To appear in M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, pp. 331–346, Springer-Verlag, Lecture Notes in Artificial Intelligence, 1996.

Foundations of a Logical Approach to Agent Programming*

Yves Lespérance[a], Hector J. Levesque[b], Fangzhen Lin[b],
Daniel Marcu[b], Raymond Reiter[b], and Richard B. Scherl[c]

[a]Department of Computer Science, Glendon College, York University,
2275 Bayview Ave., Toronto, ON, Canada M4N 3M6
lesperan@yorku.ca

[b]Department of Computer Science, University of Toronto
Toronto, ON, Canada M5S 1A4
{hector,fl,marcu,reiter}@ai.toronto.edu

[c]Department of Computer and Information Science,
New Jersey Institute of Technology,
University Heights, Newark, NJ 07102 USA
scherl@vienna.njit.edu

Abstract

This paper describes a novel approach to high-level agent programming based on a highly developed logical theory of action. The user provides a specification of the agents' basic actions (preconditions and effects) as well as of relevant aspects of the environment, in an extended version of the situation calculus. He can then specify behaviors for the agents in terms of these actions in a programming language where one can refer to conditions in effect in the environment. When an implementation of the basic actions is provided, the programs can be executed in a real environment; otherwise, a simulated execution is still possible. The interpreter automatically maintains the world model required to execute programs based on the specification. The theoretical framework includes a solution to the frame problem, allows agents to have incomplete knowledge of their environment, and handles perceptual actions. The theory can also be used to prove programs correct. A simple meeting scheduling application is used to present the approach. Ongoing work on implementing the approach and handling outstanding problems is also described.

1 Introduction

The notion of computational *agents* has become very fashionable lately [18, 26]. Building such agents seems to be a good way of congenially providing services to users in networked computer systems. Typical applications are information retrieval over the internet, automation of common user activities (e.g. scheduling meetings), smart user interfaces, integration of heterogenous software tools, intelligent robotics, business and industrial process modeling, etc. The term "agent" is used in many different ways, so let us try to clarify

*A version of this paper appeared in French in Actes des Troisièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents, Chambéry-St-Badolph, France, March, 1995. This research received financial support from the Information Technology Research Center (Ontario, Canada), the Institute for Robotics and Intelligent Systems (Canada), and the Natural Science and Engineering Research Council (Canada). Levesque and Reiter are fellows of the Canadian Institute for Advanced Research. Many of our papers are available at http://www.cs.toronto.edu/∼cogrobo/.

what we mean by it. We take an agent to be any active entity whose behavior is usefully described through mental notions such as knowledge, goals, abilities, commitments, etc. (This is pretty much the standard usage in artificial intelligence, in contrast to the common view of agents as scripts that can execute on remote machines). Moreover, we will focus on the approach to building applications that involves designing a system as a collection of interacting agents.

Agent programming [24] can be viewed as a generalization of object-oriented programming. But the notion of an agent is much more complex than that of an object. Because of this, it is crucial that tools for modeling and designing agents be based on solid theoretical foundations. For some time now, we have been working on a logical theory of agency and on programming tools based on it. The theoretical framework includes a formalization of action that incorporates a solution to the frame problem, thus relieving the designer from having to specify what aspects of the world don't change when an action is performed. The framework also includes a model of what agents know and how they can acquire information by doing knowledge-producing actions, such as sensing the environment with vision or sonar, or interacting with users or other agents. And finally, we have an account of complex actions and processes that inherits the solution to the frame problem for simple actions. The framework also has attractive computational properties.

The set of complex action expressions defined can be viewed as a programming language for agents. Given a specification of the agents' basic actions (preconditions and effects), the designer can specify complex behaviors for the agents in this programming language. The behavior specification can be at a very high-level, as the language allows references to conditions in effect in the environment. When an implementation of the basic actions is provided, the programs can be executed in a real environment; otherwise, a simulated execution is still possible. The interpreter automatically maintains the world model required to execute programs based on the specification.

Most of our work so far on the theory and implementation of agents has been concerned with single agents, especially for robotics applications. In this paper, we extend our framework to deal with multi-agent systems. As will become clear, the treatment proposed is preliminary and we identify various problems that need to be solved before we have a completely satisfactory theoretical account and implementation.

The approach will be presented with the help of an example — a multi-agent system that helps users schedule meetings. Each user has a "schedule manager" agent that knows something about his schedule. When someone wants to organize a meeting, he creates a new "meeting scheduler" agent who contacts the participants' schedule managers and tries to set up the meeting. The example is a very simplified version of such a system. To be truly useful, a schedule manager agent should know the user's preferences about meeting times, when and how it should interact with the user in response to requests from meeting scheduler agents, perhaps the hierarchical structure of the office, etc. Meeting scheduler agents should have robust scheduling and negotiating strategies. Our formalization of the application includes a simple generic agent communication module that can be used for other applications. Each agent has a message queue and abstract communication acts are defined (e.g., $\text{INFORM}(agt, \phi)$, $\text{REQUEST}(agt, \delta)$, $\text{QUERYWHETHER}(agt, \phi)$, etc.).

