YVES LESPÉRANCE, HECTOR J. LEVESQUE, AND RAYMOND REITER

A SITUATION CALCULUS APPROACH TO MODELING AND PROGRAMMING AGENTS

1 INTRODUCTION

The notion of computational *agents* has become very fashionable lately [23, 31]. 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, 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 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 [29] 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

This paper is a much revised version of "Foundations of a Logical Approach to Agent Programming" which appeared in *Intelligent Agents Volume II — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*. The work described involved contributions by many people over a number of years, in particular, Giuseppe De Giacomo, Fangzhen Lin, Daniel Marcu, Richard Scherl, and David Tremaine. 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). Many of our team's papers are available at `http://www.cs.toronto.edu/~cogrobo/`.

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. It can be used to model the behavior of a set of agents and/or to actually implement them. Given a declarative specification of the agents' primitive actions, the designer/modeler can specify complex behaviors for the agents procedurally in the programming language. The behavior specification can be in terms of very high-level actions and can refer to conditions in effect in the environment — the interpreter automatically maintains a world model based on the specifications. The approach focuses on high-level programming rather than planning. But the programs can be nondeterministic and search for appropriate actions. When an implementation of the primitive actions is provided, the programs can be executed in a real environment; otherwise, a simulated execution is still possible.

Most of our work so far on the theory and implementation of agents has been concerned with single agents. Here, we extend our framework to deal with multi-agent systems. The treatment proposed is somewhat 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 set of messages waiting for it and abstract communication acts are defined (e.g., INFORM, REQUEST, QUERYWHETHER, etc.).

In the next section, we outline our theory of simple actions. Then, we discuss how knowledge and knowledge-producing actions can be modeled. Next, we present our account of complex actions, and explain how it can be viewed as an agent programming language. Section 6 develops a set of simple tools for agent communication and section 7 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 THE SITUATION CALCULUS AND THE FRAME PROBLEM

The situation calculus [17] is a first-order language (with some second-order features) for representing dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the initial situation, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol *do* and the term $do(\alpha, s)$ denotes the situation resulting from action α being performed in situation s. Actions may be parameterized. For example, $\textsc{put}(agt, x, y)$ might stand for the action of agent agt putting object x on object y, in which case $do(\textsc{put}(agt, \mathrm{A}, \mathrm{B}), s)$ denotes that situation resulting from agt putting A on B when the world is in situation s. Notice that in the situation calculus, actions are denoted by function symbols, and situations (world histories) are also first order terms. For example,

$$do(\textsc{putDown}(agt, \mathrm{A}), do(\textsc{walk}(agt, \mathrm{P}), do(\textsc{pickUp}(agt, \mathrm{A}), S_0)))$$

is a situation denoting the world history consisting of the sequence of actions

$$[\textsc{pickUp}(agt, \mathrm{A}), \textsc{walk}(agt, \mathrm{P}), \textsc{putDown}(agt, \mathrm{A})].$$

Notice that the sequence of actions in a history, in the order in which they occur, is obtained from a situation term by reading off its action instances from right to left.

Relations whose truth values vary from situation to situation, called *relational fluents*, are denoted by predicate symbols taking a situation term as their last argument. For example, $\textsc{Holding}(agt, x, s)$ might mean that agt is holding object x in situation s. Functions whose denotations vary from situation to situation are called *functional fluents*. They are denoted by function symbols with an extra argument taking a situation term, as in $\textsc{position}(agt, s)$, i.e., the position of agt in situation s.

An action is specified by first stating the conditions under which it can be performed by means of a *precondition axiom* of the following form:

$$Poss(\alpha(\vec{x}), s) \Leftrightarrow \pi_\alpha(\vec{x}, s)$$

Here, $\pi_\alpha(\vec{x}, s)$ is a formula specifying the preconditions for action $\alpha(\vec{x})$. For example, the precondition axiom for the action $\textsc{addToSched}$ might be:

$$\begin{aligned} &Poss(\textsc{addToSched}(agt, user, period, activity, organizer), s) \Leftrightarrow \\ &agt = \textsc{scheduleManager}(user) \wedge \\ &\neg\exists activity', organizer' \\ &\quad \textsc{Schedule}(user, period, activity', organizer', s) \end{aligned} \tag{1}$$

This says that it is possible for agent agt to add an activity to $user$'s schedule in situation s iff agt is $user$'s schedule manager and there is nothing on $user$'s schedule for that period in s.

