INTERNATIONAL JOURNAL OF INTELLIGENT CONTROL AND SYSTEMS
VOL.12, NO.2 JUNE 2007 167-178

Operational Semantics Rules as a Computational Coordination Mechanism in Multi-Agent Systems

Jean-Luc KONING

Abstract— **In the recent years, several methods have been proposed to formally describe the unfolding of a communication process within a multi-agent system. Among those methods, some are perceived as future standards, but all bear limitations. Our goal here, is to look for a simpler and more general method that could formally describe interaction mechanisms during communications.**

We shall begin this paper by giving a brief survey of the various methods used to represent communications within a multi-agent system. Such state of the art is a little broader than the ones regularly found in the domain's literature. Typical artificial intelligence approaches are not often open to other related domains such as distributed calculus, and distributed programming. Some powerful tools like π-calculus start being used. Other more dedicated works like the specification of distributed programming languages, with a strong formalism at the level of message passing, are still underestimated.

In a second stage, we present the initial version of a paradigm based on operational semantics called POS. A new version is then introduced and exemplified. In order to demonstrate our model's feasibility, we have implemented a generic program that enables to simulate any multi-agent system governed by POS rules. The implementation details of this generic simulator as well as examples are discussed. We then conclude on possible future research perspectives as well as improvements that still need to be added to the model.

Index Terms— **Multi-agent systems, agent interaction, operational semantics.**

1. INTRODUCTION

Social insects have been fascinating entomologists for a long time. This is mainly related to the fact that insects are able to accomplish complex tasks through cooperation, while their other cognitive and physical abilities are very limited. The observation of such microcosms has led to a developing research field in computer science that aims at simulating this type of behavior [1].

The first euphoric period where research consisted in developing and running simulations with weak theoretical bases is over. Now, this domain has become part of the distributed artificial intelligence field [2].

This paper will focus on the communication that may take place between different entities, often called agents. Thanks to the interaction phenomenon agents are able to work together and accomplish things they were not capable of doing by themselves otherwise. However, in order to model rather complex systems, one immediately needs to make use of messages that hold strong meanings, and especially know precisely how sending a message would modify the sender's as well as the receivers' behavior. As a matter of fact, attempts in simulating biological systems have often bumped into problems related to message ambiguity. Furthermore, as soon as one wants to endow agents with logical deduction means, one needs to precisely define what agents can deduct upon sending or receiving a message.

Manuscript received October 16, 2006; revised March 15, 2007.

In the recent years, several methods have been proposed to describe in a formal way the unfolding of a communication within a multi-agent system. Among those methods, some are perceived as future standards, but all bear limitations. Our goal here, is to look for a simpler and more general method that could formally describe interaction mechanisms during communications.

We shall begin this paper by giving a brief survey of the various methods used to represent communications within a multi-agent system. Such state of the art is a little broader than the ones regularly found in this domain's literature. Typical artificial intelligence approaches are not often open to other related domains such as distributed calculus, and distributed programming. Some powerful tools like π-calculus start being used. Other more dedicated works like the specification of distributed programming languages, with a strong formalism at the level of message passing, are still underestimated.

In a second stage, we present the initial version of a paradigm based on operational semantics called POS [3] [4]. A new version is then introduced and exemplified. In order to demonstrate our model's feasibility, we have implemented a generic program that enables to simulate any multi-agent system governed driven by POS rules. The implementation details of this generic simulator as well as examples are discussed in the penultimate section. We then conclude on possible future research perspectives as well as improvements that still need to be added to the model.

2. STATE OF THE ART

2.1. Protocol Specification Formalisms

2.1.1) Automata: Presumably automata theory has been one of the first specification approaches to be used for representing interaction issues in multi-agent systems [5], [6]. Such a formalism shows several assets.

- It is a visual tool. Automata are easily represented in a graphical way.
- It is easy to understand. The interpretation of such graphs quickly leads to a rather good understanding of their meaning. Therefore, maintaining protocols becomes quite simple.
- Tools are numerous. In the literature, there is a lot of tools for handling automata as well as protocols represented via automata [7], [8].

However, such a formalism suffers from two crippling weaknesses. First, the automata generated for complex protocols quickly become gigantic. Moreover, not all protocols can be represented via automata.

Second, synchronism cannot be rendered with automata modeling. Indeed, they do not have Turing power. Their expression power is limited to that of regular languages. Yet, for the set of possible interactions to be as large as possible, it is necessary to go beyond such a limit.

It is obvious that a multi-agent system's power compared to a more common system lies in the power of its interactions among its constituents [9]. The more powerful such interactions the more complex the problem tackled by the system. We shall see this in more details in section 3.3 on page 5.

2.1.2) (Colored) Petri Nets: Petri nets go farther than automata. One represents a Petri net by means of a graph with two types of nodes referred to as *places* and *transitions*. Directed arrows always either go from a place to a transition or reverse. They are labeled with an integer indicating the number of tokens required when an arrow is traversed. Tokens circulate within the graph according to the following rule: when all the places prior to a transition each hold at least the required number of tokens for the transition to be fired, they are removed from these places and new ones are added (as many as indicated) to the places found after the transition [10].

For a larger expression power, one may also use colored Petri nets where tokens may belong to several categories generally represented by colors. In this case transition rules then take into account the token's color [11].

This formalism has been successfully used by several people in the domain so far [12] [13]. A lot of tools are available for dealing with protocols represented by Petri nets. The main problems though [14] are their lack of reusability as well as modularity. A protocol's Petri net that is composed of several phases cannot always be represented by corresponding modules.

2.1.3) Temporal Logic: Temporal logic derives from modal logic with the addition of discrete time.[1] An assertion can then be true at a time t and false at a different time. Temporal connectors are also added in order to convey that a formula can be "true at any time", "at least once", "always after time t", etc.

Describing a protocol in temporal logic is performed by means of a set of rules that link two logical terms via an implication. The left term is to be true before the triggering of the rule and the right term then turns true after [15].

Temporal logic's main feature has to do with the validation of protocols that have been defined with it [16]. Indeed, it is possible to make sure that certain logical assertions be verified, like to know if there exists a time t where the protocol ends for instance.

2.1.4) Lotos, Z and other Programming Languages: These specification formalisms are the most powerful ones. They stem from the engineering of communication protocols in the domain of distributed systems. Their use in the domain of multi-agent systems is limited to some rare attempts. This is due to an important difficulty in developing and maintaining complex protocols.

Generally, a protocol described in Lotos [17] [18] can only be easily interpreted by its author. Z [19] is a semantically well defined formal specification language. It is thus often used for the specification of distributed systems. Yet, it is not well suited for representing interactions between agents [20].

