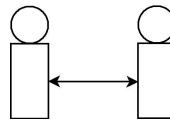
Figure 5.26: *Interacting* relation representation

Direct agent-agent interaction representation. It is not the aim of AGIL to model or detail the first type of interaction (that can be totally detached from GRID). This work is a current research domain, called agent communication, addressed by MAS community (section 2.3.3). The only assumption about this relation with STROBE agents is expressed by rule 7.

Through-service agent-agent interaction representation. The second type of interaction is on the other hand the main element of the service-based integration of GRID and MAS. Actually, from a pure MAS point of view, we should say that AGIL describes service exchange interactions in MAS; it does it using advantages of GRID mechanisms and principles. We say that agents may all together exchange several services. The *exchanging* relation formalizes the through-service agent-agent interaction. It can be decomposed in two sub-relations, *using* and *providing*, that formalize the relation between agents and

services. The *exchanging* relation is not directly represented in AGIL, however, it may be graphically simplified as figure 5.27 shows. The *using* and *providing* relations may be also simplified in AGIL as figures 5.28 and 5.29 show. Any agent exchanges with zero, one or several agents. Any service is used by zero, one or several agents. Any agent uses zero, one, or several services. Any agent provides zero, one or several services. Any service is provided by exactly one agent.

$$exchanging : A \to A \text{ (relation)}$$

$$using : \Sigma \to A \text{ (relation)}$$

$$providing : \Sigma \to A \text{ (application)}$$
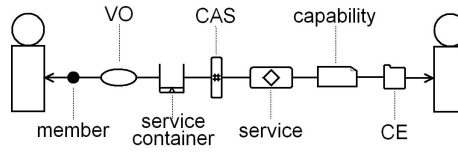
Figure 5.27: *Exchanging* simplified representation
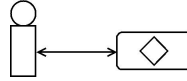
Figure 5.28: *Using* relation simplified representation
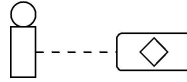
Figure 5.29: *Providing* relation simplified representation

In the following, $\forall a \in A$, we denote $exhanging(a)$ the set of agents interacting with a; $\forall \sigma \in \Sigma$, we denote $using(\sigma)$ the set of agents which use the service σ.

Rule 25 *The exchanging relation is symmetric.*

$$\forall a_1, a_2 \in A, a_1 \in exchanging(a_2) \Leftrightarrow a_2 \in exchanging(a_1)$$

Rule 26 *The exchanging relation is irreflexive.*

$$\forall a \in A, a \notin exchanging(a)$$

Rule 27 *Exchanging a service is a sub kind of interaction.*

$$\forall a_1, a_2 \in A, a_2 \in exchanging(a_1) \Rightarrow a_2 \in interacting(a_1) \wedge a_1 \in interacting(a_2)$$

Rule 28 *An agent uses a service only if it is member of a VO for which the service is available.*

$$\forall a \in A, \forall \sigma \in S \cup C, a \in using(\sigma) \Rightarrow \exists o \in O, a \in o \wedge authorizing(o) = handling(\sigma)$$

The previous rule comes directly from OGSA specifications. In OGSA a Grid user (called an agent in AGIL) has some right levels on the services available in his/her VO(s). Using a service occurs only within a VO. In AGIL, we should also detail the rules for service providing, because in the GRID-MAS integrated model, a service available within a VO is provided by an agent. Must this agent be member of the VO for which it provides the service? To answer to this question we should look back on the servicization process presented in section 5.2.3.3. When an agent servicizes one of its capability into a service available for a VO, it uses a set of services proposed by the VO e.g., add service, interfacing service, notification service, etc. In order to use these services, according to rule 28, it must be a member of the VO. This is expressed by rule 29.

Rule 29 *An agent provides a service within a VO only if it is a member of the VO and it has a capability that interfaces the service in the corresponding service container.*

$$\forall a = (b, E_a) \in A, \forall \sigma \in S \cup C, providing(\sigma) = a \Rightarrow \exists \lambda \in \Lambda, \exists o \in O, \exists e \in E_a, a \in o \wedge authorizing(o) = handling(\sigma) \wedge interfacing(\lambda) = \sigma \wedge executing(\lambda) = e$$

Therefore, within a VO, service exchanges occur when an agent uses a service another one provides. This is what append for all service exchanges in AGIL. This general case is expressed by rule 30.

Rule 30 *Two agents exchange a service only if they are members of the same VO and if one of them provides, in this VO, the service the other one uses.*

$$\forall a_1, a_2 \in A, a_2 \in exchanging(a_1) \Rightarrow \exists o \in O, \exists \sigma \in S \cup C, a_1, a_2 \in o \wedge authorizing(o) = handling(\sigma) \wedge a_1 \in using(\sigma) \wedge providing(\sigma) = a_2$$

Rules 28, 29 and 30 specify using, providing and exchanging of a normal service or a CAS. The case of service containers is special because containers are not included in containers.

Rule 31 *An agent uses a service container service only if it is a member of the corresponding VO.*

$$\forall a \in A, \forall u \in U, a \in using(u) \Rightarrow \exists c \in C, \exists o \in O, a \in o \wedge authorizing(o) = c \wedge including(c) = u$$

Remark – We do not formalize yet if an agent providing a service container service should be a member of the corresponding VO or not.

5.3.2 Service-capability instantiation

5.3.2.1 Mapping of Grid service instantiation and CE instantiation mechanisms

GRID introduces the notions of service factory and service instance. Basically, the service factory is a generator, like a class in object-oriented programming. The service instance is one of the many instances that can be generated by the service factory. Each service instance is running its own dedicated context with its own resource. For this reason this factory-instance model is well appropriated for managing service state. The distinction between a service factory and a service instance is not necessary in AGIL, because even a service factory is a service instance of another factory. What is important to address is then the bootstrap of this system: not all services are created by instantiation mechanism, some of them can have been created by core GRID functionality (out-of-band and bootstrap mechanisms).[9]

In MAS, when an agent has a conversation, it dedicates a part of its state to this conversation. It is called the conversation context. In the STROBE model, it is done by CEs. When a STROBE agent

---

[9]Notice that even if a service is created by an out-of-band mechanism, it can nevertheless be associated to an agent's capability.

receives a message from another agent for the first time, it instantiates a new local CE for this agent following three policies: (i) copying the global CE; (ii) copying a local CE; (iii) sharing a local CE. (section 3.3.4.3).[10]

In the GRID-MAS integration the key GRID idea of service instantiation is mapped with the STRO-BE's instantiation of CE mechanism. It means that instantiating a new service in GRID is equivalent to instantiating a new CE in MAS; the processes are the same thing viewed differently. The new CE contains the new capability and the service provider applies the servicization process on it in order to make available the new service instance for the service user(s). In order to map exactly the STROBE model with the WSRF specifications, we should say that a new CE can be viewed as a new WS-Resource i.e., a CE is a dedicated association between capabilities and stateful resources. In this sense STROBE agents map with GRID solution for C22[separation]. Integrating these two instantiation mechanisms make capabilities to benefit from standardization, interoperation and allocated resources from GRID, and Grid services to benefit from a dedicated context of execution and local conversation representation from MAS. The *instantiating* relation formalizes the relation between service-CE couples. The *instantiating* relation is represented in AGIL by a large arrow as figure 5.30 shows. Any service-CE couples is instantiated by at most one other couple. Any service-CE couple instantiates zero, one or several other couples.

$$instantiating : \Sigma \times E \rightarrow S \times E \text{ (function)}$$
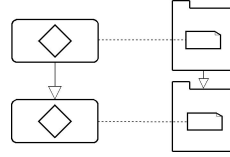
Figure 5.30: *Instantiating* relation representation

Rule 32 *Only normal services can be service factories.*

$\forall (\sigma_f, e_1), (\sigma_i, e_2) \in \Sigma \times E, instantiating(\sigma_i, e_2) = (\sigma_f, e_1) \Rightarrow \sigma_f \in S$

Rule 33 *A service-CE couple can be instantiated only by a unique couple.*

$\forall (s_1, e_1), (s_2, e_2) \in S \times E, \forall (\sigma_3, e_3) \in \Sigma \times E, instantiating(\sigma_3, e_3) = (s_1, e_1) \wedge instantiating(\sigma_3, e_3) = (s_2, e_2) \Rightarrow s_1 = s_2 \wedge e_1 = e_2$

Rule 34 *The instantiating relation (restricted to $S \times E$) is irreflexive i.e., a service-CE couple is not instantiated by itself.*

$\forall (s, e) \in S \times E, instantiating(s, e) \neq (s, e)$

Rule 35 *The instantiating relation (restricted to $S \times E$) is asymmetric i.e., a service-CE couple that instantiates another couple, cannot be instantiated by this couple.*

$\forall (s_1, e_1), (s_2, e_2) \in S \times E, instantiating(s_2, e_2) = (s_1, e_1) \Rightarrow instantiating(s_1, e_1) \neq (s_2, e_2)$

---

[10]The third situation is actually not an instantiation. A capability included in a partially dedicated CE is interfaced as a service available for several users.

Rule 36 *A service-CE couple instantiates another couple only if the two CEs execute the two capabilities interfaced by the two services.*

$\forall (s_1, e_1) \in S \times E, \forall (\sigma_2, e_2) \in \Sigma \times E, instantiating(\sigma_2, e_2) = (s_1, e_1) \Rightarrow \exists \lambda_1, \lambda_2 \in \Lambda, executing(\lambda_1) = e_1 \wedge executing(\lambda_2) = e_2 \wedge interfacing(s_1) = \lambda_1 \wedge interfacing(\sigma_2) = \lambda_2$

Remark – The service factory and the service instance are not necessary provided by the same agent because of the reproducing situations (next section).

DEFINITION: INSTANTIATION PROCESS

In AGIL, the process of creating a new service-capability couple by mapping the Grid service and STROBE's instantiation of CE mechanisms. The new cognitive environment created serves as a conversation context to execute the dedicated service.

5.3.2.2 Different cases of instantiation

Figure 5.31 shows three kinds of instantiation that GRID enables. These three cases of service instantiation may be commented at the light of the integration:

1. A service factory instantiates a new service instance in the same service container. The classical case. The service provider agent instantiates a new CE including the new capability and servicizes this capability in the same service container. The new service becomes available for the service user agent.

2. A service factory instantiates a new service container. This is a core GRID situation. It supposes the agent which provides the service container factory is able to access or request GRID middleware to create a new service container, as well as the associated VO (B on figure 5.31), and to affect the VO's virtualized resources. Similarly, another core GRID situation (not represented in figure 5.31) is the instantiation of a CAS service instance. Allowing these core GRID functionalities to be realized by agents is a powerful aspect of the AGIL's model. It realizes a part of the MAS based GRID approaches presented in section 2.4.4.2.

3. A service factory instantiates a new service instance in another service container, via some authorization provided by the CAS. In this case the service factory provider agent is necessarily member of A and B.

   Two cases are possible:

   - If the service factory user agent is also a member of B, it may ask a service instance for itself and/or other agents members of B;

   - If the service factory user agent is not a member of B, it may ask a service instance for some other agents members of B, but not itself.

In AGIL, whatever is the situation, the service factory is always a normal service. And the service instance is always an allocated interface of an agent's capability.

Remark – GRID enables also to instantiate services in sequence: a service factory instantiates a new service factory and so on. Sequences of instantiation are still possible in AGIL, because the STROBE's instantiation mechanism enables any kind of CE to instantiate a new one according to the agent policy.[11]

---

[11]Notice however these sequence of instantiations are not ordinary in GRID as well as in MAS.
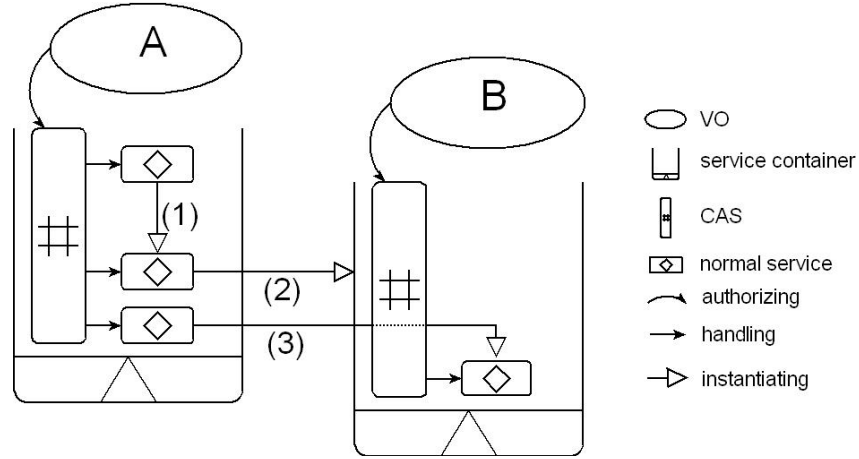
Figure 5.31: Examples of different kind of instantiations in Grid service container

5.3.3 Agent reproduction

STROBE agent reproduction was presented in section 3.3.4.5. This situation is still valid in AGIL. When created, the child agent is a member of the same VOs than its father. This situation is illustrated in figure 5.32, which completes figure 3.10.
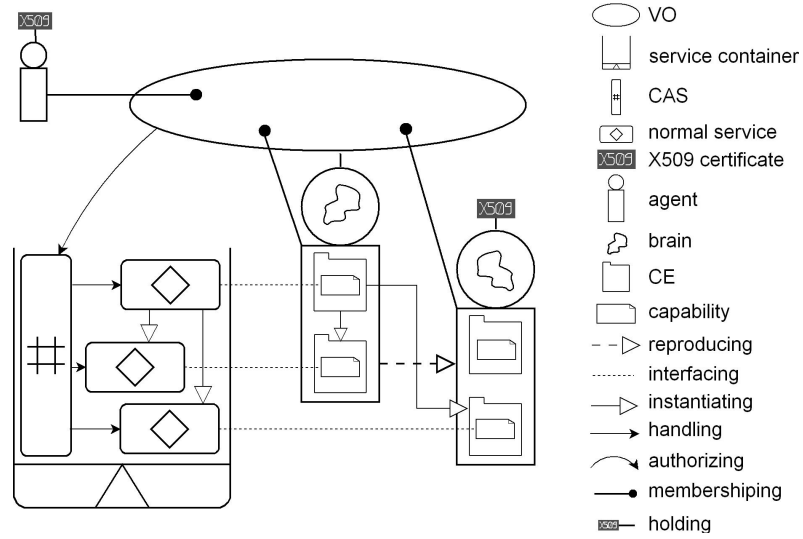
Figure 5.32: Examples of agent reproduction

Rule 37 *A child agent is member of the same VOs than its father.*

$$\forall a_1, a_2 \in A, \forall o \in O, reproducing(a_1) = a_2 \land a_2 \in o \Rightarrow a_1 \in o$$

Remark – We may have limited this rule to the VO in which the agent reproduction occurs.

Rule 38 *An agent reproduces itself only in an interactive situation with another agent. It does it because it already has a local CE dedicated to this interlocutor.*

$$\forall a_1 = (b_1, E_1), a_2 = (b_2, E_2) \in A, reproducing(a_2) = a_1 \Rightarrow \exists a_3 \in A, a_3 \in interacting(a_1) \land a_3 \in$$
$$interacting(a_2) \land \exists (s_f, e_{1_i}) \in S \times E_1, \exists (\sigma_i, e_{1_j}) \in \Sigma \times E_1, \exists (\sigma_2, e_2) \in \Sigma \times E_2, instantiating(\sigma_{i_1}, e_{1_j}) =$$
$$(s_f, e_{1_i}) \land instantiating(\sigma_2, e_2) = (s_f, e_{1_i}) \land a_3 \in dedicating(e_{1_j}) \land a_3 \in dedicating(e_2) \land$$

$$providing(s_f) = a_1 \wedge providing(\sigma_2) = a_2$$

Remark – We do not detail here the exact mechanism that allows an agent to reproduce. It may be an ad hoc mechanism for different GRID-MAS integrated systems (e.g., how an agent global CE is created, what are is brain modules, etc.) We just formalize here the set of relations that exists in such a situation.

5.3.4 Service adaptation

What is important in this integrated model is to consider how a service can be adapted by a service provider agent for a service user agent, in order to implement DSG. We sum up here four ways available within AGIL. The two firsts directly benefit from the STROBE model side of AGIL:

1. The service provider agent adapts the service according to its interactions with service user agent with it. This kind of adaptation is illustrated, for example, by the e-commerce experimentation of section 3.4.2 (dynamic specification).

    For example, the SNCF agent servicizes its `find-ticket` capability, allowing the customer agent to express its constraints, following the service interface. Each time the SNCF agent receives a message from the customer agent it changes the dedicated find-ticket function and eventually refreshes the servicization process.[12]

2. The service provider agent may offer another service to change or adapt the original service. This kind of case are close of the scenario presented in section 3.4.1.2.

    For example, a service provider agent may offer a `<and>` service, able to return the value of an `and` logical expression (using a Scheme program). It may also offer an `<scheme-eval>` service, able to evaluate any Scheme expression. A service user, by using the `<scheme-eval>` service, may change the Scheme interpreter (e.g., making it a lazy one) using techniques presented in chapter 3, and therefore change the results of the `<and>` service.

3. The service provider agent may use dynamic intelligent reflection rules or algorithms to change the service it is currently providing. This is the same type of learning/reasoning than the one presented in section 3.3.4.4. We do not detail any example, but we refer to the intelligent ability of agent, historically inspired from AI;

4. Direct agent-agent interactions may occur between the service user agent and the service provider agent and within these interactions (1) and (3) may occur in a pure ad hoc form (not via service). This is because AGIL is not a close model. Any solution to service adaptation proposed independently of AGIL is not prevented by the model and may occur in direct agent-agent interaction (not addressed in this thesis).

5.4 Towards an OGSA-STROBE model

5.4.1 Summary of the integration

In our integrated model, we consider agents exchanging services through VOs they are members of: both service users and service providers are considered to be agents. They may decide to make available some of their capabilities in a certain VO but not in another. The VO's service container is then used as a service publication/retrieval platform (the semantics may also be situated there). A service is executed by an agent but with resources allocated by the service container. Figure 5.33 shows the complete integrated model described in AGIL.

