The STROBE Model: Dynamic Service Generation on the Grid

Clement Jonquet, Stefano A. Cerri

The STROBE model: Dynamic Service Generation on the Grid*

Clement Jonquet and Stefano A. Cerri

LIRMM, CNRS & University Montpellier II
161 Rue Ada, 34392 Montpellier Cedex 5, France
{jonquet,cerri}@lirmm.fr

Abstract

This article presents the STROBE model: both an agent representation and an agent communication model based on a social approach, that means interaction centred. This model represents how agents may realise the interactive, dynamic generation of services on the Grid. Dynamically generated services embody a new concept of service implying a collaborative creation of knowledge i.e. learning; services are constructed interactively between agents depending on a conversation. The approach consists of integrating selected features from Multi-Agent Systems and agent communication, language interpretation in applicative/functional programming and e-learning/human-learning into a unique, original and simple view that privileges interactions, yet including control. The main characteristic of STROBE agents is that they develop a language (environment + interpreter) for each of their interlocutors. 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 Social Informatics, where artificial as well as human agents operate by communicating as well as by computing. The paper shows how the model may not only account for the classical communicating agent approaches, but also represent a fundamental advance in modelling societies of agents in particular in Dynamic Service Generation scenarios such as those necessary today on the Web and proposed tomorrow for the Grid. Preliminary concrete experimentations illustrate the potential of the model; they are significant examples for a very wide class of computational and learning situations.

Keywords: Interaction, Multi-Agent Systems, Agent Communication, Agent Representation, Learning, Services, Dynamic Service Generation, Grid, Functional/Applicative Programming, Social Informatics

1 Introduction

1.1 Interaction, Agents and Multi-Agent Systems

Social intelligence is a phenomenon that can not be explained by a ”linear” model: the whole is not the sum of the 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 artefacts (i.e. machines). It is therefore useful to reflect about a fundamental question: **What makes that the whole is greater than the sum of the parts?**

A common answer is interaction. Interactions in computer science is for the moment rather poor, compared to calculus. 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! However, as Turing himself said, Turing Machines could not provide in principle a complete model for all forms of computation i.e. computers can not be reduced to Turing Machines [GW04]. That is why interaction needs to become more and more important in computing aiming to overpass calculus. For example, Wegner and others try to find out the way to shift from classical Turing Machines to Interaction Machines [WG03, Weg97].

In order to progress in this direction, it is not enough to propose the correct priority: **to develop communication models of interacting societies of humans and machines** (i.e. the WHAT to do) and motivate it theoretically or

*This paper is an revised and extended version of [JC04].

practically (i.e. WHY it should be done) but one should also indicate ways to advance concretely within the current available models using technologies taking into account constraints or opportunities (i.e. the HOW to do it). In order to develop communicative models of interacting entities, the first obvious step is to adopt a unified view of these entities i.e. humans and machines. Actually, this is currently done within the agent research community. The most advanced experiences on interactive, adaptive and evolutionary software entities come from the Distributed Artificial Intelligence (DAI) community where these entities are called **agents**. The Agent concept is for the moment (or tries to be) the best operational metaphor of human in computer science. This paper deals with agents. In the rest of the paper, for the sake of simplicity, we consider two types of agents: Human Agents (HA) and Artificial Agents (AA).

Nowadays agents are the most promising way of implementing distributed systems. Current advances in agents are reported in the MAS (Multi-Agent System) literature, for example [Fer99, Woo02]. However, most of this literature, is concerned mainly with MAS consisting of AA. Our ambition goes beyond, includes HA within what we may call MAHAS (Multi Artificial and Human Agents System). A MAHAS is a system where AA and HA could interact and exchange information and knowledge easily, with no constraining limitation related to the nature of an agent. It is a team where computers could propose some ideas to humans and humans to computers. A group with a non limited collaboration and cooperation, where an agent could ask another one to do a task or help it. An autopoietic system which could evolve dynamically in time and with a nondeterministic behaviour. A society where questions (i.e. problems to solve) and their solutions may emerge through interactions.

1.2 Agents generating services one another

Some should consider Internet as the first rudimentary, yet quite impressive, MAHAS in history. This is, only partially true due to the product delivery aspect of the Internet. Actually, the notion of service is now at the centre of development of distributed systems; it plays a key role in their implementations and spreading. But, HA and AA on the Internet deliver each other "ready made informational products", more than real "on the fly generated knowledge services". Basically, the Internet allows for the moment to store and retrieve "products" (e.g. pages, software, images, function application results etc.) but not yet to participate to an agent interactive society. In fact, surfing on the Web is equivalent to ask another computer (on the network) to execute a calculus specified by an algorithm, or to give an answer to a well specified question. The classical and most widespread Web architecture, the Client/Server one, represents this view. We think that in order to become a real MAHAS, **the Web, and tomorrow the Grid, should provide services instead of products**, according to recent research in that domain (cf. Section 4.1). To go in this sense the paper introduces the concept of Dynamic Service Generation (DSG) as the next generation of services, i.e. services constructed on the fly by the provider according to the conversation it has with the user. DSG is in opposition with the classical Product Delivery (PD) approach. For going toward DSG, the notion of service on the Web (Web services, Grid services) has to surpass HTTP protocols, XML standards and the object oriented paradigm to be enriched by DAI research and especially by agent communication improvements simply because the agent paradigm differs from the object one for three important ambitions: autonomy, intelligence and interactive behaviour. **Agents on the Web/Grid have not simply to provide classical Web services; they must allow Dynamic Service Generation.** As Section 2.1 explains, the main difference between DSG and PD is the emerging *conversational process* managed by agents in the first one: future DSG systems need open and dynamic communication models allowing agents to manage these conversational processes which may occur between agents: AA-AA, AA-HA, and also HA-HA, as services may be asked/produced by any kind of agents.

Thus, to go toward MAHAS and DSG we should propose an agent representation model putting interactions between agents at their centre. **This paper presents the STROBE model, for STReam – OBject – Environment, as an agent representation and communication model based on a "social approach", that is to say interaction centred**[1]. The aim of the model is to put at the centre of the agent architecture the communication contexts in which it interprets messages from its interlocutors. The main idea is to develop a communication language for each agent's acquaintance. We assume that a language is basically a pair consisting of: i) a language expression evaluation mechanism and ii) a memory to store these mechanisms and abstractions constructed with the language. They are called *interpreter* and *environment*[2] respectively (as in the classic tradition of programming languages). In the STROBE model, agents are able to interpret messages in a given environment, with a given interpreter both dedicated to the

[1] Interaction centred models need to uniform representation and communication as they intrinsically depend one another.

[2] The term environment is here used with its programming language meaning, that is to say, a structure that binds variables and values. It does not means the world surrounding an agent as in MAS community. This meaning is used by default in the rest of the paper.

sender of these messages. Communication between agents enables to dynamically change values in an environment, and particularly enable these interpreters to change their way of interpreting messages. We will explain how STROBE is thought and constructed as a toolkit for DSG, that is to say a set of abstract components (i.e. tools) which correspond to DSG characteristics. We will also explain how the concept of DSG fits e-learning scenarios, i.e. how dynamically generated services imply learning.

1.3 Organisation of the paper

The rest of the paper is organised as follows: Section 2 is dedicated to the concept of Dynamic Service Generation (opposed to the concept of Product Delivery), and proposes a set of characteristics. The main part of the paper is Section 3 which formalises the STROBE model. The section starts by presenting the fundamental concepts of the model (concerning learning, agent communication, features from functional/applicative programming) before presenting the model itself: Cognitive Environments, Cognitive Interpreter, communication language etc. The end of the section presents some significant experimentations, which aim to illustrate the concrete potential of such a model. Section 4 details the Grid concept as an agent based and service oriented distributed system for learning scenarios. A conclusion, in Section 5, rounds off the paper.

2 Dynamic Service Generation

This section presents the concept of DSG as a new way of thinking the concept of service between entities. It begins by presenting the idea and exposing a set of fundamental differences between the Dynamic Service Generation and Product Delivery approaches. A little overview of the notion of service on today's Web (i.e. Web services) is done. Afterwards, it proposes to define DSG by a set of non-exhaustive characteristics.

2.1 What is Dynamic Service Generation?

In a future world, one may dream that intelligent computers, will operate as doctors, lawyers, teachers, architects, bankers etc. In the same dream, we will be able to acknowledge that the dynamic generation of services in computer science is accomplished. Indeed, we can say that since millennia, human beings generate "dynamically" one another many "services". But what is so difficult in the computerization of the concept of service?

The basic definition of service i.e. *use that one can make of something, someone*, has much influenced the concept of today's services. However, another definition is more relevant: *what one makes for someone* or *the contribution of an answer to the problem of another*. It defines neither a server nor an user but the two entities accomplishing some task one another. In this second definition, the service is seen like a kind of cooperation. Therefore a service does not exist alone, it is inevitably associated to a process achieving an action, a functionality. The user of the service should be able to specify this action which must be created and adapted to his needs. To sum up, **the notion of service should be seen like a relationship between two (or several) entities whose goal is to solve the problem of one among them.** The key element of the service notion is not therefore the gotten product but the relationship created. DSG tries to realise this kind of service.

DSG does not assume the user knows exactly what the service provider can offer him. He[3] finds out and constructs step by step what he wants as the result of the service provider's reactions. In fact, DSG is a nondeterministic process depending on the conversation between two agents. In computer science, 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[4], he does not exactly know what he wants and his needs and solutions appear progressively with the dialogue with the service provider until he obtains satisfaction. Actually, PD can be considered as the result of a one-shot interaction process between a pair [user, service provider] while DSG might be viewed as the result of the activation and management of a process defined by the triplet: [user, service provider, conversational process] as shown in Figure 1.

[3]DSG user are agents, AA or HA, the terms he/him will be used hereafter generically for she/her and it/its.

[4]This is a language abuse because we can not say that "a user require a service". The service is characterised by a service generation process. We would have say "when an user ask a service to be generated" or "when a service is generated for a user".