Such methods are thus left aside from the domain of multi-agent systems, mainly for the benefit of colored Petri nets.

[1]More precisely, this is the accessibility relationship between worlds that is interpreted as a temporal relation.

2.2. Semantic Aspect of Communications

The superiority of multi-agent systems compared to monolithic systems lies in the sharing of tasks between several distinct entities. In order for such common work to really be more efficient than any centralized one, one needs to develop this inter-agent interaction feature.

Such an interaction is often described as the result of explicit message passing communication. These types of communication may either be synchronous or asynchronous.

Given the need of such a communication, it is important to associate a meaning to any message. For instance, the natural language phrase "*one needs to tidy up this room*" may have three different meanings.

- *The room is out of order.*
- *The listener must tidy up the room.*
- *The speaker is going to tidy up the room.*

Such an ambiguity vanishes as soon as one associates some semantics to the messages. This current work focuses on the development of theories that enable communications where messages possess a strong semantics.

2.2.1) Speech Act Theory: One of the first pieces of work in this area is the speech act theory [21], in which each word is a communication act in itself, and not solely the verbal expression of a message any more.

A message may thus be informative, a command, a suggestion, a query, etc. Within this framework, a message decomposes into three acts: the locutory, the illocutory, and the perlocutory acts. The first is the physical act of actually uttering the message. The second consists in providing the message listener with the speaker's intention. The third corresponds to the acts that directly come from passing the message.

For example, in a command like "*tidy up your room*", the locutory act is the utterance of the different sounds of a sentence. The illocutory act has to do with the listener understanding that the speaker wants him/her to tidy up the room. The perlocutory act is related with all the things the listener does once receiving the message (in case he/she obeys, the perlocutory act then is the tidying up of the room).

2.2.2) Exchange Semantics: The issue here is: why is a message sent, or what does it become once received?

Applying speech act theory within the framework of cooperating multi-agent systems is fairly simple. Agents receiving a message need to conform to the senders intentions. Yet, for a wider application one is confronted to the problem of how

for an agent to ignore a message it has received, in order to keep its freedom.

Papers dealing with protocols semantics address what is required from the agents. In KQML [22] for instance, the semantics associated with the sending of a message formally states that the sending agent intends on sending the message and that the receiving agent noticed the message has been received. Nothing is explicitly required upon the handling of the message by the receiving agents because one should not add constraints on the internal programming of the receiving agent.

2.2.3) Communication Language Semantics: The literature mainly addresses the semantic specifications that help provide a software with the implementation of the agents' communication module. Some research works go farther in that they suggest a complete semantics for distributed programing languages [23].

Such an approach is yet sometimes balanced and enriched through the creation of programming languages that are closer to multi-agent systems, or even languages that implement these theoretical concepts. The language defined by Van Eijk et al [24] fall into this last category. Their language not only allows for the communication[2] of textual messages anymore, but also first order logic assertions (which for instance is not possible with KQML).

The theories that take advantage of such a language are based on the study of the agents' goals generally represented by logical assertions which have to become true. These theories describe the agents in terms of belief, desire and intention.

However, since this work directly stem from very specific theories, they tend to narrow their potential application field. Agents are necessarily of cognitive type, and one does not know yet to what extent protocols can be reused and constructed in a modular way.

2.3. Operational semantics and π-Calculus

Several attempts [25] [26] have been made to give an operational semantics to multi-agent systems. But such work generally heavily depends upon specific multi-agent system architectures. Some other work [27] [28] make use of formal techniques stemming from distributed and concurrent programming show similarities to our work. They mostly adapt π-calculus to multi-agent systems.

2.3.1) π-Calculus: π-calculus has been derived from λ-calculus in order to be able to take into account several processes running in a concurrent way. Since multi-agent systems intrinsically rely on parallelism, a genuine approach would consist in merging these two domains so as to better cope with exchanges between agents.

Links between processes take place via labeled communication channels. These port labels can be broadcasted as any other data.

A π-calculus expression represents a process that may contain several running in parallel. The associated syntax is:

[2]Within their framework, communication is based on rendez-vous, i.e., synchronization takes place between any two agents that want to communicate, which is a less general approach than the one of POS.

$$
\begin{array}{rcl}
\text{Proc} & ::= & 0 \\
& | & \text{Proc}|\text{Proc} \\
& | & \text{Proc};\text{Proc} \\
& | & \text{x}(z).\text{Proc} \\
& | & \uparrow \text{x}(y).\text{Proc} \\
& | & \text{C?Proc} \\
& | & \nu(\text{v})\text{Proc} \\
& | & \text{A}(y_1, \ldots, y_n)
\end{array}
$$

1) Process 0 denotes the inactive process.
2) P|Q denotes the concurrent execution of processes P and Q.
3) P;Q indicates the process behaves either as P or Q.
4) x(z).P denotes that a process is waiting for a message from port x that will be substituted for z. When message y has been received the process turns into P[y/z].
5) $\uparrow$ x(y).P denotes that message y is sent through communication port x.
6) C?P reduces to P if condition C evaluates to true and 0 otherwise.
7) ν(v)P enables to get a new channel of communication.
8) A($y_1, \ldots, y_n$) enables to instantiate variables within a process.

π-calculus expressions can be written in various ways. As in λ-calculus reduction rules enable to define an operational semantics. For example, sending a message makes use of the following reduction:

$$\uparrow \text{x}(y).\text{P}|\text{x}(z).\text{Q} \longrightarrow \text{P}|\text{Q}[y/z]$$

2.3.2) Agents as Concurrent Processes: In its application to multi-agent systems, π-calculus represents agents by processes that can run concurrently [29]. Agents communicate through either private or public channels. In order to accomplish certain tasks the agents can either create new processes that will run in parallel to them.

Let us look at a simple example where agent A asks agent B to perform a task on A's behalf. A sends its request to B on the public channel, as well as the name of a private channel they will be using for further discussion. B answers back, the discussion may then take place.