---

[12]Remember the servicization process was defined as a continuous process, allowing the service to exactly follows the corresponding capability.
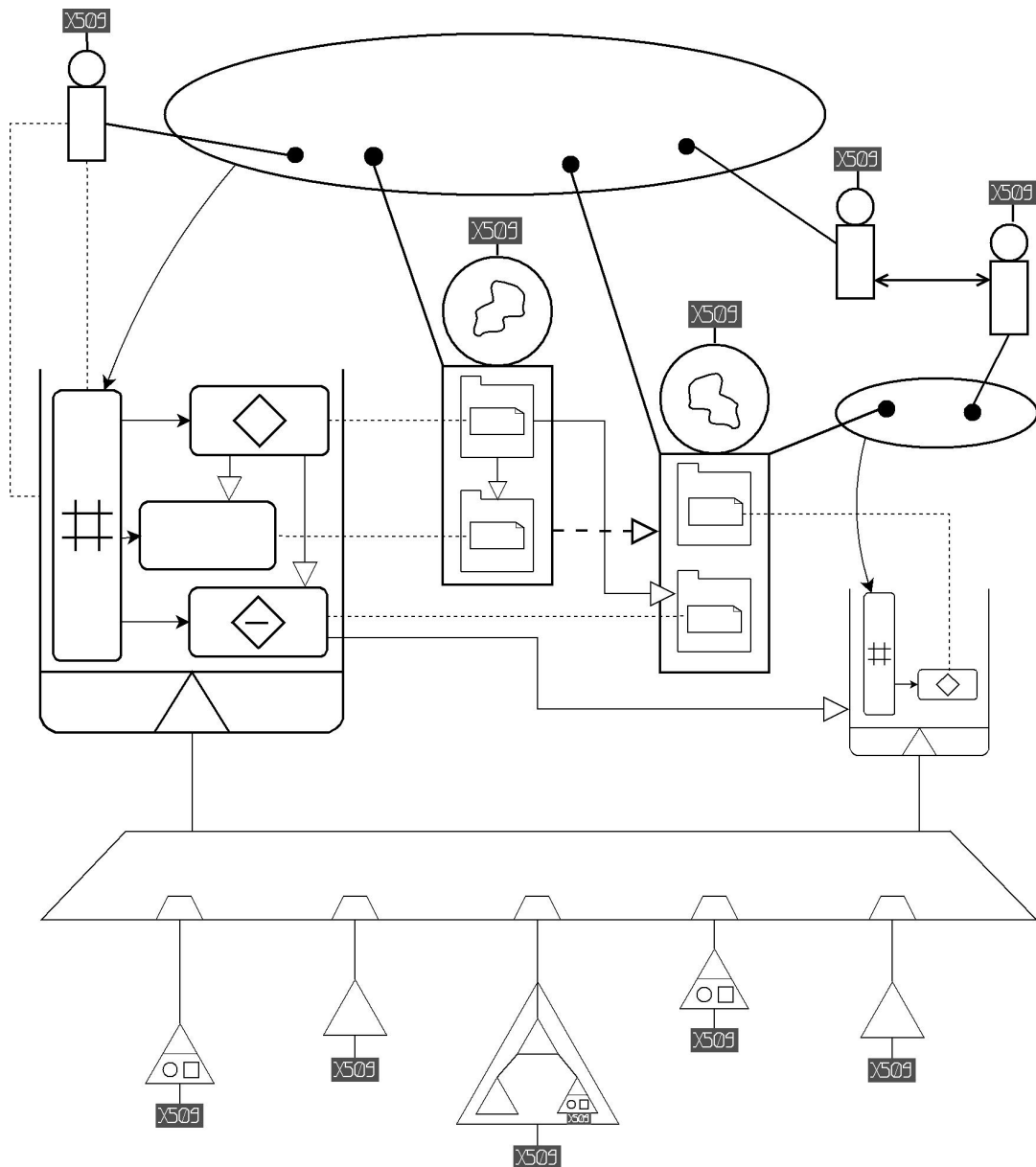
Figure 5.33: The AGIL's GRID-MAS integrated model

5.4.2 Simple example

Figure 5.34 shows some of the same elements than figure 5.33 but illustrated on a concrete simple example. Let us consider a VO of four users A, B, C and D. D servicized in the VO's service container the `<incr_count_factory>` (S1) and `<decr_count_factory>` (S2) services, corresponding to its capabilities of incrementing and decrementing a given counter. C wants to use alone D's incrementing service; A and B want to use together D's incrementing service and decrementing service. Figure 5.34 shows both the VO's service container and the D agent's body. Some services and CEs have been instantiated. S1 and S2 correspond to D's capabilities (located in E_D^D), these service factories are accessible by all agents of the VO (`1 1 1 1` line in the CAS[13]) S3 is a service instance accessible only by agent C and thus corresponding to a capability stored in a local CE dedicated totally to C, noted E_C^D. Notice that the local `cc` variable has for value 3, which means the `<incr_count_inst2>` service was used three times by agent C. S4 and S5 are both service instances accessible by A and B and thus corresponding to two capabilities stored in a local CE dedicated to A and B, noted $E_{A,B}^D$. The local `cc` variable has for value 8. Some important remarks may be done on this simple example:

- A and B share the same counter services. It means that if A changes the state (i.e., the counter value) by using the incrementing or decrementing service, it will have an influence on the next use of the services by B;

- A CE instantiation copies all the included capabilities. Therefore, the decrementing capability was created in E_C^D even if D not servicized it as a service in the VO's service container. It may be made later if C asks for a decrementing service;

- S3 and S4, are instances of the same service factory but do not exactly do the same thing. S4 was adapted by D for A and B in order to print the new counter value after incrementing it. S3 stay the same as the service factory. This adaptation may have been done by the four methods cited in section 5.3.4.

(S1) `<incr_count_factory>` : cc:=cc+1;

(S2) `<decr_count_factory>` : cc:=cc-1;

(S3) `<incr_count_inst1>` : cc:=cc+1;

(S4) `<incr_count_inst2>` : cc:=cc+1; print(cc);

(S5) `<decr_count_inst1>` : cc:=cc-1; print(cc);

E_D^D D's global CE

E_C^D D's local CE dedicated to C

$E_{A,B}^D$ D's local CE dedicated to A and B

$\frac{cc}{0}$ CC local variable binding
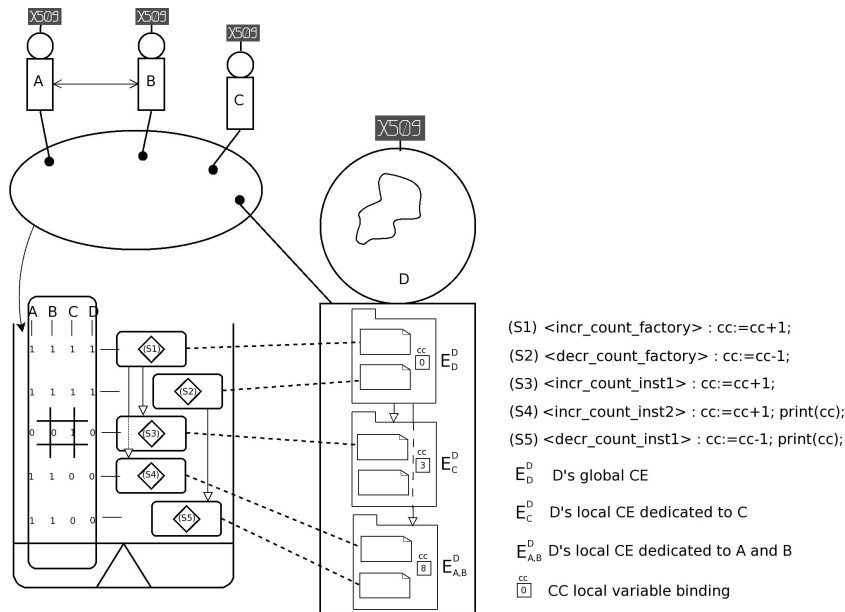
Figure 5.34: Example of the GRID-MAS integration

---

[13] In a sake of simplicity, on this simple example, only two right levels (levels of permission) are considered: accessible (`1`) or not (`0`).

Comparison with WSRF.   Figure 5.35 shows the same example but according to WSRF specification. A WS-Resource is an association between a stateful resource and a stateless service instantiated by a WS-Factory. In the example, the functionalities `cc:=cc+1` and `cc:=cc-1` represent the stateless services and the variable `cc` represent the stateful resource. The S1 WS-Factory instantiates two different WS-Resources (S3 and S4) which share the same stateless service but associated with different stateful resources (represented by a cylinder). The S2 WS-Factory instantiates only one WS-Resource (S5) which shares the same stateful resource as S4 but with another stateless service.

Notice that the incrementing service shared by S3 and S4 is unavoidably the same i.e., with WSRF a service may be involved in different WS-Resources, but it has to remain the same service for all the WS-Resources. A WS-Resource has thus only one part that can be really dedicated to its user(s). Integrating WSRF with agents allows to have the stateless service executed by an intelligent agent, which is already a good thing. However, using STROBE agents is actually more interesting because, thanks to the local CEs, a service, that interfaced a capability of a local CE, can be dedicated to the user(s). With the STROBE model, we always have an association between a capability (stateless) and a CE (stateful), several capabilities can share the same CE, but the capability is always dedicated as well as the CE. It is the fundamental difference coming from using STROBE in our GRID-MAS integrated model: a STROBE agent can make its dedicated capabilities evolve differently (following experience, learning algorithm, etc.). Then in our approach S3 and S4 can evolve differently to fit better service user's needs or wants, such as for example print the counter value. This kind of dedication of the stateless service are not possible with WSRF.
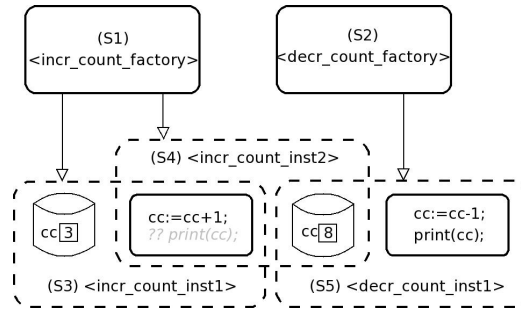
Figure 5.35: WSRF elements corresponding to the example

5.4.3   Discussions and advantages for SOC, MAS and GRID

- According to the STROBE agent's policy, when a new local CE is instantiated, all capabilities of the father CE are copied in the child CE; only some of them are servicized. Even if these capabilities are not used yet, they may help the STROBE agent in combining, or composing, its capabilities one another. Later, they may be servicized.

- There is no real standard in the MAS community to describe agents' capabilities between different agents or MAS. The integration will help MAS developers in presenting and interfacing agents' capabilities, and therefore augment MAS interoperation.

- This integrated model does not restrict MAS or GRID in any way. In particular, it does not prevent direct agent-agent interactions and thus, for example, it does not prevent agents to perform tasks to one another in a purely ad hoc manner. This is important if we want the integration to be followed by other MAS approaches and models (different from STROBE); these models can keep their internal formalisms for their internal operations.

- In this integration, VO management benefits from both GRID and MAS organizational structure formalism, e.g., AGR, CAS service, X509 certificate, etc.

- Service exchange in this integrated model benefits from the important agent communications abilities, e.g., dealing with semantics, ability to hold a conversation, etc. The challenge of modelling conversation not by a fixed structure (interaction protocol) but by a dynamic dialogue in MAS becomes the same one that dynamically create high level process of services in SOC/GRID.

- AGIL subsumes a significant number of the MAS-based GRID approaches cited in section 2.4.4.2. Indeed, thanks to the reflexivity of GRID, which defines some core GRID functionalities as (meta-)Grid services (e.g., service container, CAS), we may consider these core GRID functionalities as executed also by agents. This establishes an important part of the MAS-based GRID approaches which use MAS techniques to enhance core GRID functionalities.

- The GRID community may appreciate the key GRID concepts formalization done by AGIL. Indeed, without considering the MAS side, AGIL clearly defines and simplifies them even if it is not complete – some GRID concepts are not represented and relations are quite simplified.

5.4.4 Agent-Grid Integration Ontology

In [DJDC] we propose a formalization of AGIL's concepts, relations and integration rules by means of an ontology, called *Agent-Grid Integration Ontology* (AGIO). The ontology could be used both to model the behaviour of GRID-MAS integrated systems and to check the consistency of these systems and their instances. AGIO describes the elements involved in through-service agent-agent interactions. It is actually a meta-description, allowing agents to agree on what they are, what a service is, a host, etc. Moreover, relations are strengthened by Semantic Web Rule Language (SWRL) rules which guarantee coherence [OKT$^+$05].

Using an ontology to describe the AGIL's integrated model is interesting because we can describe using the same formalization both the model and its instances. Designers of GRID-MAS integrated systems may instantiate AGIO concepts in order to formalize their systems and check their consistency thanks to AGIO implemented rules. On a given set of instances one can check the consistency thanks to the SWRL rules implemented in AGIO e.g., if exists the right set of instances for a given relation.

Moreover, this description may be 'understood and processed' by both HAs and AAs as a consequence of a OWL/Protégé implementation briefly addressed in section C.4.2. This implementation gives us another AGIL formalization, different from a set-theory or a graphical language. Formalizing by means of an ontology is a choice often made in SOC, MAS and GRID communities to represent knowledge because is a method that AAs may understand.

5.5 Application scenario modelled with AGIL

In this section, we present a scenario modelled and specified with AGIL. This section intends both to illustrate the potentiality of AGIL as a description language and to show how such a scenario can benefit from the GRID-MAS integration. The chosen scenario was identified in section 1.1 as an example of situation requiring DSG: looking for a job. We first identified the actors of the scenario, their goals, the service they provide or use, etc. Then, we detail the service exchanges that occurs within this scenario. Afterwards, we represent them with AGIL.

5.5.1 Elements of the 'looking for a job' scenario

Let us call the job seekers of the scenario, *Helen* (H) and *Bob* (B), two human agents. Helen looks for a precise position as a project manager in an international company, whereas Bob looks for a general seller

job in a store close to his home. *Assister* (A) is an assistant artificial agent which can help job seekers by providing them an `<offer-watching>` (OW), service, watching for new job offers proposed by job agencies. *Roger* (R) is the staff manager of a company; his role consists in posting job offers in job agencies. *Jobwiner* (J) is a job agency artificial agent which collects different job offers. It proposes a `<search-offer>` (SO) a `<post-offer>` (PO) services which enable respectively a user to find a job offer in its database, and post a new job offer. The last actor of this scenario is a travel guide, said *Mappy* (M), an artificial agent which proposes a `<distance>` service able to return the distance between two places. With these agents, we can imagine the following set of service exchanges:

- Bob and Helen uses Jobwiner's service for their job searchs;

- Helen uses also Assister's service to support her for her job search;

- Assister uses Jobwiner's service to realize its service for Helen;

- Bob wants to consult only the job offers, for which job addresses are distant less than 10kms from his home. Jobwiner adapts the service it provides Bob with by using Mappy's service.

- Roger uses Jobwiner's service to post his job offers.

These service exchanges may occurred in two identified VO. *Employment* (E), that regroups all the actors playing a role in job research; *Providers* (P), that regroups many kinds of service utilities providers. All the agents, except Mappy, are members of Employement; Jobwiner and Mappy are members of Providers.

5.5.2 AGIL representation

Figure 5.36 illustrates these services exchanges in AGIL. Table 5.1 details the AGIL's set values for this scenario. Notice that we do not precise in this example scenario the Grid resource level i.e., computing resource, storage resource, hosts and virtualized resources.

With these elements, any relations of the scenario can be described by AGIL's notations, for example:

$$u_E \in reifying(R_E)$$

$$handling(s_{D1}) = c_P$$

$$instantiating(s_{OW1}, E_H^A) = (s_{OW}, E_A^A)$$

$$a_J \in exchanging(a_B)$$

$$o_P \in membershipping(a_M)$$

One can check the consistency of this GRID-MAS integrated system thanks to AGIL's rules. For example, rule 29:

$providing(s_{OW1}) = a_A \Rightarrow authorizing(o_E) = handling(s_{OW1}) \land interfacing(\lambda_{OW}{}_H^A) = s_{OW1} \land executing(\lambda_{OW}{}_H^A) = E_H^A$

Table 5.1: AGIL's set values in the 'looking for a job' scenario

$R = \{R_E, R_P\}$

$A = \{a_B, a_H, a_R, a_J, a_A, a_M\}$

$O = \{o_E = \{a_B, a_H, a_R, a_J, a_A\} \cup o_P = \{a_J, a_M\}\}$

$S = \{S_E = \{s_{OW}, s_{OW1}, s_{SO}, s_{SO1}, s_{SO2}, s_{PO}, s_{PO1}\} \cup S_P = \{s_D, s_{D1}\}\}$

$$C = \{c_E = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}, c_P = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\}$$

$U = \{u_E = \{R_E, c_E, S_E\}, u_P = \{R_P, c_P, S_P\}\}$

$\Sigma = U \cup C \cup S$

$B = \{b_B, b_H, b_R, b_J, b_A, b_M\}$

$E = \{E_B^B, E_J^B, E_H^H, E_A^H, E_J^H, E_R^R, E_J^R, E_J^J, E_B^J, E_M^J, E_{A,H}^J, E_R^J, E_A^A, E_H^A, E_J^A, E_M^M, E_J^M\}$

$\Lambda = \{\lambda_{OW}{}_A^A, \lambda_{OW}{}_H^A, \lambda_{SO}{}_J^J, \lambda_{SO}{}_B^J, \lambda_{SO}{}_{A,H}^J, \lambda_{SO}{}_R^J, \lambda_{PO}{}_J^J, \lambda_{PO}{}_B^J, \lambda_{PO}{}_{A,H}^J, \lambda_{PO}{}_R^J, \lambda_D{}_M^M, \lambda_D{}_J^M\}$

$X = \{x_B, x_H, x_R, x_J, x_A, x_M\}$
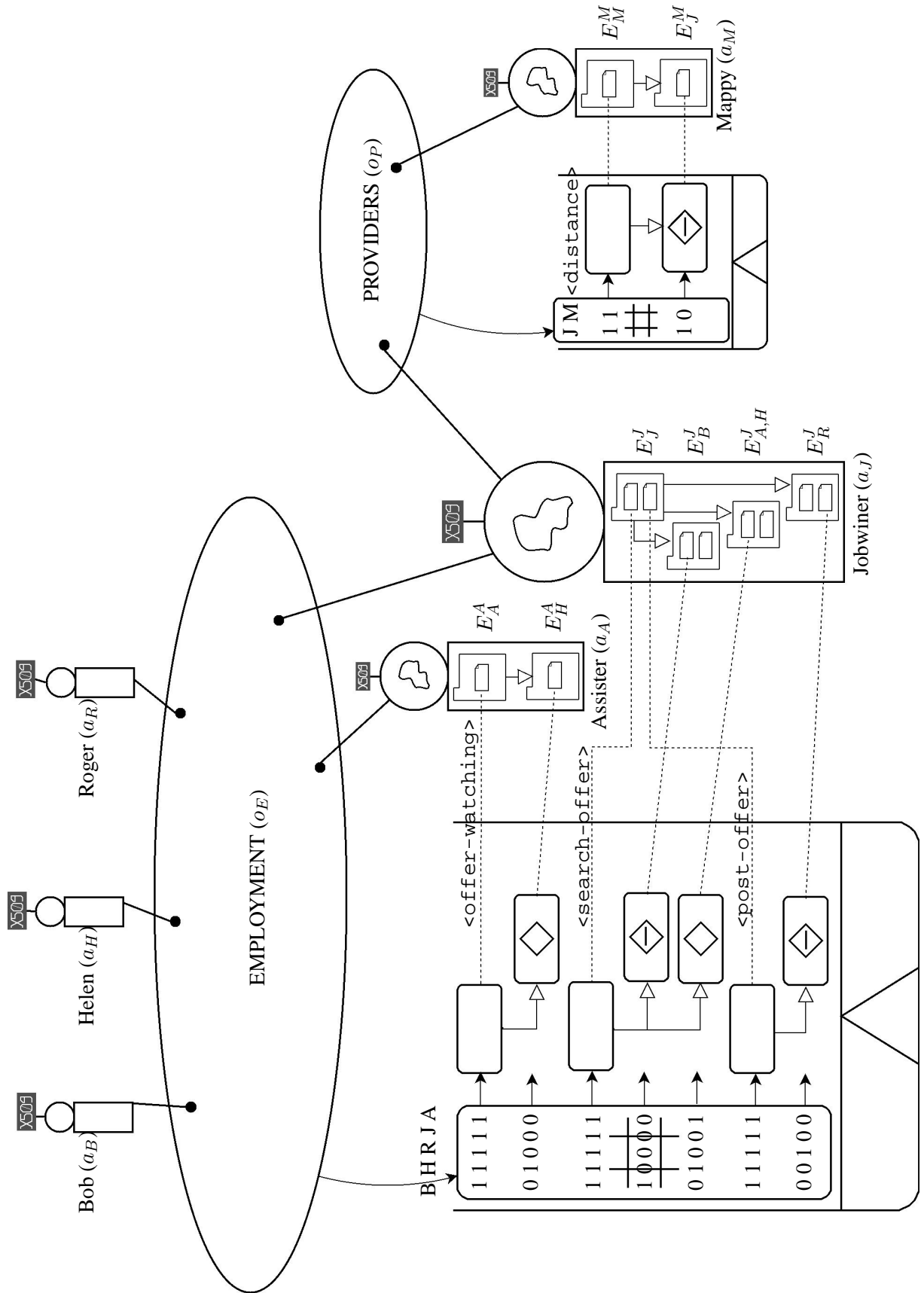
Figure 5.36: Representation of the 'looking for a job' scenario in AGIL

5.5.3 Discussion

This example illustrates several aspects of the GRID-MAS integrated model. It is not an example of DSGS, but an example of GRID-MAS integrated system, represented with AGIL. In this sense this kind of system addresses some characteristics of DSGS but not all. We view here how the different mechanisms introduced in this thesis may be-reused in this simple scenario. The added value of such mechanisms for the 'looking for a job' scenario includes:

- Each services-capability is executed in a specific dedicated context with its own state. States manipulated by Jobwiner to provide the `<search-offer>` service are not the same.