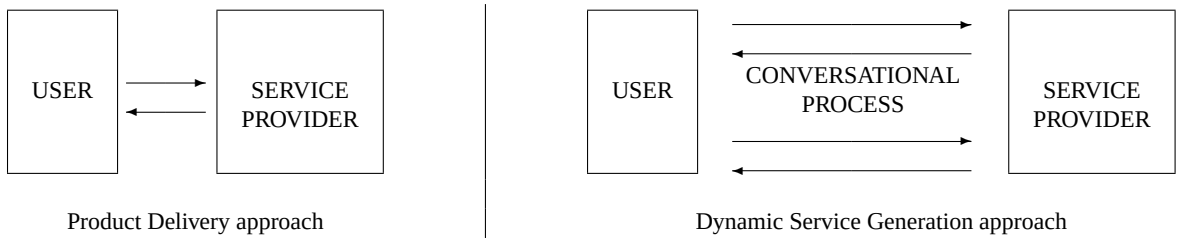


Figure 1: Difference between PD and DSG

In PD approach, the process is in principle deterministic, in the latter it is not: conversations are not a priori determined. The *conversational process* is unique, and represents the dynamic evolution of a conversation. The end of the conversational process implies the end of the DSG which is characterized by a *final result* (as the sum of all the changes implied by interactions). This final result could include a product, of course, but also knowledge, information, emotion, etc. Note that user and/or service provider can be any kind of agents (HA or AA)[5].

2.2 Dynamic Service Generation vs. Product Delivery

The classical definition of PD is: *what is gotten by an activity*. The notion of product puts forward the idea of obtaining the delivery of something already existing, or already designed, whereas the notion of service puts forward an idea of action or creation of something new. As an example in current life, when somebody looks for clothes, *buying ready-to-wear clothes* is analogue to asking for a product, whereas *having clothes made by a tailor* is analogue to requiring a service to be generated. This section defines dialectically the concept of DSG:

- In PD the user exactly knows what he wants. He knows which provider can help him to resolve a clearly defined problem. In DSG, the user does not know exactly what he wants, he can even be unaware of the fact that he needs a service to be generated.
- In PD, the user knows what the system can offer him. In DSG the user elicits and understands what the service provider can offer him by interacting with him. On the other side, the service provider constructs something to offer to the user in the same time.
- In PD, the user knows how to express his request. He should adapt 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.
- Product deliveries are identical one another, for any user. Generated services can never be the same as they depend on a conversational process and a conversation is unique.
- A product is developed by the provider (and is supposed to correspond with a well established and clearly identified need) with a clearly predefined goal for the potential user. At the contrary a service is offered within a service domain, so the user specific objectives have to be defined during the DSG.
- The value added by PD increases with the number of copies distributed. On the contrary, the value added by DSG increases proportionally with the user's satisfaction that entails an indirect publicity for the service provider and generates new users ready to invest more resources in order to have similar services (trust and reputation).
- PD is easily valuable an billable but the same is not so simple for a DSG.
- PD systems are able to announce the result of their use. They can "show" future results. DSG system should gain the trust of the users and they can not announce or guarantee a final result.
- PD systems are inactive when not engaged in a delivery phase. DSG systems are perpetually evolving, learning on their previous generation to improve the next ones. Moreover, product's evolution is slow, as it requires modifications in the conception, design and development (a revision of the whole life cycle). A service evolves

[5]Depending on the situation the *service provider* could be called server or service producer, and the term *user* could be replaced by client, consumer, customer, recipient, requestor etc.

naturally as it is a combination of basic services and products on the fly as a consequence of the conversation with a user.

- PD system are passive, they need to be solicited, whereas DSG system should be proactive, they can take the initiative even if the potential user does not ask for (or is aware he needs) the service.

The DSG approach does not aim to replace PD but rather to improve and enhance it. Generating a service means understand and construct, by interactions, what a user (an agent) needs. If a user has precise needs or precise ideas of what he wants the system to give him, then interactions are simple ones and a "product" may be rapidly delivered. Actually, a system which can generate services, can easily deliver products. However, many scenarios such as, for example, planning a travel, looking for a job, learning a skill, collaborative work etc. can not be modelled within this approach. DSG is needed.

2.3 The concept of service on the Web

The emergence of Service Oriented Architectures (SOA) and Service Oriented Computing (SOC) show the importance of the concept of service in the development of distributed systems. This is historically due to the work of the Open Group with DCE and the OMG with CORBA. Nowadays, the main way of implementing SOA (framework) is Web services[6].

As the W3C defines it, a Web service is: *a software system identified by a URI (Uniform Resource Identifier), whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols*. Web services are based on the three XML-like standard languages[7]: i) WSDL (Web Services Description Language) to describe software components, ii) SOAP (Simple Object Access Protocol) to describe methods for accessing these components, iii) UDDI (Universal Description, Discovery and Integration) to publish a service and to identify a service provider in a service registry. Due to recent work of consortia and companies on Web services composition, another set of languages has to be added to describe Web services business processes[8]. More recently, due to the emergence of Grid architectures, an evolution of Web services to Grid has been proposed with OGSA (Open Grid Services Architecture) [FKNT02] and more recently WSRF (Web Services-Resource Framework) [FFG+04]. We should also mention the recent research interest in Semantic Web Services which is the source of two main ontologies which are Ontology Web Language (OWL-S) and Web Service Modeling Ontology (WSMO)[9].

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. However, the concept of DSG as outlined above is hardly to be recognised. The practice of current Web services is still currently influenced by RPC-like (Remote Procedure Call) and CORBA (Common Object Request Broker Architecture) approaches. Therefore, it has some well recognised weaknesses, as dynamic interoperability, composition, user adaptation, memory. For example, it does not take into account the autonomy of components, or it does not use a high level, history aware, communication language. Web services do not care about persistency of the service, for instance, when a DSG begins with an agent and finishes with another one. The notion of conversation (and its history) is absent in Web services. For all these reasons, we think that Web services are concrete and quickly evolving PD systems, and are the first step of a service based network, but do not represent a solution for DSG requirements as outlined below.

2.4 A set of characteristics of DSG

This section does not propose to formalize DSG systems but rather briefly present a set of non-exhaustive characteristics (or requirements) of DSG which aim to show that DSG is at the crossing of important developments in computer science research:

[6] See `www.opengroup.org/dce` for DCE, see `www.corba.org` for CORBA, and `www.w3.org/2002/ws` for Web services.

[7] According to real specifications, only WSDL is really needed to describe a Web service, SOAP and UDDI are de-facto standards.

[8] BPEL4WS (Business Process Execution Language for Web Services) or WSFL (Web Services Flow Language) or WSCL (Web Services Conversation Language) or XLang etc.

[9] See `http://www.daml.org/services/owl-s/` and `http://www.wsmo.org/`

Interaction. DSG systems should be highly interactive as the main difference between PD and DSG is the conversational process.

Non-determinism. DSG systems should have a non-deterministic[10] behaviour. The behaviour of the system could not be predicted (thus expressed by a procedure) as it will depend on the conversation which is itself non-deterministic.

Agents. Autonomy, interactivity and intelligence of agents is needed for DSG. Agents are the current best computing metaphor for humans, and DSG system should integrate humans in the loop.

Learning. From machine-learning to human e-learning: it is a fundamental pre-requisite for evolving systems. DSG systems should evolve thanks to learning of both AA and HA composing the system.

Distributed system (MAS and Grid). As the entire knowledge of the system can not be handled by one agent, DSG systems should be part of a distributed system (MAS) sharing all kind of resources (Grid).

Semantic and Pragmatic. In addition to syntax, DSG systems should handle semantics and pragmatics of data, information and knowledge (as the explicit purpose of Semantic Web [BLHL01] and Semantic Grid [RJS05] communities).

HCI and NLP. The integration of humans in the loop strongly depends on these two domains (i.e. Human Computer Interaction and Natural Language Processing).

Emotions. DSG systems should deal with emotions. That means cause, generate, provoke and recognize emotions at least for HA.

Dynamic concept of programming. DSG systems should be highly dynamic systems. Features as: interpreted languages, dynamic typing, lazy-evaluation, stream processing, continuation passing style, constraint style programming etc. seem to be good concrete tools to implement these systems, assuming they are better integrated in higher level design and implementation methodologies.

Agent communication model. As conversational processes make the main difference between PD and DSG, these systems need strong, open and dynamic communication models to manage these processes.

Language enrichment. DSG systems should provide enrichment of languages they use in order to allow user adaption and user oriented behaviour.

Dynamic specification of program. Conversations should change ”pre-defined” functionalities (if they exist) of a DSG system. Actually, programs should be dynamically specified.

State modelling. Interaction is based on changes of state. Then DSG systems should handle stateful, history aware, resources.

The STROBE model presented hereafter is highly influenced by some of these characteristics, Section 3.7 explains how it is thought and developed as a toolkit for DSG making available some of these required characteristics as STROBE agents properties.

3 The STROBE model

This section constitutes the main contribution of the paper. It presents the ”current state” of the STROBE model[11], an agent representation and communication model which aims to play the role of toolkit for DSG presented above. The section starts by presenting the background information and the fundamental concepts of the model, then we formalise the agent architecture and the different internal modules. Afterwards, we illustrate the potential of the model by experimentations. Then we explain how to see the STROBE model as a toolkit for DSG.

[10] Nondeterministic programs execution and termination is not new in computer science since it has been introduced by Dijkstra in 1975 [Dij75]. Non determinism is encouraged in languages such as ADA or other Constraint Style Programming (CSP) languages. The main interest of non deterministic processes is that they allow to model system-user interactions with non-directed (non-controllable) environments when programs are written.

[11] A model can not be static. As the reader will see, the STROBE model is the result of our ideas for years and is permanently evolving, to fit with new situations emerging in a research perspective.

3.1 Research context and fundamental concepts of the STROBE model

The proposed model is the result of research in three domains: i) the Intelligent Tutoring Systems (ITS) or Artificial Intelligence and Education (AIED), with its historical weight as a major source of inspirations for advances in computing and human learning, ii) DAI and MAS particularly concerned with agent communication, from ACLs (Agent Communication Languages) and their semantics, to conversation modelling, 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 (3^{rd} domain) new elements brought by the communication (2^{nd} domain) in order for agents to learn (1^{st} domain).

3.1.1 Learning

Assuming that the purpose of e-learning is to create the conditions enabling and facilitating to improve human knowledge, the STROBE model is not a model of an ITS able to directly do this job. Rather, **our proposition is to reflect on interactions between agents, considering interactions as the "key of the arch" of the knowledge construction occurring in e-learning scenarios**. When humans use a computer they interact with a system. But could we really speak of "interaction"? An interaction between two entities is a process that implies an action to occur on the interacting entities. That means that interactions have some consequence on these entities, i.e. changes of state. However, each entity may only change state if the change is performed by the entity itself. For HA, these changes can be "learning", the definite purpose of ITS. In e-learning scenarios, it is quite unlikely that human learning occurs on the simple basis of interacting with a static system. **Real interactions modify both entities, including the artificial one**. The learning process should be seen as a co-construction of knowledge (social constructivism). For this process to be cumulative (i.e. unlimited) in knowledge production, the two entities have to learn from each other during this process and re-inject what they learn into the loop. Consider, for example, an *interactive CD-Rom* on 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 ITS. 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. Efforts in user modelling and user adapted interaction are remarkable but to often *ad hoc*, as it is the case for ad hoc constructed persistent transactional systems on the Web. Dynamic, evolving systems apt to improve their capacity to stimulate human learning, should be modelled accordingly. Interactions of these systems with humans – as well as with other systems in the network – should produce improvements of their ability to serve humans to learn.