A formal description of these two agents could be:

$$
\begin{array}{rcl}
A(x,t,y) & \stackrel{\text{def}}{=} & \nu(c)(\uparrow y(\mathit{request},t,c) \cdot c(m) \cdot \\
& & ((m = \mathit{agree})?\text{Q}_1 : (m = \mathit{refuse})?\text{Q}_2)) \\
B(x) & \stackrel{\text{def}}{=} & x(m,z,k) \cdot (m = \mathit{request})\ ? \\
& & (P(z)? \uparrow k(\mathit{agree}); \uparrow k(\mathit{refuse}))
\end{array}
$$

Each step in the calculus is actually a π-calculus reduction step. One therefore obtains a direct semantics such as the operational semantics. The unfolding that corresponds to the above example is:

1) $\{|\ A(a,t,b) \mid B(b)\ |\}$
2) $\{|\ \nu(c)(\uparrow b(\mathit{request},t,c) \cdot c(m) \ldots \mid b(m,t,k) \cdot (m = \mathit{request}) \ldots\ |\}$
3) $\{|\ c(m)((m = \mathit{agree})?\text{Q}_1 : (m = \mathit{refuse})?\text{Q}_2))\ |$

$(request = request)?(P(t)? \uparrow c(agree); \uparrow c(refuse))$ |}

4) {| $c(m)((m = agree)?\mathrm{Q}_1 : (m = refuse)?\mathrm{Q}_2))$ | $P(t)? \uparrow c(agree); \uparrow c(refuse)$ |}
5) {| $c(m)((m = agree)?\mathrm{Q}_1 : (m = refuse)?\mathrm{Q}_2))$ | $\uparrow c(agree)$ |}
6) {| $(agree = agree)?\mathrm{Q}_1 : (agree = refuse)?\mathrm{Q}_2)$ |}
7) {| Q_1 |}

2.3.3) Weaknesses: This calculus system shows several weaknesses.

- One agent is represented by a process but all of the processes are not necessarily agents. Thus the simulation of quite large systems may lead to useless intricacies. Among other things one may lose sight of where the processes come from (which agent are they from), or even which is an agent and which is not.
- Agents disappear. Since a simulation runs through successive reductions according to π-calculus rules, an agent's contents progressively shrinks as the simulation progresses. One solution consists in creating again the agent once the cycle has ended, but this looks awkward.
- Such formalisms does not easily allow to go from theory to practice. Indeed, agents are not independent entities here since some of the processes they need are launched separately. The description of the environment is also not sound since the description of the outside world is rendered by means of processes. One therefore needs a much more efficient mechanism (see the following sections) which stays closer to the actual multi-agent environment.

As a matter of fact, such a calculus system is not too far from pieces of work on mobile code [30] as well as ambients [31] which are labeled fragments of mobile code. This does not exactly fit our purpose.

3. Protocol Operational Semantics

3.1. The POS Formalism

POS (Protocol Operational Semantics) is a new formalism [3] based on algebraic data types and pattern-matching which enables to easily describe interaction protocols at an agent level. Such a model is not only a theoretical framework, but also to a computational one due to the existence of adequate programming languages, like Pizza/Java [32] and other ML languages [33].

A protocol is specified by rules of operational semantics [34]. Let us give a brief definition. A rule may be one of three types:

1) $\langle (sk, par), \phi(world)\rangle \xrightarrow{[Send]} \langle (sk', par'), [\mathcal{A}(world)]\rangle$
2) $\langle (sk, par), \phi(world)\rangle \xrightarrow[msg]{} \langle (sk', par'), [\mathcal{A}(world)]\rangle$
3) $\langle (sk, par), \phi(world)\rangle \xrightarrow{\varepsilon} \langle (sk', par'), [\mathcal{A}(world)]\rangle$

with the following notations:

- A parametrized state consists of a *Skeleton* (*sk* and *sk'*) which is a name, and a *Parameter* (*par* and *par'*) which is a filter that defines what types of messages can be received.
- $\phi(world)$ is a test upon the state of the world.
- $[\mathcal{A}(world)]$ is a list of actions on the world.
- $[Send]$ is a list of messages to be sent and msg is a message to be received. It is given in the form of an object with an algebraic data type on which pattern-matching can be performed

With such a formalism, it is possible to describe complex protocols like interaction protocols for cooperative learning for n agents [35]. Such a protocol allows several agents that own both a knowledge and an experience base to share their knowledge in order to reach a state where all agents have the same knowledge.

These rules help describe an agents' behavior. The whole system evolves each time one agent triggers one of its rules. This triggering is made possible when one agent is in an initial parametrized state of a rule and test ϕ evaluates to true. The agent then goes to the rule's final state by processing a message (in case of a type 2 rule), or by sending a message (in case of a type 1 rule). Then the related actions are accomplished upon the external world. With type 3 rules no messages are exchanged.

Here such a system is meant to be asynchronous. Agents choose a new production as soon as the triggering of a previous one is completed, and message passing is accomplished through FIFO mailboxes. One may synchronize the whole system though by enforcing all the agents to choose one rule at the same time, then to transmit their message, and finally enter a next stage once all are done with their actions.

Although this formalism may seem too interventionist on a cognitive agent, in the sense that it imposes a behavior on the agent according to the rules it possesses, it is actually at least as flexible as other formalisms such as finite state automata or Petri nets [36]. Indeed, starting from any agent state one may define as many transitions as wanted. Choosing a transition or choosing the values of the free parameters of a message to be sent is left in the agent's hands.

Thus, the agents that obey a protocol's rules are autonomous. They are not exclusively reactive, i.e., their behavior cannot be entirely foreseeable from reading the rules. Such cognitive agents have decisional capabilities upon their behavior.

3.2. Limitations

POS' initial purpose is to properly formalize interactions and thus help model cognitive agents. One may bring about three types of improvements on POS' content and form though.

1) POS' productions may be rewritten under one single type of rule. This would simplify their theoretical presentation as well as enable more powerful rules that combine message sending and reception.
2) One may also remove the parameter (*par*) in the parameterized states. Indeed, such parameter corresponds to a unification (on the already received messages) that can be defined at the time a message (above the arrow) is being received. Besides, such parameter has so far been used in an implementation of POS that was very much problem dependent. One may especially remove the constraint on

the type of state and leave the users free to define their own states as they see fit.

3) Finally, one may provide a more rigorous set of definitions by supplying types to the ϕ and $\mathcal{A}$ functions. This information leads to a more formal model

3.3. Power of expression

The POS formalism is at least as powerful as formalisms such as finite state automata or colored Petri nets. Indeed, one can transform an automaton or a Petri net into a set of POS rules. A full example based on Sian's protocol [35] has been thoroughly developed in a previous paper [3].

POS formalism embodies the necessary power to accept regular languages. Furthermore, it has been shown that the POS formalism exhibits the Turing power, i.e., one may describe a Turing complete language by means of POS rules [37].

Now the question is, do communication protocols in multi-agent systems have to be more expressive than what finite state automata allow them to be. An easy answer would be as much expressive as possible because expressiveness cannot harm. Unfortunately such a gain in power of expression is compensated by an increase in complexity regarding the definition of protocols. Because of the simplicity of its rules, POS affords to concentrate on the core of a protocol as opposed to what specification languages can do while avoiding the pitfalls of specification languages (cf. section 2.1.4 on page 2).