In the next section, we outline our theory of action. Then, we present our account of complex actions and explain how it can be viewed as an agent programming language. Section 4 develops a set of simple tools for agent communication and section 5 completes our specification of the meeting scheduling application. In the following sections, we discuss various architectural issues that arise in implementing our framework, describe the status of the implementation, and sketch what experimental applications have been implemented. We conclude by summarizing the main features of our approach and discussing the problems that remain.

2 A Theory of Action

Our approach is based on an extended version of the situation calculus [14], a predicate calculus dialect for representing dynamically changing worlds. The world is taken to be in a certain state (or situation).

That state can only change as a result of an agent performing an action. The term $do(agt, act, s)$ represents the state that results from agent agt's performance of action act in state s. For example, the formula $\mathrm{ON}(\mathrm{B}_1, \mathrm{B}_2, do(\mathrm{AGT}, \mathrm{PUTON}(\mathrm{B}_1, \mathrm{B}_2), s))$ could mean that B_1 is on B_2 in the state that results from AGT's performance of $\mathrm{PUTON}(\mathrm{B}_1, \mathrm{B}_2)$ in state s. Predicates and function symbols whose value may change from state to state (and whose last argument is a state) are called *fluents*.

An action is specified by first stating the conditions under which it can be performed by means of a *precondition axiom*. For example,

$$\begin{array}{l} Poss(agt, \mathrm{ADDTOSCHED}(user, period, activity, organizer), s) \equiv \\ agt = \mathrm{SCHEDULEMANAGER}(user) \wedge \\ \neg\exists activity', organizer'\ \mathrm{SCHEDULE}(user, period, activity', organizer', s) \end{array} \tag{1}$$

means that it is possible for an agent to add an activity to a user's schedule iff it is the user's schedule manager and nothing is scheduled for that period. Then, one specifies how the action affects the world's state with *effect axioms*, for example:

$$\begin{array}{l} Poss(agt, \mathrm{ADDTOSCHED}(user, period, activity, organizer), s) \supset \\ \mathrm{SCHEDULE}(user, period, activity, organizer, \\ \qquad do(agt, \mathrm{ADDTOSCHED}(user, period, activity, organizer), s)) \end{array} \tag{2}$$

The above axioms are not sufficient if one wants to reason about change. It is usually necessary to add frame axioms that specify when fluents remain unchanged by actions. The frame problem [14] 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 [17] (who extends previous proposals by Pednault [16], Schubert [22] and Haas [5]). The basic idea behind this is to collect all effect 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 applied to obtain a *successor state axiom* for the fluent, for example:

$$\begin{array}{l} Poss(agt, act, s) \supset \\ [\mathrm{SCHEDULE}(user, period, activ, org, do(agt, act, s)) \equiv \\ \ act = \mathrm{ADDTOSCHED}(user, period, activ, org) \vee \\ \ \mathrm{SCHEDULE}(user, period, activ, org, s) \wedge \\ \ act \neq \mathrm{RMVFROMSCHED}(user, period)]. \end{array} \tag{3}$$

This says that an activity is on a user's schedule following the performance of action act by agt in state s iff either the action is to add the activity to the user's schedule, or the activity was already on the user's schedule in s and the action is not that of removing it from the user's schedule. This treatment avoids the proliferation of axioms, as it only requires a single successor state axiom per fluent and a single precondition axiom per action.[1]

Scherl and Levesque [21] have generalized this account to handle knowledge-producing actions. Such actions affect the mental state of the agent rather than the state of the external world. For example, after performing the action SENSEMSG, an agent knows whether his message queue is empty, and when it is not, also knows what the first message in the queue is:

$$\begin{array}{l} Poss(agt, \mathrm{SENSEMSG}, s) \supset \\ \quad \mathbf{KWhether}(agt, \mathrm{EMPTY}(\mathrm{MSGQ}(agt)), do(agt, \mathrm{SENSEMSG}, s)) \wedge \\ \quad [\neg\mathrm{EMPTY}(\mathrm{MSGQ}(agt, s)) \supset \\ \quad \exists m \mathbf{Know}(agt, \mathrm{FIRST}(\mathrm{MSGQ}(agt)) = m, do(agt, \mathrm{SENSEMSG}, s))] \end{array}$$

[1]This discussion ignores the ramification and qualification problems; a treatment compatible with our approach has been proposed by Lin and Reiter [11].

Know(agt, ϕ, s) is an abbreviation which is defined below. **KWhether**(agt, ϕ, s) is also an abbreviation that stands for **Know**$(agt, \phi, s) \vee$ **Know**$(agt, \neg\phi, s)$. We represent knowledge by adapting the possible world model to the situation calculus (as first done by Moore [15]). $K(agt, s', s)$ represents the fact that in state s, the agent agt thinks the state of the world could be s'. **Know**(agt, ϕ, s) is an abbreviation for the formula $\forall s'(K(agt, s', s) \supset \phi(s'))$. For clarity, we sometimes use the pseudo-variable now to represent the state bound by the enclosing **Know**; so **Know**$(agt, \phi(\mathsf{now}), s)$ stands for $\forall s'(K(agt, s', s) \supset \phi(s'))$.