In a distributed system such as the Web and the Grid, three types of interaction between entities exist, and all of them may imply learning: i) HA-HA, via special devices such as for example enhanced presence tools as presented in [EKD03]. ii) HA-AA – the classical human-machine interfaces – when a human user requests a system to do something (perform a task, answer a question, solve a problem). iii) AA-AA, when autonomous and intelligent AA collaborate with each others (e.g. by delegating a sub-task to another). In this paper our focus is on AA-AA as well as HA-AA interactions but the model is "generic" with respect to any agent to agent interaction. Indeed, if we consider than the model is adapted for modelling AA but also "a part" of HA (cf. Section 3.3.1) then STROBE communications represent interactions of any type.

Therefore, the interaction centred aspect of the STROBE model is inspired by the e-learning/ITS community assuming that modelling and making possible better interactions between agents implies AA-learning which helps to improve HA-learning (e-learning) and vice versa. Learning, for any kind of agents, depends on interactions.

3.1.2 Agent communication

Communication in MAS. Simply grouping together several agents is not enough to form a MAS. It is communication[12] between these agents that makes it. Communication allows cooperation and coordination between agents [Fer99, Woo02]. Communication in MAS is classified in two modes, direct and indirect:

[12]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

- In the indirect mode, agents communicate via the environment (the world where they live and evolve). The indirect mode of communication is widely used in agent simulations. 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, historically introduced by Hewitt [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.

The STROBE model deals with asynchronous direct mode communication. That means, a communication throws directly messages (that can be buffered) between agents. The STROBE communication language is presented in Section 3.3.5.

Modelling communication has always been a problem. The traditional communication model is the Shannon statistical communication theory considering 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. However, it is limited because it is basically a statistical model of information transmission. In real communicative situations it is impossible to consider communication as simple information transmission, all the dialogue should be taken into account.

To model dialogues, (i.e. collaboration among human and artificial agents to solve collaboratively real world problems) we have to consider important semantic and pragmatic aspects as the communication effects on the interlocutors, as well as the history of the dialogue. In fact, let us assume that the first step in modelling dialogues is to define and use agent communication languages, while the second step is to model sets of communications, what is usually called conversation[13].

Agent Communication Languages. In dialoguing, 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 realised by speech acts expressing an illocutionary force. Speech act theory was reused many times in agent communication. Speech acts, that can be viewed as the linguistic unit of communication, were firstly classed by performatives by Searle and Vanderveken. 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 need of architectures[14] based on mental states representation, such as BDI (Believe Desire Intention) [RG91] as well as of communication languages such as KQML (Knowledge Query and Manipulation Language) [LF97] and FIPA-ACL (Foundation for Intelligent Physical Agents - Agent Communication Language)[15] [Fip02]. The former language was often criticised about the semantics of the messages [EBHM00]. The latter, its descendant, proposes therefore a strong semantics with pre and post conditions defined as agent mental states. These 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] precisely known in advance, is a great advantage when creating heterogeneous agent systems that are designed to be easily extensible and broadly used. 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: 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 protocol, 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, and the conversation. ACLs are interesting and make consensus in the agent communication community, but they have also weaknesses. For example, approaches based on mental states modelling presuppose the notion of sincerity and that an agent can know what another one thinks. This is a big limit to one of the primitive characteristic of agents that is autonomy. Singh [Sin98] demonstrates the insufficiency of classic approaches centred on a mental state architecture rather than on a social architecture. He proposes a social architecture lying on an organisation founded on group, role and commitment for each agent playing a role.

internal structure (e.g. changing the way of 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.

[13] A conversation is seen as a set of communications, each consisting of at least two messages exchanged.

[14] We can also use the term of policy or theory of agency.

[15] This standard is heavily influenced by Arcol developed by France Télécom in the early 90s.

About conversation modelling. Conversation models can be classified according to two approaches. The first approach is the most specified and widespread. It comes from distributed systems and it is based on interaction protocols [DG00, Hug03]. These protocols represent the interaction structure and specify interactions that should be respected during conversations. They elicit message exchanges between agents but also define the internal states agents should follow. An interaction protocol has four properties [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: interaction protocols consist of transitions between states. The notion of interaction protocol is greatly bound to an architecture based on mental states, such as envisioned in KQML and FIPA-ACL. A famous example of interaction protocol is the Contract Net protocol proposed by Smith [DS83] but also Berthet's Introduction protocol [BDB92] and Sian's Cooperative Learning protocol [Sia91]. Among the first languages using interaction protocols we notice COOL [BF95] and IL [Dem95]. Interaction protocols are either described by the language itself, such as in COOL, or are often represented with Finite State Machine or with Petri Nets or Coloured Petri Nets (CPN)[16] [CCF+00]. 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]). Nowadays, new approaches inspired by interaction protocols appear. For instance, conversation policies [GHB00] 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 [ESH00] proposes the concept of task model which takes account of the global coherence in conversation. Huget [HK03] proposes an interaction protocol engineering: the generation of protocols after the interactions analysis. The interaction protocol approach seems to be the main one in the MAS community, such that agents have to be acquainted with if they want to be the mostly autonomous and capable to interact with any other agent. However this approach has some weaknesses, especially interoperability, composition and verification of protocols which depend on a common semantics. KQML and FIPA-ACL lie on speech act theory and suppose that the meaning of a message is predetermined. Moreover, they are intimately bound to interaction protocols which control the exchange structures. Therefore, **there is no dynamic interpretation of messages, everything is specified in order to have only one way to interpret them**. 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 go toward that.

The second conversation modelling approach is linked 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 on hidden intentions. Some researchers call it *dialogism*[17]. Dialogism is widely inspired by language philosophy, cognitive science and social psychology research [MDCd99]. In dialogism the intention of a message is not predetermined 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] or [RPD99]. Dialogism is strongly bound to a social architecture such as the one proposed in [Sin98] and consequently to utterances [GGT94] and representation of the other (or partner 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? Dialogism gives some solutions to conversation modelling, for example, the sincerity prerequisite is no longer needed with this approach since agents develop a strong partner model. But the main weakness of this approach is that there is no concrete way to realise it.

The STROBE model is highly inspired from both these two approaches, Section 3.4 explains how the model fits with the emerging standards in MAS community i.e. FIPA-ACL and protocols. However, one important aim of the STROBE model is to enable to conceive and implement dialogism.

[16] CPN are very interesting to model multiple parallel conversations.

[17] Or dialogic approach, or simply dialogue but it is more ambiguous.

Agent communication requirements. From the agent communication literature we extracted 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.
- Agents should interact with other agents (AA or HA) following the same principles, the agent communication model should be generic with respect to the type of the communicating agents. What is important is the agent representation of each of its interlocutors i.e. the partner model.

3.1.3 Learning in MAS and levels of learning

Learning-by-being-told paradigm. Learning and knowledge communication are very important in MAS, they provide evolution and adaptation of these agent societies through time by automatically inferring knowledge from previous knowledge and/or from communication. Learning was firstly limited to AI, but is nowadays extended to DAI and MAS [BGS+91]. This paragraph details a simple overview of learning as it could be seen in the STROBE model. Considering agent learning, i.e. reflecting about what an agent learns by itself and by/about its interlocutors, three types of learning are distinguishable:

- The first one could be called *machine learning*. It is the classical approach as old as AI itself. Agents learn from a KB abstracting and generalizing, applying some abduction, deduction, induction/inference, rules on the KB. The most widespread approach in machine learning is reinforcement learning reported in [KLM96].
- The second one could be called *learning-by-being-told*. It comes from DAI and is issued from the instructional effect where an agent changes or creates representations according to received messages.
- The third one could be called *learning as a side effect of communication*. It is quite "serendipitous"[18] as it occurs when an agent learns without being aware of it, when it learns because of a beneficed communication context such as enhanced presence (of other AA or HA) or any kind of a posteriori useful interaction with any agent not specifically called a priori for the purpose of mutual teaching or learning.

The model describes extensively just an instance of the second type of learning, but it fits the all three types (cf. Section 3.3.4). STROBE agents learn-by-being-told by other agents, and change their representations according to their interlocutors' information given by messages.

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 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 i.e. shared, language appears. We rather weaken this constraint, assuming just a kernel common communication language to be adopted and propose to enrich the 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, facts, rules and languages are such as *Data*, *Control* and *Interpreter* levels in computing. These three abstraction levels (or learning levels) may be found in all programming languages[19]. Every language provides an abstraction tool[20] which allows to enrich the language. For example, abstraction at the Data level consists in assigning values to already existing variables, or defining new data from already existing data (e.g. in Scheme it is done with expressions such as `(define weekend 3)` or `(set! weekend '(friday saturday sunday))` or `(define (make-point x y) (cons x y))`). Abstraction at the Control level consists in defining new functions/procedures abstracting on the existing ones (e.g. in Scheme it is done with expressions such as `(define foo`

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

[19] These three levels are clearly explained in Abelson & Sussman's famous book *Structure and Interpretation of Computer Programs* [ASS96] Building Abstractions with Procedures (chapter 1), Building Abstractions with Data (chapter 2), Metalinguistic Abstraction (chapter 4).

[20] For example, the `define` special form in Scheme.

`(lambda ...))` using the `lambda` special form). 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 level 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, 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.

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. The challenge is to allow Interpreter level modifications at run time, in order to generate dynamic processes. The STROBE model maps these three levels schema into agents, considering them as a set of pairs: environment + interpreter. Data and Control levels learning are made by changing the environment and meta-level learning by changing the interpreter. While enabling its interpreter(s) to evolve, an agent learns more than simple information, it completely changes its way of perceiving and processing this information. This is the difference between learning a datum and learning how to process a class of data.

3.1.4 The influence of applicative/functional programming

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 based on lambda calculus, hence it offers a clear semantics. Further, it is easy to learn and use [ASS96]. Scheme provides especially 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[21]. 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 realised with other languages such as Java but it is very heavy due to the fact that it is not intrinsic to the language[22].

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). Streams, objects and environments are three Scheme first class primitives. Actually, Scheme is used in STROBE as specification language (such as lambda calculus or denotational semantics) as well as an implementation language (cf. Appendix A). However, the model is implementation independent.