Therefore POS is a tool which enables to better study protocols that necessitate a Turing power. We have worked in developing and improving it according to this perspective.

4. Improving POS

4.1. Definition

In order to improve POS one may replace the three types of rule advocated by the following unique type:

$$\langle R, \phi \rangle \xrightarrow{msg} \langle S, \mathcal{A} \rangle$$

where *msg* is a possibly empty finite list of messages of either type

- "SEND *[destination] text*", or
- "RECEIVED *[origin] text*"

R and S being some agent's states. The type of a state is free as long as an equality relation is being defined.

ϕ is a test function on the world which helps determine whether the rule is applicable when a rule is to be chosen. Such a test is not necessarily true during the whole time the rule is applicable. Otherwise one would have to impose that the mappings $\mathcal{A}$ be atomic.

$\mathcal{A}$ is the mapping related to the rule, i.e., the consequences upon the world of the rule that has been chosen by the agent. Particularly, the actions of an agent are modeled through $\mathcal{A}$.

The parameters have been removed since the shape of the awaited messages may in fact be specified in *msg*. It is thus not necessary to over-specify an agent's state anymore.

Here the mappings ϕ and $\mathcal{A}$ are respectively of the following type:

$$\begin{aligned} \phi: &\quad World \rightarrow boolean \\ \mathcal{A}: &\quad World \rightarrow World \end{aligned}$$

$World$ is the type of the world perceived by an agent. This type is more fully described through its implementation at the end of this section.

All the examples that follow to illustrate POS have been implemented using an object-oriented dialect of ML called *OCaml* [38] [39]. For instance, here is the types' definition for the rules:

```
type rule = {init:     state;
             phi:      test;
             messages: (msg list);
             final:    state;
             action:   action}
```

One will see later which types should be defined in order to implement actions and tests.

One imposes that mapping ϕ be without side effects in order to insure that a test made to see whether a rule is applicable does not directly change the world, i.e., without control. Indeed, for example, it would be absurd to break an egg to know whether it is full or empty.

In order to limit the problems related to this test's innocuousness constraint, we have made a distinction between the worlds which the tests bear on from the ones concerned by the actions. A test should not alter the world. Therefore a copy of the world is given as an argument to a test. On the other hand, in our simulations, the world itself is going to be modified by an action. Consequently, a test is called by value and an action is called by reference.

The implemented version of ϕ's and $\mathcal{A}$'s types[3] is thus:

```
type
int_world_actions = agent * (ext_world ref)
and int_world_tests = agent * ext_world
and test = int_world_tests -> bool
and action = int_world_actions -> unit
```

However some problems are left unsolved. Indeed, if a test is a random drawing (tails: the rule is fireable, heads: it is not), the value of an obtained drawing is not necessarily the same during the sorting of the possible rules and during the firing of a chosen rule. This is why a test is performed again when a rule is fired. One thus validates that a test is correct whenever a rule is being chosen.

4.2. Application Example: *The* Collect *Protocol*

Although not mundane, protocol *Collect* is a relatively simple protocol. This interaction protocol allows to solve a set of fundamentally distributed problems. It is a representative example of a large family of typical protocols that can be found in the domain of agent-based systems.

The problem *Collect* is meant to solve consists in handling *objects* spread in a world by means of cognitive *agents*. For

[3] One notices here that the type given back by the actions is the type `unit` which has the only value `()`. Indeed a call by reference implies that the world is itself modified and it is not necessary to give its value back after an action has been performed.

example, in a virtual open world such as the Internet, the objects could be pieces of information, and software agents would have to collaborate in order to process them. In a physical world, agents could be autonomous robots that would have to cooperate like in a harbor setting for instance [40].

One single agent cannot entirely handle any object. Agents need to coordinate their actions in order to handle the objects. Within this project let us suppose that 5 agents are needed.

In order to model such a world one considers an imaginary space in which a set of agents are going to live. Such a space could be represented by a 20×20 square where each agent can only be located at one place (unit cell) at a time although there could be several agents in a single cell (which is a necessary condition for any two agents to cooperate). Initially the objects as well as the agents available are randomly spread in that space for a simulation run.

The policy according to which the agents work is rather simple. They look for an object around them as long as they are not requested. When an agent finds an object, it requests the other agents' help and wait for their answer. If an agent receives a request for help it may give a positive answer if it is free, otherwise it does not answer. When the agent which is waiting for help gets enough positive answers it asks the agents that accepted to help it to come to its location in order to handle the object. Once the handling has been performed the agents scatter. On the other hand, if after a certain time, some positive answers are still awaited the handling is canceled in order to prevent any agent deadlock. With such a protocol, when an agent stops working the whole system does not get affected. Such an overall behavior is represented by means of a transition graph in section 5.

The overall simulation runs have dealt with a number of agents and a number of objects varying from 10 to 100.

5. A Complete Multi-Agent System Simulator

5.1. General Principles

In order to test our proposal on several models and check its implementation we have built a complete multi-agent system simulator and used it on a number of examples.

The whole work has been split up into modules, some related to context- (application-) dependent parts and others related to more general aspects. See section 6 for a detailed overview of the modules needed for the *Collect* application.

One may readily notice two points.

- The principle beneath POS is quite simple. About 250 lines of code (including comments) are sufficient to describe a fully operational simulator.
- The heavy part lies in the implementation of the protocol itself. For example, about 600 lines of code are necessary for the *Collect* protocol. Yet, the *Collect* protocol is a relatively simple although not a mundane protocol.
  We have decided to make use of heterogeneous agents as well as asynchronous message passing to model such a protocol. This has lead to designing 26 rules. Most of them aim at avoiding agents to get stuck upon the reception of unexpected messages. Had one used a synchronous set of agents governed by a global clock, the number of rules would have shrink to 17. This number of rules is even lower than the one necessary for a complete model with the initial POS formalism since now rules may both send and receive messages at the same time (one example will be discussed in section 5.2).

A whole simulation run operates as follows:

1) An agent chooses among the possible rules the one to use, so that the initial state matches the current state, the predicate evaluates to true and the expected messages have been received.
2) These messages are processes one by one according to the order indicated in the rule. The processing firstly consists of deriving the values contained in the received messages, and secondly sending to the right agents the useful values. Indeed, the rules are sometimes defined with generic values that need to be dynamically changed.
3) The action attached to the rule is then applied to the world.
4) Finally, the rule's final state becomes the agent's current state.

