DYNAMIC CHOICE OF AGENT INTERACTION PROTOCOLS

Jean-Luc Koning
Inpg-CoSy
50, rue Laffemas, BP 54
26902 Valence cedex 9, France
Jean-Luc.Koning@esisar.inpg.fr

Abstract One issue that is never tackled when dealing with conversation policies handled by agents is the way they get to choose the right interaction protocol one in order to answer the problem at hand. This paper is a first attempt to automate such a procedure.

This raises a number of questions when trying to find a suited framework that allows to dynamically gather the various agents into “conversation groups”. Several groups should be able to coexist at the same time, each solving a specific problem. A single agent should also be able to be part of several groups, i.e., carry several conversations at the same time.

Such a mechanism can be handled via a POS protocol, in exactly the same way as the other regular interaction protocols, except for the fact that this one will be triggered from the beginning and will keep on running the whole time. This paper is a first attempt to automate such a procedure.

Keywords: Autonomous software agents, conversation policies

Introduction

Multiagent systems are based on the notion that if one gathers simple autonomous systems (agents) within a society and that they are endowed with interaction skills, then such a society may show complex behaviors (compared to each of the agent’s skills) that are not implemented in any particular individual. The society’s intelligence is distributed among its constituents. Besides, one fundamental characteristic of such systems is its ability of interaction between its constituents, i.e., its agents.

In the multiagent system community Cammarata et al., 1988 the communication issue is typically addressed by means of interaction protocols

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35602-0_35](https://doi.org/10.1007/978-0-387-35602-0_35)

M. A. Musen et al. (eds.), *Intelligent Information Processing*

© IFIP International Federation for Information Processing 2002

that give more or less flexible guidelines to cognitive agents in their communication with each other. Given its knowledge bases, what it perceives from its outside world and the possible messages it gets from other agents, a cognitive agent communicates with the others both through message passing (direct communication) and modification of the world (indirect communication). More precisely, interaction protocols should enable cognitive agents to know which types of message they may receive, which types they may send and at what time, and they should clarify to a certain extent the agents' side effects on the outside world.

The design of protocols Burmeister et al., 1995 classically relies on building a graph, where vertices represent the system's conversational states, i.e., the current state of the conversation the agents are participating in, and where edges correspond to conversational events, i.e., the sending or receiving of a message. Such protocol graphs describe the possible conversation courses. Yet several inherent weaknesses are attached to this approach.

This kind of representation only offers a syntactic description of the protocol. No semantic information can be taken into consideration. Moreover, the particular family of protocols that are thus addressed falls in the class of regular languages. This means multiagent systems with such protocols deny themselves any chance of a larger interaction expression power.

Making such agent interaction protocols operational requires each agent to possess a general and quite flexible protocol execution algorithm, i.e., primitives for reading, processing, and sending messages. This piece of code is totally independent of any specific protocol. Modifying a multiagent system strategy comes down to substituting a state transition graph description for a new one. A protocol is not actually hard-coded within the agents. Even though schemes have been devised to automate the translation of a graphical representation to an equivalent computable piece of code, such a solution is not fully satisfying. Among other things, they do not lead to a modular enough way of implementing strategies. For instance, adding or removing conversational states to a protocol forces a complete change of protocol.

Note, that when implementing a limited number of heterogeneous roles in a team of few cognitive agents, the standard approach is, though being tedious, still more or less feasible. But for larger teams, based on additional heterogeneity, this is obviously not the way to go. Therefore, we propose here a Protocol Operational Semantics (POS) Koning and Oudeyer, 2000, inspired by the structural operational semantics Plotkin, 1981 of programming languages Hennessy, 1990.

1. Modeling Cognitive Agent Interaction Protocols

1.1. The POS Formalism

POS (Protocol Operational Semantics) is a formalism Oudeyer and Koning, 2000 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 Odersky and Wadler, 1997 and other ML languages Milner, 1987.

A protocol is specified by rules of operational semantics Plotkin, 1981. Let us give a brief definition of a simplified version of POS.

A rule is of the following 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* are 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:

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

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 Leroy, 2000. For instance, here is the types’ definition for the rules:

```
type rule = {init:      state;
```

```

phi:      test;
messages: (msg list);
final:    state;
action:   action}

```

We shall see later what are the types 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 master this test's innocuousness constraint one could bring into play the uniqueness typing developed with the *clean* language Plasmeijer and van Eekelen, 1998 Barendsen and Smetsers, 1996. Such typing enables to guaranty the absence of side effects when needed while allowing some if necessary.

An expression is identified as unique if there is only one reference to that expression. One may then destructively change its value without modifying anything though. On the other hand, if that expression is not identified as unique, there cannot be any modification in it.

Therefore, if this kind of type is forced on function ϕ with no uniqueness identification, then this function's argument is the external world and returns a boolean without modifying the world since it may be referenced somewhere else.

Function $\mathcal{A}$ may use the following type:

$$\mathcal{A} : *World \rightarrow *World$$

where $*$ is the uniqueness identification. This function's argument is thus a unique world et returns it once it has been modified.

