Distributed Software Agents and Communication in the Situation Calculus[1]

Daniel Marcu, Yves Lespérance, Hector J. Levesque[2], Fangzhen Lin, Raymond Reiter[2], and Richard B. Scherl[3]
Department of Computer Science
University of Toronto
Toronto, ON, M5S 1A4 Canada
{marcu,lesperan,hector,fl,reiter,scherl}@ai.toronto.edu

Keywords: Multi-agent communication, Agent-oriented programming, Reasoning about actions, Situation calculus

Abstract

This paper describes a theoretical framework that can be used for the implementation of software agents that reason, act, and perceive in incompletely known, dynamic software environments. Special attention is given to describing the way inter-agent communication is treated in this framework. We assume that the user provides a specification of each agent's primitive actions, and the relevant information about the software environment in which the agent is supposed to operate. On the basis of the primitive actions, the user can specify complex behaviors in a programming language, GOLOG, whose execution reflects the way the environment and the knowledge of the agent change from state to state. A simple domain consisting of two elevator controllers that negotiate the serving of a floor is used to present incrementally the details of our approach.

1 Introduction

A number of formal languages have been proposed recently for modeling inter-agent communication [Finin *et al.*, 1993, Shoham, 1993, Sidner, 1994], and a number of implemented agents have already started to reveal the research issues one should address when dealing with real-world systems [Etzioni *et al.*, 1993, Kautz *et al.*, 1994]. Theoretical work [Shoham, 1993, Sidner, 1994] emphasizes the role of knowledge or mental state of the agents that participate in a cooperative process, while practical work [Etzioni *et al.*, 1993, Kautz *et al.*, 1994] emphasizes the importance of agents' trustworthiness, robustness, and autonomy. We are interested in an approach that is able to address successfully both classes of problems.

This paper presents a solution for the programming of intelligent agents that is based on an advanced logical theory of reasoning about actions, knowledge, and perception. The programmer is supposed to provide a logical description of the primitive actions that each agent can perform. This includes both the precondition axioms, i.e., the conditions that must be satisfied in order to make an action possible, and the effect axioms, i.e., the axioms that

[1]This research received financial support from the Information Technology Research Center (Ontario, Canada), the Institute for Robotics and Intelligent Systems (Canada), and the Natural Sciences and Engineering Research Council (Canada).

[2]Hector J. Levesque and Raymond Reiter are fellows of the Canadian Institute of Advanced Research.

[3]Current address: Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102 USA.

formalize the way the environment or the knowledge of an agent changes after performing a given action. A completeness assumptions allows one to automatically derive a set of *successor state axioms* [Reiter, 1991, Scherl and Levesque, 1993] that solve the frame problem [McCarthy and Hayes, 1969] for primitive actions. Once primitive actions have been described, the programmer can specify the behavior of a given agent using a programming language that handles complex control structures of the kind found in classical procedural languages such as conditionals, loops, and procedures and that inherits the solution to the frame problem exhibited by primitive actions.

In multi-agent environments, an essential issue is the characterization of the communication process among agents. In this paper we investigate the way in which communication can be formalized in the case where a distributed control is assumed. The approach to modeling communication that we propose here allows one to implement agents that communicate, cooperate, and solve problems even when they are located on different machines.

We use a very simple domain to delineate our approach to communication that has the following components: an external agent randomly generates PUSH($button$) exogenous requests. The domain contains two more agents, each controlling an elevator:

The ActiveElevatorBot constantly monitors the world "to see" if some new button has been pressed. When this is the case, it initiates a communication process with the PassiveElevatorBot, in order to determine its position. Knowing its own position, and the position of the other elevator, the ActiveElevatorBot can now decide on the basis of a pre-specified algorithm, which elevator is going to serve the requested floor. If it is the ActiveElevator who is going to serve a given floor, it does so. Otherwise, it will send a command to the PassiveElevatorBot to do it.

The PassiveElevatorBot answers requests from the ActiveElevatorBot about its current position. It also fires a SERVE($floor$) procedure, if it receives a command to do so, from the ActiveElevator.

A diagram representing the two elevators and the communication channels is given in figure 1.

We use the simple elevator domain to give a brief introduction of the logical foundations of our approach, and the programming language, GOLOG, derived from them. Special emphasis is given to the treatment of communication. We conclude with implementation remarks.