Secondly, one must specify how the action affects the state of the world with *effect axioms*. For example,

$$\begin{aligned}&\mathit{Poss}(\text{ADDTOSCHED}(agt, user, period, activity, organizer), s) \Rightarrow\\ &\text{SCHEDULE}(user, period, activity, organizer,\\ &\quad \mathit{do}(\text{ADDTOSCHED}(agt, user, period, activity, organizer), s))\end{aligned}$$

Effect axioms provide the "causal laws" for the domain of application.

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; for example,

$$\begin{aligned}&\mathit{Poss}(\text{RAISESAL}(comp, user_1, amt), s) \wedge\\ &\text{SCHEDULE}(user_2, per, activ, org, s) \Rightarrow\\ &\text{SCHEDULE}(user_2, per, activ, org, \mathit{do}(\text{RAISESAL}(comp, user, amt), s))\end{aligned}$$

The frame problem arises because the number of these frame axioms is very large, in general, of the order of $2 \times \mathcal{A} \times \mathcal{F}$, where $\mathcal{A}$ is the number of actions and $\mathcal{F}$ the number of fluents.

Our approach incorporates the solution to the frame problem described in [21]. First, for each fluent F, one can collect all effects axioms involving F to produce two *general effect axioms* of the following form:

$$\mathit{Poss}(a, s) \wedge \gamma_F^+(\vec{x}, a, s) \Rightarrow F(\vec{x}, \mathit{do}(a, s))$$

$$\mathit{Poss}(a, s) \wedge \gamma_F^-(\vec{x}, a, s) \Rightarrow \neg F(\vec{x}, \mathit{do}(a, s))$$

Here $\gamma_F^+(\vec{x}, a, s)$ is a formula describing under what conditions doing the action a in situation s leads the fluent $F(\vec{x})$ to become true in the successor situation $do(a, s)$ and similarly $\gamma_F^-(\vec{x}, a, s)$ is a formula describing the conditions under which performing action a in situation s results in the fluent $F(\vec{x})$ becoming false in situation $do(a, s)$. For example, the following might be the general effect axioms for the fluent SCHEDULE:

$$\begin{aligned}&\mathit{Poss}(\text{ADDTOSCHED}(agt, user, per, activ, org), s) \Rightarrow\\ &\text{SCHEDULE}(user, per, activ, org,\\ &\quad \mathit{do}(\text{ADDTOSCHED}(agt, user, per, activ, org), s))\end{aligned} \tag{2}$$

$$\begin{aligned}&\mathit{Poss}(\text{RMVFROMSCHED}(agt, user, per), s) \Rightarrow\\ &\neg\text{SCHEDULE}(user, per, activ, org,\\ &\quad \mathit{do}(\text{RMVFROMSCHED}(agt, user, per), s))\end{aligned} \tag{3}$$

The solution to the frame problem rests on a *completeness assumption*. This assumption is that the general effect axioms characterize all the conditions under which action a can lead to a fluent $F(\vec{x})$'s becoming true (respectively, false) in the successor situation. Therefore, if action a is possible and $F(\vec{x})$'s truth value changes from *false* to *true* as a result of doing a, then $\gamma_F^+(\vec{x}, a, s)$ must be *true* and similarly for a change from *true* to *false*. Additionally, *unique name axioms* are added for actions and situations. From the general effect axioms and the completeness assumption, one can derive a *successor state axiom* of the following form for the fluent F:

$$\begin{array}{l} \mathit{Poss}(a, s) \Rightarrow [F(\vec{x}, \mathit{do}(a, s)) \Leftrightarrow \\ \quad \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))] \end{array}$$

By quantifying over actions, this single axiom provides a parsimonious solution to the frame problem. Similar successor state axioms can be written for functional fluents.[1]

Applying this to our example, from 2 and 2 we obtain the following successor state axiom for SCHEDULE:

$$\begin{array}{ll} & \mathit{Poss}(act, s) \Rightarrow \\ & [\textrm{SCHEDULE}(user, per, activ, org, \mathit{do}(act, s)) \Leftrightarrow \\ (4) & \exists agt\ act = \textrm{ADDTOSCHED}(agt, user, per, activ, org) \\ & \vee\ \textrm{SCHEDULE}(user, per, activ, org, s) \wedge \\ & \quad \neg\exists agt\ act = \textrm{RMVFROMSCHED}(agt, user, per)] \end{array}$$

i.e., an activity is on $user$'s schedule following the performance of action act in situation s iff either the action is some agent adding the activity to $user$'s schedule, or the activity was already on $user$'s schedule in s and the action is not some agent removing the activity from $user$'s schedule. Now note for example that if $\textrm{SCHEDULE}(u, p, a, o, S_0)$, then by the unique names axioms for actions, it also follows that $\textrm{SCHEDULE}(u, p, a, o, \mathit{do}(\textrm{RAISESAL}(c, u', d), S_0))$.