Scherl and Levesque show how to obtain a successor state axiom for the fluent K in the single agent case. In multi-agent domains, the problem is more complex since an agent need not be aware of the actions of other agents (exogenous actions). For our meeting scheduling application, the successor state axiom appearing below is sufficient. Note that we write $s < s'$ iff s' is the result of doing some sequence of actions in s, where the actions are possible in the situation where they are done; $s \leq s'$ stands for $s < s' \vee s = s'$.

$$\begin{array}{l}
Poss(actor, act, s) \supset \\
[K(knower, s'', do(actor, act, s)) \equiv \\
\ \exists s'(K(knower, s', s) \wedge \\
\quad (actor \neq knower \supset Exo(knower, s', s'')) \wedge \\
\quad (actor = knower \supset \exists s^*(Exo(knower, s', s^*) \wedge \\
\qquad s'' = do(knower, act, s^*) \wedge Poss(knower, act, s^*) \wedge \\
\qquad [act = \textsc{senseMsg} \supset \\
\qquad\quad (\textsc{Empty}(\textsc{msgQ}(knower, s^*)) \equiv \textsc{Empty}(\textsc{msgQ}(knower, s))) \wedge \\
\qquad\quad (\neg\textsc{Empty}(\textsc{msgQ}(knower, s^*)) \supset \\
\qquad\qquad \textsc{first}(\textsc{msgQ}(knower, s^*)) = \textsc{first}(\textsc{msgQ}(knower, s)))]))))] \\
\text{where } Exo(agt, s, s') \overset{def}{=} s \leq s' \wedge \\
\qquad\qquad\qquad \forall s^* \forall agt' \forall act(s < do(agt', act, s^*) \leq s' \supset agt' \neq agt)
\end{array} \tag{4}$$

Let us explain how this works. When an action act is performed by some agent $actor$ other than the $knower$, the specification says that in the resulting situation, $knower$ considers possible any situation that is the result of any number of exogenous actions occurring in a state that used to be considered possible; this means that $knower$ allows for the occurrence of an arbitrary number of exogenous actions, and thus loses any knowledge he may have had about fluents that could be affected by exogenous actions. When act is a non-knowledge-producing action (e.g., $\textsc{addToSched}(u, p, a, o)$) performed by $knower$, the specification states that in the resulting situation, $knower$ considers possible any situation that is the result of his doing act preceded by any number of exogenous actions occurring in a state that used to be considered possible; thus, $knower$ acquires the knowledge that he has just performed act, but loses any knowledge he may have had about fluents that could be affected by exogenous actions and are not reset by act. Finally, when act is a knowledge-producing action (i.e., $\textsc{senseMsg}$) performed by $knower$, we get the same as above, plus the fact that the agent acquires knowledge of the values of the fluents associated with the sensing action, in this case whether his message queue is empty and if not what the first message is.[2] This can be extended to an arbitrary number of knowledge-producing actions in a straightforward way.

In general, allowing for the occurrence of an arbitrary number of exogenous actions at every step as we do here would probably leave agents with too little knowledge. But our meeting scheduling domain is neatly partitioned: a user's schedule can only be updated by that user's schedule manager. Thus, schedule manager

[2]To see how we get this, note for instance that we require that $\textsc{Empty}(\textsc{msgQ}(knower, s^*)) \equiv \textsc{Empty}(\textsc{msgQ}(knower, s))$. Thus, $\textsc{Empty}(\textsc{msgQ}(knower))$ will have the same truth value in all s^* such that $K(knower, do(knower, \textsc{senseMsg}, s^*), do(knower, \textsc{senseMsg}, s))$. Observe that for any situation s, $\textsc{Empty}(\textsc{msgQ}(knower))$ is true at $do(knower, \textsc{senseMsg}, s)$ iff it is true at s. Therefore, $\textsc{Empty}(\textsc{msgQ}(knower))$ has the same truth value in all worlds s'' such that $K(knower, s'', do(knower, \textsc{senseMsg}, s))$, and so **KWhether**$(knower, \textsc{Empty}(\textsc{msgQ}(knower, \mathsf{now})), do(knower, \textsc{senseMsg}, s))$ holds.

agents always know what their user's schedule (as recorded) is. In other circumstances, it may be appropriate to assume that actions are all "public", that is, that all agents are aware of every action that occurs; this case is similar to the single agent case and is axiomatized in a similar way. In other cases, it seems preferable for agents to assume that no exogenous actions occur and to revise their beliefs when an inconsistency is discovered; a formalization of this approach is in progress.

3 Complex Actions and CONGOLOG

Actions in the situation calculus are primitive and determinate. Any approach to agent programming requires complex actions. Moreover, since we want to deal with multi-agent systems, we also need concurrent processes.

Our approach to complex actions/processes is to introduce them into the formalism as abbreviations (macros). This yields a much simpler account than any alternative and complex actions automatically inherit the solution to the frame problem. We will thus define an abbreviation $Do(\delta, s, s')$, where δ is a complex action/process expression, meaning that the performance of δ in state s leads to the (not necessarily unique) state s'. It is relatively straightforward to provide an inductive definition of $Do(\delta, s, s')$ when δ does not involve concurrent actions [9, 6]; but this simple approach will not work with concurrency.