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 (nondeterministic evaluation)

As in all computer languages an expression is evaluated in an *evaluation context*. In Scheme this context consists of the expression to evaluate itself (`e`), the environnement in which this expression should be evaluated (`r`), sometimes

[21] First-class objects may be named by variables, may be passed as argument to procedures, may be returned as results and may be included in (and retrieved from) data structures.

[22] 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).

the continuation(s), that means the next expression(s) to evaluate with the result of this evaluation (`k`) and of course the interpreter itself (`eval`).

Moreover, we also decided to use Scheme 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 feature to access to an expression evaluation 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 put into work and very heavy in computing resources.
2. Using first class interpreters such those present in [IJF92].
3. Using two levels of evaluation in order to use the evaluation function (`eval`) which is part of the language and reachable as any other procedure.

We experiment solution 1 and 3 in the STROBE model. Notice, that doing this kind of Interpreter's level modification could be very difficult with other types of languages, especially compiled ones (e.g. Java, C++) which have an execution machine not consisting of a simple interpreter, but of a compiler and a virtual machine.

3.1.5 McCarthy inspirations and Wegner's view compliance

This paragraph has the purpose to compare the STROBE model with related famous visionary work aiming to shift the classical algorithm view of computer science to an interaction centred approach. Some concepts of the model may be recognised in McCarthy predictions described in [McC89]:

- *A language should be interactions 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 both 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.*

3.2 The STROBE model's first ideas

The STROBE model was firstly described in [Cer96, Cer99]. 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*. These three primitives are supposed to be first class objects giving to the model an important dynamic behaviour. The STROBE model is highly influenced by applicative/functional programming features and traditional ACLs but goes beyond them as a consequence of lying on the three main concepts adopted:

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[23]. This principle is important because it regards agents as autonomous entities whose behaviour and interactions are controlled by a concrete evaluation 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 emphasises the agent and interprets the message contents, and `evaluate-kqmlmsg`, the sub-procedure able to interpret messages.

Cognitive Environments. The concept of *Cognitive Environments* was firstly presented in [Cer96]. It explains that multiple environments are simultaneously available within the same agent. These environments represent the agent knowledge – for example, the value of a variable, the definition of a procedure – they embody the agent Knowledge Base (KB) as they represent a part of the different languages known by an agent (Data and Control levels, cf. Section 3.1.3). Actually, for the representation of each interlocutor, STROBE proposes to have a partner model to be able to reconstruct as much as possible, the interlocutor's internal state. The Cognitive Environment plays this role. A STROBE agent interprets each messages' communication in a specific environment dedicated to the interlocutor. They have two types of environments:

- the first one is unique – global and private – it belongs to the agent, representing its own beliefs. It is called the *Global Environment* (after denoted E_X^X);
- the other ones – locals – are dedicated to the interlocutors, each of them represents a partner model as it evolves during conversations. They are called *Local Environments* (after denoted E_Y^X as the X's dedicated environment to Y). Each time an agent receives a message, its selects the corresponding environment dedicated to the message sender to interpret the message.

Remark — The first idea of STROBE was to dedicate to the interpretation of interlocutors' messages a part of the execution context (i.e. Cognitive Environment). We will see how STROBE now dedicates beside another part (i.e. Cognitive Interpreter) and how perspectives (cf. Section 3.6.1) aim to dedicate the rest of the execution context (i.e. Cognitive Continuation).

Messages as streams. STROBE agents' inputs and outputs flow of messages are represented by streams[24]. As [ASS96] explains, streams allows 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 to model agent conversation because they allow defining recursive data structures that can be used to represent sequences that are undetermined and infinitely long such as inputs and outputs. This feature of applicative/functional programming seems particularly adapted for modelling agent conversation 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 first message. It allows to remove the necessity of global conversation planning (i.e. the classic interaction protocol) substituting it by history memorisation and one-step, opportunistic planning. Section 3.6.2 details work in progress on this special point.

3.3 The STROBE model today

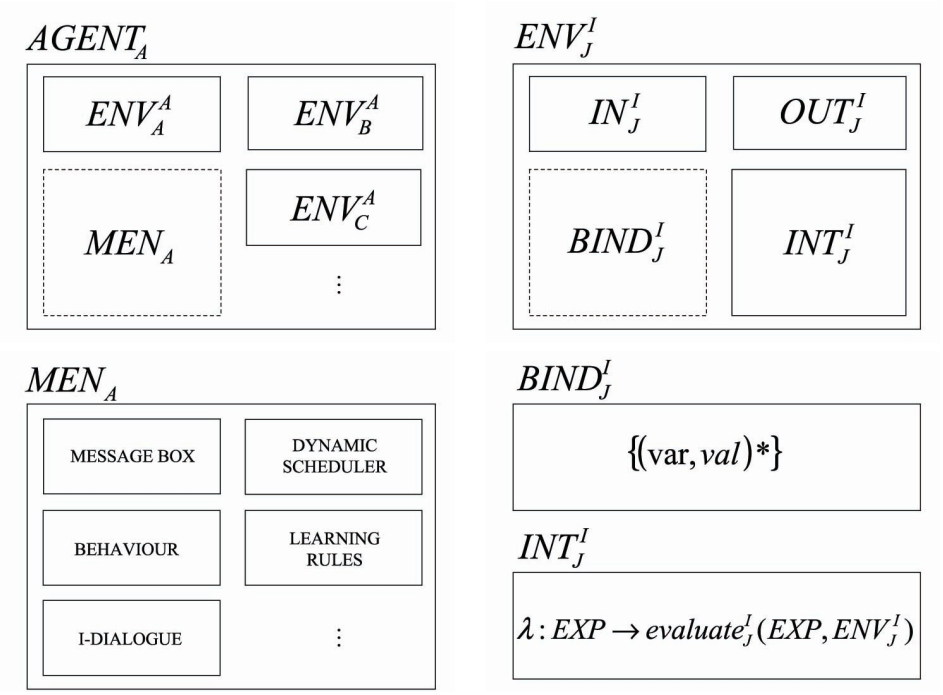
The current extension of the STROBE model described in this paper can be synthetically outlined by one sentence: to augment the concept of Cognitive Environments, by including in them Cognitive Interpreters, which can be dynamically modified, allowing agents to develop a complete language (Interpreter + Environment) for each of their interlocutors, increasing agent dynamics and giving to them an evolutionary partner model.

[23]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 all their time to accomplish what they are conceived for, so the step L has got a sense.

[24]Streams are "lazy lists" (i.e. the natural representation of sequences that are partially instantiated (evaluated)). The basic idea consists in constructing a stream only partially, and to pass the partial construction to the program that consumes the stream [ASS96]. Most of the times streams lie on lazy evaluation mechanism (also called delayed evaluation). Lazy evaluation consists in delaying evaluation of function arguments until the last possible moment (e.g. until they are required by a primitive, or printed as an answer)[FW76].

3.3.1 The STROBE agent architecture

Cognitive Environments provide to agents a global environment and several local environments representing the partner models. We significantly improve the representation of agents by considering them as a set of interpreters, including a global one and local ones for each agent they have a representation of. The previous concept of Cognitive Environments is actually extended with Cognitive Interpreters. These interpreters (i.e. `evaluate` procedures) are stored in environments as any other usual procedure. The use of the terms "cognitive" has to be explained: i) for Cognitive Environments, because, according to the three learning levels seen above, the Data and Control levels can be modified by communications; and ii) for Cognitive Interpreters, because the last level, the Interpreter level, can be also modified using devices presented in Section 3.1.4. The three levels are reachable at run time while communicating, allowing agents to dynamically modify their meta-level knowledge, increasing their autonomy and adaptability, allowing them "to change their mind" i.e. their way of interpreting things. **STROBE agents therefore develop, while communicating, a language (Cognitive Environment + Cognitive Interpreter) for each of their interlocutors.** The STROBE agent architecture is given in Figure 2.



$$\text{With, } ITEM_Y^X = \begin{cases} \text{X local item dedicated to Y} & \text{if} \quad X \neq Y \\ \text{X global item} & \text{if} \quad X = Y \end{cases}$$

Figure 2: The STROBE agent architecture

An agent A, $AGENT_A$, is considered as a set of Cognitive Environments, $ENV_{...}^A$ and mental states, MEN_A (i.e. agent's private modules). The set of environments is not null, because an agent unavoidably has its own global environment, ENV_A^A, and eventually a other environments corresponding to its partner models ENV_B^A, ENV_C^A, etc. A Cognitive Environment, ENV_J^I, is a set of bindings with a least, an interpreter, INT_J^I, which is the Cognitive Interpreter included in the Cognitive Environment, and the rest of the bindings, $BIND_J^I$, which is a set of pairs variable – value. Two other bindings are also distinguished IN_J^I and OUT_J^I as I's input and output streams of messages dedicated to J. A Cognitive Interpreter, INT_J^I, is a procedure $evaluate_J^I(EXP, ENV_J^I)$, which evaluates an expression EXP, in the corresponding environment, ENV_J^I. EXP is an expression of a language, i.e. a message or the content of a message. The mental state, MEN_A, is a set of modules depending on the role of the agent. It could be composed for example of a *message box*, which is used to store messages before interpreting them, a *dynamic scheduler*, which

corresponds to the policy of choice of messages in the message box[25], a *behaviour*, corresponding to the role of the agent i.e. its intrinsic purpose, *learning rules* to change environments in a global way (cf. Section 3.3.4), an *i-dialogue* function, to model conversation by streams (cf. Section 3.6.2) etc.

We can now illustrate the STROBE model REPL loop more precisely as Figure 3 shows. Each time an agent *Read*s a message, it picks up both the dedicated environment and interpreter to interpret it, then it *Eval*s it (i.e. applies the selected interpreter on the message in the selected environment, and *Print*s the corresponding answer (i.e. sends the answer message(s) and/or processes the evaluation result), before *Listen*ing the next message.

Agent X REPL loop

- *LISTEN*
  - Accomplish its own objectives while waiting for new messages; and/or choose (with dynamic scheduler) the next message M to process;
- *READ*
  - Pick up the Cognitive Environment (E_Y^X) and Cognitive Interpreter (I_Y^X) dedicated to the interlocutor Y (if Y is the sender of M);
- *EVAL*
  - Interpret the message M : $evaluate_Y^X(M,E_Y^X)$;
- *PRINT*
  - Send to Y the answer message(s) and/or process the evaluation result;

Figure 3: Agent REPL loop

We can now detail the procedure `evaluate`. If the expression to interpret is a message then it calls `evaluate -kqmlmsg` (cf. Section 3.3.5), 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.