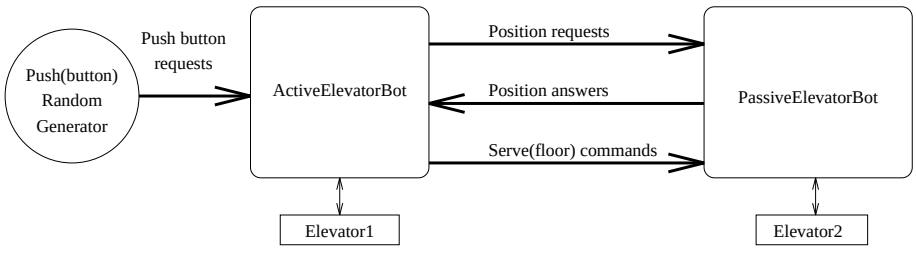

Figure 1: Two elevators negotiate about which one serves a floor.

2 Logical foundations

The variant of the situation calculus that we use [Reiter, 1991] is a first-order language for representing dynamically changing worlds in which all of the changes are the result of named *actions* performed by some agent. Throughout this paper, we use the convention that free variables are universally quantified, that variables start with lower case letters and constants with upper case letters. In the situation calculus, terms are used to represent states of the world, i.e., *situations*. If α is an action and s a situation, the result of performing α in s is represented by $do(\alpha, s)$. The constant S_0 is used to denote the initial situation. Relations whose truth values vary from situation to situation, called *fluents*, are denoted by predicate symbols taking a situation term as the last argument. For example, $\text{ON}(n, s)$ means that the button corresponding to floor n is "on" in situation s. A possible set of actions for modeling the behavior of the ActiveElevatorBot, for example, is given below:

Actions

$\text{TURNOFF}(n)$	turn off button n
OPEN	open the door
CLOSE	close the door
$\text{UP}(n)$	go up to the n^{th} floor
$\text{DOWN}(n)$	go down to the n^{th} floor
$\text{PUSH}(n)$	push button n

States of the ActiveElevatorBot are modeled using the following fluents:

Fluents

$\text{ON}(n, s)$	button n is "on" in state s
$\text{FLOOR}(n, s)$	the ActiveElevator is at floor n
$\text{NEXT\_FLOOR}(n, s)$	the next floor to be served by the ActiveElevatorBot is n
$\text{FLOOR\_PASSIVEBOT}(n, s)$	the PassiveElevator is at floor n
$\text{NEGOTIATE}(n, s)$	the ElevatorBots negotiate about which one will serve floor n

It is assumed that the axiomatizer has provided for each agent and each action an *action precondition axiom* specifying the conditions under which that action can be performed. For example, the following axiom says that an elevator can go UP to a floor n whenever the elevator is located at a floor m, below n.

$$Poss(\text{UP}(n), s) \equiv \text{FLOOR}(m, s) \wedge m \leq n \tag{1}$$

Furthermore, it is assumed that the axiomatizer has provided for each fluent F, two *general effect axioms*. For example, sentence 2 is a positive effect axiom for the fluent ON, while sentence 3 is a negative effect axiom.

$$Poss(a, s) \wedge a = \text{PUSH}(n) \rightarrow \text{ON}(n, do(a, s)) \tag{2}$$

$$Poss(a, s) \wedge a = \text{TURNOFF}(n) \rightarrow \neg\text{ON}(n, do(a, s)) \tag{3}$$

Effect axioms provide the "causal laws" for the domain of application but they are not sufficient if one wants to reason about change. It is usually necessary to add frame axioms that specify when fluents are not changed by actions. The frame problem [McCarthy and Hayes,

1969] arises because the number of these frame axioms is of the order of the product of the number of fluents and the number of actions. Our approach incorporates a treatment of the frame problem due to Reiter [1991] who extends previous proposals by Pednault [1989], Schubert [1990] and Haas [1987]. The basic idea behind this is to collect all effects axioms about a given fluent and assume that they specify all the ways the value of the fluent may change. A syntactic transformation can then be used to obtain a *successor state axiom* for each fluent [Reiter, 1991]. The following successor state axiom specifies that a button is "on" if it has just been pressed or if it was "on" before and it was not turned off.

$$Poss(a, s) \rightarrow [\text{ON}(n, do(a, s)) \equiv a = \text{PUSH}(n) \vee \text{ON}(n, s) \wedge a \neq \text{TURN\_OFF}(n)] \quad (4)$$