Here, we will develop a different account that handles concurrent action by interleaving. We will say that the concurrent execution of δ_1 and δ_2, written $(\delta_1 \,||\, \delta_2)$, occurs between s and s' iff both δ_1 and δ_2 occur between s and s', interleaved in some fashion. Thus, we need to capture the idea of a program δ executing between s and s', but allowing for other actions in the interval as well. Do is defined inductively on the structure of its first argument as follows:

- *Primitive actions:*

$$Do(\mathbf{by}(agt, act), s, s') \overset{def}{=} \exists s^*(s < do(agt, act, s^*) \leq s').$$

- *Test actions:*

$$Do(\phi?, s, s') \overset{def}{=} \exists s^*(s \leq s^* \leq s' \wedge \phi(s^*)).$$

- *Sequences:*

$$Do([\delta_1; \delta_2], s, s') \overset{def}{=} \exists s^*(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s')).$$

- *Concurrent actions:*

$$Do((\delta_1 \,||\, \delta_2), s, s') \overset{def}{=} Do(\delta_1, s, s') \wedge Do(\delta_2, s, s').$$

- *Nondeterministic choice of action:*

$$Do((\delta_1 \mid \delta_2), s, s') \overset{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$

- *Nondeterministic choice of argument:*

$$Do(\pi x\ \delta(x), s, s') \overset{def}{=} \exists x\ Do(\delta(x), s, s').$$

- *Nondeterministic iteration:*

$$\begin{aligned}Do(\delta^*, s, s') \overset{def}{=} \forall P\{ & \\ & \forall s_1[P(s_1, s_1)] \quad \wedge \\ & \forall s_1, s_2, s_3[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\ \} \\ & \qquad \supset\ P(s, s').\end{aligned}$$

This definition has a slight problem however: it does not ensure that every action in the interval has been accounted for by some part of the program. In [8], we give a more complex version that solves this problem. This formalization draws somewhat from the standard treatment of concurrency in dynamic logic [4].

Conditionals and while-loops can be defined in terms of the above constructs as follows:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \overset{def}{=} [\phi?;\delta_1]|[\neg\phi?;\delta_2],$$

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \overset{def}{=} [[\phi?;\delta]^*;\neg\phi?].$$

Procedures and recursion can also be handled.

For convenience, we use the form **by**(agt, δ) with arbitrary complex actions δ, and take it to stand for δ with every occurrence of a primitive action expression act replaced by **by**(agt, act). So for example,

$$\begin{array}{l}\textbf{by}(agt, \textbf{while } \phi \textbf{ do } [act_1; act_2] \textbf{ endWhile}) \overset{def}{=} \\ \qquad \textbf{while } \phi \textbf{ do } [\textbf{by}(agt, act_1); \textbf{by}(agt, act_2)] \textbf{ endWhile}.\end{array}$$

We also have a version of Do that handles interrupts and concurrent processes with different priorities. As well, we have been experimenting with various techniques to represent and reason about actions with long durations (e.g., filling a bathtub), where the interleaving model is inappropriate.

This set of complex action expressions forms a programming language suitable for applications in distributed AI, robotics, and simulation. We call the portion without concurrency GOLOG (alGOl in LOGic), and the whole language CONGOLOG (CONcurrent GOLOG). We give some examples in the next two sections.

CONGOLOG differs from ordinary programming languages in many ways. It has a formal semantics based on the situation calculus. Its complex actions decompose into primitives that in most cases refer to actions in the external world (e.g. picking up an object or telling something to another agent, as opposed to actions which merely change machine states, such as an assignment to a register). This supports the specification of agent behaviors at a very high level of abstraction. CONGOLOG programs are evaluated with a theorem prover. The user supplies precondition axioms, one per action, successor state axioms, one per fluent, a specification of the initial state of the world, and a CONGOLOG program specifying the behavior of the agents in the system. To execute a program (roughly), one attempts to prove $Axioms \models \exists s Do(program, S_0, s)$, and if a (constructive) proof is obtained, one extracts a binding for the variable $s = do(agt_n, act_n, \ldots do(agt_2, act_2, do(agt_1, act_1, S_0)) \ldots)$, and then sends the sequence of actions $\langle agt_1, act_1\rangle, \langle agt_2, act_2\rangle, \ldots, \langle agt_n, act_n\rangle$ to the primitive action execution module. The CONGOLOG interpreter automatically maintains a world model for the agents. The approach focuses on high-level programming rather than plan synthesis at run-time. But sketchy plans are allowed; nondeterminism can be used to infer the missing details. For example, the plan **while** $\exists b$ ONTABLE(b) **do** πb REMOVE(b) leaves it to the CONGOLOG interpreter to find a legal sequence of actions that clears the table. For a more thorough discussion of the sequential portion of our language and its implementation, see [9].