- Jobwiner realizes a dynamic composition of its own `<search-offer>` service with the `<dist-ance>` service of Mappy. Notice this kind of service composition is the same that the one in section 4.3.4, using i-dialogue. Jobwiner may use i-dialogue, with the two CEs, E_B^J, E_M^J, to model the conversation it has with Bob and Mappy. The same kind of dynamic composition of services is also done by Assister.

- This composition was realized after a meta-level learning of Jobwiner: INT_B^J was modified in order to produce a message for Mappy and INT_M^J was modified to produce a message for Bob. This service adaptation may occur following the ways summed up in section 5.3.4. Its can be done directly by Jobwiner, or it can be suggested (such as in the teacher-learner scenario of STROBE, section 3.4.1) by Bob.

- Jobwiner uses the same CE ($E_{A,H}^J$) for the conversation it has with Assistant and with Helen. For it, they are the same entity using the same service. No matter if $\lambda_{SO}{}_{A,H}^J$ capability is executed for Helen or Assistant, this CE stores Helen's requirements for searching job offer.

- Jobwiner's procedures implementing its capabilities $\lambda_{SO}{}_B^J$ and $\lambda_{SO}{}_A^J$ should be implemented as the `find-ticket` procedure of section 3.4.2. They may use dynamic specification mechanisms i.e., non-deterministic interpretation and constraint expressions to implement the service they provide.

- If Jobwiner needs to use another instance of `<distance>` service, then Mappy would reproduce in M', in order for Jobwiner to keep only one local CE dedicated to Mappy.

5.6 Conclusion and perspectives

Identifying key factors to demonstrate the convergence of MAS and GRID models is not an easy task. We pointed out that the current state of GRID and MAS research activities is sufficiently mature to enable justifying the exploration of the path towards an integration of the two domains. Besides describing why GRID and MAS need each other, we explained how they can become synergic. The main contributions of the chapter are:

- Through an analysis of concrete models (mainly OGSA and STROBE), we extracted from the previous chapters a few key concepts of GRID and MAS;

- Inspired by the analogies and related work, we proposed an integrated model that respects all the constraints and foundations of both GRID and MAS;

- We proposed a new set-theory formalization and a graphical description language, called AGIL, to describe the GRID-MAS integrated model;

- We illustrated on an example how this formalization allows to describe GRID-MAS integrated systems.

AGIL raises the challenge to be both a model for GRID-MAS integrated systems and a description language for those systems. The graphic symbols and the set-theory formalization address together this double objective.

The set-theory formalization consists of a set of concepts, relations between them and rules. Their first role is to rigorously formalize the AGIL's model. It may be useful in order to specify some of the mechanisms occurring in the model as it is illustrated in section 5.2.1.2, on the evaluation of the amount of virtualized resource reified for a service container. Moreover, they can also be used to formalize instances of the model and determine if those GRID-MAS integrated systems are consistent with the model. These three elements directly express the semantics of GRID-MAS integrated systems e.g., agent A provides service S will be expressed *providing(S)=A*. An example was presented with the 'looking for a job' scenario, in the previous section. Besides, the adopted notation allowed the model to evolve and new concepts, relations and especially rules to be added.

The graphical description language gives to each AGIL concepts and relations a graphic symbol that can be used in different diagrams. This language is less powerful than the set-theory formalization, because the rules cannot be expressed graphically. However, it has some important benefits. In particular, it formalizes the model (such as UML) and specifies a reusable and pertinent graphical language i.e., both the model and the instances of this model can be represented by AGIL (this is illustrated respectively by figures 5.33 and 5.36. These graphical representations are an easy way to illustrate and define a GRID-MAS integrated system. It is also very didactic.

Even if AGIL does not give yet an answer to all the questions concerning GRID-MAS integration, it proposes a rigourous way to formalize these systems. Any new answer to the remarks done in the chapter (concerning X509 certificates, service container, hosts) or any new rule may be expressed later.

At the core of this integration is the concept of service. The bottom-up vision of service in GRID combined with the top-down vision of service in MAS bring forth a richer concept of service, integrating both GRID and MAS properties. In this chapter, we put this enhanced concept of service into the perspective of DSG. Integrating SOC, MAS and GRID allows to address some characteristics of DSG intrinsic to GRID such as C20[grid], C18[community] or C8[SOA]. Moreover, the integration of GRID and MAS enhances the WSRF approach to address C22[separation]. The separation between capabilities and their execution context (CE) matches with the WSRF model of separating stateless service and stateful resource. Using STROBE agents allows both the stateful resource (i.e., the CE) and the stateless service (i.e., the capability executed in the CE) to evolve dynamically.

From a simplified point of view, the concepts of the GRID-MAS integrated model can be summarized by analyzing the interactions in this model (figure 5.37). **The AGIL's integrated model may be seen as a formalization of agent interactions for service exchange on the Grid.** The most significant contribution of the model links together all the elements of figure 5.37, it is the relation introduced between a Grid service instance and its state and an agent's capability and its context (the *interfacing* relation). OGSA fits well for the first part (left hand side) of the interaction and the STROBE model fits well for the second part (right hand side).
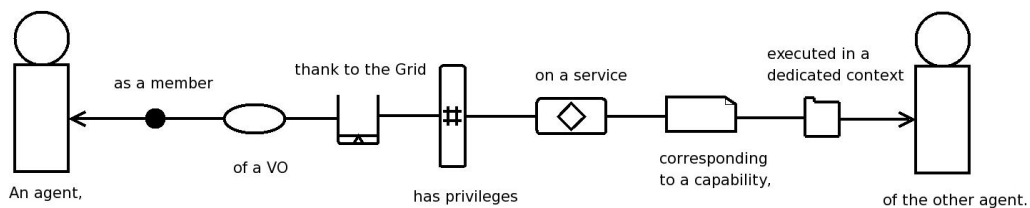
Figure 5.37: AGIL as a model of service exchange interaction

The integration proposed in this chapter is feasible considering today's state of SOC, MAS and GRID technologies. Future developments in GRID and MAS may leverage from this integration in order to progress. For example, here are aspects of the STROBE model and some aspects of the OGSA model that need to evolve:

Better support for more than one interlocutor dedicated for a CE. Most of the scenarios using the STROBE model take as a rule that STROBE agents must have a local CE dedicated to a group of only one interlocutor. However, this viewpoint is not adequate for a GRID-MAS integration because it would imply that each service instance in a service container can be used by one and only one agent. Therefore, the STROBE model needs to evolve to fit better the cases where a local CE is used for a group of interlocutors. This is part of a larger perspective proposing within AGIL a real mapping between organizational structures of GRID and MAS.

Support for non-synchronous protocols and semantics. GRID and SOC must evolve to fit better agent communication properties. In particular, the question of synchronicity (HTTP and thus SOAP are still synchronous communication protocols) and the question of semantics (which starts with the work of the Semantic Web community) need to be addressed.

Following the evolutions of SOC, GRID and MAS, AGIL will have to evolve as well. Even today, a number of perspectives may be highlighted. For example:

- To introduce into AGIL more concepts coming from (i) GRID specifications such as certification authority, trust, etc; (ii) MAS approaches when they are considered as standard e.g., environment (surrounding world of an agent) or objects in that environment, etc;

- To add into AGIL new rules, corresponding to non yet answered questions, or to new aspects that GRID-MAS integrated systems will develop;

- To propose AGIL extensions such as the representation of an agent's capability by several services in different service containers e.g., an agent may want to use a given service in two different VOs it is a member of;

- To detail the processes occurring in the AGIL's model e.g., the capability-service instantiation, we did not really specify the process behind such an instantiation. This could be done by using a Process Description Language (PDL) such as the ones introduced in section 2.2.2.3;

- To give a better specification of how core GRID functionalities are executed by agents (using work presented in section 2.4.4.2);

- To continue the implementation of AGIL as an ontology, that may rigorously check consistency of the modelled GRID-MAS integrated systems.

Finally, the GRID and MAS communities, mainly industrial for the former and mainly academic for the latter, have addressed the question of services in distributed systems from completely different angles and have thus developed different complementary aspects. Integrating these aspects according to the guidelines given in this chapter seems to us a good way to capitalize past, present and future work in order to progress towards DSG.

Conclusion

Our knowledge can only be finite, while our
ignorance must necessarily be infinite.
Karl R. Popper

This thesis is a reflection about the concept of service in Informatics. We proposed an integration theoretical framework called Dynamic Service Generation that suggests new directions in service exchange modelling. We adopted a characterization approach that helps us both to elicit DSG main aspects and to open a new path towards tomorrow's DSG systems. DSG served us as a large theoretical and challenging perspective. DSG characterization gives us an idea of the problem to solve. We chose some of the identified characteristics and we tried to propose models and mechanisms to address these characteristics. These experimental results are sometime called proof of concept, as we explained in the thesis introduction. They aim to demonstrate that our approaches reach their objectives and that an application of these results is realizable. These theoretical solutions have a strong potential considered on simple, but significant experimentations or scenarios.

Considering DSG as it was presented in chapter 1 and 2, we finally achieve only some preliminary solutions. Finally, simplified to its maximum, we should say that our analysis revealed that service exchange in computing is currently not enough interactive. This is a proof to the thesis statement of introduction (table 2. Therefore, we proposed a model of interaction for service-exchange. This model is SOC, MAS and GRID oriented. The thesis contribution can be summarized by the thesis subtitle: a formalization of agent interactions for service exchange on the Grid. This is illustrated in figure 5.37. We identified in chapter 1 conversation as the key aspect of DSG. We formalized in chapter 5 the service exchange interaction between two agents. The thesis main results may be summarized as follows:

- One analysis about the concept of service that defends an integration of SOC, MAS and GRID, and gives to these domains a common theoretical framework;

- The proposal of a new agent communication and representation model, interaction oriented, that gives agents dedicated dynamic conversation contexts and interlocutor's models;

- The proposal of a new computing abstraction that uses streams and lazy evaluation to model multi-party dialogues among agents;

- A GRID-MAS integrated model rigorously formalized which defines key GRID and MAS concepts as well as their relations and the rules of their integration;

- A formalization of agent interactions for service exchange on the Grid.

185

These results are an answer to the 'how to do it' question raised in the thesis introduction. They reflect our global methodology that privileged simple and concrete solutions in order to address large class of complex problems. The reusability of these results may be also summarized as follows (respectively):

- The analysis is not directly reusable, however it may open some research directions. Complementarily, the state of the art may offer a good overview of these three domains and their integration;

- The STROBE model may be implemented in any platform or language allowing its internal constructs to execute effectively (e.g., dynamic modification of an interpreter, CSP, stream, etc.). We illustrate this kind of implementation in Scheme and on the MadKit [GFM00] multi-agents platform in appendix C.

- The i-dialogue abstraction should be used inside and outside the STROBE model. The function may be implemented in any language that supports streams, lazy-evaluation and higher-order procedures. We give an example of such an implementation in Scheme in section C.3;

- The GRID-MAS integrated model has to be implemented. However, AGIL may already be used to describe and formalize GRID-MAS integrated systems. The Agent-Grid integration ontology that formalize AGIL by means of an ontology, described in section C.4.2, may already be used as a description understandable by HAs and processable by AAs, which can automatically check system consistency;

- The proposed formalization of agent interactions for service exchange on the Grid is reusable modulo the four previous points.

Integration played an important role in our choices. We tried to always keep in mind the fact that, as explained in the conclusion tyo chapter 2, even if the number of domains and the set of Informatics applications augments rapidly, the number of real problems keeps quite constant. Simplification and a sharp focus on real problems are necessary to progress, by linking and integrating results emerged from different approaches. For example:

- DSGS defined by opposition with PDS (section 1.3.3);

- CEs defined by analogy with programming languages execution contexts (section 3.3.2);

- Meta-level learning as the transfer of the techniques of language enrichment of applicative/functional programming (section 3.4.1);

- Dynamic specification which used CSP principles (section 3.4.2);

- I-dialogue as a use of streams, lazy evaluation and higher-order procedures to model agent conversation (chapter 4);

- AGIL as a GRID-MAS integrated model (chapter 5).

Working on SOC, MAS and GRID, we have demonstrated the importance of considering interactions as the core concept in future applications. A strong interaction based system might be the only way to integrate humans, then users, in the loop. We are now facing new kinds of problems that only interaction can help to resolve. This social approach to Informatics (i.e., interaction centred), as well as to AI or cognition, sometimes called Social Informatics, may be considered highly promising because in these agent societies we are allowed to envision, from the beginning, a partition of the responsibilities among members according to their so called 'best' capacities, i.e: motivation, trust, reflection, etc. for HA and remembering, accessing, computing, transmitting, etc. for AA, then looking for the system evolution. Humans will not 'use' computer anymore but consider them as members of the 'team' achieving an

effective collective dynamic behaviour that is not just the sum of each member's pre-coded contribution i.e., the whole is greater than the sum of the parts.

The non-rigorous specification of DSGS is also a choice in the scientific process. We were not guided by an accurate goal, but by the elicitation and exploration of a need. We did not determine a subject three years ago for which we bring a solution today, but on the contrary, we applied to our scientific process the same ideas as the ones we outlined as the result of this process (i.e., DSG). It means that we tried to apply in our own methodology the same principles that we put forward for DSG. DSG should be formalized step-by-step and should permanently evolve to reconsider new perspectives in service exchange that will appear. In other words, we should dynamically generate DSG!

Perspectives. We finally present hereafter some perspectives to our work. They correspond both to research aspects that we should have followed, as well as new ideas generated by the work we have accomplished. These perspectives are complementary to the ones we mentioned at the end of the chapters. Some of the general perspectives are:

- Explore solutions to address the already identified DSG characteristics;

- Continue the DSG characterization process, which of course is not terminated (as explained before, this process needs itself to be generated dynamically);

- Explore how to integrate current and new solutions (elicited by others, or by us in the future) with our today's approaches i.e., STROBE/i-dialogue/AGIL.

Some precise issue may be:

Provenance of dynamically generated services. A recent research interest on service history is the concept of provenance. The provenance of a piece of data is the process that leads to the data [CTX+05]. Provenance would enable users to trace and identify the individual services (or high level process of services) as well as their corresponding inputs and outputs that were involved in the production of a specific result data. For example, in [CTX+05] the authors show an example of how to extract and store provenance from a process of baking a cake. Each agent of the process (user, baker, whisk, etc.) sends to a provenance service information about messages they exchange and their states along the process. As we explained with D5, history plays an important role in service exchange. The set of elements, characterizing the generation process, that leads to a particular dynamically generated service is perhaps more important than the service itself, for both the service provider and the service user. It is a very important aspect for the evolution of DSGS. It would enable agents to learn about generated services and enhance the way the next ones will be generated. Integrating provenance on DSGS is thus an interesting research perspective.

Service containers as Semantic Web/Grid services platform. In GRID, (as well as in the AGIL's integrated model), the service container is mainly considered as a hosting environment for services. Priority was given to the mechanisms enabling the allocation of specific resources (for a certain time) to services. However, service containers may play a more important role. Let us see for example the role played by the CAS included in the service container, which specifies the right levels of the VO members on the services. It could be interesting to consider service containers as semantic platforms enabling the development of Semantic Grid services. This kind of platform already exists for Semantic Web services, such as the Internet Reasoning Service (IRS) [CLMM03, DCH+04, DCG05]. Each agent who wants to servicize one of its capabilities in a service container, would have to add to its service semantic descriptions that connect this service with the already existing semantic elements of the service container e.g., in IRS goals, services,

ontologies and mediators are the four basic semantic elements. Integrating the work of the Semantic Web community for the development of Semantic Grid service containers is an interesting research perspective.

Modelling agent dialogues with. Working towards the realization of dialogue for agents within the DSG framework (C16[dialogue]) is for us one very challenging perspective. We really think that agents will bring an answer to a large number of problems in Informatics when they will be able to have dynamic dialogues. In this perspective, we are further convinced that working on streams, as suggested by chapter 4 and envisioned by Wegner et al. [WG99], and early by [Cer96b] may open an important range of solutions.

Planning. Explore the planning and distributed planning [AHT90] approaches, developed for a long time in AI as an inspiration source of modelling DSG process, could be also very interesting. The main challenge of planning which consists in dynamically re-adapt a plan according to circumstances seems very close to the one in DSG which consists in dynamically changing the way a service is provided.

Appendix A

Service-oriented architecture technologies

Table A.1: SOA technologies and consortiums URLs

CONSORPTIUMS AND TECHNOLOGIES	URL
W3C (World Wide Web Consortium)	`www.w3.org`
WSDL (Web Service Description Language) SOAP (Simple Object Access Protocol) WSCL (Web Services Conversation Language) WSCI (Web Service Choreography Interface) WS-CDL (Web Services Choreography Description Language) OWL (Web Ontology Language) RDF (Resource Description Framework) XML (eXtensible Markup Language)	
OASIS (Organization for the Advancement of Structured Information Standards)	`www.` `oasis-open.org`
UDDI (Universal Description, Discovery and Integration) WS-BPEL (Web Services Business Process Execution Language) BTP (Business Transaction Protocol) ebXML registries SAML (Security Services) WSRM (Web Services Reliable Messaging)	
WfMC (Workflow Management Coalition)	`www.wfmc.org`
XPDL (XML Processing Description Language)	
UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business)	`www.unece.org/` `cefact`
ebXML BPSS (Business Process Specification Schema)	
BPMI (Business Process Management Initiative)	`www.bpmi.org`
BPML (Business Process Modeling Language) BPQL (Business Process Query Language)	
Industrials (IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems, etc.)	
XLANG WSFL (Web Services Flow Language) BPEL4WS (Business Process Execution Language for Web Services) WS-Coordination (composed of WS-AtomicTransaction, WS-BusinessActivity) WS-Transaction, WS-Inspection, WS-Addressing, WS-Security WS-Trust, WS-reliability	

Appendix B

AGIL's concepts and relations

Table B.1: AGIL's concepts sum up
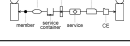
AGIL'S CONCEPT	SET	ELEMENT	GRAPHICS
computing resource	Ω	ω	
storage resource	Θ	θ	
host	H	h	
virtualized resource	R	r	nothing
service container	U	u	
normal service	S	s	
CAS	C	c	
service	Σ	σ	nothing
agent	A	a	
virtual organization	O	o	
brain	B	b	
cognitive environment	E	e	
capability	Λ	λ	
X509 certificate	X	x	X509

Table B.2: AGIL's relations sum up

AGIL's relation	Domain	Range	Type	Graphics
$resource - coupling$	$\Theta \times \Omega$	H	function	
$host - coupling$	H	H	function	
$virtualizing$	H	$\mathcal{P}(R)$	application	
$reifying$	U	$\mathcal{P}(R) - \{\emptyset\}$	application	
$authorizing$	O	C	bijection	
$handling$	$S \cup C$	C	surjection	
$including$	$S \cup C$	U	surjection	
$membershipping$	A	O	relation	
$holding$	X	$A \cup H$	application	
$interfacing$	Σ	Λ	injection	
$executing$	Λ	E	surjection	
$owning$	$E \cup B$	A	surjection	
$dedicating$	E	$\mathcal{P}(A) - \{\emptyset\}$	application	nothing
$interacting$	A	A	relation	
$exchanging$	A	A	relation	
$using$	Σ	A	relation	
$providing$	Σ	A	application	
$instantiating$	$\Sigma \times E$	$S \times E$	function	
$reproducing$	A	A	function	

In mathematics, a *binary relation* is an arbitrary association of elements of one set (domain) with elements of another set (range). Each time its was possible, respecting the semantics of AGIL, the relations were specialized to function, application, surjection, injection or bijection. Let us recall the properties of such relations. A relation may have four properties:

- *functional*. Each input of the domain has at most one output;

- *applicative*. Each input of the domain has at least one output;

- *injective*. Each output of the range has at most one input;

- *surjective*. Each output of the range has at least one input.