5.2. Implementing the Collect Protocol

5.2.1) Set of POS Rules Modeling the Interaction Protocol: Let us see how to implement protocol *Collect* with such a simulator. Figure 1 shows the various possible states an agent can enter based on the informal specification given in section 4.2 on the previous page. The solid lines correspond to transitions that lead to the handling of an object. The doted lines represent cases where not enough agents are available to help thus leading to possibly giving up the handling of an object.

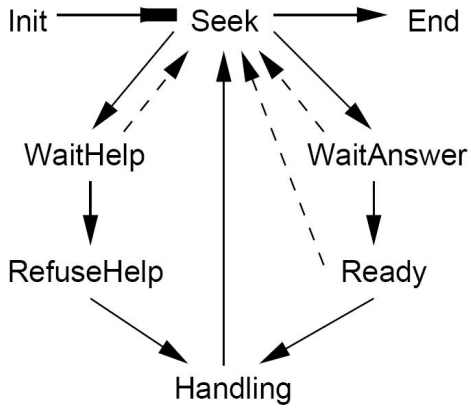

Fig. 1. Agents' states for protocol *Collect*.

The transition between state *Init* and state *Seek* is only given to tell the agents to start. Conversely, the transition between state *Seek* and state *End* is provided to stop the agents once the world is empty.

In state *Seek*, an agent that is not called upon seeks an object to handle. When one such object is found, it goes to state *WaitHelp* and broadcasts a request for help. Agents in state *Seek* that get a request for help may send a positive answer back and go to state *WaitAnswer*. When a requesting

agent obtains enough help, it goes to state *RefuseHelp* and waits for its helping agents to come while turning down any other offer to help. When agents in state *WaitAnswer* get an acknowledgment from a requesting agent, they move toward it and let it know they are *Ready* to help. Finally, when all agents have arrived they deal with the *Handling* of the object, then go back to seeking another one. When a requesting agent does not get enough help, it may cancel the gathering already taking place.

Having presented this, the interaction protocol's rules are gathered into table table I on the following page.

(1) $\langle Init,\ True\rangle \xrightarrow{[]} \langle Seek,\ NOP\rangle$

(2) $\langle Seek,\ not\ \ CloseBy\rangle \xrightarrow{[]} \langle Seek,\ (Move\ 3)\rangle$

(3) $\langle Seek,\ CloseBy\rangle \xrightarrow{[Send(others,NeedHelp(!myPosition,0))]} \langle WaitHelp,\ GoToObject\rangle$

(4) $\langle Seek,\ True\rangle \xrightarrow{[Received(Someone,NeedHelp(!myPosition,0));Send([Someone],Yes\ 0)]} \langle WaitAnswer,\ NOP\rangle$

(5-1) $\langle Seek,\ True\rangle \xrightarrow{[Received(Someone,NeedHelp(!myPosition,0))]} \langle Seek,\ NOP\rangle$

(5-2) $\langle WaitAnswer,\ True\rangle \xrightarrow{[Received(Someone,NeedHelp(!myPosition,0))]} \langle WaitAnswer,\ NOP\rangle$

(5-3) $\langle Ready,\ True\rangle \xrightarrow{[Received(Someone,NeedHelp(!myPosition,0))]} \langle Ready,\ NOP\rangle$

(5-4) $\langle WaitHelp,\ True\rangle \xrightarrow{[Received(Someone,NeedHelp(!myPosition,0))]} \langle WaitHelp,\ Tick\rangle$

(5-5) $\langle RefuseHelp,\ True\rangle \xrightarrow{[Received(Someone,NeedHelp(!myPosition,0))]} \langle RefuseHelp,\ NOP\rangle$

(6) $\langle Seek,\ True\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],TooLate)]} \langle Seek,\ NOP\rangle$

(7) $\langle WaitHelp,\ and\ \ Help2\ \ MyCall\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],Come\ !Goal)]} \langle WaitHelp,\ NOP\rangle$

(8-1) $\langle WaitHelp,\ and\ \ Help1\ \ MyCall\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],Come\ !Goal)]} \langle RefuseHelp,\ NOP\rangle$

(8-2) $\langle WaitHelp,\ Not\ \ MyCall\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],TooLate)]} \langle WaitHelp,\ NOP\rangle$

(9-1) $\langle RefuseHelp,\ True\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],TooLate)]} \langle WaitHelp,\ NOP\rangle$

(9-2) $\langle RefuseHelp,\ AllThere\rangle \xrightarrow{[Send(Helpers,Handling)]} \langle Handling,\ RemoveObject\rangle$

(10) $\langle Handling,\ True\rangle \xrightarrow{[]} \langle Seek,\ Erase\rangle$

(11-1) $\langle WaitAnswer,\ True\rangle \xrightarrow{[Received(Someone,TooLate)]} \langle Seek,\ NOP\rangle$

(11-2) $\langle WaitAnswer,\ True\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],TooLate)]} \langle WaitAnswer,\ NOP\rangle$

(12) $\langle WaitAnswer,\ True\rangle \xrightarrow{[Received(Someone,Come\ !Goal)]} \langle Ready,\ GoHelp\rangle$

(13-1) $\langle Ready,\ True\rangle \xrightarrow{[Received(Someone,Handling)]} \langle Handling,\ NOP\rangle$

(13-2) $\langle Ready,\ True\rangle \xrightarrow{[Received(Someone,Cancel)]} \langle Seek,\ Erase\rangle$

(13-3) $\langle Ready,\ True\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],TooLate)]} \langle Ready,\ NOP\rangle$

(14) $\langle WaitHelp,\ TimeOut\ i\rangle \xrightarrow{[Send([Helpers],Cancel)]} \langle Seek, lambda(x) = ((Move\ 7)\ x; Erase\ x)\rangle$

(15) $\langle Seek,\ EmptyWorld\rangle \xrightarrow{[]} \langle End,\ NOP\rangle$

(16) $\langle WaitHelp,\ True\rangle \xrightarrow{[]} \langle WaitHelp,\ Tick\rangle$

(17) $\langle End,\ True\rangle \xrightarrow{[Received(Someone,Yes\ 0);Send([Someone],TooLate)]} \langle End,\ NOP\rangle$

TABLE I

INTERACTION PROTOCOL'S RULES

5.2.2) Discussion: Let us detail the rule which allows an agent that requests for help to handle an object to send a command to start handling the object to all the agents that have come to help it. A semantic definition of such a rule (*R9-2*) is as follows:

$$\langle \textit{RefuseHelp}, \textit{the agents are there} \rangle \xrightarrow{[\textbf{Send}\ \text{list\_of\_helping\_agents}\ \textit{Handling}]} \langle \textit{Handling}, \textit{Remove the object} \rangle$$

With our implementation choices one may encode this rule in a virtually identical way thanks to the types of rule mentioned above (see section 4.1 on page 5).

```
let r9-2:rule =
     {init:     "RefuseHelp";
     phi:      AllThere;
     messages: [Send (Helpers, Handling)];
     final:    "Handling";
     action:   RemoveObject}
```

Now, what leaves to be done is to write the code related to the rule's test and action. Very often, one of these is commonplace: either the test is always true or the action does not do anything. Here, for illustration purposes we chose one of the rare rules where the messages are not empty, and the test and action are somewhat complex.