One project that is closely related to ours is work on the AGENT-0 programming language [24]. But it is hard to do a systematic comparison between CONGOLOG and AGENT-0 as there are numerous differences. The latter includes a model of commitments and capabilities, and has simple communication acts built-in; its agents all have a generic rule-based architecture; there is also a global clock and all beliefs are about time-stamped propositions. However, there is no automatic maintenance of the agents' beliefs based on a specification of primitive actions as in CONGOLOG and only a few types of complex actions are handled; there also seems to be less emphasis on having a complete formal specification of the system.

Another agent language based on a logic is Concurrent MetateM [3]. Here, each agent's behavior is specified in a subset of temporal logic. The specifications are executed using iterative model generation techniques. A limitation of the approach is that neither the interactions between agents nor their mental states are modeled within the logic. In [25], Wooldridge proposes a richer logical language where an agent's knowledge and choices could be specified; he also sketches how model generation techniques could be used to synthesize automata satisfying the specifications. This follows the situated automata view of Rosenschein and Kaelbling [19], which allows knowledge to be attributed to agents without any commitment to a symbolic architecture.

4 Communication in CONGOLOG

Most multi-agent applications require some kind of agent communication facility. A popular choice is the KQML communication language [2] and its associated tools. However according to Cohen and Levesque [1], the KQML definition has many deficiencies, in particular the lack of a formal semantics. One of our objectives is to show that CONGOLOG is suitable for various implementation tasks, so here we chose to define our own simple communication tools. The specification can be viewed as a generic package that can be included into specific applications. We first specify a set of basic message passing actions; later, some abstract communication actions are defined in terms of the primitives. Each agent agt is taken to have a message queue, whose content in a state s is denoted by the functional fluent $\textsc{msgQ}(agt, s)$. There are three primitive action types that operate on message queues:

- $\textsc{sendMsg}(recipient, msg)$: sending message msg to agent $recipient$, which results in the pair $\langle senderId, msg\rangle$ being tacked onto $recipient$'s queue;
- $\textsc{senseMsg}$: sensing what the sender-message pair at the head of one's queue is; and
- $\textsc{rmvMsg}$: removing the sender-message pair at the head of one's queue.

The preconditions of these actions are as follows: $\textsc{rmvMsg}$ is possible for an agent iff his message queue is not empty:

$$Poss(agt, \textsc{rmvMsg}, s) \equiv \neg\textsc{Empty}(\textsc{msgQ}(agt, s)); \tag{5}$$

$\textsc{sendMsg}(rcpt, msg)$ and $\textsc{senseMsg}$ are always possible (we leave out the formal statements).

The effects of these actions are as described above, which yields the following successor state axiom for the $\textsc{msgQ}$ fluent:

$$\begin{aligned}
&Poss(agt, act, s) \supset \\
&[\textsc{msgQ}(agt', do(agt, act, s)) = q \equiv \\
&\ \exists m(act = \textsc{sendMsg}(agt', m) \wedge q = \textsc{append}(\textsc{msgQ}(agt', s), [\langle agt, m\rangle])) \\
&\ \vee act = \textsc{rmvMsg} \wedge agt = agt' \wedge q = \textsc{rest}(\textsc{msgQ}(agt', s)) \\
&\ \vee q = \textsc{msgQ}(agt', s) \wedge \\
&\qquad \forall m(act \neq \textsc{sendMsg}(agt', m)) \wedge (act \neq \textsc{rmvMsg} \vee agt \neq agt')].
\end{aligned} \tag{6}$$

Given these primitives, we can now define some useful abstract communication actions:

```
proc INFORM($agt, \phi$)
    Know($\phi$)?; SENDMSG($agt$, ⌜INFORM($\phi$)⌝)
end

proc INFORMWHETHER($agt, \phi$)
    INFORM($agt, \phi$) | INFORM($agt, \neg\phi$) | INFORM($agt$, ¬KWhether($\phi$))
end
```

```
proc $\text{REQUEST}(agt, \delta)$
    $\text{SENDMSG}(agt, \ulcorner\text{REQUEST}(\delta)\urcorner)$
end

proc $\text{QUERYWHETHER}(agt, \phi)$
   $\text{REQUEST}(agt, \text{INFORMWHETHER}(\mathsf{self}, \phi))$
end
```

Note that the pseudo-variable self refers to the agent running the program. The above definitions use quotation. There are a number of well known ways of handling this. A formal treatment compatible with our framework is under development.

We can show that the INFORM abstract communication act behaves as one would expect. Let us take $\text{SINCERE}(sender, rcpt, s)$ as meaning that up to state s, every INFORM message sent to $rcpt$ by $sender$ was truthfully sent:

$$\begin{aligned}&\text{SINCERE}(sender, rcpt, s) \overset{def}{=} \\ &\quad \forall s'[do(sender, \text{SENDMSG}(rcpt, \ulcorner\text{INFORM}(\phi)\urcorner), s') \leq s \supset \mathbf{Know}(sender, \phi, s')].\end{aligned}$$

Then, we can show that after an agent sends an $\text{INFORM}(\phi)$ message to someone and the recipient senses his messages, the recipient will know that at some prior time the sender knew that ϕ, provided that the recipient knows that his message queue is empty at the outset and that the sender has been sincere with him over that period:

Proposition

$$\begin{aligned}&\mathbf{Know}(rcpt, \text{EMPTY}(\text{MSGQ}(rcpt), \mathsf{now}), S_0) \wedge \\ &\mathbf{Know}(rcpt, \forall agt'\ \text{SINCERE}(agt', rcpt, \mathsf{now}), \\ &\qquad do(rcpt, \text{SENSEMSG}, do(sender, \text{SENDMSG}(rcpt, \ulcorner\text{INFORM}(\phi)\urcorner), S_0))) \supset \\ &\mathbf{Know}(rcpt, \exists s'[s' \leq \mathsf{now} \wedge \mathbf{Know}(sender, \phi, s')], \\ &\qquad do(rcpt, \text{SENSEMSG}, do(sender, \text{SENDMSG}(rcpt, \ulcorner\text{INFORM}(\phi)\urcorner), S_0)))\end{aligned}$$

At this point, it is impossible to prove interesting general results about REQUEST, because we lack a formalization of goals and intentions. We plan to remedy this in future work (see [23] for some preliminary results). In the next section, we show that the simple communication tools specified above are sufficient for developing interesting applications. We are in the process of refining the specification and extending it to handle other types of communicative acts. Eventually, we would like to have a comprehensive communication package that handles most applications.

5 Meeting Scheduling Agents in CONGOLOG

To define our simple meeting scheduling system, we first have to complete our specification of the primitive actions that manipulate users' schedule databases. The precondition axiom for ADDTOSCHED was given earlier (1). We take it to be possible for an agent to remove the activity scheduled for a user at a period iff the agent is the user's schedule manager and there is currently something on the user's schedule for that period:

$$\begin{aligned}Poss(agt, \text{RMVFROMSCHED}(user, period), s) \equiv& \\ & agt = \text{SCHEDULEMANAGER}(user) \wedge \\ & \exists activ, org\ \text{SCHEDULE}(user, period, activ, org, s)\end{aligned} \qquad (7)$$

```
**proc** $\text{SCHEDULEMEETING}(organizer, Participant, period)$
   **for** $p : Participant(p)$ **do**
      $\text{REQUEST}(\text{SCHEDULEMANAGER}(p), \text{ADDTOSCHEDULE}(p, period, \text{MEETING}, organizer));$
      $\text{QUERYWHETHER}(\text{SCHEDULEMANAGER}(p), \text{AGREEDTOMEET}(p, \mathsf{self}, period, organizer))$
   **endFor**;
   **while** $\neg$**KWhether**$(\mathsf{self}, \forall p[Participant(p) \supset$
                  $\text{AGREEDTOMEET}(p, \mathsf{self}, period, organizer)], \mathsf{now})$ **do**
      $\text{SENSEMSG};$
      **if** $\neg\text{EMPTY}(\text{MSGQ}(\mathsf{self}))$ **then** $\text{RMVMSG}$ **endIf**
   **endWhile**;
   $\text{INFORMWHETHER}(organizer,$
         $\forall p[Participant(p) \supset \text{AGREEDTOMEET}(p, \mathsf{self}, period, organizer)]);$
   **if** $\neg\forall p[Participant(p) \supset \text{AGREEDTOMEET}(p, \mathsf{self}, period, organizer)]$ **then**
      % release participants from commitment
      **for** $p : Participant(p)$ **do**
         $\text{REQUEST}(\text{SCHEDULEMANAGER}(p), \text{RMVFROMSCHED}(p, period))$
      **endFor**
   **endIf**
**endProc**
```

Figure 1: Procedure run by the "meeting scheduler" agents.

The effects of these actions on the SCHEDULE fluent are captured in the successor state axiom given earlier (3).

We are now ready to use CONGOLOG to define the behavior of our agents. We start with the "meeting scheduler" agents. These will be running the procedure in figure 1. The procedure uses the following abbreviation:

$$\begin{aligned}&\text{AGREEDTOMEET}(participant, requester, period, organizer, s) \overset{def}{=}\\ &\quad \exists s', s''(s'' \leq s \wedge Do(\text{SCHEDULEMANAGER}(participant), [\\ &\qquad \text{FIRST}(msgQ(\text{SCHEDULEMANAGER}(participant))) = \langle requester, \ulcorner\text{REQUEST}(\\ &\qquad\qquad \text{ADDTOSCHEDULE}(participant, period, \text{MEETING}, organizer))\urcorner\rangle ?;\\ &\qquad \text{SENSEMSG};\\ &\qquad \text{ADDTOSCHEDULE}(participant, period, \text{MEETING}, organizer)],\\ &\quad\ s', s'')).\end{aligned}$$

Thus, we take the participant to have agreed to a meeting request iff at some prior state, the participant's schedule manager has sensed a request to add the meeting to the participant's schedule, and then immediately fulfilled that request.[3]

Meanwhile, users' "schedule manager" agents run the procedure in figure 2. Note that the procedure does not handle cases where a user wants to be released from a commitment.