Data: EXP, ENV_J^I
Result: value or message(s)
if EXP *is not a message* **then**
  $evaluate - exp_J^I(EXP,ENV_J^I)$;
else
  $evaluate - kqmlmsg_J^I(EXP,ENV_J^I)$;
end

Procedure: $evaluate_J^I(EXP,ENV_J^I)$

The advantage of seeing STROBE agents as a set of interpreters is that they can process programs for other agents or even exchange their interpreters as simple procedures just like in Grid architectures where it is sometime more interesting to move programs than data. Their interpreters can also be transmitted before a conversation, just like an ontology. As [ASS96] states:

> 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.

[25] This idea proposed originally by [GB99] and [Cer99] allows an agent to choose the next message to process without interpreting it, only using message metadata (i.e. communication level and message level). For example, your dynamic scheduler may help you to answer your boss before your colleague.

With the STROBE model, agents have a set of environments and interpreters that represents their knowledge and its evolution through time. Indeed, these pairs correspond to their recognised sub-languages and thus to their faculties to carry out a task.

3.3.2 Partner model representation

In the STROBE model, the Cognitive Environment and its included Cognitive Interpreter, emphasise the partner model. Therefore, message interpretation is done in a given environment and with a given interpreter both dedicated to the conversation (we deal with conversations because Cognitive Environment modifications[26] are persistent along communications). For instance, Figure 4 points out representations of an agent, $AGENT_A$ which knows two other agents, $AGENT_B$ and $AGENT_C$.

$$AGENT_A = \left\{ \begin{array}{l} ENV_A^A = \left\{ \begin{array}{l} INT_A^A = \lambda : EXP \rightarrow evaluate_A^A(EXP, ENV_A^A) \\ BIND_A^A = \{(u,3)(v,5)(square, \lambda : x \rightarrow x * x) \ldots\} \end{array} \right. \\ ENV_B^A = \left\{ \begin{array}{l} INT_B^A = \lambda : EXP \rightarrow evaluate_B^A(EXP, ENV_B^A) \\ BIND_B^A = \{(u,3)(v,5)(carre, \lambda : x \rightarrow x * x) \ldots\} \end{array} \right. \\ ENV_C^A = \left\{ \begin{array}{l} INT_C^A = \lambda : EXP \rightarrow evaluate_C^A(EXP, ENV_C^A) \\ BIND_C^A = \{(u,6)(v,2)(id, \lambda : x \rightarrow x) \ldots\} \end{array} \right. \end{array} \right.$$

Figure 4: Representations of $AGENT_A$, with two partners, $AGENT_B$ and $AGENT_C$, models

3.3.3 Determination of the local environment dedicated to a new partner

When an agent meets another one for the first time, three cases are possible:

1. it produces a new local environment for this agent by cloning its own global environment and uses this new one to communicate with this agent, for example, when two agents meet together for the very first time, and when they do not have any idea of the type of the new partner agent (e.g. HA, AA, service provider, etc...). This case is the most logic and the most frequent;

2. it produces a new local environment for this agent by cloning another local environment it already has, for example, when another agent finishes a conversation (e.g. generates a service) that a first one has begun i.e. keeping the same state;

3. it uses an already existing local environment to communicate with this new agent (and the previous ones), for example, when an agent identical, or playing the same role, or belonging to the same group, replaces another one.

Figure 5 illustrates these three cases representing $AGENT_A$ when it meets $AGENT_D$ for the first time.

The third situation is heavy in consequences, because it shows that several agents can be represented by another one by means of the same partner model. Therefore, an agent may have, a partner model of lonely agents, but also representations of groups of agents eventually playing different roles. **It brings closer the model to a organisational structure** such as AGR (Agent, Group, Role) [FGM03] as it is proposed by Singh in [Sin98]. It is one important point of a social agent view.

[26] We generally use the term of Cognitive Environment, or simply environment, but the reader should not forget that it concerns also Cognitive Interpreters due to the fact that the latter is included in the former.

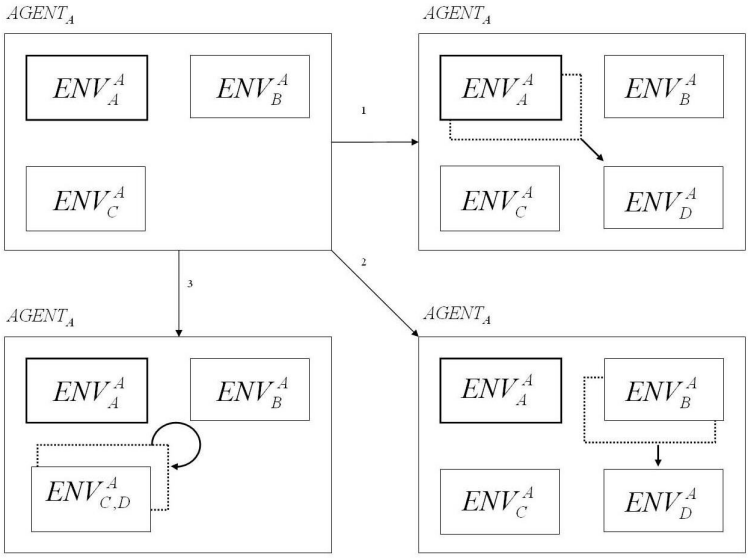


Figure 5: New interlocutor environment instantiation

3.3.4 Learning/Reasoning on Cognitive Environments

It is important to notice that the global environment, (i.e. ENV_X^X), is private and changes only for precise reasons. However, caring readers could have noticed in Figure 4, that $BIND_C^A$ was different from $BIND_B^A$ and particularly from $BIND_A^A$. Which value $AGENT_A$ must use for *u* or *v*? How to infer that the `square` procedure of $BIND_A^A$ and `carre` of $BIND_B^A$ are the same? When does $BIND_A^A$ change?

Even if the STROBE model is an instance of the learning-by-being-told paradigm, i.e. Cognitive Environment changes (i.e. learning) is done according to messages "told" by another agent, it does not prevent at all the two other types of learning. For the first one, *machine learning*, notice that the set of environments, $ENV_{...}^X$, corresponds to X's KB. On this KB STROBE agents may have two kinds of learning processes which should be considered locally (i.e. on one environment) and globally (i.e. on all the environments):

- The first kind consist on learning on a particular environment, $ENV_I^X \; \forall I$, 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 an environment are the same. This kind of machine learning processes can be applied on an data structure which binds variables with values, such as the one used for Cognitive Environments. For example, in Scheme, programs can detect if two functions produce the same result, or if they have the same body (i.e. detect that `(square (lambda (x) (* x x)))` and `(carre (lambda (y) (* y y)))` are the same functions assuming that `*` is the same function in the two definition environments).

- The second learning process concerns the modification of the global environment, ENV_X^X. Agents must transfer knowledge they learned in the different $ENV_I^X \; \forall I \neq X$ (local environments) to ENV_X^X. 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 use with another agent knowledge learned with another one. Then an agent must have a global environment *evolution policy* (or a *learning rule* module), which emphasises 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 environment) x will be 5. Or, 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 will be 3. The second example shows that an agent global environment policy may use metadata on agent interlocutors. Notice that these kinds of processes can be similar to the ones seen just above but globally. For example, detecting that two functions in two different local environments are the same.

Concerning the last type of learning of Section 3.1.3, *learning as a side effect of communication*, we do not detail

this point here as it is the subject of other ongoing research (for example, [JEC05] proposes a bootstrap to go toward "serendipitous" learning with enhanced presence by HA and AA communication).

Remark — The STROBE model, as number of agent model, is highly inspired by human representation. In one sense we could say that each HA has is own internal representations and states, what (s)he thinks, what (s)he believes, what (s)he wants to do or become etc. Each HA has a set of acquaintances and different representations of these acquaintances, actually (s)he develops different languages for each of them. Other HAs do not know what are their real thoughts, they can just deduce facts from messages these HAs send one another. Each time a HA interacts (and for HA not just communicate) with another one (s)he changes her/his representations and mental states. HAs are able to infer, or deduce knowledge from what they know or learn. The purpose of STROBE is to model AA as close as possible from HA, since it is assumed that HAs perform better as AA the social construction of solutions to problems.

3.3.5 Communication language

This section details the communication language used by STROBE agents (i.e. message structure and performatives used). It is considered minimal. Messages are speech act oriented and use performatives. They are inspired by the KQML and FIPA-ACL message structure but use only the main parameters (sender, receiver, performative, content). Other ones (e.g. ontology, in-reply-to, reply-with, protocol) may eventually be added on the fly as the `evaluate-kqmlmsg` procedure is part of the interpreter. Their form is:

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

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

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[27]. In order to construct a minimal agent STROBE agents process six performatives by default ([Cer99]). Another one, broadcast, is the subject of one of the experimentations hereafter:

assertion messages modify the interlocutor behaviour or some of its representations (bindings in the Cognitive Environment). 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 learning.

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. 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 or for instance, to dynamically modify an agent interpreter.

For example, Figure 6 illustrates a dialogue with the three kind of performatives.

As we see in the experimentation, Section 3.5.1, the list of known performative is not closed. Surely, due to the fact that performatives are processed by `evaluate-kqmlmsg` procedure, which is a sub-procedure of `evaluate`, it is possible to dynamically modify it and thus add to it new performatives. For example, the broadcast performative:

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.

[27] 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 (quasi-)all the language.

Teacher (A_T)	Student (A_S)
$\{A_T, A_S, request,$ `square` $\}$	$\{A_S, A_T, answer,$ `undefined` $\}$
$\{A_T, A_S, assertion,$ `(define (square x) (* x x))` $\}$	$\{A_S, A_T, ack,$ `(*.*)` $\}$
$\{A_T, A_S, order,$ `(square 3)` $\}$	$\{A_S, A_T, executed,$ `9` $\}$

Figure 6: Examples of message between an agent teacher and an agent student.

We now can detail the `evaluate-kqmlmsg` procedure:

Data: MSG=$(A_I, A_J, PERFORM, CONTENT), ENV_J^I$
Result: value or message(s)
switch *PERFORM* **do**
  case *assertion*
    send($A_J, A_I, ack, evaluate_J^I(CONTENT, ENV_J^I)$);
  case *request*
    send($A_J, A_I, answer, evaluate_J^I(CONTENT, ENV_J^I)$);
  case *order*
    send($A_J, A_I, executed, evaluate_J^I(CONTENT, ENV_J^I)$);
  otherwise
    $analyse - answer_J^I(CONTENT)$;
end

Procedure: $evaluate - kqmlmsg_J^I(EXP, ENV_J^I)$

Note that for each performative the interpretation of the content is performed. However, different performative may be used to distinct "pragmatic effect" of messages. That is, being able to know 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).