3 Communication in the language of situation calculus

3.1 Communication channels

One of the most sensitive issues in formalizing the behavior of groups of agents, in the case where distributed control is assumed, occurs when one wants to consider the effects on the world, or the effects on an agent of a communication action. If one axiomatizes a domain in which there is only one controller for all agents, the effects of a SEND_MESSAGE action, for example, will be reflected by a change in the status of the queue of messages that have been received by another agent. This is possible because the controller has access to the knowledge of all the agents involved in a communication process. Unfortunately, this methodology is not applicable when agents are located on different machines and cooperate over Internet connections for example. We overcome this problem by treating communication channels as special entities that are characterized by pseudo-fluents of the type given below:

$$\text{CHANNEL}(channel\_id, sender, recipient, content, s) \quad (5)$$

Formula 5 says that in state s, channel *channel_id* contains the message *content* that was sent by a *sender* for a *recipient*. We use the term *pseudo-fluent* because physically, a channel can be a permanent link such as a TCP/IP protocol or a temporary link such as an email message. The evolution in time of such an entity is controlled by two actions: a sender agent can make a message of the form given in 5 available for a recipient agent by sending a message, when a connection exists or can be instantiated between two agents; a recipient can recognize it by sensing it. A communication succeeds only when such a link is created and the full message is sent between the sender and the recipient. Note that a communication process does not modify any fluent at the recipient level: it only assures the sender that a sensing action performed by the recipient with respect to a communication channel will recognize the message. This approach to communication is consistent with sensing actions in physical environments. If an obstacle is put in front of a robot, for example, this obstacle does not explicitly modify the state of knowledge of that robot, but it can modify the result of some sensing action that the robot performs. The robot may sense the obstacle or not. If it doesn't, the obstacle has no effect on its behavior. If it does, the effect is that the robot will *know* about that obstacle. If the considered robot is able to sense every obstacle one adds, we can say that the action of adding an obstacle has a "direct" effect on the knowledge of the robot. Exactly the same thing happens in the software domain. Sending a message is possible only if the success of the sending is assured. We can program the recipient to monitor constantly

the incoming messages. The incoming messages will have a "direct" effect on the recipient's state of knowledge. The only difference between a software agent and a robot is that for the former, we prefer sensing to have a destructive effect. Once a message is read, it is no longer needed.

Each agent has a communication channel associated with it that is recognized by the other agents involved in solving a given task. In this sense a channel is a shared object. Sender-agents add messages to a channel and recipient-agents read and destroy these messages. This treatment of communication ensures the expected behavior even when the communication takes place over Internet links.

To simplify things, we will make the following assumptions: the communication channels among different agents (bots) are first-class entities. They have unique names, and they are read/written by the agents that they link. We assume that the communication channels are perfect, i.e., no messages are lost[4]. We assume that perfect channels can be created at any moment. A more complex behavior can be modeled if one removes this assumption: actions such as "establish-connection" will have to be considered.

For our running example, in accomplishing the task of serving a floor, the following types of messages will be exchanged among different agents:

- a $push\_button(floor)$ command will be sent by an external process to the ActiveElevator, requesting a new $floor$ to be served;
- an $ask\_where\_is\_passive\_elevator$ request can be sent by the ActiveElevatorBot to the PassiveElevatorBot, to ask for its position;
- an $answer(position)$ will be sent by the PassiveElevatorBot in response to an $ask\_where\_is\_passive\_elevator$ message;
- a $serve\_passive\_elevator(floor)$ command will be sent by the ActiveElevatorBot to the PassiveElevatorBot to serve a given floor.

For our domain, we consider a pseudo-fluent associated with the input channel that is afferent to the PassiveElevatorBot (PEB), and one pseudo-fluent associated with the communication channel afferent to the ActiveElevatorBot (AEB). Possible instances of these pseudo-fluents are given below, where Rg is the name of the push button random-generator agent.

$\textsc{channel}(AEB, Rg, ActiveElevatorBot, push\_button(5), s)$
$\textsc{channel}(AEB, PassiveElevatorBot, ActiveElevatorBot, answer(3), s)$
$\textsc{channel}(PEB, ActiveElevatorBot, PassiveElevatorBot, ask\_where\_is\_passive\_elevator, s)$
$\textsc{channel}(PEB, ActiveElevatorBot, PassiveElevatorBot, serve\_passive\_elevator(5), s)$

3.2 Communication specific actions and knowledge

In order to make communication possible, the agents should be able to read and write their afferent communication channels. This means that external commands of the following type should be considered:

[4]These constraints can be relaxed at the price of introducing extra fluents and choosing a more complex communication protocol.