In multi-agent domains, it is often useful to refer to the agent of an action. We use the term $\mathit{agent}(act)$ for this. We require that for each primitive action one provide an axiom specifying who its agent is, for example:

$$\mathit{agent}(\textrm{ADDTOSCHED}(agt, user, per, activ, org)) = agt$$

In general, a particular domain of application will be specified by the union of the following sets of axioms:

- Axioms describing the initial situation, S_0.

[1] In the above, we have assumed that there were no state constraints which might contribute ramifications, i.e., indirect effects of actions. In [15], the approach presented is extended to deal with state constraints by compiling their effects into the successor state axioms.

- Action precondition axioms, one for each primitive action.
- Successor state axioms, one for each fluent.
- Unique names axioms for the primitive actions.
- Axioms specifying the *agent* of each primitive action.
- Some foundational, domain independent axioms.

The latter foundational axioms include unique names axioms for situations, and an induction axiom. They also introduce the relation $<$ over situations. $s < s'$ holds iff s' is the result of some sequence of actions being performed in s, where each action in the sequence is possible in the situation in which it is performed; $s \leq s'$ stands for $s < s' \vee s = s'$. Since the foundational axioms play no special role in this paper, we omit them. For details, and for some of their metamathematical properties, see Lin and Reiter [15] and Reiter [22].

For domain theories of this kind, there are very clean characterizations of various reasoning tasks, for instance planning [6]:

> **Classical Planning:** Given a domain theory *Axioms* as above, and a goal formula $\phi(s)$ with a single free-variable s, the planning task is to find a sequence of actions $\vec{a}$ such that:
>
> $$\mathit{Axioms} \models S_0 \leq \mathit{do}(\vec{a}, S_0) \wedge \phi(\mathit{do}(\vec{a}, S_0))$$
>
> where $\mathit{do}([a_1, \ldots, a_n], s)$ is an abbreviation for
>
> $$\mathit{do}(a_n, \mathit{do}(a_{n-1}, \ldots, \mathit{do}(a_1, s) \ldots)).$$

In other words, the task is to find a sequence of actions that is executable (each action is executed in a context where its precondition is satisfied) and that achieves the goal (the goal formula ϕ holds in the final situation that results from performing the actions in sequence). If you have a particular planning algorithm, you can show that it is sound by proving that it only returns answers that satisfy the specification given above.

3 KNOWLEDGE AND KNOWLEDGE PRODUCING ACTIONS

Knowledge can be represented in the situation calculus by adapting the possible world model of modal logic (as first done by Moore [18]). The idea is to model someone's uncertainty (lack of knowledge) about what is true using the set of situations he/she considers possible. We introduce a fluent *K*, where $K(agt, s', s)$ means that in situation s, the agent agt thinks the world could be in situation s' (in modal logic terms, K is the knowledge accessibility relation). Then we introduce the abbreviation:

$$\mathbf{Know}(agt, \phi, s) \stackrel{\text{def}}{=} \forall s'(K(agt, s', s) \Rightarrow \phi(s')).$$

Thus, agt knows in s that ϕ holds iff ϕ holds in all the situations s' that agt considers possible in s. For clarity, we sometimes use the placeholder term *now* to represent the situation bound by the enclosing **Know**; so $\mathbf{Know}(agt, \phi(now), s)$ stands for $\forall s'(K(agt, s', s) \Rightarrow \phi(s'))$.

With this in place, we can then consider knowledge-producing actions (as they occur in perception or communication). Such actions affect the mental state of the agent rather than the state of the external world. For example, consider the action of an agent sensing what messages he has with the following effect axiom:

$$\begin{array}{l} Poss(\textsc{senseMsgs}(agt), s) \Rightarrow \\ \mathbf{KWhether}(agt, \textsc{MsgRcvd}(agt, sender, msgId, msg, now), \\ \quad do(\textsc{senseMsgs}(agt), s)) \end{array}$$

This says that after performing the action SENSEMSGS, the agent agt knows exactly which messages it has received and not yet processed, who sent them, and what their message IDs are ($\mathbf{KWhether}(agt, \phi, s)$ is an abbreviation for the formula $\mathbf{Know}(agt, \phi, s) \vee \mathbf{Know}(agt, \neg\phi, s)$).