3.4 Related works on agent communication

This section shows how the STROBE model does not prevent the use of classical approaches in agent communication (i.e. ACLs and protocols). Let us take the problem in the contrary sense as in Section 3.1.2. Let us begin with conversation modelling. STROBE messages interpretation is done with a dedicated procedure, `evaluate-kqmlmsg`. This procedure personifies the way an agent interprets messages, thus an agent may be constructed with a `evaluate-kqmlmsg` which takes into account some interaction protocols or conversation policies. Thus, the messages structure becomes:

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

with $PROTOCOL = \{contract - net, introduction, coop - learning, \ldots\}$

and the `evaluate-kqmlmsg` procedure becomes:

Data: MSG=$(A_I, A_J, PERFORM, PROTOCOL, CONTENT), ENV_J^I$
Result: value or messages(s)
switch *PROTOCOL* **do**
  case *contract-net*
    switch *PERFORM* **do**
      case *call*
        ...
      case *proposal*
        ...
      case *decision*
        ...
      case *contract*
        ...
    end
  case *introduction*
    ...
  case *coop-learning*
    ...
  otherwise
    switch *PERFORM* **do**
      case *assertion*
        send($A_J, A_I, ack, evaluate_J^I(CONTENT, ENV_J^I)$);
      case ...
        ...
    end
end

Procedure: $evaluate - kqmlmsg_J^I$ (*MSG*,ENV_J^I)

Then STROBE can embody classical approaches of conversation modelling (i.e. interaction protocols and conversation policies). 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. So STROBE should also embody these kind of architectures. Subsequently, data structures as environments allows 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 (Believe, 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 language such as PROLOG or Scheme as presented in Chapter 4 of [ASS96]. This chapter shows how to write a query system (i.e. a logic expression interpreter). STROBE agent's interpreters can become such query systems.

Mental states based ACLs are often criticised [Sin98] for some of their requirements: i) the sincerity condition, ii) the fact that they presuppose what an agent think (i.e. believe, intention, desire). For example, its strong semantic gives to FIPA-ACL these drawbacks. These requirements are a very big limit to agent autonomy. Using a strong partner model and not simply a mental states architecture avoids these drawbacks. Actually, 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 the knowledge has been communicated (has become shared).

As an example of another mapping of STROBE with other work, we can say that the Cognitive Environments are quite the same thing as Huget's *interaction module* [Hug03] which extracts protocols and semantics from agents and puts them into a framework. It can be implemented outside agents increasing their modularity and interoperability.

3.5 Experimentations: examples of scenarios

In order to express the potentiality of the model, this section suggests two examples of scenario, called experimentations, which could occur with such a model. They illustrate: i) meta-level learning by communicating: agents which modify their interpreters while communicating; ii) how STROBE enables the dynamic specification of a problem by means of communication, which is necessary for DSG as presented in Section 2. Even if these experimentations are "toy examples", they are significant as they represent solutions for a large class of problems.

3.5.1 Meta-level learning by communicating

Learning 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-kqmlmsg` which is part of the interpreter). It is a standard "teacher-student" dialogue. An agent teacher (A_T) asks to an agent student (A_S) to broadcast a message to all its interlocutors using a special performative, `broadcast`. However, student does not initially know the performative used by teacher. So, teacher transmits two messages (`assertion` to define it the way of interpreting broadcast messages and `order` to invite it to change its interpreter) clarifying to the student the way of processing this new performative. Finally, teacher formulates once more its request to student and obtains satisfaction. After the last message process, $evaluate - kqmlmsg_T^S$ procedure is modified. Thus a part of its interpreter (INT_T^S) was dynamically changed and the corresponding definition in its dedicated environment (ENV_T^S) is changed. It is thus a meta-level learning. At the end of the dialogue student agent can process `broadcast` messages coming from teacher. Figure 7 describes the exact dialogue occurring in the experimentation. The paper [JC03] details this experimentation and shows it using the reifying procedures.

Teacher (A_T)	Student (A_S)
Here is the definition of the square procedure: $\{A_T, A_S, assertion,$ `(define (square x) (* x x))`$\}$	*Ok, I know now this procedure:* $\{A_S, A_T, ack,$ `(*.*)`$\}$
Broadcast to all your current interlocutors: $\{A_T, A_S, broadcast,$ `'(order (square 3))`$\}$	*Sorry, I don't know this performative:* $\{A_S, A_T, answer,$ `'(unknown broadcast)`$\}$

Save your `evaluate-kqmlmsg` *procedure:*
$\{A_T, A_S, assertion,$
`(define old-eval-kqmlmsg evaluate-kqmlmsg)`$\}$
 	
Here is the way to process broadcast:
$\{A_T, A_S, assertion,$
`(define (evaluate-broadcast msg) ...)`$\}$
 	
Then, you can change your `evaluate-kqmlmsg` *procedure:*
$\{A_T, A_S, order,$ `(set! evaluate-kqmlmsg`
`(lambda (msg)`
`(if (eq? (performative? msg) 'broacast)`
`(evaluate-broadcast msg)`
`(old-eval-kqmlmsg msg))))`$\}$
	Ok, I have added this new binding:
	$\{A_S, A_T, ack,$ `(*.*)`$\}$
	
	Ok, I have added this new binding:
	$\{A_S, A_T, ack,$ `(*.*)`$\}$
	
	Ok, I have changed my procedure:
	$\{A_S, A_T, executed,$ `(*.*)`$\}$

Broadcast to all your current correspondents: $\{A_T, A_S, broadcast,$ `'(order (square 3))`$\}$	*Ok, I broadcast* $\{A_S, \ldots, order,$ `(square 3)`$\}$

Figure 7: Teacher-student dialogue for `broadcast` performative learning.

Remark — Note that the new `evaluate-kqmlmsg` procedure[28] integrate the new code for evaluating broadcast messages, but the previous code is not modified as it is saved and called by `old-eval-kqmlmsg`. The term integration is important, because A_S 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.

[28] Environment (`r`) and continuation (`k`) are note mentioned in Figures 7 and 8 for simplicity.

Learning 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 student agent. The teacher wants to test the way the student process the AND logic operator. Logic operators AND and OR have to be processed particularly. 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 "lazy 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 classical procedures[29]. 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 the student way of processing AND, teacher uses a global variable, `counter`, which is incremented each time the interpreter evaluates expression like `(begin (set! counter (+ 1 counter)) predicate)` which returns the value (true or false) of `predicate`. Figure 8 describes the exact dialogue occurring in the experimentation.

Teacher (A_T)	Student (A_S)
Here is the definition of the counter variable: $\{A_T, A_S, assertion,$ `(define counter 0)`$\}$	*Ok, I know now this variable:* $\{A_S, A_T, ack,$ `(*.*)`$\}$

Please, evaluate the above expression:
$\{A_T, A_S, order,$
`(and (begin (set! counter (+ 1 counter))`
`(> 2 3))`
`(begin (set! counter (+ 1 counter))`
`(= 0 0)))`$\}$
	Ok, I have executed, here is the result:
	$\{A_S, A_T, executed,$ `#f`$\}$

What is the value of counter?: $\{A_T, A_S, request,$ `counter`$\}$	*Ok, here is the value of counter:* $\{A_S, A_T, answer,$ `2`$\}$

Save your `evaluate` *procedure:*
$\{A_T, A_S, assertion,$ `(define old-eval evaluate-exp)`$\}$
 	
Here is the way to process AND in a lazy way:
$\{A_T, A_S, assertion,$ `(define (evaluate-and exp) ...)`$\}$
 	
Then, you can change your `evaluate` *procedure:*
$\{A_T, A_S, order,$ `(set! evaluate-exp`
`(lambda (exp)`
`(if (and-exp? exp) (evaluate-and exp)`
`(old-eval exp))))`$\}$
	Ok, I have added this new binding:
	$\{A_S, A_T, ack,$ `(*.*)`$\}$
	
	Ok, I have added this new binding:
	$\{A_S, A_T, ack,$ `(*.*)`$\}$
	
	Ok, I have changed my `evaluate` *procedure:*
	$\{A_S, A_T, executed,$ `(*.*)`$\}$

Please, evaluate the above expression:
$\{A_T, A_S, order,$
`(and (begin (set! counter (+ 1 counter))`
`(> 2 3))`
`(begin (set! counter (+ 1 counter))`
`(= 0 0)))`$\}$
	Ok, I have executed, here is the result:
	$\{A_S, A_T, executed,$ `#f`$\}$

What is the value of counter?: $\{A_T, A_S, request,$ `counter`$\}$	*Ok, Here is the value of counter:* $\{A_S, A_T, answer,$ `3`$\}$

Figure 8: Teacher-student dialogue for lazy AND operator learning.

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.

[29] You can also do it using continuation passing style programming.

3.5.2 Enabling dynamic specification by communication

STROBE agent's interpreters could of course be nondeterministic ones. The key idea is that expressions, in nondeterministic language, can have more than one possible value. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Nondeterministic evaluation is very useful for CSP, but moreover AI in general is intrinsically nondeterministic. We invite the reader to refer to [ASS96] (Chapter 4) for more details on nondeterministic evaluation with Scheme or elsewhere in CSP literature.

It is interesting for agents to be considered as nondeterministic interpreters to solve, for instance 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 (to achieve a task) for another agent. The constraints, defining a nondeterministic program[30], can be determined progressively with the conversation using STROBE's features to dynamically change procedures and the way of interpreting them.

Remark — We should be careful with the meaning of "nondeterministic". It means that the process achieves, finds an answer (a solution or a statement on the fact that there is no solution), but the way it is done is nondeterministic, can be each time different and not foreseeable.

An e-commerce scenario. Let us consider, a standard e-commerce dialogue for a train ticket booking. A ticket is characterised by a departure city, a destination city, a price, and a date. An SNCF[31] agent dialogues with a customer agent which tries to book a ticket. A realistic dialogue 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, 150€*
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 result of application of ticket research 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. Figure 9 illustrates how this dialogue can be represented into Scheme expressions in order to be realised by our agents.[32]

The idea of this experimentation is interesting because it is the dialogue that builds the computation 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) that specifies the problem before coding it, is changed into a dynamic specification approach during coding. That is to say, specification and realisation are carried out at the same time.

Constraints on Figure 9 are pretty simple, but with such a system more complex constraints are possible. 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:

[30] Actually, CSP formalizes problem with: X the set of variables, D the restricted domains of these variables, and C the set of constraints on these variables. In the following we consider that the elements X and D are determined and let the user express C.