External actions

READ_EXTERNAL_REQUEST($channel\_id, sender, recipient, content$) read the message *content* from channel *channel_id* that was sent by *sender* for the *recipient*

SEND_EXTERNAL_COMMAND($channel\_id, sender, recipient, content$) send a message *content* on channel *channel_id* to agent *recipient*

Note that the desired effect of a SEND_EXTERNAL_COMMAND action is the modification of the specified communication channel. In order to make communication successful, a READ_EXTERNAL_REQUEST action should not only destroy the message, but also it should modify the mental state of the agent that performs the reading operation, or the sensing. This kind of communicative action can be accommodated in an extension of the language described so far that handles perception and knowledge [Scherl and Levesque, 1993].

A binary relation $K(s', s)$ is introduced in order to adapt the standard possible-world model of knowledge to the situation calculus, as first done by Moore [1980]. The binary relation $K(s', s)$, read as "s' is accessible from s", is used to define the fluent **Knows**(P, S), read as P is known in situation s. For example, the following formula, written for the ActiveElevatorBot, says that this agent knows that the PassiveElevator is on floor n.

$$\textbf{Knows}(\text{FLOOR\_PASSIVEBOT}(n, s)) \stackrel{\text{def}}{=} \forall s'(K(s', s) \rightarrow \text{FLOOR\_PASSIVEBOT}(n, s')) \quad (6)$$

Consider now a knowledge producing action READ_EXTERNAL_REQUEST that determines whether or not some fluent P is true. At $do(\text{READ\_EXTERNAL\_REQUEST}_P, s)$ as far as the agent knows, he can be in any of the worlds $do(\text{READ\_EXTERNAL\_REQUEST}_P, s')$ for all s' such that $K(s', s)$ and $\text{P}(s) \equiv \text{P}(s')$. In our domain, for the ActiveElevatorBot, a successor state axiom for the knowledge fluent K, will be:

$$\begin{array}{l} Poss(a, s) \rightarrow [K(s'', do(a, s)) \equiv \\ \quad \exists s' \, (K(s', s) \wedge s'' = do(a, s)) \wedge \\ \quad\quad (a = \text{READ\_EXTERNAL\_REQUEST}(AEB, PassiveElevatorBot, \\ \qquad\qquad\qquad\qquad\qquad\qquad ActiveElevatorBot, answer(n)) \rightarrow \\ \quad\quad\quad \text{FLOOR\_PASSIVEBOT}(n, s) \equiv \text{FLOOR\_PASSIVEBOT}(n, s')) \wedge \\ \quad\quad (a = \text{READ\_EXTERNAL\_REQUEST}(AEB, Rg, ActiveElevatorBot, push(m)) \rightarrow \\ \quad\quad\quad \text{ON}(m, s) \equiv \text{ON}(m, s'))] \end{array} \quad (7)$$

Formula 7 says that after reading a message that was sent by the *PassiveElevatorBot* on channel AEB containing an answer n, the ActiveElevatorBot knows that the PassiveElevatorBot is on floor n; and after reading a push button request that was sent by the random generator agent on channel AEB, the ActiveElevatorBot knows that the corresponding button has been pressed.

In the style of formula 4, we can write successor state axioms for channels as well, as they are shared objects among agents. The following axiom specifies that channel AEB contains message *content* sent by *sender* to the *ActiveElevatorBot* if and only if the last communication action with respect to channel AEB was issued by *sender* to *ActiveElevatorBot* and had the message *content*, or if the message was already on the channel and it has not been

read by the *ActiveElevatorBot*.

$$\begin{array}{l} Poss(a,s) \rightarrow [\textsc{Channel}(AEB, sender, ActiveElevatorBot, content, do(a,s)) \equiv \\ \quad a = \textsc{send\_external\_command}(AEB, sender, ActiveElevatorBot, content) \vee \\ \quad \textsc{Channel}(AEB, sender, ActiveElevatorBot, content, s) \wedge \\ \quad a \neq \textsc{read\_external\_request}(AEB, sender, ActiveElevatorBot, content)] \end{array} \tag{8}$$