One needs 12 rules (1, 3, 14, 8-1, 9-2, 10, 4, 11-1, 12, 13-1, 13-2, 15) in order to directly handle the objects, plus a seeking rule (2) which moves an agent when it does not find any object close by, as well as an accumulation rule (7) which takes into account the coming offers help until they are numerous enough for the handling to happen, or until the agent decides to give up handling that particular object (in that case, an extra transition —a fifteenth rule (16)— assesses the decreasing patience of the agent). This number of rule is smaller than the one used with POS. In our proposal it is possible to have rules that allow both receiving and sending at the same time, such as the following:

```
let r4:rule =
  {init:     "Seek";
   phi:      true;
   messages:
    [Received
       (Someone, NeedHelp (!MyPosition,0));
     Send ([Someone], Yes 0)];
   final:    "WaitAnswer";
   action    Nothing}
```

From a practical perspective, the set of fifteen rules just mentioned has been extended with additional transitions meant to prevent from deadlocks. Indeed an agent waiting for a message gets stuck when receiving new messages that it does not know what to do with. A simple solution could have been to add a low-priority rule to each state which immediately removes the first message in the mailbox. A more explicit solution has been chosen in order to have a direct control upon the agents' behavior.

There are five transitions for refusing to help another agent when this is not possible (5-1, 5-2, 5-3, 5-4, 5-5,), and six transitions for refusing an offer for help when this help arrives too late (6, 8-2, 9-1, 11-2, 13-3, 17). In a first implementation we have considered that pattern-matching on the transitions' initial and final states was not a first priority. Had we implemented it, this latter set of eleven transitions would have shrunk to only two, which would have led to the same figures (seventeen rules) as with POS. The difference though is that we have added some rules in order to deal with cases that did not take place in the simulation with POS since it was a synchronous one.

Our proposal allows for both an easy development of automatic protocol generation as well as the combination of already existing protocols. Automatic generation of protocols enables to endow the agents to build their own protocols themselves provided they possess some general rules. One also may endow the agents with rules that allow them to learn new rules from other agents. The combination of protocols is a set of methods whose aim is to build new protocols based on several existing protocols.

Some methods for automatically enriching protocols (such as, merging of rules, association of protocols, etc.) will be presented in another paper.

6. Architecture and Implementation Issues

6.1. General Architecture

In order to ease the understanding of POS' application examples let us detail the architecture and implementation of our multi-agent simulator and have a deeper look at its generic part. The whole program is divided into two parts: the one related to the simulator itself and the other to the multi-agent application to be simulated.

Altogether there are 12 modules which are related to each other according to the dependence graph shown in figure 2 on the following page.

In this figure, the dependencies are represented by means of arrows. The underlined modules are related to the simulator's generic part, i.e., they may be reused as such for the simulation of other multi-agent applications. The modules given in the dotted rectangles could be gathered into one but have been kept separated for code readability purpose.

The three top modules are modules related to types. Among them, only *AdditionalTypes* is related to the application to be simulated.

In order to better understand the technicalities let us detail the various basic types. Rules, tests and actions have already been discussed in section 4.1 on page 5. A remaining type deals with the agents themselves. Such a type is given in the *GenericTypes* module since its structure stays the same in any multi-agent system provided the agents communicate in an asynchronous manner.

```
type agent =
        {id:              identity;
         mutable stat:    state;
         mutable mailbox: message list;
         body:            ad_agent;
```

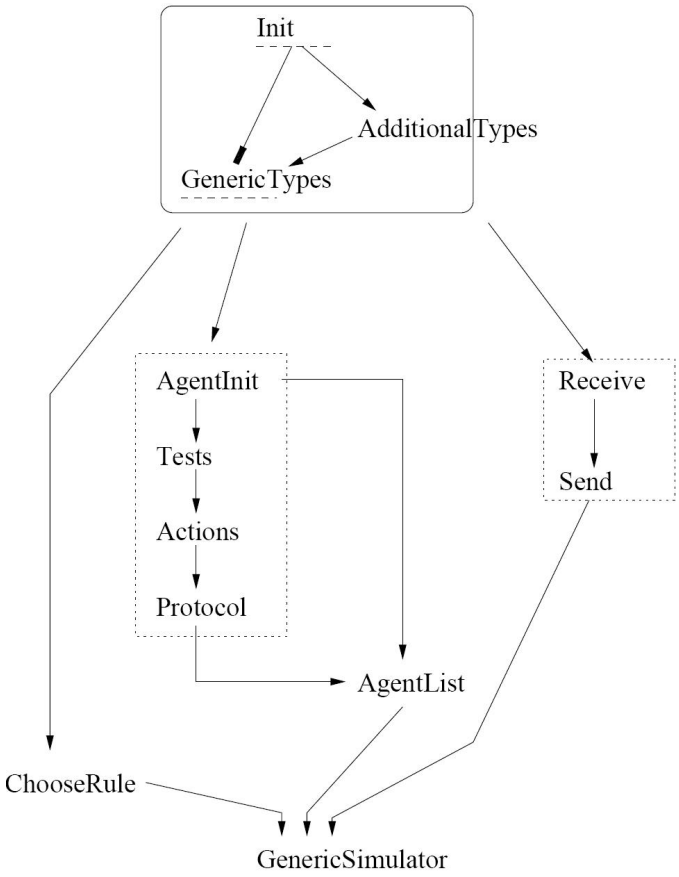

Fig. 2. Module dependence graph.

```
mutable prot:    protocol};;
```

- An agent has its own identity. The `identity` type correspond to integers plus an "`unknown`" identity.
- An agent state may change along the way. It is usually denoted by a string but this could be modified depending on the application. There is no explicit constraint upon the states type, except that one needs to be able to perform some unification in order to be able to choose among applicable rules from a given state.
- Each agent owns a mailbox where all the messages are stored prior to being read.
- The agent's body corresponds to anything related to the application. For example, this could be the location in case the agents represent physical entities.
- The protocol is the list of rules an agent can use. In case the agents learn new rules along the way, this field needs to be `mutable`.