[31] The acronym of the French railway company: Société Nationale des Chemins de Fer.

[32] We should say a word about the `amb, require, try-again` expressions evaluation: the expression `(amb exp1 exp2 ... expn)` returns the value of one of the n expressions `expi`. For example, the following expression can have six possible values:
`(list (amb 1 2 3) (amb 'a 'b))` ⇒ `(1 a), (1 b), (2 a), (2 b), (3 a), (3 b)`
The developer can add constraints (as predicates) on the values returned by `amb`. These constraints are expressed with the `require` special form. 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. Just like in the PROLOG language, with which the user can ask, one by one, all the solutions of a logical expression, with a nondeterministic interpreter, the `try-again` key-word allows to ask the next successful evaluation for a procedure calling `amb`. The traditional interpretation loop is modified to take into account these backtracks (e.g. by using continuations).

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, 150€
	`((depart montpellier) (dest paris) (price 150)`
	`(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 user of the service, transmits its requests with `require` *and* `try-again` *expressions. At the beginning of the conversation, the SNCF agent (A_S) 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 environment dedicated to the customer agent, E_C^S.*

Figure 9: Scheme expressions in dialogue between the SNCF agent and the customer agent.

```
(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€, else 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)))
```

Or, *"I want a first class ticket only if the price overcharge with respect to a second class ticket, is lower than 10€"* etc. 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 constraint, and let it communicates its ideas. The travel agency example is often used in the Web research community (Semantic Web, ontology building, Data mining, Web service, etc.), but seldom associated to a dynamic interaction with the user as our approach presents.

This experimentation is a first step toward DSG because instead of asking values to user, it asks constraints on values, which is substantially different from the user viewpoint

Remark — Section 2.2 mentions that DSG does not aim to substitute 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*))` etc. In this case, DSG is downgraded to PD. It is equivalent to fill a Web form, what is a current practice in many Web sites.

Remark — 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-kqmlmsg`). The same principles can be used to modify any part of an agent interpreter. We can imagine a prelude to the second experimentation with a scenario where an agent teaches to another one how to transform its interpreter into nondeterministic one.

3.6 The STROBE model tomorrow

3.6.1 Increasing control with dedicated continuations

Continuation is a fundamental notion in programming languages. When a program is interpreted, it represents the following of the current process and allows to reach it. In Scheme there are two ways of doing continuation passing style programming:

- Use the `call/cc` special form 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 evaluation context itself (cf. Section 3.1.4). For example a Scheme nondeterministic interpreter uses two continuations: a succeed one and a failure one.

Our interest concerns only the second way for the moment. Using continuation in the evaluation context means to give to the interpreter not simply the environment where the expression must be evaluated but also 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)))` returns `44`.

Indeed, to increase dynamic control it is interesting to have programs that can access the continuation of the current evaluation, in order to dynamically generate processes. It means to access another part of the evaluation context. It can be done with devices presented in Section 3.1.4 (for example a reifying procedure). Considering that, the STROBE model should regard Cognitive Environment as the embodiment of the complete evaluation context of an expression and then integrate an interpreter (the Cognitive Interpreter) but also continuation(s)[33] (a function called `continuation`). As interpreter and classical functions this continuation can also be dynamically modified, giving to agents a way to improve their capability to adjust their partner model and as consequence increase their adaptability to communicate. For example, we can imagine an agent global environment evolution policy which consists in interpreting messages of a special interlocutor in both its dedicated environment and in the global environment (e.g. if the interlocutor is the creator of the agent). It can be done via the dedicated continuation coding that any message must be interpreted in the global environment after being interpreted in the local environment. Due to the propriety to be dynamically changed, this new addition can be called *Cognitive Continuation*. This perspective for the STROBE model gives to agents access to all the expression evaluation context. Then, **a communication message is interpreted by a dedicated interpreter in a dedicated environment, with a dedicated continuation**. Therefore, the evaluation process is totally controlled and dynamic.

[33] In fact, it is one or several continuations, as in nondeterministic interpretation. What is important is to note that the number of continuations depend on the evaluation procedure `(evaluate e r k)` or `(evaluate e r ks kf)`. In the following, we simply consider one continuation.

3.6.2 i-dialogue: modelling agent conversation by streams

A recent work in progress presented in [JC05] introduces the *i-dialogue* abstraction which aims to model communicative situations such as those where an agent conducts multiple concurrent conversations with other agents. It models conversations among agents by means of fundamental constructs of applicative/functional languages. (i.e. streams, lazy evaluation and higher order functions). The i-dialogue abstraction spreads the elegant model of O'Donnell *dialogue* [O'D85] to more complex situations and considers it for agent communication. An i-dialogue is an interactive session between n agents (A, B_i, ...) which take turns sending messages to each other (messages from A to B_i and vice and versa). Each agent has a state (α_j, β_{ij}, ...) that contains personal information (internals) and information about the history of the conversation. Each agent models the conversation with the interlocutor with the i-dialogue abstraction. During the conversation, each agent computes a new state ($\alpha_{i+1}, \beta_{ij+1}, \ldots$) and a new output ($O^A_{Bi}, \ldots, O^{Bi}_A, \ldots$) from its previous state and the last input ($I^A_{Bi}, \ldots, I^{Bi}_A, \ldots$) it received from other agents, using its transition function ($f^A_{Bi}, \ldots, f^{Bi}_A, \ldots$).

The i-dialogue abstraction seems particularly synergic with the STROBE model: the Cognitive Interpreters are the transitions functions which produce an answer message (i.e. output) according to an incoming message (i.e. input) and an environment (i.e. state). The i-dialogue abstraction has two main advantages: i) to not presuppose anything about the internals of the partner agent and only deal with output streams of messages; ii) to deal with the entire conversation (expressed by potentially infinite and not predetermined streams of messages) and not simply with a message alone.

3.7 The STROBE model as a toolkit for DSG

One important requirement for a progress in the area is conceptual simplicity for the researcher/developer. We think that our proposed model could be perceived as simple enough to be not only understood in its potential, but also used and reused in concrete settings, as we are convinced that it may be the kernel of complex applications, by composition, abstraction and generalisation, of its principles. Our idea is to integrate in a quite evolutive model some concrete tools able to improve fulfilling of requirements of DSG. The STROBE model is thus presented as a toolkit to go towards DSG. That is why we developed this model to fit a maximal of DSG's characteristics: interaction centred, agent oriented, learning centred etc. For example, the two kinds of experimentations (meta-level learning and dynamic specification of programs) illustrate concrete solutions to fulfill two of the characteristics of DSG mentioned in Section 2.4. Meta-level learning is needed to enable *language enrichment* at the three levels (Data, Control, Interpreter), *dynamic specification* shows how to let the user express his own constraints (even if the set of variables and the domains are fixed) concerning the functionality of a program he needs to use. As told in the beginning of the section, the STROBE model is not static: it is developed progressively with the elicitation of characteristics of DSG.

4 Dynamic Service Generation on the Grid for e-learning scenarios

4.1 A service oriented Grid

A fundamental notion which appears in DSG is the notion of network. A PD system can be considered like an unique, independent, and disconnected system, but a DSG system plays its role only connected to a network for which it is at the same time agent user and agent provider. The agent plays then a role of point of entry on the network, he must be capable, while interconnecting many other agents to find any kind of solution for the agent user. An agent is therefore a member of a community. On the Web, the concept of *virtual organization/community* does not exist, it appears with the emergence of the Grid. A virtual organization is a dynamic collection of individuals, institutions and resources bundled together in order to share resources as they share common goals.

The essence of the Grid concept is nicely reflected 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. The Grid aims to enable "resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organisation" [FKT01]. Actually, it was originally designed to be an environment with a large number of networked computer systems where computing and storage resources could be shared as needed and on demand. 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.

Grid technologies have evolved from ad hoc solutions, and de facto standards based on the Globus Toolkit, to Open Grid Services Architecture (OGSA) [FKNT02] which adopts Web service standards and extend services to all kind of resources (not only computing and storage). Today, the Web Service Resource Framework (WSRF) [FFG+04] defines uniform mechanisms for defining, inspecting, and managing stateful resources in Web/Grid services. However, the notion of Grid services do not fully correspond to the service requirements of the Grid. Grid services are quite the same as Web services, adapted to the Grid. However, they represent a quite good step towards DSG as they add to the classical Web services framework two main things: management of state (stateful service) and lifetime (transient service).

4.2 Grid and agents

Foster et al. call service [FKNT02]: *A (potentially transient) stateful service instance supporting reliable and secure invocation (when required), lifetime management, notification, policy management, credential management, and virtualization*. Actually, such a service could be better called an "agent". We further think that the Grid [FK99, FKT01, RJS05] can be seen as the evolution of both Web and agent research. For example, you can not pass from a Client/Server model based network (as the Web) to a distributed resource sharing system (as the Grid) without considering autonomous and intelligent entities on this network and the communication between this entities. Even the simplest functionality claimed by the Grid – the availability of information resources, including computer power, in a way transparent to the Grid user – needs the agent user to fully delegate to the Grid the generation of the service, by accepting at the same time that the Grid adapts dynamically to the run time conditions in order to offer the best service to the user. This delegation is absent in classical Client/Server architectures.

In the same line as Foster et al. [FJK04] who show that Grid (brawn) and agent (brain) need each other, we take the challenge of semantic Grids by interpreting the role of Grids as societies of autonomous interacting agents offering dynamically defined as well as dynamically delivered (generated) services (i.e. DSG) by means of conversations among AA and HA available in the society. This view is not only consistent with current developments on Grid, but with the above outlined state of the art in agents and agent communication, and with our own extension to include HA. Learning Grids (i.e. societies of learning agents) become in this view at the same time societies of agents (HA and AA) supporting human learning. No doubt that HA are autonomous (what is yet to be realised for AA) and solve problems by interacting (what is partially realised by AA). No doubt that their own "internals" are hard to identify (what calls for a social, better than a mental state based, approach to modelling cooperation in mixed societies of agents).

4.3 Dynamic Service Generation implies learning

One requirement of DSG is that the user does not certainly know what he wants. This statement supports our view that DSG implies 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 learned what is the train 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 which imply very complex learning. In this sense, our work is inscribed in a global logic to provide an e-learning infrastructure based on DSG. That means that an agent able to generate dynamically services embodies perfectly Learning Services. If services are embodied by STROBE agents, then STROBE agents can realise Learning Grid Services.