Using this axiom we can prove, for example, for the ActiveElevatorBot that after the random generator agent, *Rg* sends a $push\_button(n)$ command to the PassiveElevatorBot, and after the ActiveElevatorBot reads that command, it will know that button n is "on":

$$\begin{array}{l} \textbf{Knows}(\textsc{On}(n, do(\textsc{read\_external\_request}(AEB, Rg, ActiveElevatorBot, \\ \quad push\_button(n)), do(\textsc{Send\_external\_command}(AEB, Rg, \\ \quad ActiveElevatorBot, push\_button(n)), s)))) \end{array} \tag{9}$$

4 Golog — a language for complex actions

Golog treats complex actions as abbreviations for expressions in the situation calculus. This macro approach is accomplished by defining a predicate *Do* as in $Do(\delta, s, s')$ where δ is a complex action that takes the agent from state s to state s'. The inductive definition of *Do* follows:

- $Do(a, s, s') \stackrel{\text{def}}{=} Poss(a, s) \wedge s' = do(a, s)$ — simple actions
- $Do([\delta_1; \delta_2], s, s') \stackrel{\text{def}}{=} \exists s''(Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s'))$ — sequence
- $Do([\delta_1 | \delta_2], s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$ — nondeterministic choice of actions
- $Do(\Pi x(\delta), s, s') \stackrel{\text{def}}{=} \exists x\, Do(\delta, s, s')$ — nondeterministic choice of action parameters

Other cases handle tests ($\phi?$), conditionals (**if** ϕ **then** δ_1 **else** δ_2), loops (**while** ϕ **do** δ and **for** $x : \phi(x)$ **do** δ), and recursive procedures.

This set of complex action expressions forms a programming language that we call GOLOG (alGOl in LOGic), which is suitable for high-level programming of robots [Lesperance *et al.*, 1994] and software agents, as well as discrete event simulation. GOLOG differs from ordinary programming languages in that: it has a situation calculus semantics; its complex actions decompose into primitives that in most cases refer to actions in the external world; executing a complex action may involve arbitrary first-order reasoning (e.g., executing **while** $\exists n$ $\textsc{On}(n)$ **do** Serve(n), requires inferring whether $\exists n$ $\textsc{On}(n)$ in the current state) — the interpreter for this programming language is a theorem prover.

GOLOG is designed as a compromise between classical planning and detailed programming. It is a high-level nondeterministic language in which one can express schematic plans. These schemas give advice to an agent about how to achieve certain effects, without necessarily specifying in detail how to perform the corresponding action. The details are to be figured out by the theorem prover when the program is executed.

For the ActiveElevatorBot the following procedures can be recursively defined. In order to serve a floor, the elevator will go to that floor, it will turn off the button, and it will open and close the door.

proc SERVE(n)
GO_FLOOR(n); TURNOFF(n); OPEN; CLOSE;
end.

In order to go to a given floor, the elevator will determine the direction in which it should move, and it will act accordingly.

proc GO_FLOOR(n)
if FLOOR(m) $\wedge$ $m < n$ **then** UP(n)
else DOWN(n)
end.

The control procedure for the ActiveElevator, called CONTROL, is an infinite loop in which the ActiveElevatorBot monitors $push(button)$ requests, finds where the PassiveElevator is, calls a decision procedure, I_SERVE?, to determine which elevator is going to solve the request, and solves this request or sends a command to the other controller to do it. Similar control procedures are provided for the PassiveElevatorBot as well.

proc CONTROL
while TRUE
READ_EXTERNAL_REQUEST($AEB, Rg, ActiveElevatorBot, push\_button(floor)$);
SEND_EXTERNAL_COMMAND($PEB, ActiveElevatorBot, PassiveElevatorBot,$
$ask\_where\_is\_passive\_elevator$);
READ_EXTERNAL_REQUEST($AEB, PassiveElevatorBot, ActiveElevatorBot,$
$answer(floor2)$);
if I_SERVE? **then** SERVE($floor$)
else SEND_EXTERNAL_COMMAND($PEB, ActiveElevatorBot,$
$PassiveElevatorBot, serve\_passive\_elevator(floor)$))
end
end.