To run a meeting scheduling system, one could, for example, give the following program to the CON-

[3] This is quite a simplistic way of modeling agreement to a request. We could talk about the most recent instance of the request sensed, and if we modeled intentions, we could avoid requiring the requestee to handle the request immediately.

```
proc MANAGESCHEDULE($user$)
  while True do
    SENSEMSG;
    if $\neg$EMPTY(MSGQ(self)) then
      % if message is INFORM($\phi$), nothing to do
      if FIRST(MSGQ(self)) $= \langle queryer, \ulcorner$QUERYWHETHER($\phi$)$\urcorner\rangle$ then
        INFORMWHETHER($queryer, \phi$) endIf;
      if FIRST(MSGQ(self)) $= \langle requester,$
                  $\ulcorner$REQUEST(ADDTOSCHEDULE($user, period, activ, organizer$))$\urcorner\rangle$
         $\wedge$ PERMITTEDTOADDTOSCHED($organizer$)
         $\wedge$ $Poss$(self, ADDTOSCHEDULE($user, period, activ, organizer$), now)
        then ADDTOSCHEDULE($user, period, activ, organizer$) endIf;
      if FIRST(MSGQ(self)) $= \langle requester, \ulcorner$REQUEST(RMVFROMSCHEDULE($user, period$))$\urcorner\rangle$
         $\wedge$ $Poss$(self, RMVFROMSCHEDULE($user, period$), now)
         $\wedge$ $\exists activ$ SCHEDULE($user, period, activ,$ OWNER($requester$), now)
        then RMVFROMSCHEDULE($user, period$) endIf;
      RMVMSG
    endIf
  endWhile
endProc
```

Figure 2: Procedure run by the "schedule manager" agents.

GOLOG interpreter:

by(SM$_1$, MANAGESCHEDULE(USER$_1$)) $\|$
by(SM$_2$, MANAGESCHEDULEG(USER$_2$)) $\|$
by(SM$_3$, MANAGESCHEDULE(USER$_3$)) $\|$
by(MS$_1$, SCHEDULEMEETING(USER$_1$, $\{$USER$_1$, USER$_3\}$, NOON)) $\|$
by(MS$_2$, SCHEDULEMEETING(USER$_2$, $\{$USER$_2$, USER$_3\}$, NOON)).

Here, the meeting schedulers will both try to obtain USER$_3$'s agreement for a meeting at noon; there will thus be two types of execution sequences, depending on who obtains this agreement.

6 Architectural Issues

CONGOLOG must maintain a world model for each of the agents, so that it can evaluate the tests that agents' programs contain. Aspects of the agents' environment may also have to be modeled, for instance, if one is doing a simulated execution. In previous work, we have identified various useful techniques for performing this reasoning. In [17, 21], we propose a method for reasoning about whether a condition holds after a sequence of actions. It uses a form of regression to reduce the given query to one involving only the initial state; the resulting query can then be handled with an ordinary atemporal theorem proving method. The approach is more efficient than plain theorem proving and is very general — it handles incompletely known initial states as well as actions with context-dependent effects. It has been proven sound and complete. Regression can also be used to incorporate the information acquired through sensing into the world model:

the information acquired is regressed until it refers only to the initial state and the result is added to the theory of the initial state.

One problem with the regression approach is that its cost increases with the number of actions performed. It is clear that in general, we need to roll the agent's knowledge forward in time as actions are performed. But as Lin and Reiter [10, 12] have shown, the progression of a knowledge base (KB) need not be first-order representable. They do however identify a useful special case where the progression of a KB is first-order representable and easily computable: KB's where the state dependent sentences consist of ground literals only and where the actions have no context-dependent effects. Note that the closed world assumption is not made, so the agent's knowledge may be incomplete.

Exogenous actions cause problems for both of these techniques; in such cases, the agent does not know what sequence of actions has occurred. Our current implementation handles some of these cases by doing sensing when regression cannot be performed. It may be useful to precompute the effects of arbitrary sequences of exogenous actions on fluents. We are working on a general account.

Clearly, for anything other than short simulations, we need to progress the KB eventually. Work is under way to generalize the progression method. In some cases, it may be possible to partition the KB and handle some fluents by regression and some by progression. One could also have the agent strive to reduce its KB to the form required for progression by performing appropriate sensing acts. Finally, in some cases it may be necessary to be content with an efficiently computable approximation to the progression (i.e., something weaker than the strongest postcondition of the KB for the action).

Nondeterminism is another feature of our language that raises architectural issues. Our current implementation does arbitrary lookahead until a sensing action (or the end of the program) is reached and then commits to the actions. Clearly, this is not always the best strategy. Perhaps, we should provide a way for the user to specify when the interpreter should commit (something similar to PROLOG's cut). Also, we may even want to lookahead over sensing actions in some cases by considering all possible outcomes.

We are working on a formal specification of the CONGOLOG interpretation process for multi-agent programs. This will facilitate the evaluation of different implementations. The specification will draw on our account of when an agent *knows how* to execute a program (i.e., of the knowledge preconditions of actions) [7].