5 Conclusion

We present in this paper the STROBE model as a social oriented model of agent representation and agent communication. We have thoroughly adopted the two requirements of Section 3.1.2: the model i) is highly dynamic, allowing the three levels of learning (Data, Control, Interpreter) at run time (thanks to the concept of Cognitive Interpreters) and ii) is based on a strong representation of partner models (thanks to the concept of Cognitive Environments). STROBE agents are able to interpret communication messages in a given environment, including an interpreter, dedicated to the current conversation. We 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 shown how a STROBE agent can build, dynamically while communicating, a language dedicated to each of its interlocutors. Be-

sides, the model was illustrated with experimentations showing its adequateness and the feasibility of its prototypical applications.

As we saw, the STROBE model is thought and developed as a toolkit for Dynamic Service Generation, a quite new, but nevertheless very natural, concept of service based on a conversational process. Finally we saw that Grid has a strong agent and service oriented future. Thus, the aim of the paper was to explain that:

- Learning depends on interaction;
- DSG is based on interaction;
- DSG implies learning;
- STROBE is oriented interaction and DSG;
- Grid is service oriented;
- Grid and agents need each other;
- **Dynamically generated Learning Grid Services by conversational agents seem to be a significative part of the future of e-learning, Grid, agent, and service research.**

As told, the STROBE model is the result of research in different domains and therefore has the advantage of gluing them together in a coherent way. Working on agents, the Web, applicative/functional programming languages, technologies supporting human learning and more recently the Grid, we have demonstrated **the importance of considering interactions as the core concept in future Social Informatics 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 Computer Science (i.e. interaction centred), as well as to Artificial Intelligence or Cognition that is 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 member of the "team" achieving an effective collective behaviour that is not just the sum of each members' pre-coded contribution.

Acknowledgements

Work partially supported by the European Community under the Information Society Technologies (IST) programme of the 6^{th} Framework Programme for RTD - project ELeGI, contract IST-002205. This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

A Implementation

The STROBE model is under current implementation: 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 can exchange with each other messages using the communication language presented in Section 3.3.5. Their autonomy is characterised by the fact that they can learn alone, by communicating (we do not mind, for the moment, of the agent evolution policy). Their behaviour consists in the REPL loop seen above. Our Scheme interpreters are inspired from [JF92] for reflection and reifying procedure mechanism, [ASS96], and [Que96], for continuations and nondeterministic evaluation. Implementation of such a model is not easy to do because in order to work on language interpretation a meta-interpreter[34] is needed and it increases a lot the complexity and computation load of programs. However, the first and third experimentation of Section 3.5 were implemented and tested. It illustrates such a model is viable and proposes a concrete way to go toward DSG. Nevertheless, we have to mention as well

[34] An interpreter written in the same language that the one it interprets is called a metacircular interpreter or simply meta-interpreter.

restrictions on the agents interacting in these experimentations: i) they have to respect our model and have the same construction (i.e. the same interpreter structure), which is still a contradiction to heterogeneous agents' interactions; 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 AA presented above, and an 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.

An implementation in the the multi-agents platform MadKit (`www.madkit.org`) is in progress. On this platform, STROBE agents are implemented in Java and use Kawa constructs i.e. interpreter and environment to link Java and Scheme (`http://www.gnu.org/software/kawa`).

References

[AP80] James Allen and C. Raymond Perrault. Analysing intention in utterances. *Artificial Intelligence*, 15:143–178, 1980.

[ASS96] Harold Abelson, Geral Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, USA, 2^{nd} edition, 1996.

[Aus62] John L. Austin. *How to do things with words*. Clarendon Press, Oxford, UK, 1962.

[BDB92] S. Berthet, Yves Demazeau, and Olivier Boissier. Knowing each other better. In *11^{th} International Workshop on Distributed Artificial Intelligence*, pages 23–42, Glen Arbor, Michigan, USA, February 1992.

[BF95] Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi agent systems. In *1^{st} International Conference and Multi-Agent Systems, ICMAS'95*, pages 17–24, Cambridge, Massachusetts, USA, 1995. AAAI/MIT Press.

[BGS+91] P. Brazdil, M. Gams, S. Sian, L. Torgo, and W. Van De Velde. Learning in distributed systems and multi-agent environments. In Y. Kodratoff, editor, *Machine Learning: European Working Session on Learning, EWSL'91*, pages 424–439, Porto, Portugal, March 1991.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic Web. *Scientific American*, May 2001.

[BP99] Christian Brassac and Sylvie Pesty. *Simuler la conversation : un défi pour les systèmes multi-agents*, chapter 9, pages 317–345. L'interdisciplinaire, Limonest, France, 1999.

[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 Heidelberg New York, 2000.

[Cer96] Stefano A. Cerri. Computational mathetics tool kit: architecture's for dialogue. In C. Frasson, G. Gauthier, and A. Lesgold, editors, *Intelligent Tutoring Systems, 3^{rd} International Conference ITS'96*, volume 1086 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, Berlin Heidelberg New York, 1996.

[Cer99] Stefano A. Cerri. Shifting the focus from control to communication: the STReam OBjects Environments model of communicating agents. In Padget J.A., 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 Heidelberg New York, 1999.

[CL90] Philip R. Cohen and Hector J. Levesque. Intentions is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[CP79] Philip R. Cohen and C. Raymond Perrault. Elements of a plan-based theory of speech acts. *Cognitive Science*, 3:177–212, 1979.

[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.

[DG00] Frank Dignum and Mark Greaves, editors. *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin Heidelberg New York, 2000.

[Dij75] E.-W. Dijkstra. Non-determinacy and formal verification of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DS83] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[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 Heidelberg New York, 2000.

[EKD03] Marc Eisenstadt, Jiri Komzak, and Martin Dzbor. Instant messaging + maps = powerful collaboration tools for distance learning. In *Simposio internacional de Tele-Educación y Formación Continua, TelEduc'03*, Havana, Cuba, May 2003. See also www.buddyspace.org.

[ESH00] Renée Elio, Ajit Singh, and Afsaneh Haddadi. Task models, intentions, and agent conversation policies. In R. Elio, A. Singh, and A. Haddadi, editors, *Task models, intentions, and agent communication, Proceedings of the Pacific Rim Conference on AI, PRICAI'00*, volume 1886 of *Lecture Notes in Artificial Intelligence*, pages 394–403. Springer-Verlag, Berlin Heidelberg New York, 2000.

[Fer99] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman, Harlow, UK, 1999.

[FFG+04] Ian Foster, Jeffrey Frey, Steve Graham, Steve Tuecke, Karl Czajkowski, Don 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, Web Services Resource Framework, May 2004.

[FGM03] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: An organizational view of multi-agent systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *4th International Workshop on Agent-Oriented Software Engineering, AOSE'03*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer-Verlag, Berlin Heidelberg New York, 2003.

[Fip02] Foundation for Intelligent Physical Agents, FIPA ACL message structure specification, December 2002. www.fipa.org/specs/fipa00061/.

[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 City, New York, USA, July 2004.

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[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. Extended version of Grid Services for Distributed System Integration.

[FKT01] Ian Foster, Carl Kesselman, and Steve Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3), 2001.

[FW76] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming: Third International Colloquium*, pages 257–284. Edinburgh University Press, 1976.

[GB99] Zahia Guessoum and Jean-Pierre Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, 1999.

[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.

[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 Heidelberg New York, 2000.

[Gue02] Francis Guerin. *Specifying Agent Communication Languages*. PhD thesis, Imperial College of Science, University of London, London, UK, June 2002.

[GW04] Dina Goldin and Peter Wegner. The origins of the turing thesis myth. Technical Report CS 04-13, Brown University, Providence, Rhode Island, USA, June 2004.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[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 Heidelberg New York, 2003.

[Hug03] Marc-Philippe Huget, editor. *Communication in Multiagent Systems, Agent Communication Languages and Conversation Poliscies*, volume 2650 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg New York, 2003.

[IJF92] John W. Simmons II, Stanley Jefferson, and Daniel P. Friedman. Language extension via first-class interpreters. Technical Report TR362, Indiana University, Bloomington, Indiana, USA, September 1992.

[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.

[JC04] Clement Jonquet and Stefano A. Cerri. Agents communicating for Dynamic Service Generation. In *1^{st} International Workshop on GRID Learning Services, GLS'04*, pages 39–53, Maceio, Brazil, September 2004.

[JC05] 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.

[JEC05] Clement Jonquet, Marc Eisenstadt, and Stefano A. Cerri. Learning agents and enhanced presence for generation of services on the Grid. In *Towards the Learning GRID: advances in Human Learning Services*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2005. To appear.

[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.

[KLM96] Leslie P. Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[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, Maryland, USA, February 1997. www.cs.umbc.edu/kqml/.

[Mau02] Nicolas Maudet. A la recherche de la structure intentionnelle dans le dialogue. *Traitement automatique des langues*, 43(2):71–98, 2002.

[McC89] John McCarthy. Elephant 2000: A programming language based on speech acts. Unpublished draft, Stanford University, CA, USA, 1989.

[MDCd99] B. Moulin, S. Delisle, and B. Chaib-draa, editors. *Analyse et Simulation de conversation : De la théorie des actes de discours aux systèmes multiagents*. L'interdisciplinaire, Limonest, France, 1999.

[Nor91] Kurt Normark. Simulation of object-oriented and mechanisms in Scheme. Technical report, Institute of Electronic Systems, Aalborg University, Denmark, 1991.

[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, Washington, USA, June 1985.

[OPB00] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *1^{st} International Workshop on Agent-Oriented Software Engineering, AOSE'00*, pages 121–140, Limerick, Ireland, June 2000. Springer-Verlag.

[Que96] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, UK, 1996.

[RG91] A.S. Rao and M.P. Georgeff. Modelling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *3^{rd} International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, San Mateo, California, USA, April 1991. Morgan Kaufmann.

[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.

[Rob83] John Alan Robinson. Logic programming - past, present and future. *New Generation Computing*, 1(2):107–124, 1983.

[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.

[Sea69] John Searle. *Speech acts: An essay in the philosophy of language*. Cambridge University Press, Cambridge, UK, 1969.

[Sia91] S. S. Sian. Adaptation based on cooperative learning in multi-agent systems. In Y. Demazeau and J.-P. Müller, editors, *Decentralized AI, 2^{nd} European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 2, pages 257–272. Elsevier Science Publishers, Amsterdam, Holland, 1991.

[Sin98] Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

[Weg97] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.

[WG03] Peter Wegner and Dina Goldin. Computation beyond turing machines. *Communications of the ACM*, 46:100–102, April 2003.

[Woo02] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, Chichester, UK, February 2002.