5 Implementation

We have used Quintus Prolog to implement the GOLOG interpreter. Communication channels are implemented through TCP/IP protocols. Each of the software agents described in this paper has its own TCP/IP address and knows the addresses of the other agents in the domain. Quintus Prolog constraints allow us to create only one communication channel for each agent. Under these circumstances, it is possible that while the elevatorbots negotiate the serving of a given floor, some new PUSH($button$) requests occur on the same channel. Special subroutines have been developed for handling this type of message interleaving. TCP/IP protocols ensure the user that all messages that are sent arrive at their destination in a chronological order; therefore, it is acceptable to treat TCP/IP connections as perfect channels. For the moment, the epistemic operators are not used: instead, we assume that the robot knows everything about the world, i.e, the facts derivable from the successor state axioms.

We are currently building more complex softbots for assisting users in performing tedious tasks, such as meeting scheduling for example. We are also developing a theory of communication among software agents, using an extension of our theory of complex actions that

handles concurrency, and its programming language counterpart, CONGOLOG [Lespérance *et al.*, 1995].

6 Conclusion

The approach described here for programming software agents is concerned with a theory and an implementation of software agents that reason, act, and perceive in dynamically changing software environments. The theory we have described allows agents to exhibit higher level cognitive functions that involve reasoning and planning, inquiring about the cognitive states of other agents, and collaboration. In short, we have presented a uniform theoretical and implementation framework that integrates reasoning, perception, and action. From this perspective, we argue that our approach meets the requirements specified by previous work on agent-programming techniques [Shoham, 1993]. We have paid special attention to explaining a way in which communication can be treated in this framework, that allows agents to cooperate even though they are not located on the same machine. The theory described here has been fully implemented in GOLOG. Because each agent is described by its own set of actions and programs, and because all communication issues fall under the scope of communicative actions, our agents deal responsibly with the information they manage. Concerns raised by Kautz [1994] about information privacy, for example, are appropriately dealt with: each agent can be programmed such that it answers only a given subset of queries. Moreover, note that other agents may influence only the state of knowledge of a given agent; the behavior of each agent is fully predictable on the basis of its own programs.

References

[Etzioni *et al.*, 1993] O. Etzioni, N. Lesh, and R. Segal. Building softbots for Unix (preliminary report). Technical Report 93-09-01, University of Washington, 1993. Available via anonymous FTP from pub/etzioni/softbots at cs.washington.edu.

[Finin *et al.*, 1993] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, and C. Beck. Specification of the KQLM agent-communication language. The DARPA Knowledge Sharing Initiative, June 1993.

[Haas, 1987] A.R. Haas. The case for domain-specific frame axioms. In F.M. Brown, editor, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pages 343–348, Lawrence, KA, April 1987. Morgan Kaufmann Publishing.

[Kautz *et al.*, 1994] H. Kautz, B. Selman, M. Coen, S. Ketchpel, and C. Ramming. An experiment in the design of software agents. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume one, pages 438–443, Seattle, Washington, July 31 - August 4 1994.

[Lesperance *et al.*, 1994] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A logical approach to high-level programming — a progress report. In *Control of the Physical World by Intelligent Systems, Working Notes of the 1994 AAAI Fall Symposium*, pages 109–119, New Orleans, LA, November 1994.

[Lespérance *et al.*, 1995] Y. Lespérance, H.J. Levesque, F. Lin, D. Marcu, R. Reiter, and R.B. Scherl. Fondements d'une approche logique à la programmation d'agents. In *Actes des Troisièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents*, Chambéry-St. Badolph, France, March 1995. To appear.

[McCarthy and Hayes, 1969] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Moore, 1980] R.C. Moore. Reasoning about knowledge and action. Technical Report 191, SRI International, AI Center, Menlo Park, CA, 1980.

[Pednault, 1989] E.P.D. Pednault. ADL: Exploring the middle ground between strips and the situation calculus. In R.J. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, ON, May 1989. Morgan Kaufmann Publishing.

[Reiter, 1991] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Scherl and Levesque, 1993] R.B. Scherl and H.J. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, July 1993. AAAI Press/The MIT Press.

[Schubert, 1990] L.K. Schubert. Monotonic solution to the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Loui, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, MA, 1990.

[Shoham, 1993] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[Sidner, 1994] C.L. Sidner. An artificial discourse language for collaborative negotiation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume one, pages 814–819, Seattle, Washington, July 31 - August 4 1994.