7 Implementation and Experimentation

A prototype GOLOG interpreter [9] has been implemented in PROLOG and experiments are under way with various types of applications. The most advanced involves a robotics application — mail delivery in an office environment [6]. The high-level controller of the robot programmed in GOLOG is interfaced to a robotics software package that support path planning and local navigation. The system currently works in simulation mode; experiments with a real robot are planned.

Two versions of the CONGOLOG interpreter are also being implemented. A monolithic version supports the concurrent execution of agents on a single processor. Another version supports the truly distributed execution of agents, with the communication between agent being implemented using TCP/IP tools [13]. An application involving tools for home banking [20] has been implemented; it includes a number of software agents that handle various parts of the banking process (responding to buttons on an ATM terminal, managing the accounts at a bank, monitoring account levels for a user etc.). The meeting scheduling application is also being implemented.

8 Conclusion

We believe that much of the brittleness of current AI systems derives from a failure to provide an adequate theoretical account of the task to be accomplished. Accordingly, we are trying to develop tools that are based on a solid theoretical foundation. The results we have obtained so far represent a significant step towards this objective. But clearly, more work is required on both implementation issues and the underlying theory. As mentioned earlier, we are examining various ways of implementing the mechanism for modeling world change and the effects of knowledge-producing actions. We are looking at how nondeterminism can be used to make "incomplete" programs executable and how run-time planning can be incorporated. Also under investigation are issues such as handling uncertainty and belief revision, as well as agent-relative (indexical) representations for robotics.

In terms of its support for multi-agent interaction, the current framework is rather limited. When agents interact with others without having complete knowledge of the situation, it is advantageous for them to view other agents as having goals, intentions, commitments, and abilities, and as making rational choices. This allows them to anticipate and influence the behavior of other agents, and cooperate with them. It also supports an abstract view of communication acts as action that affect other agents' mental states as opposed to mere message passing. We have started extending our framework to model goals, intentions, ability, and rational choice [23, 7], and considering possible implementation mechanisms. Communication raises numerous issues: How far should agent design tools go in handling intricacies that arise in human communication (e.g., deception, irony)? What's a good set of communication primitives and how do we implement them? With respect to coordination, what sort of infrastructure should we provide? Can we come up with a set of general purpose coordination policies? We are examining all these questions. Of course in the end, the usefulness of our approach will have to be evaluated empirically.

References

[1] Philip R. Cohen and Hector J. Levesque. Communicative actions for artificial agents. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems*, San Francisco, CA, June 1995. AAAI Press/MIT Press.

[2] ARPA Knowledge Sharing Initiative External Interfaces Working Group. Specification of the KQML agent-communication language. Working Paper, June 1993.

[3] M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag, July 1994.

[4] Robert Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes No. 7. Center for the Study of Language and Information, Stanford University, Stanford, CA, 2nd. edition, 1987.

[5] Andrew 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.

[6] Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. A logical approach to high-level robot programming – a progress report. In Benjamin Kuipers, editor, *Control of the Physical World by Intelligent Agents, Papers from the 1994 AAAI Fall Symposium*, pages 109–119, New Orleans, LA, November 1994.

[7] Yves Lespérance, Hector J. Levesque, Fangzhen Lin, and Richard B. Scherl. Ability and knowing how in the situation calculus. In preparation, 1995.

[8] Hector J. Levesque. Concurrency in the situation calculus. In preparation, 1995.

[9] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. Submitted to the *Journal of Logic Programming*, special issue on Reasoning about Action and Change, Sept. 1995.

[10] Fangzhen Lin and Raymond Reiter. How to progress a database (and why) I. logical foundations. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, pages 425–436, Bonn, Germany, 1994. Morgan Kaufmann Publishing.

[11] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[12] Fangzhen Lin and Raymond Reiter. How to progress a database II: The STRIPS connection. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 2001–2007, Montréal, August 1995. Morgan Kaufmann Publishing.

[13] D. Marcu, Y. Lespérance, H. Levesque, F. Lin, R. Reiter, and R. Scherl. Distributed software agents and communication in the situation calculus. In *Proceedings of the International Workshop on Intelligent Computer Communication*, pages 69–78, Cluj-Napoca, Romania, June 1995.

[14] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, UK, 1979.

[15] Robert C. Moore. A formal theory of knowledge and action. In J. R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Common Sense World*, pages 319–358. Ablex Publishing, Norwood, NJ, 1985.

[16] 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.

[17] Raymond 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.

[18] D. Riecken (editor). Communications of the ACM **37** (7), special issue on intelligent agents, July 1994.

[19] Stanley J. Rosenschein and Leslie P. Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149–173, 1995.

[20] Shane Ruman. GOLOG as an agent-programming language: Experiments in developing banking applications. Master's thesis, Department of Computer Science, University of Toronto, Toronto, ON, 1995. In preparation.

[21] Richard B. Scherl and Hector 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.

[22] 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.

[23] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. Goals and rational action in the situation calculus — a preliminary report. In *Working Notes of the AAAI Fall Symposium on Rational Agency: Concepts, Theories, Models, and Applications*, Cambridge, MA, November 1995. To appear.

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

[25] M. J. Wooldridge. Time, knowledge, and choice. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996. (In this volume).

[26] M.J. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995. To appear.