Then, a *function* is a functional relation. An *application* is an applicative function. An *injection* is an injective application. A *surjection* is a surjective application. A *bijection* is a injective and surjective application.

Appendix C

Implementation

It is vain to do with more what can be done with fewer
(Entities should not be multiplied beyond necessity)
Occam's razor principle, English philosopher, 1295-1349

T HE WORK presented in this thesis was partially implemented or experimented under different forms. These implementations and experimentations aims to show the viability of the principles presented in the previous chapters. They do not have for goal a large-scale development, which can be the objective of future engineering work based on this thesis. In particular, in this appendix, we quickly present the two partial implementations that have been done of the STROBE model as well as the mechanisms or tools that support these implementations (e.g., Scheme interpreters, the MadKit platform, etc.). We present a Scheme implementation of lazy lists, dialogue and i-dialogue functions as well as an example. An AGIL implementation plan is also proposed. The OWL/Protégé implementation of AGIO is briefly addressed.

Contents

C.1 Scheme implementation of the STROBE model

We developed some object based agents with techniques presented in [Nor91] and we endowed them with cognitive environments and cognitive interpreters. These simple agents are able to communicate, they exchange messages one another using the communication language presented in section 3.3.4.2. Their main characteristics is the fact that they can learn by communicating (learning-by-being told) at the three levels: data, control and interpreter. Their behaviour consists in the REPL loop seen above.

These agents were used to implement the experimentations (first and third) presented in section 3.4. Nevertheless, we have to mention the restrictions on the STROBE agents interacting in these experimentations: (i) they should follow the same construction (i.e., the same interpreter structure), which is a strong contradiction with heterogeneous agents modelling; (ii) in the first experimentation, the agent teacher is supposed to 'know' how to teach a new performative; (iii) the third experimentation deals with three agent, the two AAs presented above, and a HA who specify the constraints along the conversation via an interface managed by the customer agent; (iv) in all experimentations, the conversation needs a bootstrap.

In the experimentations, Scheme is supposed to be the minimal language agents all have in common. This is one of the constraint of the model, but as it is explained here after, Scheme is very interesting because nearly all the language can be re-defined from five special forms. At any time, if an agent does not know (i.e., does not bound in a CE) a specific primitive, it can ask to another agent. So primitives are bound locally in each CE. Notice that most of the time, when a programmer designs a community of agents that use a specific language he/she defines them with a set of primitives. We do not use a kind of super-environment binding all the primitives for all the agents; this is for us in opposition with the agents autonomy requirement. Moreover, scope assumed in the experimentations is lexical (static), the one used by Scheme i.e., in a function call, a variable is substituted by the value of the symbol bound during the definition of the function (not during the execution such as in Lisp).

C.1.1 Scheme meta-evaluation

Meta-programming is the term for the art of developing methods and programs to read, manipulate, or write other programs. Scheme uses the same syntax to represent data structures and expression of the language: *s-expression*. For that reason, languages such as Scheme – but it is also true for Lisp – easily enable to write meta-programs e.g., macro, interpreters or compilers. Scheme is often used in CS courses to write meta-circular evaluators [ASS96, Que96].[1] Writing a meta-circular evaluator for a subset of a language is a very good exercise to get familiar with the details of the semantics of the language, and the general techniques involved in implementing it. Furthermore, Scheme is one of the simplest language: nearly all the language can be re-defined with five special forms:

- `(lambda (formal-parameters) body)`, for procedure definitions;

- `(define variable value)`, for variable definitions;

- `(set! variable value)`, for variable assignments;

- `(if predicate consequent alternative)`, as control structure;

- `(quote expression)`, for symbol manipulation.

These are the reasons why we used Scheme to facilitate our work on meta-interpretation and reflection when experimenting the STROBE model's features – in particular meta-level learning presented in section 3.4.1. We wrote a Scheme meta-interpreter (`evaluate`) that was used by agents to interpret messages in a conversation. The content of messages was limited to Scheme expressions.

---

[1] An evaluator written in the same language as the one it evaluates is called a meta-circular evaluator or simply meta-evaluator or meta-interpreter.

C.1.2 Dynamic modification of an interpreter

With our meta-interpreter, we experimented the first and third methods of section 3.2.3. The first one was more an exercise realized to understand the art of reflection than a concrete experimentation. The third one is more trivial and depends mainly on the fact the Scheme language provides the `eval` function as part of the language itself (it was used in the second implementation of STROBE agents, STROBEKit, detailed hereafter).

C.1.2.1 First method: reifying procedures

Our meta-interpreter was mainly inspired from the famous article written by Jefferson and Friedman, *A Simple Reflective Interpreter* [JF92]. This interpreter is user friendly (both easy to use and learn) and provides the reflexivity property that we need. This interpreter has the following signature: `(evaluate e r k)`. It is mainly defined by two procedures `evaluate` and `apply-procedure`, corresponding to the two steps of the applicative-order evaluation in the environment model of evaluation.

The interpreter proposed in [JF92] is already implemented in a reflexive way.[2] In order to be evaluated by itself the interpreter code is written in the sub-language of Scheme that it recognizes. Jefferson and Friedman also propose an architecture called *reflexive tower*, as a tower of interpreters. The interpreter at the bottom of the tower executes user input, and every other interpreter in the tower executes the interpreter immediately below it. Two mechanisms allow 'moving' in the tower in order to evaluate the code at any level: reflexivity allows going up and reification allows going down. These two points are particularly interesting to us.

Reifying procedures provide the mechanism required to access to the execution context of the interpreter below. The main idea is that user's programs could have the same access rights as the interpreter itself. Thus, procedures implementing the interpreter can be accessed and modified by user's programs in the same way as the environment and the continuation. This property makes reifying procedures the ideal tool to dynamically modify our interpreters. Indeed, access to the execution context means access to the environment in which procedures defining the interpreter are stored and thus, means being able to modify them. Reification is the method to go down in the evaluation levels, therefore, to apply a reifying procedure, we need two levels of evaluation. The first level is carried out by evaluating the experimentation programs with a meta-interpreter. The second level being carried out by the agent itself when it receives a message (REPL loop) with the same interpreter (enriched with a `evaluate-strobemsg` procedure). Such an implementation is therefore not easy to build and it increases the complexity and computation load of programs. This implementation of the STROBE model meta-level learning is therefore consistent, but absolutely not scalable.

C.1.2.2 Third method: the `eval` function

Scheme is an interpreted language. That means that the interpreter evaluates dynamically expressions and gives them a value. As the language of the Lisp family, the evaluation procedure is part of the language itself. That means that the `eval` procedure is available and can be used by programs e.g., to evaluate dynamically generated code. What we have to know is that it is not this procedure which is called by Scheme in order to evaluate expressions. The one called by Scheme produces the same result, but it is not `eval`, it is a primitive one. Thus, an expression such as `(set! eval (lambda (x y) 'ok))` would change the `eval` procedure of the global environment, but not change the way expressions are interpreted. In order to dynamically modify an interpreter of Scheme expressions we have to write a meta-interpreter which uses the `eval` function given by the language instead of the primitive interpreter. Then, the simplest meta-evaluator is `meval`:

```
(define (meval exp env)
```

---

[2]Reflective programming refers to meta-programming which consists of writing programs that can deal with themselves.

```
(eval exp env))
```

A program implementing the REP loop with `meval` is:

```
(define global-env (interaction-environment)) ; With DrScheme
;(define global-env (make-environment ())) ; With MITScheme

(define (mloop)
  (newline)
  (display "MEVAL> ")
  (let ((userexp (read)))
    (if (eq? userexp 'exit)
    'ok
    (begin (display ": ")
           (display (meval userexp global-env))
           (mloop)))))
```

Therefore, using `meval` to evaluate expression we really use the `eval` procedure given by the language, then an evaluation with `meval` of an expression such as `(set! meval (lambda (x y) 'ok))` change the `meval` interpreter itself. The Scheme primitive interpreter which is not accessible is not modified but the one given in the language is. We actually use two levels of evaluation, as in the first method, but without the heavy mechanism of reifying procedures. This technique is the one used in the STROBEKit implementation of STROBE agents, where the `meval` interpreter is achieved by a Java method, that calls the `eval` function to evaluate an expression instead of evaluating it directly.

C.1.3   Non-deterministic interpreters

In order to implement the dynamic specification feature of the model (section 3.4.2) we changed the meta-interpreter previously designed in a non-deterministic one. This section is inspired by [ASS96] (chapter 4).

Nondeterministic evaluation is based on the special form `amb`. The expression `(amb exp1 exp2 ... expn)` returns the value of one of the n expressions `expi`. For example, the expression:

```
(list (amb 1 2 3) (amb 'a 'b))
```

can have six possible values:

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

The evaluation of the expression `(amb)` without arguments corresponds to a failure.

The interest of such an evaluator is that functions can call `amb` special form by adding constraints (as predicates) on the values returned by `amb`. These constraints are expressed with the function `require` defined like this:[3]

```
(define (require p)
   (if (not p) (amb)))
```

The evaluation of an `amb` expression can be seen as a tree solution exploration where a function is processed until a solution respecting all the constraints is found as long as the complete tree is not traversed. The form `(amb)`, without arguments, corresponds to a leaf of this tree, and then another

---

[3]The form `(if cond exp)` returns `exp` value if `cond` is `true`. Else it return no value.

branch must be explored. To recognize the special form `amb`, we have to modify the traditional evaluator to embody the change. Just like PROLOG language, with which, one can call, one by one, for all solutions to a logical expression; with a nondeterministic evaluator, the `try-again` special form[4] allows displaying of the next success evaluation to a function calling `amb`. The traditional interpretation loop is modified in nondeterministic evaluation, to take into account these backtracks. For example, consider the `an-element-of` function which returns an element of a list. Its evaluation could be:

```
> (an-element-of '(a b c))
: b
> try-again
: a
> try-again
: c
> try-again
: no more values
```

We could just take a look at the body of the above function; here, the constraint is that the list can not be null. If it is null?, then (amb) is evaluated and the non deterministic interpreter explores another branch of the solution tree:

```
(define (an-element-of items)
    (require (not (null? items)))
    (amb (car items)
         (an-element-of (cdr items))))
```

Consider now a harder constraint problem, taken from [ASS96]. It is a classic logic puzzle:

> 'Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?'

We can construct the following procedure which determine who lives on each floor by enumerating all the possibilities and imposing the given restrictions via require forms:

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
```

---

[4]The term special form is actually not the good one, because the `try-again` expression is not an expression of the language i.e., recognized by the interpreter, but an ad-hoc symbol treated by the program implementing the REP loop. This is not the case in the STROBE model, where the (`try-again`) expression is interpreted by CIs as a request for another proposition.

```
(require (> miller cooper))
(require
  (not (= (abs (- smith fletcher)) 1)))
(require
 (not (= (abs (- fletcher cooper)) 1)))
(list (list 'baker baker)
      (list 'cooper cooper)
      (list 'fletcher fletcher)
      (list 'miller miller)
      (list 'smith smith))))
```

Evaluating the expression `(multiple-dwelling)` produces the result:

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

C.1.4 Copying a Scheme environment

In order to implement the CE instantiation mechanism, we had to address the question of copying Scheme environments. This is not an easy task because of the fact that Scheme procedures include in their closure the environment in which they are defined. A problem of circularity raises to change the definition environment of a procedure by the environment in which this procedure is added.

C.2 STROBEkit

An implementation in the multi-agents platform MadKit (`www.madkit.org`) [GFM00] was also partially achieved.[5] MadKit (Multi-Agent Development Kit) is a MAS development platform founded on the AGR model [FGM03]. The platform is written in Java but enables the implementation of agents in several languages such as Java, Python, Scheme, Jess. MadKit is in particular interesting for us because it does not enforce any consideration about the internal structure of agents, thus allowing a developer to freely implements its own agent architectures. Moreover, MadKit is also a distributed platform i.e., different agents on different MadKit platforms may interact with one another. It allows the development of efficient distributed systems such as the one DSGS should tend towards.

STROBE agents are implemented in MadKit in Java but use Scheme environments for CE implementation. They use Kawa constructs i.e., interpreter and environment, to link Java and Scheme [Bot98] (`www.gnu.org/software/kawa`). The STROBEKit API enables a programmer to write agents that respect the STROBE model's requirements concerning agent structure and agent interaction (the i-dialogue abstraction was not yet integrated).

The STROBEKit API. The STROBEKit API consists mainly of three classes:

- `StrobeAbstractAgent`. The main class of the STROBEKit agent API. It extends the MadKit class `Agent`. STROBE agents must inherit from this abstract class. This class contains methods for: (i) cognitive environment (class `CogintiveEnv`) management; (ii) STROBE message (class `StrobeMsg`) management; (iii) defining the agent REPL loop. A STROBE agent is associated to a `Scheme` object which enables an agent to call a static method `eval`: a Scheme interpreter available as a Java method. This Java method is equivalent to `meval`, explained in the previous section.

---

[5]It was part od the subject of an applicative research internship done by four Computer Science bachelor students (Julien Bonjean, Loïc Calvino, Benjamin Chapus, and Christophe Di Pirro).

- CognitiveEnv. This class defines the CE's structure of the STROBEKit API. A Cognitive-Env belongs to a STROBE agent and is dedicated to an interlocutor agent (only one for the moment). It has two input and output 'streams'[6] of messages. A CognitiveEnv is associated to a Scheme environment (class SimpleEnvironment, provided by Kawa). This Scheme environment contains the CE's bindings (and thus the eval function). Then, the CognitiveEnv class contains methods for: (i) input and output 'streams' management; (ii) Scheme environment management; (iii) evaluation of messages; (iv) constructing Scheme environment by copy.

- StrobeMsg. This class defines the type of messages exchanged by STROBE agents. It extends the MadKit class Message. A StrobeMsg is logically viewed as a set of four attributes (sender receiver performative content). This class contains methods for: (i) accessing StrobeMsg attributes; (ii) converting StrobeMsg to String (in order to be processed by Java).

STROBEKit demo.   We have implemented some 'demo' STROBE agents (sub-class of StrobeAbstractAgent) with graphical user interfaces (GUI) in order to demonstrate how STROBE agents exchange messages. This demo illustrates the structure of STROBE agents: each STROBE agent has a set of CEs (global and locals). The three CE instantiation policies are available and CEs' bindings may be consulted.
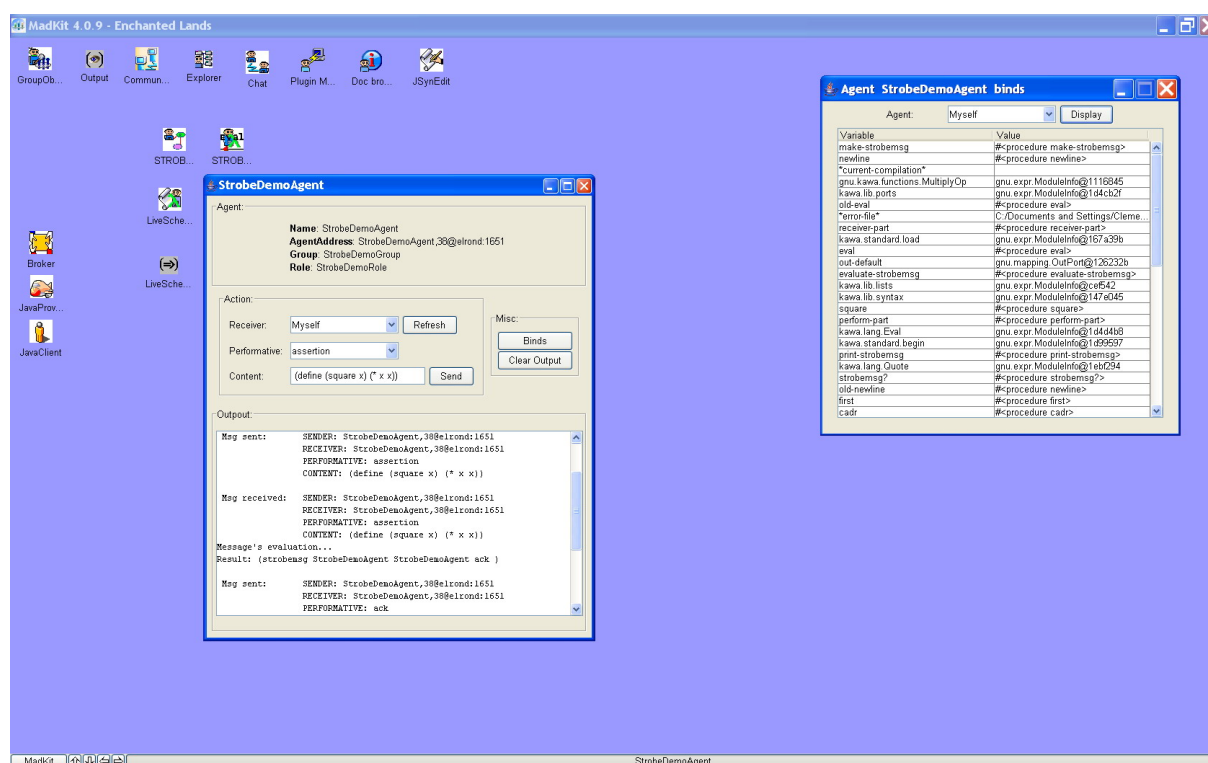

Figure C.1: Programming of an agent in STROBEKit

Programming in Scheme may be viewed as a dialogue with the Scheme interpreter following the *Read-Eval-Print* loop. In STROBEKit, we tried to keep this idea in agent-programmer interaction. STROBEKit enables a programmer to interact with the agent he wants to program following exactly the same rules of interaction than the ones the STROBE model proposes for agent-agent interaction. Actually, programmer's instructions are considered as messages directly interpreted in the agent's global CE. Programming the agent's global CE (at the three abstraction levels presented in section 3.2.1) means

---

[6]Streams is noted between quotes, because it is not really yet a stream of message as defined above.

programming the agent's generic capabilities, representations, knowledge, etc. Figure C.1 is a snapshot of the MadKit's desktop interface which shows: (i) on the left-hand window: a `StrobeDemoAgent`'s GUI, enabling a HA programmer to interact with the STROBE agent. He may send it messages or make it send messages to other agents. For instance, on the figure, the HA programmer learns `StrobeDemo-Agent` how to compute squares; (ii) on the right-hand window: `StrobeDemoAgent`'s global CE's bindings. For instance, the `eval` procedure, or the `square` procedure that has just been taught by the HA programmer.

Figure C.2 is a snapshot of the MadKit's desktop interface which shows three agents exchanging messages. `StrobeDemoAgent-2` and `StrobeDemoAgent-3` have both just sent `order` messages to `StrobeDemoAgent`, requesting it to compute `(cube 3)`. `StrobeDemoAgent` has just interpreted these messages in different local CEs, and has send back different `executed` messages as answers.
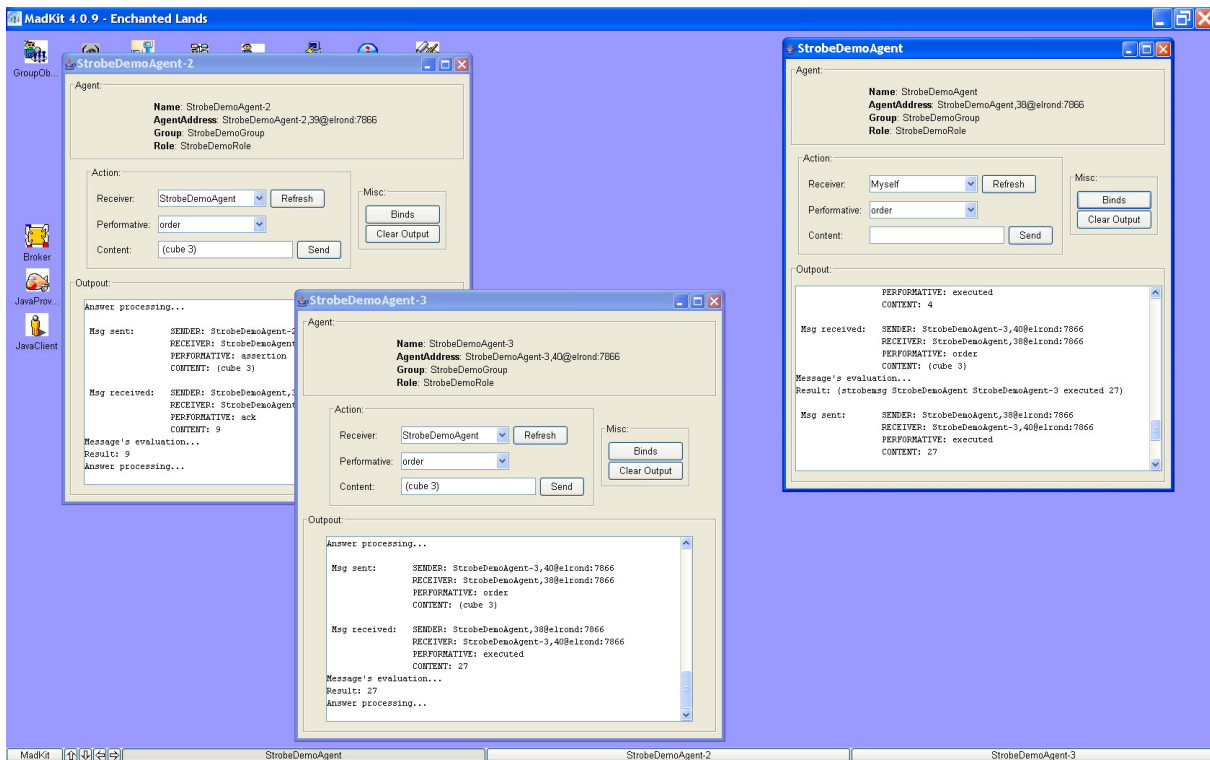

Figure C.2: Agent interaction in STROBEKit

The STROBEKit demo agents is the only set of agents developed following the STROBE model. There is no other application for the moment.

C.3 I-dialogue implementation

The i-dialogue abstraction is currently under development in STROBEKit. However, to experiment the work described in chapter 4, we have implemented the i-dialogue function in Scheme, a purely applicative language. The two ways of implementing i-dialogue (section 4.3.1) i.e., with several *step-fcn*s or with a unique *step-fcn*, were tested. The first way requires only an implementation of streams and is easily implementable in Scheme (following section). The second way needs all the language to be lazy. It was tested and verified with a lazy Scheme meta-evaluator that is not addressed here. For both implementations (i.e., streams and lazy evaluation) we invite the reader to refer to the chapter 4 of [ASS96].

C.3.1 Scheme implementation of streams

Scheme R^{5}RS is not a lazy language, but it offers the function `force` and the macro `delay` in the library which allow, combined with the macro operator (special form) `define-syntax`, to implement streams and lazy lists. An implementation of lazy lists may be:

```
; (CONS-LL EXP1 EXP2)-> LL, constructs a LL with EXP1 and EXP2
(define-syntax cons-ll
  (syntax-rules ()
    ((cons-ll e1 e2) (cons (delay e1) (delay e2)))))

; (LAZYLIST EXP1...EXPN)-> LL, constructs a LL with EXP1...EXPN
(define (lazylist . exps)
  (if (null? exps)
      ll-null
      (cons-ll (car exps) (apply lazylist (cdr exps)))))

; (LL-CAR,LL-CDR LL)-> ELEMENT, first element and rest of the elements of LL
(define (ll-car ll) (force (car ll)))
(define (ll-cdr ll) (force (cdr ll)))

; (CSTE) LL-NULL
(define ll-null (cons-ll 'eofll ll-null))

; (LL-NULL? LL)-> BOOL, true if LL is the ll-null
(define (ll-null? ll) (eq? (ll-car ll) 'eofll))

;(LL-FIRST,LL-SECOND,... LL)-> ELEMENT, first,second... element of LL
(define (ll-first ll) (ll-car ll))
(define (ll-second ll) (ll-car (ll-cdr ll)))
(define (ll-third ll) (ll-car (ll-cdr (ll-cdr ll))))
(define (ll-fourth ll) (ll-car (ll-cdr (ll-cdr (ll-cdr ll)))))

; (APPEND-LL LL1 LL2)-> LL, append for lazy lists
(define (append-ll ll1 ll2)
  (cond ((ll-null? ll1) ll2)
        ((ll-null? ll2) ll1)
        (else (cons-ll (ll-car ll1) (append-ll (ll-cdr ll1) ll2)))))

; (APPEND-LL-WITH-LIST LIST1 LIST2)-> LIST, applies append-ll to the arguments
; of LIST1 and LIST2 respectively
(define (append-ll-with-list list1 list2)
  (define (iter l1 l2 list-acc)
    (cond ((and (null? l1) (null? l2))
             list-acc)
          ((null? l1)
           (iter ()
                 (cdr l2)
                 (append list-acc (list (append-ll ll-null (car l2))))))
          ((null? l2)
           (iter (cdr l1)
                 ()
                 (append list-acc (list (append-ll (car l1) ll-null)))))
          (else
           (iter (cdr l1)
                 (cdr l2)
                 (append list-acc (list (append-ll (car l1) (car l2))))))))
```

```
  (iter list1 list2 ())))


; (LL-MAP-UN PROC-UN LL)-> LL, map but for 1 lazy lists
(define (ll-map-un proc ll)
  (if (ll-null? ll)
      ll-null
      (cons-ll (proc (ll-car ll))
               (ll-map-un proc (ll-cdr ll)))))
```

C.3.2 Scheme implementation of i-dialogue

The dialogue abstraction as it is defined in figure 4.3 may be implemented as:

```
(define (dialogue inputs initial-state step-fcn result-fcn)
  (define (run inputs state)
    (let* ((4long-ll (step-fcn inputs state))
           (inputs1 (ll-first 4long-ll))
           (outputs1 (ll-second 4long-ll))
           (state1 (ll-third 4long-ll))
           (done1 (ll-fourth 4long-ll)))
      (if done1
          (lazylist inputs1 outputs1 (result-fcn state1))
          (let* ((3long-ll (run inputs1 state1))
                 (inputs2 (ll-first 3long-ll))
                 (future-outputs2 (ll-second 3long-ll))
                 (result2 (ll-third 3long-ll)))
            (lazylist inputs2 (append-ll outputs1 future-outputs2) result2)))))
  (run inputs initial-state))
```

The i-dialogue abstraction as it is defined in figure 4.8 may be implemented as (first way):

```
(define (i-dialogue list-inputs initial-state list-step-fcn result-fcn)

  (define (iter listi listf listui listo s)
     (if (null? listi)
         (lazylist listui listo s #f)
         (let* ((4long-ll1 ((car listf) (car listi) s))
                (inputs1 (ll-first 4long-ll1))
                (outputs1 (ll-second 4long-ll1))
                (state1 (ll-third 4long-ll1))
                (done1 (ll-fourth 4long-ll1)))
           (if done1
               (lazylist (append listui (cons inputs1 (cdr listi)))
                         (append listo
                                 (cons outputs1
                                       (map (lambda (x) ll-null) (cdr listi))))
                         state1
                         #t)
               (iter (cdr listi)
                     (cdr listf)
                     (append listui (list inputs1))
                     (append listo (list outputs1))
                     state1)))))

  (define (run list-inputs state)
    (let* ((4long-ll2 (iter list-inputs list-step-fcn () () state))
```

```
             (list-inputs1 (ll-first 4long-ll2))
             (list-outputs1 (ll-second 4long-ll2))
             (state1 (ll-third 4long-ll2))
             (done1 (ll-fourth 4long-ll2)))
       (if done1
           (lazylist list-inputs1 list-outputs1 (result-fcn state1))
           (let* ((3long-ll (run list-inputs1 state1))
                  (list-inputs2 (ll-first 3long-ll))
                  (future-list-outputs2 (ll-second 3long-ll))
                  (result2 (ll-third 3long-ll)))
             (lazylist list-inputs2
                    (append-ll-with-list list-outputs1 future-list-outputs2)
                    result2)))))

  (run list-inputs initial-state))
```

C.3.3 I-dialogue example

The REP loop may be implemented with i-dialogue, as suggested by [O'D85]. The REP program consumes an input stream of Scheme expressions, evaluates the expressions (with the `eval` function) and produces an output streams. Inversely, the programmer produces and consumes these two streams and decide at each step the next expression he wants the REP programs to evaluate. The Scheme implementation is therefore:[7]

```
(define (REP-loop number)
  (define read-ll (cons-ll '(begin (display " > ") (read)) read-ll))

  (define (step-evaluate ll state)
    (let ((value (eval (ll-car ll) state)))
      (display "=> ")(display value)(newline)
      (lazylist (ll-cdr ll)
                (cons-ll value ll-null)
                state
                (if (eq? value 'quit) #t #f))))

  (define (result-fcn state) 'end)

  (display "------EVALUATE------")(newline)
  (set! sauv (cond
    ;; REP loop with dialogue
    ((= 1 number) (dialogue (ll-map-un eval read-ll)
                            (interaction-environment)
                            step-evaluate
                            result-fcn))
    ;; REP loop with i-dialogue
    ((= 2 number) (i-dialogue (list (ll-map-un eval read-ll))
                              (interaction-environment)
                              (list step-evaluate)
                              result-fcn))
    (else
     (display "Wrong parameter"))))
  (display "------END OF EVALUATE------")(newline))
```

---

[7]The function (`interaction-environment`) is a DrScheme (`www.drscheme.org`) primitive that returns an environment that can be passed as parameter of the `eval` function.

C.4 AGIL's implementation

C.4.1 AGIL's implementation plan

In this section, we propose some implementation directions for the GRID-MAS integrated model proposed in chapter 5. The integration of two so much important domains of Informatics will become a reality only if a large part of members of the corresponding communities invest many time and research effort towards. Whatever is the underlying integrated model, AGIL or another one. This effort has been done, for example, for GRID-SOC integration which is today considered as achieved. Some steps in such an implementation may be:

- To implement STROBE agents on a multi-agent platform. This is what we have begun with STROBEkit presented before;

- To enable MadKit agents to be compliant with SOA standards. For example, to enable agents (STROBE and others) to publish and request Web services;

- To implement agents in MadKit to play specific roles such as registry (compliant with the UDDI format), mediators, etc. To endow these agents with negotiation abilities (such as the one existing today i.e., protocols);

- To enable MadKit agents to interoperate with a Grid tool such as the Globus Toolkit.[8] It means to develop techniques enabling agents to manipulate GRID resources, allocated to the service they provide;

- To implement Madkit agents able to provide core GRID service functionalities e.g., CAS management and service container management. Related work on MAS based GRID approaches, cited in section 2.4.4.2, may be used at this step;

C.4.2 AGIO OWL/Protégé implementation

OWL is a common standard for representing ontologies [Hel04]. The Protégé knowledge-modelling environment (`http://protege.stanford.edu`) is a de facto standard ontologies editor. It frees the designer from writing directly XML/OWL code by automatically generating it. Moreover, Protégé allows to express rules on ontology concepts with the Semantic Web Rule Language (SWRL) [OKT$^+$05]. AGIL was implemented in Protégé by means of an ontology, the Agent-Grid Integration Ontology.[9] For instance, the class[10] *agent* and the property *interfaces* (i.e., *interfacing* relation) are defined in OWL as:

```
<owl:Class rdf:about="#Agent">
    <owl:disjointWith rdf:resource="#X509"/>
    ...
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="members"/>
        </owl:onProperty>
        <owl:someValuesFrom>
          <owl:Class rdf:about="#VO"/>
        </owl:someValuesFrom>
```

---

[8]This is the subject of a new thesis recently started by Saber Mansour between the Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour (LIUPPA – `liuppa.univ-pau.fr`) and the Laboratory of Informatics, Robotics, and Microelectronics of Montpellier (LIRMM – `www.lirmm.fr`).

[9]It was part of the subject of a research internship done by Frederic Duvert for his Master degree.

[10]Protégé calls class and property respectively concept and relation.