The implemented version of ϕ 's and $\mathcal{A}$'s types¹ 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

¹One 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.

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.

1.2. 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. Such 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.

Now the question is, do communication protocols in multiagent systems have to be more expressive than what finite state automata allow them to be. An easy answer would be to 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.

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.

2. Dynamic Choice of Protocols by an Agent

2.1. The Issue

One issue that is never tackled when dealing with conversation policies handled by agents is the way they get to choose the right interaction protocol in order to answer the problem at hand. This paper is a first attempt to automate such a procedure.

Let us suppose that a set of agents have at their disposal a number of protocols which can potentially be used in any conversation about a given problem submitted to them. In other words, one steps aside the standard setting where one problem is being solved via interaction protocols within the context of a unique conversation. One is faced here with a set of problems that may be given to a society of agents in any order, or even at the same time.

This raises a number of questions when trying to find a suited framework that allows to dynamically gather the various agents into "conversa-

tion groups". Several groups should be able to coexist at the same time, each solving a specific problem. A single agent should also be able to be part of several groups, i.e., carry several conversations at the same time.

2.2. A Solution Based on Recursive Use of Protocols

Such a mechanism can be handled via a POS protocol, in exactly the same way as the other regular interaction protocols, except for the fact that this one will be triggered from the beginning and will keep on running the whole time. Let us call it the *Init Protocol* (whose name stems from the "init" process in a Unix operating system). All the agents of the system will then always be in contact through the related conversation.

Its role will be to manage the requests received any agent for participating in a given conversation, as well as the spontaneous need for an agent to start a new or enter an ongoing conversation. This *Init Protocol* will enable to fork new protocols in order for agents to enter a conversation.

Another important element is the table that is associated to every conversation. It will list all the participating agents along with the conversation's description. In case of a standard conversation (not operated by the *Init Protocol*), a table holds the following information :

- t_i denotes the table's address. This characterizes any table and allows the agents to access it.
- the description of the corresponding conversation, which amounts to the name/number of the protocol P_i used. One supposes here that one protocol is being used for any given conversation.
- the list of identities/addresses of the agents involved in the conversation.

The *Init Protocol* will also possess such a corresponding table with the following information.

- its address denoted by t_{init} .
- the list of addresses of the tables associated to the running conversations, as well as their related description.
- the list of all the agents of the society that can participate in the conversations. Such a list may be dynamic.

3. The Init Protocol

3.1. Its POS definition

Here is a POS definition of the *Init Protocol*:

- (1) $\langle \text{Wait}, \text{True} \rangle \xrightarrow{[\text{Received}(\text{SomeOne}, \text{Enter}(P_i, t_i))]} \langle \text{Ponder}(P_i, t_i), \text{NOP} \rangle$
- (2) $\langle \text{Ponder}(P_i, t_i), \text{InterestedIn}(P_i, t_i) \rangle \xrightarrow{\parallel} \langle \text{Wait}, \text{Start}(P_i, t_i) \rangle$
- (3) $\langle \text{Ponder}(P_i, t_i), \text{!InterestedIn}(P_i, t_i) \rangle \xrightarrow{\parallel} \langle \text{Wait}, \text{NOP} \rangle$
- (4) $\langle \text{Wait}, \text{CurrentlyInterestedIn}() = P_j \text{ and } \text{ExistsConversation}(P_j) = t_j \rangle \xrightarrow{\parallel} \langle \text{Wait}, \text{Start}(P_i, t_i) \rangle$
- (5) $\langle \text{Wait}, \text{CurrentlyInterestedIn}() = P_j \text{ and } \text{ExistsConversation}(P_j) = \text{null} \rangle \xrightarrow{\parallel} \langle \text{Wait}, \text{CreateConversation}(P_j) \rangle$

3.2. Its side effects

Three specific side effects need to be introduced depending whether a conversation is being created, entered or left. Such side effects apply to the protocols themselves.

Almost all messages exchanged among the agents belong to a given conversation. However there is one exception where a message is sent from a standard protocol to the *init* conversation, i.e., $\text{Enter}(P_i, t_i)$. Such a message is sent by an agent participating in conversation P_i whose table is at address t_i and who would like to call on a receiving agent whose identity/address has been found in table t_{init} .

In order to make up its mind the receiving agent evaluates predicate $\text{InterestedIn}(P_i, t_i)$. The first side effect $\text{Start}(P_i, t_i)$ is triggered and consists for the calling agents in first adding its name into the table at address t_i , and second “fork”² a protocol P_i .

Besides, evaluating $\text{CurrentlyInterestedIn}()$ looks up the set of protocols at disposal and returns one (its identity/address) at random among the ones the agent considers interesting, or returns *null* if none are available. The expression $\text{ExistsConversation}(P_j)$ evaluates to the address of the corresponding table if a conversation that uses P_j already exists. Such a piece of information is available in table t_{init} . The expression otherwise returns *null*.

The second side effect $\text{CreateConversation}(P_j)$ amounts for the calling agents to create a new table t_j , add this table's address into table t_{init} , and register its own address into table t_j . Let us note that the first agent to launch a conversation is the one to create the related table. Since this

²The word “fork” is used here in the same way as it applies to processes in a Unix system.

table is an independent entity, the creating agent may well leave the conversation while other agents keep on interacting.