The four modules at the center of the dependence graph have to do with the agents and thus entirely depend upon the simulated system. The first module (*AgentInit*) defines constants such as the number of agents. The other module names are self explanatory. The last one (*Protocol*) holds a function that allows to define the list of agents needed for starting the simulation stage.

The *ChooseRule* module contains a function that enables to know which rule to trigger depending on the current agent and the outside world. Such a function holds two parts. First, it determines which rules are applicable. Second, it chooses one of them.[4]

In our implementation, we did not rely on this way of choosing among rules. Therefore the two tested strategies consisted either in choosing the first applicable rule or picking up one at random.

The *receive* and *send* modules' purpose is to handle the messages' contents. Indeed, the rules are not precisely defined for all the possible message values. In an auction system for instance it would be cumbersome to define one rule per possible price. One therefore needs to substitute the actual values for the symbolic ones when sending messages. Conversely, one needs to extract these actual values upon reception.

6.2. *The* Collect *Application*

Overview of the *OCaml* modules needed for the *Collect* application.

Generic code	Application dependent code
init.ml (4)	additionalTypes.ml (49)
genericTypes.ml (40)	agentInit.ml (29)
genericSimulator.ml (180)	tests.ml (67)
main.ml (37)	actions.ml (66)
	protocol.ml (195)
	agentList.ml (24)
	chooseRule.ml (59)
	receive.ml (35)
	send.ml (56)
Number of lines: 261	Number of lines: 580

TABLE II
LINES OF CODE AND MODULES OF THE POS SIMULATOR

7. CONCLUSION AND PERSPECTIVES

7.1. *Concluding Remarks*

While implementing our model we uncovered both practical and theoretical problems. The theoretical ones have been solved along the way as were most of the practical ones. But some still need to be considered.

For example, our rules are static ones. Their initial state is fixed upon definition. However some rules could be defined for any possible state. Let us suppose one would like to add a rule that stops immediately the systems in any given state, our current implementation forces us to define it n times, n being the number of states.

Similarly, sending a value in a message is done by substitution when the predefined value is to be sent, whenever the rule is being created. A better solution would be to create rules that could be modified by the agent itself. However, implementing such a system then becomes more complex. The current implementation is a good alternative.

One other issue is that when an unexpected message is being received the related agent gets stuck. In the examples

[4]This choice function could have been included as a generic field within the agents. We did not do so in our implementation for ease of access reasons. Eventually, one intend to make such a generic field mutable in order for the agents to possess a rule choice strategy that could change over time.

we have tested specific rules have been added to prevent such deadlocks. On the other hand, when dealing with complex protocols, or in case of systems where the agents do not necessarily know all the messages they may receive, such a solution becomes impossible. One should then either choose to add a rule to be triggered that eliminate the problematic message when encountering deadlocks, or to enable the agents not only to check their mailbox' first message but also any other message they want.

7.2. Possible Perspectives

The approach presented in this paper focuses most on the interaction issue at the agents level. This work is promising and needs to be applied and used on still more actual systems in order to fully convince of its added value. So far, this has been the case with a soccer robot application [41], [4].

From a theoretical standpoint, no extension directly need to be added to the model. On the other hand, it would be interested to demonstrate some properties of protocols that have been defined using POS rules. For example, one may show in a statistic manner that the *Collect* protocol discussed in this paper allows to collect all the objects in the world provided that the number of agents is greater than or equal to the necessary number of agents to handle a single object. Such a proof has not been achieved here since it obviously is a statistic one and is out of this paper's scope.

Writing such mathematical proofs would help arguing about the power and usefulness of POS' formalism compared to others. One would like to go even farther than that by building validation tools that would for instance demonstrate the termination of a protocol or the accessibility of some of the protocol's states, such as the ones found for validating Petri nets. Such work is currently being tackled.

ACKNOWLEDGMENT

The author would like to thank François Dutot for his help on this work.

REFERENCES

[1] G. N. Gilbert and R. Conte, Eds., *Artificial societies: the computer simulation of social life*. London: UCL Press, 1995.

[2] L. Steels and R. Brooks, *The 'Artificial Life' Route to 'Artificial Intelligence' - Building Situated Embodied Agents*. New Haven: Lawrence Erlbaum Ass., 1994.

[3] J.-L. Koning and P.-Y. Oudeyer, "Introduction to POS: A protocol operational semantics," *International Journal on Cooperative Information Systems*, vol. 10, no. 1 & 2, pp. 101–123, 2001, special Double Issue on "Intelligent Information Agents: Theory and Applications".

[4] P.-Y. Oudeyer and J.-L. Koning, "Modeling soccer-robots strategies through conversation policies," in *2nd IEEE International Symposium on Agent Systems and Application (ASA/MA-00)*, F. Mattern and D. Kotz, Eds. Zürich, Switzerland: Springer Verlag, September 2000.

[5] B. Burmeister, A. Haddadi, and K. Sundermeyer, "Generic, configurable, cooperation protocols for multi-agent systems," in *From Reaction to Cognition*, ser. Lecture notes in AI, C. Castelfranchi and J.-P. Muller, Eds., vol. 957. Berlin, Germany: Springer Verlag, 1995, pp. 157–171, appeared also in MAAMAW-93, Neuchatel.

[6] A. Haddadi, *Communication and Cooperation in Agent Systems: A Pragmatic Theory*, ser. Lecture Notes in Computer Science. Springer Verlag, 1996, vol. 1056.

[7] I. R. Alpes, "Cadp (caesar/aldebaran development package)," http://www.inrialpes.fr/vasy/cadp.html.

[8] H. Garavel, "An overview of the eucalyptus toolbox," in *Proceedings of the COST247 International Workshop and Applied Formal Methods in System Design*, Z. Vrezocnik and T. Kapus, Eds., University of Maribor, Slovenia, June 1996.

[9] M. P. Singh, "Toward interaction oriented programming," in *Second International Conference on Multi-Agent Systems (ICMAS-96)*, Tokyo, Japan, December 1996.

[10] C. Xiong, T. Murata, and J. Tsai, "Modeling and simulation of routing protocol for mobile ad hoc networks using colored petri nets," *Research and Practice in Information Technology*, vol. 12, pp. 145–153, 2002, australian Computer Society.