```
   </owl:Restriction>
</rdfs:subClassOf>
...
<owl:ObjectProperty rdf:about="#interfaces">
    <owl:inverseOf rdf:resource="#isInterfacedBy"/>
    <rdfs:range rdf:resource="#Capability"/>
    <rdfs:domain rdf:resource="#Service"/>
</owl:ObjectProperty>
```

An example of AGIL integration rules expressed in AGIO by means of SWRL may be: *All agents members of a VO hold a X509 certificate* (rule 23, page 164):

$$Agent(?x) \land X509(?y) \land VO(?z) \land$$
$$\textbf{members(?x,?z)} \rightarrow$$
$$holds(?x, ?y)$$

We invite the reader to refer to [Duv06], for a quite complete specification of AGIO.

List of Figures

List of Tables

List of Definitions

Agent

Computational metaphor that represents active entities (i.e., human or artificial) involved in dynamic service generation or product delivery. Agents are supposed autonomous, intelligent and interactive; they can play the role of service provider or service user. [**page** 37]

Agent-Grid integration language

GRID-MAS integrated systems description language which formalizes both key GRID and MAS concepts, their relations and the rules of their integration with graphical representations and a set-theory formalization. AGIL represents also a GRID-MAS integrated model which formalizes the service exchange interactions of STROBE agents through the Grid infrastructure. [**page** 154]

Artificial agent

A clearly identifiable physical or virtual autonomous entity which: (i) is situated in a particular environment of which it has only a partial representation; (ii) is capable of perceiving (with sensors) and acting (with effectors) in that environment; (iii) is designed to fulfil a specific role; (iv) communicates directly with other agents; (v) possesses its own state (and controls it) and skills; (vi) offers services (in the sense of particular problem solving capabilities); (vii) may be able to reproduce itself; (viii) has a behaviour that tends to satisfy its objectives. [**page** 69]

Cognitive environment

A structure which binds variables and values together. It includes a cognitive interpreter. The pair represents an agent's interlocutor model. STROBE agents use dedicated cognitive environments as conversation contexts to interpret messages (execution contexts) and to develop a different language for each of their interlocutors. [**page** 110]

Community

The dynamic social group (virtual or not) in which the agents evolve by using and providing different dynamically generated services with one another. The community is the context of conversational processes and is defined by the set of relationships created between agents. [**page** 37]

Conversational process

A long-lived, dynamic, unique, and not a pre-determined set of interactions (i.e., conversation or dialogue) allowing user and provider to respectively express their needs and constraints during the service generation. A conversational process starts with a bootstrap, produces elements local to agents and finishes by a final (internal or external) result. [**page** 38]

Dialogue abstraction

A recursive function run by an agent to model a conversation with another one. The dialogue abstraction produces an output stream of messages and a new agent state by consuming an input stream of messages and an old state. [**page** 139]

Dialogue modelling

Identifies the research problem of agent communication that aims to model dynamic conversations. It expects agents to consider only messages from interlocutors and allow them to handle the entire conversation dynamically and not simply interpret messages one-by-one following a previously determined structure. [**page** 78]

Dynamic service generation

The process of exchanging services, constructed on the fly, and created specifically following the conversation between a service provider and a service user. This unique dynamic process establishes relationships within communities and improve user and provider's mastery by solving a problem or answering a not necessarily clearly specified need or wish. [**page** 35]

Execution context

The set of elements (expression, environment, interpreter, continuation) intervening in the evaluation of an expression. The value of an expression depends on the execution context. [**page** 107]

Grid (the)

Service oriented architecture based on a flexible, secure and coordinated resource sharing infrastructure allowing dynamic service exchange among members of several virtual communities/organizations. The Semantic Grid highlights the information and knowledge dimension of these service exchanges. [**page** 82]

Grid service

A service instantiated with its own dedicated resources (state) for a certain amount of time (lifetime) available within a VO. Grid services are compliant with the Web service framework. [**page** 88]

High level process of services

A (business) process that realizes the coordinated composition, or any form of putting together, of several services (composed) in a unique service (composite) performance. This composition may be static (orchestration/workflow) or dynamic (choreography/conversation). [**page** 61]

I-dialogue abstraction

A generalization of the dialogue abstraction applied in the STROBE model for modelling multi-party intertwined agent conversation. I-dialogue models conversations that must occur when an agent provides a composite service. [**page** 142]

Instantiation process

In AGIL, the process of creating a new service-capability couple by mapping the Grid service and STROBE's instantiation of CE mechanisms. The new cognitive environment created serves as a conversation context to execute the dedicated service. [**page** 170]

Interaction protocol

Interaction structure that (semantically) specifies rules that must be respected, messages that must be exchanged, and the states that agents must take, during a conversation. Interaction protocols model pre-fixed conversations. [**page** 76]

Language enrichment/learning

The process of abstracting at the data, control and in particular at the interpreter level of a given language in order to enrich both the set of recognized expressions and change the way they are evaluated. Language enrichment is said to be dynamic when it is done during the evaluation of an expression of the language. [**page** 106]

Product delivery

The process of supplying products, obtained as the result of the activation of a provider's functionality in order to answer to a well specified user's need or want. This one-shot invocation generic process does not improve user or provider's mastery and hardly help to answer a need or wish not expressed following a specified set of parameters. [**page** 35]

Semantics

Semantics is related to the meaning of things. Semantics is taken into consideration every time that a set of concepts, the relations between them and their rules need to be formalized. This formalization often takes the form of an ontology. [**page** 67]

Service

A solution, identified and chosen among many possible ones, offered to the problem of someone. This solution may be physical, psychological, concrete, abstract, real, virtual. This service exchange may be viewed as a kind of collaboration and establishes a specific relationship between a service user and a service provider. [**page** 31]

Service-oriented architecture

A model for the execution of loosely coupled and dynamic service-based software applications. A service is seen as a standardized and interoperable interface of a specific function. The Web service framework is the current main implementation of service-oriented architecture. [**page** 55]

Servicization process

The virtualization process of an agent's capability into a Grid service within a service container. It is the key element of AGIL to represent a service as an interface of a capability executed with Grid resources but managed by an intelligent, autonomous and interactive agent. [**page** 164]

Stream of messages

A data structure which represents a potentially infinite sequence of messages consumed and produced by agents within a dialogue. [**page** 137]

STROBE model

An agent representation and communication model inspired by constructs of functional/programing languages. STROBE agents behave like interpreters that evaluate streams of messages in multiple dedicated cognitive environments. [**page** 109]

Web service

Web services are standardized and interoperable software components that perform some function and that use Internet protocols and technologies. They are describable, discoverable and executable according to SOA standards. Web services may be included in high level process of services in order to address specific collaboration problems. [**page** 64]

List of Characteristics

Characteristic 8 [SOA]

DSGS should respect fundamental SOA principles such as loose coupling, implementation neutrality, standardization, message based communication, interoperability, etc. The more SOAs are loosely coupled and dynamic, the more they will fit with DSG requirements. [**page** 55]

Characteristic 17 [agent]

DSGS should be agent oriented. Firstly, because of the adapted properties of agents for providing and using services. Secondly, because agents are the current best metaphor for humans in computing i.e., agents enable to use a unique model for HAs and AAs. [**page** 79]

Characteristic 18 [community]

DSGS should be structured in (virtual) communities. They represent the set of relationships created between service provider and service user agents during DSG. It is the social dimension of DSG. [**page** 83]

Characteristic 15 [conversation]

DSGS need strong open and dynamic communication models that are able to manage conversational processes. They should not reduce agent autonomy and heterogeneity and they should allow an interpretation of messages that is as dynamical as possible. [**page** 75]

Characteristic 16 [dialogue]

DSGS conversational processes should be made of dialogues. They should be dynamic and not pre-determined by a data structure that guides the conversational processes, thus limiting agent interpretation. [**page** 78]

Characteristic 20 [grid]

DSGS should be GRID oriented because the Grid is the first service-oriented architecture and infrastructure addressing the question of secure and reliable resource sharing and service exchange by virtualization (resource and service level) among VOs. [**page** 86]

Characteristic 6 [human]

DSGS should integrate human agents by means of highly suitable interfaces, visualization mechanisms and miscellaneous means of interaction. Moreover, they should be able to understand and deal with the languages employed by HA users. More generally, DSGS should deal with all domains of Informatics that tend towards a better integration of humans in the loop. [**page** 47]

Characteristic 2 [interaction]

DSGS should be interaction oriented as the main difference between PD and DSG is the conversational process. Interaction should be intrinsic to the underlying model of DSGS. The behaviour of a DSGS should emerge from interactions between agents of this DSGS. [**page** 44]

Characteristic 4 [learning]

DSGS should support learning, knowledge creation and sharing in order to fulfill the pedagogical dimension of DSG. This is both true at the agent individual level (service user and provider change of states) and at the social level (new knowledge common to and shared between agents emerges from DSG). [**page** 46]

Characteristic 11 [message]

DSGS agents should interact with an asynchronous direct message-passing mode of communication. [**page** 59]

Characteristic 10 [negotiation]

DSGS should allow agents to dynamically negotiate (and re-negotiate) the terms and conditions under which the desired service is dynamically generated. A service contract may exist, but some parts are fixed and some parts may evolve during the DSG. [**page** 59]

Characteristic 3 [nondeterminism]

DSGS should be non-deterministic as the service exchanges that may occur are non predictable and depend on the conversations. The non-deterministic behaviour of the system should not be predicted (thus expressed by a procedure) but should emerge from interactions. [**page** 45]

Characteristic 14 [pragmatics]

DSGS should deal with pragmatics to ensure a context aware interpretation of messages occurring during the service conversational process. [**page** 69]

Characteristic 12 [process]

DSGS should enable dynamic high level processes of services by modelling agent communication. This may be true both for workflow/orchestration or conversation/choreography based approaches. [**page** 61]

Characteristic 7 [programming]

DSGS should use techniques and methods allowing a dynamic behaviour of the system as much as possible. Dynamic concepts of programming such as interpreted languages, stream processing, lazy-evaluation, dynamic typing, continuation passing style, constraint satisfaction programming are all promising examples. [**page** 49]

Characteristic 5 [reasoning]

DSGS agents should be able to reason and learn according to their current state and history in order to evolve and improve the way they provide and use dynamically generated services. This reasoning process should be dynamic during DSGs, and static between DSGs. [**page** 46]

Characteristic 9 [registry]

DSGS need symmetric dynamic registries where service providers and service users publish respectively what they know, or what can be published from their offers and demands. These registries should play the role of marketplaces and facilitate negotiation in order to match offers and demands. [**page** 57]

Characteristic 13 [semantics]

DSGS should deal with semantics at the data, information and knowledge levels, through the use of ontologies. The elements of DSG messages (i.e., descriptions, requirements, constraints, etc.) should be described at a semantic level. [**page** 69]

Characteristic 22 [separation]

DSGS should provide agents with the functionalities enabling several accesses and managements of stateful resources allocated to service exchange. Agents should be able to manage and dynamically change services and resources separately. [**page** 90]

Characteristic 21 [stateful]

DSGS agents should be able to manage autonomously the state and lifetime of resources allocated for a certain period of time to the service exchanges. This state is the key element supporting interaction. The state also plays the role of history. [**page** 88]

Characteristic 19 [trust]

DSGS need trust environments to favour real collaboration and knowledge creation through service exchange. Security plays an important role for trust. [**page** 86]

Characteristic 1 [web]

DSGS should be Web oriented. They have to be accessible through Web standards and technologies but the Web also has to evolve to fit other DSGS requirements (e.g., statefulness, semantics, conversation, etc.) [**page** 43]

List of Acronyms

ACL   Agent Communication Language, [**page** 73]
ADEPT   Advanced Decision Environment for Process Tasks, [**page** 71]
AGIL   Agent-Grid Integration Language, [**page** 151]
AGIO   Agent-Grid Integration Ontology, [**page** 176]
AGR   Agent-Group-Role, [**page** 96]
AI   Artificial Intelligence, [**page** 29]
AIED   Artificial Intelligence in EDucation, [**page** 103]
AOP   Agent Oriented Programming, [**page** 71]
ARMS   Agent based Resource Management System, [**page** 92]
AUML   Agent Unified Modelling Language, [**page** 76]

BDI   Belief-Desire-Intention, [**page** 70]
BPEL4WS Business Process Execution Language for Web Service, [**page** 65]
BPM   Business Process Management, [**page** 60]
BPMI   Business Process Management Initiative, [**page** 187]
BPML   Business Process Modeling Language, [**page** 187]
BPQL   Business Process Query Language, [**page** 187]
BPSS   Business Process Specification Schema, [**page** 187]

CA   Certification Authority, [**page** 84]
CAS   Community Authorization Service, [**page** 84]
CC   Cognitive Continuation, [**page** 131]
CE   Cognitive Environment, [**page** 109]
CI   Cognitive Interpreter, [**page** 110]
CIA   Cooperative Information Agent, [**page** 72]
CoABS   Control of Agent-Based Systems, [**page** 90]
CONOISE-G Grid-enabled Constraint-Oriented Negotiation in an Open Information Services Environment, [**page** 93]
CORBA   Common Object Request Broker Architecture, [**page** 53]
CPDL   Communication Protocol Description Language, [**page** 76]
CPN   Coloured Petri Nets, [**page** 76]
CSP   Constraint Satisfaction Programming, [**page** 45]

D/COM   Distributed/Component Object Model, [**page** 53]
DAML   DARPA Agent Markup Language, [**page** 68]
DCE   Distributed Computing Environment, [**page** 53]
DL   Description Logic, [**page** 68]

219

DSG   Dynamic Service Generation, [**page** 29]
DSGS   Dynamic Service Generation Systems, [**page** 34]

ebXML   electronic-business XML initiative, [**page** 64]
ESCE   Experimental Computer Science and Engineering, [**page** 17]

FIPA   Foundation for Intelligent Physical Agents, [**page** 74]
FIPA-ACL   FIPA - Agent Communication Language, [**page** 74]
FIPA-SD   FIPA - Service Description, [**page** 80]
FIPA-SL   FIPA - Semantic Language, [**page** 74]
FSM   Finite State Machine, [**page** 76]

GGF   Global Grid Forum, [**page** 83]
GRAM   Grid Resource Allocation and Management, [**page** 83]
GSD   Grid Shared Desktop, [**page** 98]
GSFL   Grid Service Flow Language, [**page** 98]
GSH   Grid Service Handle, [**page** 88]
GSI   Grid Security Infrastructure,Fos99-GridBlueprint, [**page** 84]
GSR   Grid Service Reference, [**page** 88]
GT   Globus Toolkit, [**page** 83]

HTTP   Hyper Text Transfer Protocol, [**page** 29]

IRS   Internet Reasoning Service, [**page** 65]
ITS   Intelligent Tutoring Systems, [**page** 36]

KB   Knowledge Base, [**page** 46]
KQML   Knowledge Query and Manipulation Language, [**page** 74]

MANTA   Modeling an ANThill Activity, [**page** 70]
MAS   Multi-Agent Systems, [**page** 69]
MATS   Mobile Agents Team System, [**page** 93]
MD   Message Digest, [**page** 84]
MDS   Meta Directory Service, [**page** 84]
MIME   Multipurpose Internet Mail Expansion, [**page** 66]

OASIS   Organization for the Advancement of Structured Information
                Standards, [**page** 187]
OCMAS   Organization Centred Multi-Agent Systems, [**page** 96]
OCML   Operational Conceptual Modelling Language, [**page** 122]
OGSA   Open Grid Service Architecture, [**page** 87]
OGSI   Open Grid Services Infrastructure, [**page** 88]
OIL   Ontology Information Language, [**page** 68]
OMG   Object Management Group, [**page** 53]
OWL   Web Ontology Language, [**page** 68]
OWL-S   OWL-based Web Service ontology, [**page** 22]

PD   Product Delivery, [**page** 29]
PDL   Process Description Language, [**page** 64]

PDS	Product Delivery Systems, [**page** 34]
PRS	Procedural Reasoning System, [**page** 71]
RDF	Resource Description Framework, [**page** 68]
REPL	Read - Eval - Print - Listen, [**page** 108]
RIE	Recursive Interactive Environment, [**page** 135]
RMI	Remote Method Invocation, [**page** 53]
RPC	Remote Procedure Call, [**page** 29]
SAML	Security Assertion Markup Language, [**page** 187]
SIP	Session Initiation Protocol, [**page** 66]
SLA	Service Level Agreement, [**page** 71]
SMTP	Simple Mail Transfer Protocol, [**page** 66]
SOA	Service-Oriented Architecture, [**page** 53]
SOAP	Simple Object Access Protocol, [**page** 63]
SOC	Service-Oriented Computing, [**page** 53]
STROBE	STReam OBject Environment, [**page** 108]
SWRL	Semantic Web Rule Language, [**page** 202]
SWS	Semantic Web Service, [**page** 68]
TCP/IP	Transmission Central Protocol, [**page** 66]
UDDI	Universal Description, Discovery and Integration, [**page** 63]
UML	Unified Modeling Language, [**page** 65]
UN/CEFACT	United Nations/Centre for Trade Facilitation and Electronic Business, [**page** 187]
VO	Virtual Organization, [**page** 83]
W3C	World Wide Web Consortium, [**page** 63]
WfMC	Workflow Management Coalition, [**page** 187]
WS	Web Service, [**page** 61]
WS-BPEL	BPELWS, [**page** 187]
WS-CDL	Web Services Choreography Description Language, [**page** 187]
WSAG	Web Service Agent Gateway, [**page** 81]
WSCI	Web Service Choreography Interface, [**page** 65]
WSCL	Web Services Conversation Language, [**page** 65]
WSDL	Web Service Description Language, [**page** 63]
WSFL	Web Services Flow Language, [**page** 65]
WSIGS	Web Service Integration Gateway Service, [**page** 80]
WSMF	Web Service Modeling Framework, [**page** 97]
WSMO	Web Service Modelling Ontology, [**page** 68]
WSRF	Web Service Resource Framework, [**page** 88]
WSRM	Web Services Reliable Messaging, [**page** 187]
XML	eXtensible Markup Language, [**page** 29]
XPDL	XML Processing Description Language, [**page** 187]

Bibliography

[ACD⁺05]   Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). GGF Proposed recomendation, Global Grid Forum, September 2005. 65

[ACR⁺05]   Colin Allison, Stefano A. Cerri, Pierluigi Ritrovato, Angelo Gaeta, and Matteo Gaeta. Services, semantics and standards: elements of a learning Grid infrastructure. *Applied Artificial Intelligence, Special issue on Learning Grid Services*, 19(9-10):861–879, October-November 2005. 82, 98

[Agh86]   Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 112

[AGP03]   Liliana Ardissono, Anna Goy, and Giovanna Petrone. Enabling conversations with Web services. In *2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS'03*, pages 819–826, Melbourne, Australia, July 2003. ACM Press. 95, 125

[AHT90]   James Allen, James Hendler, and Austin Tate, editors. *Readings in planning*. Morgan Kaufmann, Palo Alto, CA, USA, 1990. 60, 188

[ALd05]   Ronald Ashri, Michael Luck, and Mare d'Inverno. From SMART to agent systems development. *Engineering Applications of Artificial Intelligence*, 18(2):129–140, March 2005. 72, 121

[AP80]   James Allen and C. Raymond Perrault. Analysing intention in utterances. *Artificial Intelligence*, 15(3):143–178, December 1980. 74

[ASS96]   Harold Abelson, Geral Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. 47, 48, 107, 109, 111, 112, 119, 125, 128, 134, 140, 194, 196, 197, 200

[Aus62]   John L. Austin. *How to do things with words*. Clarendon Press, Oxford, UK, 1962. 73

[BD01]   Jean-Pierre Briot and Yves Demazeau, editors. *Principes et architectures des systèmes multi-agents*. Traité IC2 - Informatique et Systèmes d'Information. Hermès, Paris, France, 2001. 36, 69

[BDB92]   Sabine Berthet, Yves Demazeau, and Olivier Boissier. Knowing each other better. In *11th International Workshop on Distributed Artificial Intelligence*, pages 23–42, Glen Arbor, MI, USA, February 1992. 76

[BF95]   Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi agent systems. In V. Lesser and L. Gasser, editors, *1st International Conference and*

Multi-Agent Systems, ICMAS'95, pages 17–24, Menlo Park, CA, USA, June 1995. AAAI Press. 76

[BFH03]   Fran Berman, Geoffrey Fox, and Anthony J.G. Hey, editors. *Grid computing: making the global infrastructure a reality*. John Wiley & Sons, 2003. 87

[BHM$^+$04]   David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. W3C working group note NOTE-ws-arch-20040211, World Wide Web Consortium, February 2004. www.w3.org/TR/2004/NOTE-ws-arch-20040211/. 61