This raises the issue of the end of a conversation. Let us first introduce the third side effect that deals with leaving a conversation. *Exit*(P_j, t_j) consists for the calling agent to unsubscribe from table t_j and “kill”³ protocol P_j . One therefore needs to endow each table the ability to check whether there still remains any agents which subscribed to the conversation. If such is not the case a table would get rid of itself once it has unsubscribed from table t_{init} .⁴

Regarding the implementation issue, a suggestion would be to have each agent handle one message box per conversation it is involved in. It creates it whenever a protocol is forked and discards it when it leaves the conversation. In order to broadcast the messages into the various message boxes, one may label each message with the table’s address t_i related to the conversation at hand.

4. Discussion

The system’s architecture described in this paper does not mention any constraint regarding whether the operations should be atomic. In any case this seems difficult to bring into play. This means that when designing protocols, one needs to take into account the possibility that by the time an agent receives a messages (sent by another agent) it has unsubscribed from the conversation.

This is the general problem of interactions in open systems where agents dynamically enter or leave the system. One way to address this is to design protocols that are not sensitive to such dynamic behavior. This for example is the case for the second POS protocol in the RoboCup application detailed in Oudeyer and Koning, 2000.

Another way would consist in explicitly implementing the synchronization within the protocol. A message would inform of its intent to leave a conversation and then would pause until some acknowledgment is sent back to it.

So far, one implicitly forced the conversations to be based on different protocols (P_i) at any time. One could nonetheless contemplate the use of several conversations based on a single protocol. This raises the problem of how two simultaneous conversations run by an agent get mutually influenced. Such a problem seriously needs to be tackled when considering the building of an actual multiagent system. One could imagine

³The word “kill” is an analogy to the killing of processes in Unix systems.

⁴Such tables could almost be considered as agents.

that a designer provide constraints such as “ P_3 and P_5 cannot be run at the same time”. An agent would then be able to use such constraints in order to decide among *InterestedIn*(...) and *CurrentlyInterestedIn*(...).

5. Summary and Perspectives

This paper dealt with the issue for an agent to dynamically choose among interaction protocols.

From a practical standpoint, one is left here with the question of making such a paradigm operational. Several implementation examples have been suggested so far, with typed functional languages (ML/Haskel), or object oriented languages (Java/Pizza). The next stage consists in defining a means to implement POS protocols without assuming the use of any particular programming languages for the agent’s internal architecture.

One suited solution to implement this is through the XML technology, in the same way and possibly as a complement to SOAP, which is an XML language that allows to accomplish some remote procedure call in an easy and light approach.

Acknowledgments

I would like to thank Pierre-Yves Oudeyer and Pierre-François Dutot for their help in preparing this paper.

References

- Barendsen, E. and Smetsers, S. (1996). Uniqueness typing for functional languages with graph rewriting semantics. *Math. Struct. in Computer Science*, pages 1–36. Cambridge University Press.
- Burmeister, B., Haddadi, A., and Sundermeyer, K. (1995). Generic, configurable, cooperation protocols for multi-agent systems. In Castelfranchi, C. and Muller, J.-P., editors, *From Reaction to Cognition*, volume 957 of *Lecture notes in AI*, pages 157–171, Berlin, Germany. Springer Verlag. Appeared also in MAAMAW-93, Neuchatel.
- Cammarata, S., Arthur, D. M., and Steeb, R. (1988). Strategies of cooperation in distributed problem solving. In Bond, A. and Gasser, L., editors, *Readings in Distributed Artificial Intelligence*, pages 102–105. Morgan Kaufmann Publishers, Inc., San Mateo, CA.
- Hennessy, M. (1990). *The Semantics of Programming Languages: An Introduction Using Structured Operational Semantics*. Wiley.
- Koning, J.-L. and Oudeyer, P.-Y. (2000). Modeling and implementing conversation policies using POS. In d’Auriol, B., editor, *International Conference on Communications in Computing (CIC-00)*, Las Vegas, NV. CSREA Press.
- Leroy, X. (2000). *The Objective Caml system, release 3.00*. <http://caml.inria.fr/ocaml/htmlman>.

- Milner, R. (1987). A proposal for standard ML. In *ACM Conference on Lisp and Functional Programming*.
- Odersky, M. and Wadler, P. (1997). Pizza into Java: Translating theory into practice. In *4th ACM Symposium on Principles of Programming Languages*, Paris, France.
- Oudeyer, P.-Y. and Koning, J.-L. (2000). Modeling soccer-robots strategies through conversation policies. In Mattern, F. and Kotz, D., editors, *2nd IEEE International Symposium on Agent Systems and Application (ASA/MA-00)*, Zürich, Switzerland. Springer Verlag.
- Plasmeijer, R. and van Eekelen, M. (1998). *Concurrent Clean Language Report*. HILT - High Level Software Tools B.V. and University of Nijmegen. <http://www.cs.kun.nl/~clean>.
- Plotkin, G. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus university, Computer Science Department, Denmark.