[11] J. Fernandes and O. Bello, "Modeling of multi-agent system activities through colored petri nets: An industrial production system case study," in *16th International Conference on Applied Informatics*, Anaheim, CA, 23–25 February 1998, pp. 17–20. [Online]. Available: citeseer.ist.psu.edu/fernandes98modeling.html

[12] R. S. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng, "Modeling agent conversation with colored Petri nets," in *Autonomous Agents'99, Special Workshop on Conversation Policies*, J. Bradshaw, Ed., May 1999.

[13] J.-L. Koning, G. François, and Y. Demazeau, "Formalization and pre-validation for interaction protocols in multiagent systems," in *13th European Conference on Artificial Intelligence (ECAI-98)*, H. Prade, Ed. Brighton, UK: John Wiley & Sons, August 1998, pp. 298–302.

[14] J.-L. Koning and M.-P. Huget, "A semi-formal specification language dedicated to interaction protocols," in *Information Modeling and Knowledge Bases XII*, ser. Frontiers in Artificial Intelligence and Applications, H. Kangassalo, H. Jaakkola, and E. Kawaguchi, Eds. Amsterdam: IOS Press, 2001.

[15] M. Wooldridge, *Temporal Belief Logics for Modeling Distributed AI Systems*. Wiley Interscience, 1995, ch. 10, g.M.P. O'Hare and N.R. Jennings.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[17] S. Gallouzi, L. Logrippo, and A. Obaid, "Le lotos : théorie, outils, applications," in *Conférence Francophone de l'Ingénierie des Protocoles (CFIP)*, O. Rafiq, Ed. Hermes, 1991, pp. 385–404.

[18] H. Garavel, "Introduction au langage lotos," INRIA Alpes, Tech. Rep., 1990.

[19] J. Spivey, *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.

[20] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge, "Formalisms for multi-agent systems," *The Knowledge Engineering Review*, vol. 12, no. 3, 1997.

[21] J. Searle, *Speech Acts: An Essay in the Philosophy of Language*. Cambridge: Cambridge University Press, 1969.

[22] T. Finin, Y. Labrou, and J. Mayfield, "KQML as an agent communication language," in *Software Agents*, J. Bradshaw, Ed. MIT Press, 1997.

[23] K. Takeuchi, K. Honda, and M. Kubo, "An interaction-based language and its typing system," in *PARLE*, ser. Lecture Notes in Computer Science, vol. 817. Springer-Verlag, 1994, pp. 398–413.

[24] R. van Eijk, F. S. de Boer, and W. van der Hoek, "Operational semantics for agent communication languages," University of Utrecht, Tech. Rep. UU-1999-08, 1999.

[25] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl, "Foundations of a logical approach to agent programming," in *Intelligent Agent II - ATAL'95*, ser. Lecture Notes in Computer Science, M. Wooldridge, J. Muller, and M. Tambe, Eds., vol. 1037. Springer-Verlag, 1995, pp. 331–346.

[26] A. Rao, "Agentspeak(l): Bdi agents speak out in a logical computable language," in *Agents Breaking Away - Maamaw-96*, ser. Lecture Notes

in Computer Science, W. van de Welde and J. Perram, Eds., vol. 1038. Berlin: Springer-Verlag, 1996, pp. 42–55.

[27] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. C. Meyer, "Control structures of rule-based agent languages," in *Intelligent Agents V (ATAL-98)*, ser. Lecture Notes in Computer Science, J. Muller and A. Rao, Eds., vol. 1555. Springer-Verlag, 1999, pp. 381–396.

[28] J. Ferber and O. Gutknecht, "Operational semantics of a role-based agent architecture," in *6th Int. Workshop on Agent Theories, Architectures and Languages (ATAL-99)*. Springer-Verlag, 1999.

[29] ——, "Pour une sémantique opérationelle des systèmes multiagents," in *Systèmes Multiagents : Méthodologie, Technologie et Expériences (JFIADSMA-00)*, S. Pesty and C. Sayetta-Fau, Eds. Hermes, October 2000, pp. 39–56.

[30] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy, "A calculus of mobile agents," in *7th International Conference on Concurrency Theory (CONCUR-96)*, ser. Lecture Notes in Computer Science, vol. 1119. Pisa, Italy: Springer-Verlag, August 1996, pp. 406–421.

[31] L. Cardelli and A. Gordon, "Mobile ambients," in *ETAPS'98*, ser. Lecture Notes in Computer Science, vol. 1378. Springer-Verlag, 1998, pp. 140–155.

[32] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice," in *4th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[33] R. Milner, "A proposal for standard ML," in *ACM Conference on Lisp and Functional Programming*, 1987.

[34] G. Plotkin, "A structural approach to operationnal semantics," Aarhus university, Computer Science Department, Denmark, Tech. Rep. DAIMI FN-19, 1981.

[35] S. S. Sian, "Adaptation based on cooperative learning in multi-agent systems," in *Decentralized AI*, Y. Demazeau and J.-P. Müller, Eds., vol. II. Amsterdam, The Netherlands: Elsevier Science Publishers B.V., 1991, pp. 257–272.

[36] J.-L. Koning and P.-Y. Oudeyer, "Modeling and implementing conversation policies using POS," in *International Conference on Communications in Computing (CIC-00)*, B. d'Auriol, Ed. Las Vegas, NV: CSREA Press, June 2000.

[37] J.-L. Koning, "Dynamic choice of agent interaction protocols," in *Intelligent Information Processing (IIP-2002)*, M. Musen, B. Neumann, and R. Studer, Eds., 17th IFIP World Computer Congress. Montréal, Canada: Kluwer Academic Publishers, August 2002, pp. 167–176.

[38] E. Chailloux, P. Manoury, and B. Pagano, *Developing Applications with Objective Caml*. O'Reilly, 2002.

[39] X. Leroy, *The Objective Caml system, release 3.00*, 2000, http://caml.inria.fr/ocaml/htmlman.

[40] A. Drogoul, M. Tambe, and T. Fukuda, Eds., *Collective Robotics Workshop 98*, ser. LNAI, no. nř1456. Springer-Verlag, 1998.

[41] J.-L. Koning and P.-Y. Oudeyer, "Designing soccer-robot strategies using POS, a protocol operational semantics," *Applied Intelligence*, 2006.

Jean-Luc Koning received his Ph.D. in Computer Science from the University of Toulouse, France, in 1990. He was Visiting Scientist at Carnegie-Mellon University, Pennsylvania, in 1990 and 1991, and became Associate then Full Professor of Computer Science at the Grenoble Institute of Technology.

He is a co-founder of the Complex Cooperative Systems (COSY) research team. He headed this team from 2000 to 2006 and was Vice-Director of the LCIS research Laboratory from 2003 till 2006. Dr. Koning's research focuses on multi-agent systems, agent interaction, and methodologies.