[BL00]   Mihai Barbuceanu and Wai-Kau Lo. Conversation oriented programming for agent interaction. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 220–234. Springer-Verlag, Berlin, Germany, 2000. 76

[BLHL01]   Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001. 21, 67

[BMMS03]   Joanna J. Bryson, David Martin, Sheila A. McIlraith, and Lynn A. Stein. Agent-based composite services in daml-s: the behavior-oriented design of an intelligent semantic web. In N. Zhong, J. Liu, and Y. Yao, editors, *Web Intelligence*, pages 37–58. Springer, 2003. 95

[Boi01]   Olivier Boissier. Modèles et architectures d'agents. In J-P. Briot and Y. Demazeau, editors, *Principes et architectures des systèmes multi-agents*, Traité IC2 - Informatique et Systèmes d'Information, chapter 2, pages 71–107. Hermès, Paris, France, 2001. 71

[Bot98]   Per Bothner. Kawa: compiling Scheme to Java. In *Lisp User Group Meeting, LUGM'98*, Berkeley, CA, USA, November 1998. 198

[BP99]   Christian Brassac and Sylvie Pesty. Simuler la conversation : un défi pour les systèmes multi-agents. In B. Moulin, S. Delisle, and B. Chaib-draa, editors, *Analyse et simulation de conversations : De la théorie des actes de discours aux systèmes multiagents*, chapter 9, pages 317–345. L'interdisciplinaire, Limonest, France, 1999. 78

[Bre97]   Joost Breuker. Problems in indexing problem solving methods. In D. Fensel, editor, *Workshop on Problem Solving Methods for Knowledge Based Systems*, pages 19–35, Nagayo, Japan, August 1997. 67

[Bro91]   Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, January 1991. 70

[Bur75]   William H. Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1):12–25, January 1975. 140

[BV04]   Paul A. Buhler and José M. Vidal. Integrating agent services into BPEL4WS defined workflows. In *4th International Workshop on Web-Oriented Software Technologies, IW-WOST'04*, Munich, Germany, July 2004. 81, 94

[BVV03]   Paul A. Buhler, José M. Vidal, and Harko Verhagen. Adaptive workflow = Web services + agents. In *1st International Conference on Web Services, ICWS'03*, pages 131–137, Las Vegas, NV, USA, July 2003. CSREA Press. 65, 94

[Cao01]   Junwei Cao. *Agent-based resource management for Grid computing*. PhD thesis, University of Warwick, Coventry, UK, October 2001. 82, 93

[CCF⁺00]   R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Using colored Petri nets for conversation model. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 178–192. Springer-Verlag, Berlin, Germany, 2000. 76

[CDM⁺04]   Liliana Cabral, John Domingue, Enrico Motta, Terry Paynec, and Farshad Hakimpour. Approaches to Semantic Web services: an overview and comparisons. In *1st European Semantic Web Symposium, ESWS'04*, Heraklion, Crete, Greece, May 2004. 22, 68

[Cer94]   Stefano A. Cerri. Models and systems for collaborative dialogues in distance learning. In M. F. Verdejo and S. A. Cerri, editors, *Collaborative Dialogue Technologies in Distance Learning*, volume 133 of *ASI Series F: Computers and Systems Sciences*, pages 119–125. Springer-Verlag, Berlin, Germany, 1994. 46, 105

[Cer96a]   Stefano A. Cerri. Cognitive Environments in the STROBE model. In P. Brna, A. Paiva, and J. Self, editors, *European Conference in Artificial Intelligence and Education, EuroAIED'96*, pages 254–260, Lisbon, Portugal, October 1996. 110, 111

[Cer96b]   Stefano A. Cerri. Computational mathetics tool kit: architecture's for dialogue. In C. Frasson, G. Gauthier, and A. Lesgold, editors, *3rd International Conference on Intelligent Tutoring Systems, ITS'96*, volume 1086 of *Lecture Notes in Computer Science*, pages 343–352, Montreal, Canada, June 1996. Springer-Verlag. 24, 110, 111, 135, 188

[Cer99a]   Stefano A. Cerri. Dynamic typing and lazy evaluation as necessary requirements for Web languages. In *European Lisp User Group Meeting, ELUGM'99*, Amsterdam, Netherlands, June 1999. 48, 150

[Cer99b]   Stefano A. Cerri. Shifting the focus from control to communication: the STReam OBjects Environments model of communicating agents. In J.A. Padget, editor, *Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computing*, volume 1624 of *Lecture Note in Artificial Intelligence*, pages 74–101. Springer-Verlag, Berlin, Germany, 1999. 110, 115, 121

[Cer05]   Stefano A. Cerri. An integrated view of Grid services, agents and human leaning. In P. Ritrovato, C. Allison, S.A. Cerri, T. Dimitrakos, M. Gaeta, and S. Salerno, editors, *Towards the Learning Grid: advances in Human Learning Services*, volume 127 of *Frontiers in Artificial Intelligence and Applications*, pages 41–62. IOS Press, November 2005. 29, 98

[CFF⁺04a] Karl Czajkowski, Donald F. Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Tom Maguire, David Snelling, and Steve Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: refactoring and evolution. Whitepaper Ver. 1.0, The Globus Alliance, May 2004. 90

[CFF⁺04b] Karl Czajkowski, Donald F. Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe. The WS-Resource Framework. Whitepaper Ver. 1.0, The Globus Alliance, May 2004. 87, 90

[CFFK01]   Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *10th IEEE International Symposium on High Performance Distributed Computing, HPDC-10'01*, pages 181–184, San Francisco, CA, USA, August 2001. IEEE Computer Society. 85

[CFK⁺98]   Karl Czajkowski, Ian Foster, Nicholas Karonis, Carl Kesselman, Stuart Martin, War-
           ren Smith, and Steven Tuecke. A resource management architecture for metacomput-
           ing systems. In *4th Workshop on Job Scheduling Strategies for Parallel Processing,
           IPPS/SPDP'98*, volume 1459 of *Lecture Notes In Computer Science*, pages 62–82, Or-
           lando, FL, USA, March 1998. Springer-Verlag. 84

[Cha96]   Jacques Chazarain. *Programmer avec Scheme*. International Thomson Publishing, Paris,
           France, 1996. 106

[CL90]   Philip R. Cohen and Hector J. Levesque. Intentions is choice with commitment. *Artificial
           Intelligence*, 42(2-3):213–261, March 1990. 74, 125

[CL95]   Philip R. Cohen and Hector J. Levesque. Communicative actions for artificial agents. In
           V. Lesser and L. Gasser, editors, *1st International Conference on Multi-Agents Systems,
           ICMAS'95*, Menlo Park, CA, USA, June 1995. AAAI Press. 74

[Cla05]   William J. Clancey. Towards on-line services based on a holistic analysis of human ac-
           tivities. In P. Ritrovato, C. Allison, S.A. Cerri, T. Dimitrakos, M. Gaeta, and S. Salerno,
           editors, *Towards the Learning Grid: advances in Human Learning Services*, volume 127
           of *Frontiers in Artificial Intelligence and Applications*, pages 3–11. IOS Press, November
           2005. 21, 28, 35

[CLMM03]   Monica Crubézy, Wenjin Lu, Enrico Motta, and Mark A. Musen. Configuring online
           problem-solving resources with the Internet Reasoning Service. *Intelligent Systems*,
           18(2):34–42, March-April 2003. 65, 68, 187

[Cou94]   National Research Council. *Academic Careers for Experimental Computer Scientists and
           Engineers*. National Academy Press, Washington, D.C., USA, 1994. 15, 16, 17

[CP79]   Philip R. Cohen and C. Raymond Perrault. Elements of a plan-based theory of speech
           acts. *Cognitive Science*, 3(3):177–212, July-August 1979. 74, 120

[CSJN05]   Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Grid load bal-
           ancing using intelligent agents. *Future Generation Computer Systems*, 21(1):135–149,
           January 2005. 93

[CT03]   Mario Cannataro and Domenico Talia. The Knowledge Grid. *Communications of the
           ACM*, 46(1):89–93, January 2003. 82, 83, 97

[CTT05]   Carmela Comito, Domenico Talia, and Paolo Trunfio. Grid services: principles, imple-
           mentations and use. *Web and Grid Services*, 1(1):48–68, 2005. 82, 90

[CTX⁺05]   Liming Chen, Victor Tan, Fenglian Xu, Alexis Biller, Paul Groth, Simon Miles, John
           Ibbotson, Michael Luck, and Luc Moreau. A proof of concept: provenance in a service-
           oriented architecture. In *4th All Hands Meeting, AHM'05*, Nottingham, UK, September
           2005. 82, 187

[DCG05]   John Domingue, Liliana Cabral, and Stefania Galizia. Choreography in IRS-III - Coping
           with heterogeneous interaction patterns in Web services. In *4th International Semantic
           Web Conference, ISWC'05*, Galway, Ireland, November 2005. 65, 95, 187

[DCH⁺04]   John Domingue, Liliana Cabral, Farshad Hakimpour, Denilson Sell, and Enrico Motta.
           IRS-III: a platform and infrastructure for creating WSMO-based Semantic Web services.
           In *1st Workshop on WSMO Implementations, WIW'04*, CEUR Workshop Proceedings,
           Frankfurt, Germany, September 2004. 65, 68, 187

[DDCP05]   Hywel R. Dunn-Davies, Jim Cunningham, and Shamimabi Paurobally. Propositional stat-echarts for agent interaction protocols. *Electronic Notes in Theoretical Computer Science*, 134:55–75, June 2005. 76

[Dec03]   Rina Dechter. *Constraint processing*. Morgan Kaufmann, San Francisco, CA, USA, 2003. 45, 49, 130

[Dem95]   Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *1st European Conference on Cognitive Science*, pages 117–132, Saint-Malo, France, April 1995. 71, 76

[DG00]   Frank Dignum and Mark Greaves, editors. *Issues in agent communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2000. 75, 76, 125

[DHKV03]   Jonathan Dale, Akos Hajnal, Martin Kernland, and Laszlo Zsolt Varga. Integrating Web services into agentcities recommendation. Agentcities technical recommendation actf-rec-00006, Agentcities Web Services Working Group, November 2003. 81

[Dij75]   Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. 45

[DJDC]   Frédéric Duvert, Clement Jonquet, Pascal Dugénie, and Stefano A. Cerri. Agent-Grid Integration Ontology. In *International Workshop on Agents, Web Services and Ontologies Merging, AWeSOMe'06*. 67, 178

[DLJC06]   Pascal Dugénie, Philippe Lemoisson, Clement Jonquet, and Monica Crubézy. The Grid Shared Desktop: a bootstrapping environment for collaboration. *Advanced Technology for Learning, Special issue on Collaborative Learning*, 2006. Accepted for publication - Expected end of 2006. 67, 82, 86, 98

[Dro93]   Alexis Drogoul. *De la simulation multi-agents à la résolution collective de problèmes*. PhD thesis, University Paris 6, November 1993. 70

[DS83]   Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, January 1983. 76

[Duv06]   Frederic Duvert. An ontology of Grid and Multi-agent systems integration. Master's thesis, University Montpellier 2, Montpellier, France, July 2006. 205

[EBHM00]   Rogier M. Van Eijk, Frank S. De Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer. Operational semantics for agent communication languages. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 80–95. Springer-Verlag, Berlin, Germany, 2000. 74

[EH00]   Renée Elio and Afsaneh Haddadi. On abstract models and conversations policies. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 301–313. Springer-Verlag, Berlin, Germany, 2000. 76

[Eij02]   Rogier M. Van Eijk. Semantics of agent communication: an introduction. In M. d'Inverno, M. Luck, M. Fisher, and C. Preist, editors, *Foundations and Applications of Multi-Agent Systems, Workshops of the UK Special Interest Group on Multi-Agent Systems, UK-MAS 1996-2000*, volume 2403 of *Lecture Notes in Artificial Intelligence*, pages 152–168. Springer-Verlag, Berlin, Germany, 2002. 74, 121

[FB02]   Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002. 68, 98

[Fer92]   Innes A. Ferguson. *TouringMachines: an architecture for dynamic, rational mobile agents*. PhD thesis, University of Cambridge, Cambridge, UK, 1992. 72, 125

[Fer99]   Jacques Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*. Addison Wesley Longman, Harlow, UK, 1999. 36, 69, 73, 75, 154, 163

[FF96]   Daniel P. Friedman and Matthias Felleisen. *The little Schemer*. MIT Press, Cambridge, MA, USA, 4th edition, 1996. 106

[FFG⁺04]   Ian Foster, Jeffrey Frey, Steve Graham, Steve Tuecke, Karl Czajkowski, Donald F. Ferguson, Frank Leymann, Martin Nally, Igor Sedukhin, David Snelling, Tony Storey, William Vambenepe, and Sanjiva Weerawarana. Modeling stateful resources with Web services. Whitepaper Ver. 1.1, The Globus Alliance, May 2004. 22, 82, 87

[FFJ90]   John Franco, Daniel P. Friedman, and Steven D. Johnson. Multi-way streams in Scheme. *Computer Languages*, 15(2):109–125, 1990. 140

[FGM03]   Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: an organizational view of multi-agent systems. In P. Giorgini, J. P. Müller, and J. Odell, editors, *4th International Workshop on Agent-Oriented Software Engineering, AOSE'03*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230, Melbourne, Australia, July 2003. Springer-Verlag. 75, 96, 198

[Fip02a]   FIPA-ACL message structure specification. FIPA specifications SC00061G, Foundation for Intelligent Physical Agents, December 2002. www.fipa.org/specs/fipa00061/. 74

[Fip02b]   FIPA communicative act library specification. FIPA specifications SC00037J, Foundation for Intelligent Physical Agents, December 2002. www.fipa.org/specs/fipa00037. 74, 120

[Fip02c]   FIPA contract net interaction protocol specification. FIPA specifications SC00029H, Foundation for Intelligent Physical Agents, December 2002. www.fipa.org/specs/fipa00029. 76

[FJK04]   Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: why Grid and agents need each other. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'04*, volume 1, pages 8–15, New York, NY, USA, July 2004. 22, 92, 153, 154

[FK99a]   Ian Foster and Carl Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5-6):607–621, October 1999. 84

[FK99b]   Ian Foster and Carl Kesselman, editors. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, 1999. 82, 84, 87

[FK03]   Ian Foster and Carl Kesselman, editors. *The Grid 2: blueprint for a new computing infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, 2003. 87

[FKNT02]   Ian Foster, Carl Kesselman, Jeff Nick, and Steve Tuecke. The physiology of the Grid: an Open Grid Services Architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*. The Globus Alliance, June 2002. 87, 154

[FKT01]   Ian Foster, Carl Kesselman, and Steve Tuecke. The anatomy of the Grid: enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, 2001. 82, 84, 85, 86

[FKTT98]   Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational Grids. In *5th ACM conference on Computer and communications security*, pages 83–92, San Francisco, CA, USA, November 1998. ACM Press. 84

[FM01]   Dieter Fensel and Enrico Motta. Structured development of problem solving methods. *Transactions on Knowledge and Data Engineering*, 13(6):913–932, November/December 2001. 67

[FW76]   Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *3rd International Colloquium on Automata, Languages and Programming, ICALP'76*, pages 257–284, Edinburgh, UK, July 1976. Edinburgh University Press. 140

[GB99]   Zahia Guessoum and Jean-Pierre Briot. From active objects to autonomous agents. *Concurrency*, 7(3):68–76, July-September 1999. 115

[GC04]   Dominic Greenwood and Monique Calisti. Engineering Web service - agent integration. In *IEEE Systems, Cybernetics and Man Conference, SMC'04*, The Hague, Netherlands, October 2004. IEEE Computer Society. 80, 81

[GCL04]   Aram Galstyan, Karl Czajkowski, and Kristina Lerman. Resource allocation in the Grid using reinforcement learning. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'04*, volume 3, pages 1314–1315, New York, NY, USA, August 2004. IEEE Computer Society. 82, 93

[Gel04]   Marije Geldof. The Semantic Grid: will Semantic Web and Grid go hand in hand? Technical report, European Commission DG Information Society Unit 'Grid technologies', June 2004. 82, 97

[GFM00]   Olivier Gutknecht, Jacques Ferber, and Fabien Michel. The MadKit agent platform architecture. Tecnical report, University Montpellier 2, Montpellier, France, April 2000. www.madkit.org. 186, 198

[GGT94]   Marie-Pierre Gleizes, Pierre Glize, and Sylvie Trouilhet. Étude des lois de la conversation entre agents autonomes. *Revue Internationale de Systémique*, 8(1):39–50, 1994. 78

[GHB00]   Mark Greaves, Heather Holmback, and Jeffrey Bradshaw. What is a conversation policy? In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 118–131. Springer-Verlag, Berlin, Germany, 2000. 76, 77

[GHCN99]   Robert Ghanea-Hercock, Jaron C. Collis, and Divine T. Ndumu. Co-operating mobile agents for distributed parallel processing. In *3rd International Conference on Autonomous Agents*, pages 398–399, Seattle, WA, USA, May 1999. ACM Press. 93

[GI89]   Michael P. Georgeff and François F. Ingrand. Decision making in an embedded reasoning system. In *11th International Joint Conference on Artificial Intelligence, IJCAI'89*, volume 1, pages 972–978, Detroit, MI, USA, August 1989. Morgan Kaufmann. 71

[GMM98]   Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated electronic commerce: a survey. *The Knowledge Engineering Review*, 13(2):147–159, July 1998. 80

[GO94]   Tom R. Gruber and Gregory R. Olsen. An ontology for engineering mathematics. In J. Doyle, P. Torasso, and E. Sandewall, editors, *4th International Conference on Principles of Knowledge Representation and Reasoning, KR'04*, Bonn, Germany, May 1994. Morgan Kaufmann. 66

[GR04]   Carole Goble and David De Roure. The Semantic Grid: myth busting and bridge building. In R. López de Mántaras and L. Saitta, editors, *16th European Conference on Artificial Intelligence, ECAI'04*, pages 1129–1135, Valencia, Spain, August 2004. IOS Press. 22

[Gru93]   Tom R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, June 1993. 66

[Gue02]   Francis Guerin. *Specifying agent communication languages*. PhD thesis, Imperial College of Science, University of London, London, UK, June 2002. 74, 76, 95

[GW04]   Dina Goldin and Peter Wegner. The origins of the Turing thesis myth. Technical report CS 04-13, Brown University, Providence, RI, USA, June 2004. 21, 44, 105

[Had95]   Afsaneh Haddadi. Towards a pragmatic theory of interactions. In V. Lesser and L. Gasser, editors, *1st International Conference on Multi-Agent Systems, ICMAS'95*, pages 133–139, Menlo Park, CA, USA, June 1995. AAAI Press. 71, 125

[Hel04]   OWL Web Ontology Language use cases and requirements. W3C recommendation, World Wide Web Consortium, February 2004. 66, 204

[Hew77]   Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977. 59, 73

[HJM05]   Nadim Haque, Nicholas R. Jennings, and Luc Moreau. Resource allocation in communication networks using market-based agents. *Knowledge-Based Systems*, 18(4-5):163–170, August 2005. 93

[HK03]   Marc-Philippe Huget and Jean-Luc Koning. Interaction protocol engineering. In M-P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *Lecture Notes in Artificial Intelligence*, pages 209–222. Springer-Verlag, Berlin, Germany, 2003. 76

[HNL02]   James E. Hanson, Prabir Nandi, and David W. Levine. Conversation-enabled Web services for agents and e-business. In *3rd International Conference on Internet Computing, IC'02*, pages 791–796, Las Vegas, NV, USA, June 2002. 65, 95

[HS98]   Michael N. Huhns and Munindar P. Singh, editors. *Readings in agents*. Morgan Kaufmann, San Francisco, CA, USA, 1998. 36, 69

[HSB+05]   Michael N. Huhns, Munindar P. Singh, Mark Burstein, Keith Decker, Ed Durfee, Tim Finin, Les Gasser, Hrishikesh Goradia, Nicholas R. Jennings, Kiran Lakkaraju, Hideyuki Nakashima, Van Parunak, Jeffrey S. Rosenschein, Alicia Ruvinsky, Gita Sukthankar, Samarth Swarup, Katia Sycara, Milind Tambe, Tom Wagner, and Laura Zavala. Research directions for service-oriented multiagent systems. *Internet Computing*, 9(6):65–70, November-December 2005. 22, 27, 80, 92

[Hug89]   John Hughes. Why functional programming matters. *Computer*, 32(2):98–107, 1989. 140

[Hug01]   Marc-Philippe Huget. *Une ingénierie des protocoles d'interaction pour les systèmes multi-agents*. PhD thesis, University Paris IX, Paris, France, June 2001. 76

[Hug03]   Marc-Philippe Huget, editor. *Communication in multiagent systems, agent communication languages and conversation policies*, volume 2650 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2003. 75

[Huh02]   Michael N. Huhns. Agents as Web services. *Internet Computing*, 6(4):93–95, July-August 2002. 80

[IJF92]   John W. Simmons II, Stanley Jefferson, and Daniel P. Friedman. Language extension via first-class interpreters. Technical report TR362, Indiana University, Bloomington, IN, USA, September 1992. 110

[IT83]   Tetsuo Ida and Jiro Tanaka. Functional programming with streams. In R. E. A. Mason, editor, *9th IFIP World Computer Congress, Information Processing 83*, pages 265–270, Paris, France, September 1983. 140

[IYT04]   Fuyuki Ishikawa, Nobukazu Yoshioka, and Yasuyuki Tahara. Toward synthesis of Web services and mobile agents. In *2nd International Workshop on Web Services and Agent Based Engineering, WSABE'04*, pages 48–55, New York, NY, USA, July 2004. 80, 81

[JC03]   Clement Jonquet and Stefano A. Cerri. Cognitive Agents Learning by Communicating. In *Colloque Agents Logiciels, Coopération, Apprentissage et Activité Humaine, ALCAA'03*, pages 29–39, Bayonne, September 2003. 127

[JC05a]   Clement Jonquet and Stefano A. Cerri. i-dialogue: modeling agent conversation by streams and lazy evaluation. In *International Lisp Conference, ILC'05*, pages 219–228, Stanford University, CA, USA, June 2005. 22

[JC05b]   Clement Jonquet and Stefano A. Cerri. The STROBE model: Dynamic Service Generation on the Grid. *Applied Artificial Intelligence, Special issue on Learning Grid Services*, 19(9-10):967–1013, October-November 2005. 22

[JC06]   Clement Jonquet and Stefano A. Cerri. Characterization of the Dynamic Service Generation concept. Research report 06007, University Montpellier 2, France, February 2006. www.lirmm.fr/˜ jonquet/Publications. 22

[JDC06a]   Clement Jonquet, Pascal Dugenie, and Stefano A. Cerri. AGIL specifications. Research report 06030, University Montpellier 2, France, May 2006. www.lirmm.fr/˜ jonquet/Publications. 22, 231

[JDC06b]   Clement Jonquet, Pascal Dugenie, and Stefano A. Cerri. Service-based integration of Grid and multi-agent systems models. Research report 06012, University Montpellier 2, France, February 2006. www.lirmm.fr/˜ jonquet/Publications. 22

[JDC07]   Clement Jonquet, Pascal Dugenie, and Stefano A. Cerri. Agent-Grid Integration Language. *Multiagent and Grid Systems*, 2007. Accepted for publication - Expected middle 2007 - Se also [JDC06a]. 22

[JEC05]   Clement Jonquet, Marc Eisenstadt, and Stefano A. Cerri. Learning agents and Enhanced Presence for generation of services on the Grid. In P. Ritrovato, C. Allison, S.A. Cerri, T. Dimitrakos, M. Gaeta, and S. Salerno, editors, *Towards the Learning GRID: advances in Human Learning Services*, volume 127 of *Frontiers in Artificial Intelligence and Applications*, pages 203–213. IOS Press, November 2005. 47, 99

[Jef99]   Keith Jeffery. Knowledge, Information and Data. Research report, Central Laboratory of the Research Councils, UK, September 1999. 82

[Jen01]   Nicholas R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, April 2001. 36, 69, 154

[JF92]   Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. In *International Workshop on Reflection and Meta-level architecture, IMSA'92*, Tokyo, Japan, November 1992. 109, 119, 195

[JFL$^+$01]   Nicholas R. Jennings, Peyman Faratin, Alessio R. Lomuscio, Simon Parsons, Carles Sierra, and Michael Wooldridge. Automated negotiation: prospects, methods and challenges. *Group Decision and Negotiation*, 10(2):199–215, March 2001. 79

[JFN$^+$00]   Nicholas R. Jennings, Peyman Faratin, Timothy J. Norman, Paul O'Brien, and Brian Odgers. Autonomous agents for business process management. *Applied Artificial Intelligence*, 14(2):145–189, February 2000. 71

[Joh89]   Steven D. Johnson. How Daisy is lazy: suspending construction at target levels. Technical report 286, Indiana University Computer Science Department, Bloomington, IN, USA, August 1989. 48, 136, 140

[Jon03]   Clement Jonquet. Communication agent et interprétation Scheme pour l'apprentissage au méta-niveau. Master's thesis, University Montpellier 2, Montpellier, France, June 2003. 127

[Jon05]   Clement Jonquet. A framework and ontology for Semantic Grid services: an integrated view of WSMF and WSRF. Unpublished draft research report, University Montpellier 2, France and KMi, Open University, UK, May 2005. 97

[KCe98]   Richard Kelsey, William Clinger, and Jonathan Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998. 106, 110

[KLM96]   Leslie P. Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: a survey. *Artificial Intelligence Research*, 4:237–285, January-June 1996. 46

[KP01]   Jean-Luc Koning and Sylvie Pesty. Modèles de communication. In J-P. Briot and Y. Demazeau, editors, *Principes et architectures des systèmes multi-agents*, Traité IC2 - Informatique et Systèmes d'Information, chapter 3, pages 109–137. Hermès, Paris, France, 2001. 74, 75, 121

[KWvL02]   Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL: a workflow framework for Grid services. Draft paper, Globus Alliance, July 2002. 82, 99

[Lan65]   Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, February 1965. 140

[Lau99]   Diana Laurillard. A conversational framework for individual learning applied to the 'learning organisation' and the 'learning society'. *Systems Research and Behavioral Science*, 16(2):113–122, March 1999. 46

[LF97]   Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical report TR-CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland, Baltimore, MD, USA, February 1997. www.cs.umbc.edu/kqml/. 74, 121

[LL04]   Chunlin Li and Layuan Li. Competitive proportional resource allocation policy for computational Grid. *Future Generation Computer Systems*, 20(6):1041–1054, August 2004. 82, 93

[LLA$^+$03]   Rubén Lara, Holger Lausen, Sinuhé Arroyo, Jos de Bruijn, and Dieter Fensel. Semantic Web services: description requirements and current technologies. In *International Workshop on Electronic Commerce, Agents, and Semantic Web Services*, Pittsburgh, PA, USA, September 2003. 68

[LRCSN03] Margaret Lyell, Lowell Rosen, Michelle Casagni-Simkins, and David Norris. On software agents and Web services: usage and design concepts and issues. In *1st International Workshop on Web Services and Agent Based Engineering, WSABE'03*, Melbourne, Australia, July 2003. 80, 81

[LSL⁺00] Diana Laurillard, Matthew Stratfold, Rose Luckin, Lydia Plowman, and Josie Taylor. Affordances for learning in a non-linear narrative medium. *Interactive Media in Education*, 2, August 2000. 46

[LT03] Craig Lee and Domenico Talia. Grid programming models: current tools, issues and directions. In G. Fox F. Berman and A. J. G. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 21, pages 555–578. John Wiley & Sons, 2003. 87

[Mau02] Nicolas Maudet. A la recherche de la structure intentionnelle dans le dialogue. *Traitement automatique des langues*, 43(2):71–98, 2002. 78

[MBB⁺05] Luc Moreau, Jeff Bradshaw, Maggie Breedy, Lary Bunch, Matt Johnson, Shri Kulkarni, James Lott, Niranjan Suri, and Andrzej Uszok. Behavioural specification of Grid services with the KAoS policy language. In *5th IEEE International Symposium on Cluster Computing and the Grid, CCGRID'05*, volume 2, pages 816– 823, Cardiff, UK, May 2005. IEEE Computer Society. 65, 82

[MBG03] Libero Maesano, Christian Bernard, and Xavier Le Galles. *Services Web avec J2EE et .NET Conception et implémentations*. Eyrolles, Paris, France, 2003. 57, 58, 59

[MBP05] Sunilkumar S. Manvi, M. N. Birje, and Bhanu Prasad. An agent-based resource allocation model for computational Grids. *Multiagent and Grid Systems*, 1(1):17–27, 2005. 93

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960. 106

[McC89] John McCarthy. Elephant 2000: a programming language based on speech acts. Unpublished draft, Stanford University, CA, USA, 1989. 105, 118

[MCd02] Nicolas Maudet and Brahim Chaib-draa. Commitment-based and dialogue-game based protocols - News trends in agent communication language. *The Knowledge Engineering Review*, 17(2):157–179, June 2002. 76, 78

[McG04] OWL Web Ontology Language overview. W3C recommendation, World Wide Web Consortium, February 2004. 68

[MDCd99] Bernard Moulin, Sylvain Delisle, and Brahim Chaib-draa, editors. *Analyse et Simulation de conversation : De la théorie des actes de discours aux systèmes multiagents*. L'interdisciplinaire, Limonest, France, 1999. 78

[MML05] Zakaria Maamar, Soraya K. Mostéfaoui, and Mohammed Lahkim. Web services composition using software agents and conversations. In D. Benslimane, editor, *Les services Web*, volume 10 of *RSTI-ISI*. Lavoisier, 2005. 95

[Mor02] Luc Moreau. Agents for the Grid: a comparison with Web services (part 1: the transport layer). In H. E. Bal, K-P. Lohr, and A. Reinefeld, editors, *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID'02*, pages 220–228, Berlin, Germany, May 2002. IEEE Computer Society. 80, 81, 94

[Mot99]   Enrico Motta. *Reusable components for knowledge modelling: case studies in parametric design problem solving*, volume 53 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, Netherlands, 1999. 124

[MPD⁺03]   Simon Miles, Juri Papay, Vijay Dialani, Michael Luck, Keith Decker, Terry Payne, and Luc Moreau. Personalised Grid service discovery. *IEE Proceedings Software, Special issue on Performance Engineering*, 150(4):252–256, August 2003. 65, 82

[MPM⁺04]   David L. Martin, Massimo Paolucci, Sheila A. McIlraith, Mark H. Burstein, Drew V. McDermott, Deborah L. McGuinness, Bijan Parsia, Terry R. Payne, Marta Sabou, Monika Solanki, and Naveen Srinivasan andKatia P. Sycara. Bringing semantics to Web services: The OWL-S approach. In J. Cardoso and A. P. Sheth, editors, *1st International Workshop on Semantic Web Services and Web Process Composition, SWSWPC'04*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42, San Diego, CA, USA, July 2004. Springer-Verlag. 68

[MRG98]   Luc Moreau, Daniel Ribbens, and Pascal Gribomont. Advanced programming techniques using Scheme. In *9th Journées Francophones des Langages Applicatifs*, pages 69–90, Como, Italy, February 1998. INRIA, Collection Didactique. 107

[MS03]   E. Michael Maximilien and Munindar P. Singh. Agent-based architecture for autonomic Web service selection. In *1st International Workshop on Web Services and Agent Based Engineering, WSABE'03*, Sydney, Australia, July 2003. 80

[MT99]   Frank Manola and Craig Thompson. Characterizing the agent Grid. Technical report 990623, Object Services and Consulting, Inc., June 1999. 91, 154

[Mus98]   Mark A. Musen. Modern architectures for intelligent systems: reusable ontologies and problem-solving methods. In C.G. Chute, editor, *American Medical Informatics Association Annual Symposium, AMIA'98*, pages 46–52, Orlando, FL, USA, November 1998. 67

[NGW05]   Roger Nkambou, Guy Gouardères, and Beverly P. Woolf. Toward learning Grid infrastructures: an overview of research on Grid learning services. *Applied Artificial Intelligence, Special issue on Learning Grid Services*, 19(9-10):811–824, October-November 2005. 82, 98

[Nif04a]   Roger Nifle. Le sens du service – Analyse des sens et cohérences humaines. Le journal permanent de l'Humanisme Méthodologique, http://journal.coherences.com, July 2004. 31, 32

[Nif04b]   Roger Nifle. Les services sur Internet – Echec mode d'emploi. Le journal permanent de l'Humanisme Méthodologique, http://journal.coherences.com, July 2004. 31, 32

[Nor91]   Kurt Normark. Simulation of object-oriented and mechanisms in Scheme. Technical Report R-90-01, Institute of Electronic Systems, Aalborg University, Denmark, 1991. 110, 194

[NPC⁺04]   Timothy J. Norman, Alun Preece, Stuart Chalmers, Nicholas R. Jennings, Michael Luck, Viet D. Dang, Thuc D. Nguyen, Vikas Deora, Jianhua Shao, Alex Gray, and Nick J. Fiddian. Agent-based formation of virtual organisations. *Knowledge Based Systems*, 17(2-4):103–111, May 2004. 93

[O'D85]   John T. O'Donnell. Dialogues: a basis for constructing programming Environments. *ACM SIGPLAN Notices*, 20(7):19–27, June 1985. Proceedings of the symposium on Language issues in programming environments, Seattle, WA, USA, June 1985. 24, 135, 136, 137, 138, 151, 203

[OKT+05]   Martin O'Connor, Holger Knublauch, Samson W. Tu, Benjamin N. Grosof, Mike Dean, William E. Grosso, and Mark A. Musen. Supporting Rule System Interoperability on the Semantic Web with SWRL. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *4th International Semantic Web Conference, ISWC'05*, volume 3729 of *Lecture Note in Computer Science*, pages 974–986, Galway, Ireland, November 2005. Springer-Verlag. 178, 204

[OPB00]   James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *1st International Workshop on Agent-Oriented Software Engineering, AOSE'00*, pages 121–140, Limerick, Ireland, June 2000. Springer-Verlag. 76

[Pel03]   Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003. 61, 65

[Pet05]   Jan Peters. Integration of mobile agents and Web services. In *1st European Young Researchers Workshop on Service-Oriented Computing, YR-SOC'05*, pages 53–58, Leicester, UK, April 2005. Software Technology Research Laboratory, De Montfort University. 80, 81

[PTJ+05]   J. Patel, W. T. L. Teacy, N. R. Jennings, M. Luck, S. Chalmers, N. Oren, T. J. Norman, A. Preece, P. M. D. Gray, G. Shercliff, P. J. Stockreisser, J. Shao, W. A. Gray, N. J. Fiddian, and S. Thompson. Agent-based virtual organisations for the Grid. *Multiagent and Grid Systems*, 1(4):237–249, 2005. 93

[PW05]   Shamimabi Paurobally and Michael Wooldridge. A Critical Analysis of the WS-Agreement Specification. Technical report EU Ontogrid project FP6-511513, University of Liverpool, Liverpool, UK, November 2005. 66

[PWF+02]   Laura Pearlman, Von Welch, Ian Foster, Carl Kesselman, and Steven Tuecke. A Community Authorization Service for group collaboration. In *3rd International Workshop on Policies for Distributed Systems and Networks, POLICY'02*, pages 50–59, Monterey, CA, USA, June 2002. IEEE Computer Society. 84

[Que96]   Christian Queinnec. *Lisp in small pieces*. Cambridge University Press, Cambridge, UK, 1996. 48, 106, 109, 119, 194

[Que00]   Christian Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In *5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, number 9, pages 23–33, Montreal, Canada, September 2000. 48

[RAC+05]   Pierluigi Ritrovato, Colin Allison, Stefano A. Cerri, Theo Dimitrakos, Mateo Gaeta, and Saverio Salerno, editors. *Towards the learning Grid: advances in human learning services*, volume 127 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, November 2005. 82, 98

[RG91]   Anand S. Rao and Michael P. Georgeff. Modelling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, San Mateo, CA, USA, April 1991. Morgan Kaufmann. 70, 71, 125

[RJS01]   David De Roure, Nicholas R. Jennings, and Nigel Shadbolt. Research agenda for the Semantic Grid: a future e-science infrastructure. Technical report, University of Southampton, UK, June 2001. Report commissioned for EPSRC/DTI Core e-Science Programme. 82, 83, 91, 96, 97, 154

[RJS05]   David De Roure, Nicholas R. Jennings, and Nigel R. Shadbolt. The Semantic Grid: past, present, and future. *Proceedings of the IEEE*, 93(3):669–681, March 2005. 82, 92, 96

[RKL⁺05]   Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005. 68

[RM00]   Omer F. Rana and Luc Moreau. Issues in building agent based computational Grids. In *3rd Workshop of the UK Special Interest Group on Multi-Agent Systems, UKMAS'00*, Oxford, UK, December 2000. 82, 83, 91, 154

[Rob83]   John A. Robinson. Logic programming - Past, present and future. *New Generation Computing*, 1(2):107–124, 1983. 125

[RPD99]   Pierre-Michel Ricordel, Sylvie Pesty, and Yves Demazeau. About conversations between multiple agents. In *1st International Workshop of Central and Eastern Europe on Multi-agent Systems, CEEMAS'99*, pages 203–210, St. Petersburg, Russia, June 1999. 77, 78

[RvSBS03]   Debbie Richards, Sander van Splunter, Frances M.T. Brazier, and Marta Sabou. Composing Web services using an agent factory. In *1st Workshop on Web Services and Agent-Based Engineering, WSABE'03*, pages 57–66, Melbourne, Australia, July 2003. 65

[Sea69]   John Searle. *Speech acts: an essay in the philosophy of language*. Cambridge University Press, Cambridge, UK, 1969. 73

[SH99]   Munindar P. Singh and Michael N Huhns. Multiagent systems for workflow. *Intelligent Systems in Accounting, Finance and Management*, 8(2):105–117, June 1999. 94

[SH05]   Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing, Semantics, processes, agents*. John Wiley & Sons, 2005. 20, 21, 22, 29, 42, 43, 53, 80, 85, 111, 155, 156

[SHMS04]   Amal El Fallah Seghrouchni, Serge Haddad, Tarak Melitti, and Alexandru Suna. Interopérabilité des systèmes multi-agents à l'aide des services Web. In O. Boissier and Z. Guessoum, editors, *12èmes Journées Francophones sur les Systèmes Multi-Agents, JF-SMA'04*, pages 91–104, Paris, France, November 2004. Hermès. 80

[Sho93]   Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993. 70, 71, 74, 120

[SHP03]   Evren Sirin, James Hendler, and Bijan Parsia. Semi-automatic composition of Web services using semantic descriptions. In *1st Internatonal Workshop Web Services: Modeling, Architecture and Infrastructure*, Angers, France, April 2003. 65

[Sia91]   Sati S. Sian. Adaptation based on cooperative learning in multi-agent systems. In Y. Demazeau and J.-P. Müller, editors, *Decentralized AI, 2nd European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 2, pages 257–272. Elsevier Science Publishers, Amsterdam, Netherlands, 1991. 76

[Sin98]   Munindar P. Singh. Agent communication languages: rethinking the principles. *Computer*, 31(12):40–47, December 1998. 72, 75, 78

[SJ75]   Gerald J. Sussman and Guy L. Steele Jr. Scheme: an interpreter for extended lambda calculus. AI lab memo AIM-349, MIT AI Lab., Cambridge, MA, USA, December 1975. 106

[SK03]   Biplav Srivastava and Jana Koehler. Web service composition - Current solutions and open problems. In *Workshop on Planning for Web Services*, pages 28–35, Trento, Italy, June 2003. 65

[SLGW02]   Weiming Shen, Yangsheng Li, Hamada H. Ghenniwa, and Chun Wang. Adaptive negotiation for agent-based Grid computing. In *1st International Agentcities Workshop on Challenges in Open Agent Environments*, pages 32–36, Bologna, Italy, July 2002. 93

[SMZ01]   Tran C. Son Sheila McIlraith and Honglei Zeng. Semantic Web services. *Intelligent Systems*, 16(2):46–53, March-April 2001. 68

[SS04]   Steffen Staab and Rudi Studer, editors. *Handbook on ontologies*. International Handbooks on Information Systems. Springer, 2004. 67

[SvdAB$^+$03] Steffen Staab, Wil van der Aalst, V. Richard Benjamins, Amit Sheth, John A. Miller, Christoph Bussler, Alexander Maedche, Dieter Fensel, and Dennis Gannon. Web services: been there, done that? *Intelligent Systems*, 18(1):72–85, January-February 2003. 64

[Tia05]   Huaglory Tianfield. Towards agent based Grid resource management. In *5th IEEE International Symposium on Cluster Computing and the Grid, CCGRID'05*, volume 1, pages 590–597, Cardiff, UK, May 2005. IEEE Computer Society. 93

[TV01]   Orazio Tomarchio and Lorenzo Vita. On the use of mobile code technology for monitoring Grid systems. In *1st International Workshop on Agent-based Cluster and Grid computing*, pages 450–455, Brisbane, Australia, May 2001. IEEE Computer Society. 94

[VBH04]   José M. Vidal, Paul Buhler, and Christian Stahl Humboldt. Multiagent systems with workflows. *Internet Computing*, 8(1):76–82, January-February 2004. 94

[VDB97]   Egon Verharen, Frank Dignum, and Sander Bos. Implementation of a cooperative agent architecture based on the language-action perspective. In M. P. Singh and A. S. Rao, editors, *4th International Workshop on Intelligent Agents, Agent Theories, Architectures, and Languages, ATAL'97*, volume 1365 of *Lecture Notes In Computer Science*, pages 31–44, Providence, RI, USA, July 1997. Springer-Verlag. 72, 125

[Vog03]   Werner Vogels. Web services are not distributed objects. *Internet Computing*, 7(6):59–66, November-December 2003. 63

[WBPB03]   Rich Wolski, John Brevik, James S. Plank, and Todd Bryan. Grid resource allocation and control using computational economies. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*, pages 747–772. John Wiley & Sons, 2003. 93

[Weg96]   Peter Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285–287, March 1996. 48, 54

[Weg97]   Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997. 44, 105

[WF86]   Terry Winograd and Fernando Flores. *Understanding computers and cognition: a new foundation for design*. Ablex Publishing Co., Norwood, NJ, USA, 1986. 75, 76

[WG99]   Peter Wegner and Dina Goldin. Interaction, computability, and Church's thesis. Draft paper. Accepted to the British Computer Journal, May 1999. 44, 48, 139, 188

[WG03]   Peter Wegner and Dina Goldin. Computation beyond turing machines. *Communications of the ACM*, 46(4):100–102, April 2003. 44, 105

[Wit92]   Thies Wittig, editor. *ARCHON an architecture for multi-agent systems*. Ellis Horwood, Chichester, UK, 1992. 71, 125

[WJK00]   Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000. 75, 96

[Woo02]   Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, Chichester, UK, February 2002. 36, 69, 73

[WOvSB02]   Niek J.E. Wijngaards, Benno J. Overeinder, Marteen van Steen, and Frances M.T. Brazier. Supporting internet-scale multi-agent systems. *Data & Knowledge Engineering*, 41(2-3):229–245, June 2002. 93

[WW02]   Stefan Wesner and Konrad Wulf. How Grid could improve e-learning in the environmental science domain. In *1st LeGE-WG International Workshop on Educational Models for Grid Based Services*, Lausanne, Switzerland, September 2002. Electronic Workshops in Computing. 98

[Zhu04]   Hai Zhuge. *The Knowledge Grid*. World Scientific, Singapore, 2004. 83, 97

Abstract. _____

This thesis deals with modelling dynamic service exchange. The notion of service is now at the centre of distributed system development ; it plays a key role in their implementation and success. The thesis proposes firstly a **reflection about the notion of service and introduces the concept of Dynamic Service Generation (DSG)** as a different way to provide and use services in a computer-mediated context: services are dynamically constructed, provided and used by agents (human or artificial) within a community, by means of a conversation. In particular, two major characteristics of DSG are highlighted: an **agent** and **Grid** oriented aspect of service exchange. Therefore, the thesis proposes an integration of three research domains in Informatics: Service-Oriented Computing (SOC), Multi-Agents System (MAS) and GRID. The thesis contributions consists of three main aspects: The proposal of (i) a **new agent representation and communication model, called *STROBE***, that enables agents to develop different languages for each agent they communicate with. STROBE agents are able to interpret communication messages and execute services in a given dynamic and dedicated conversation context; (ii) a **computational abstraction, called *i-dialogue*** (intertwined dialogues) that models multi-agent conversations by means of fundamental constructs of applicative/functional languages (i.e., streams, lazy evaluation and higher-order functions); (iii) a **service-oriented GRID-MAS integrated model** based on the representation of agent capabilities as Grid services. In this model, concepts of GRID and MAS, relations between them and the integration rules are semantically described by a set-theory formalization and a common graphical description language, **called *Agent-Grid Integration Language* (AGIL)**. AGIL integrates the thesis results together by formalizing agent interactions for service exchange on the Grid.

Keywords: Service-Oriented Computing (SOC), Multi-Agents System (MAS), GRID, agent interaction, Web service, Grid service, dynamic service generation

Résumé. _____

L'objet de cette thèse est la modélisation de l'échange dynamique de services. La notion de service joue désormais un rôle clé dans le développement, la diffusion et l'implémentation des systèmes distribués. Cette thèse propose **une réflexion sur la notion de service et introduit le concept de Génération Dynamique de Service (GDS)** comme une approche différente de l'échange de service en informatique, dans laquelle des services sont dynamiquement construits, fournis et utilisés par des agents (humains ou artificiels). Ces échanges de services sont basés sur des conversations qui ont lieu au sein de différentes communautés. Deux caractéristiques de la GDS sont particulièrement mises en avant : l'aspect orienté **agent** et l'aspect orienté **Grid**. La thèse se situe donc à l'intersection de trois domaines : le Service-Oriented Computing (SOC), les Systèmes Multi-Agents (SMA) et GRID. Les trois contributions majeures sont : (i) la proposition d'un **nouveau modèle de représentation et de communication agent, appelé *STROBE***, qui permet aux agents de développer dynamiquement un langage différent pour chacun de leurs interlocuteurs. Ils sont capables d'interpréter des messages et d'exécuter des services dans des contextes de conversation dédiés; (ii) **une fonction, appelée *i-dialogue*, qui modélise les conversations agents** à l'aide des principes de la programmation applicative/fonctionnelle (i.e., flots, évaluation paresseuse, procédures de première classe) ; (iii) **un modèle d'intégration GRID-SMA** qui représente les capacités des agents par des services Grid. Dans ce modèle, un langage formel, **appelé *Agent-Grid Integration Language* (AGIL)**, décrit sémantiquement et graphiquement les concepts clés de GRID et SMA, leurs relations, ainsi que les règles de leur intégration. AGIL intègre tous les résultats de la thèse en proposant une formalisation des interactions entre agents pour l'échange de services sur le Grid.

Mots-clés: Service-Oriented Computing (SOC), Systèmes Multi-Agents (SMA), GRID (Grille), interaction agent, Service Web, Service Grid, génération dynamique de service