Scherl and Levesque [26] have shown how one can generalize the solution to the frame problem of the previous section to deal with knowledge and knowledge-producing actions. But they only consider domains where there is a single agent. For multi-agents settings, their solution can used with minimal changes provided we assume that all actions are *public*, i.e., that agents are aware of every action that happens. For instance, if we make this assumption and the only knowledge producing action in the domain is SENSEMSGS, then we can use the following successor state axiom for the knowledge fluent K:

$$\begin{array}{l} Poss(act, s) \Rightarrow \\ [K(knower, s'', do(act, s)) \Leftrightarrow \\ \exists s'(K(knower, s', s) \wedge s'' = do(act, s') \wedge Poss(act, s') \wedge \\ \quad [act = \textsc{senseMsgs}(knower) \Rightarrow \\ \quad\quad \forall sndr, mId, m(\textsc{MsgRcvd}(knower, sndr, mId, m, s') \Leftrightarrow \\ \quad\quad\quad \textsc{MsgRcvd}(knower, sndr, mId, m, s))])] \end{array}$$

Let's look at what this says. There are two cases. If the action act is not a knowledge producing action performed by the agent under consideration $knower$ (i.e., $act \neq \textsc{senseMsgs}(knower)$), then the axiom says that in the resulting situation $do(act, s)$, $knower$ considers possible any situation s'' that is the result of act being performed in a situation s' that $knower$ used to consider possible before the action. Thus, $knower$ only acquires the knowledge that the action act has been performed. If on the other hand, the action act is a knowledge producing action performed by $knower$ (i.e., $act = \text{SENSEMSGS}(knower)$), then we get that in the resulting situation $do(act, s)$, $knower$ considers possible any situation s'' that is the result of act being performed in a situation s' that $knower$ used to consider possible before the action, and where the fluent $\text{MSGRCVD}(knower, \ldots)$ holds exactly for the messages for which it holds in the "real" situation s. Thus after doing SENSEMSGS, an agent knows that it has performed this action and knows exactly which messages it has received and not yet processed. This can be extended to an arbitrary number of knowledge-producing actions in a straightforward way.

However, the assumption that all actions are public is too strong for many multi-agent domains; agents need not be aware of the actions of other agents (exogenous actions). Another workable approach is to be very "conservative" and have agents allow for the occurrence of an arbitrary number of exogenous actions at every step. For our meeting scheduling application, this yields the following successor state axiom:

$$
\begin{aligned}
&\mathit{Poss}(act, s) \Rightarrow \\
&[K(knower, s'', do(act, s)) \Leftrightarrow \\
&\;\exists s'(K(knower, s', s) \wedge \\
&\quad(\mathit{agent}(act) \neq knower \Rightarrow \mathit{ExoOnly}(knower, s', s'')) \wedge \\
&\quad(\mathit{agent}(act) = knower \Rightarrow \exists s^*(\mathit{ExoOnly}(knower, s', s^*) \wedge \\
&\qquad s'' = do(act, s^*) \wedge Poss(act, s^*) \wedge \\
&\qquad[act = \text{SENSEMSGS}(knower) \Rightarrow \\
&\qquad\;\forall sndr, mId, m(\text{MSGRCVD}(knower, sndr, mId, m, s^*) \Leftrightarrow \\
&\qquad\qquad\text{MSGRCVD}(knower, sndr, mId, m, s))]))))] \\
&\text{where } \mathit{ExoOnly}(agt, s, s') \stackrel{\text{def}}{=} s \leq s' \wedge \\
&\qquad\qquad\forall s^* \forall act(s < do(act, s^*) \leq s' \Rightarrow \mathit{agent}(act) \neq agt)
\end{aligned}
$$

Let us explain how this works. When an action act is performed by some agent 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 situation 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 it may have had about fluents that could be affected by exogenous actions. When act is a non-knowledge-producing action (e.g., $\text{ADDTOSCHED}(knower, u, p, a, o)$) performed by $knower$, the specification states that in the resulting situation, $knower$ considers possible any situation that is the result of its doing act preceded by any number of exogenous actions occurring in a situation that used to be considered possible; thus, $knower$ acquires the knowledge that it has just performed act, but loses any knowledge it 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., SENSEMSGS) 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 what messages it has received and not yet processed.

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 agent. Thus, schedule manager 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, as discussed earlier. 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 being investigated.

4 COMPLEX ACTIONS AND GOLOG

A very general and flexible approach to designing agents involves using a planner. When the agent gets a goal, the planner is invoked to generate a plan that achieves the goal, and then the plan is executed. A problem with this approach is that plan synthesis is often computationally infeasible in complex domains, especially when the agent does not have complete knowledge and there are exogenous actions. An alternative approach that is showing promise is that of *high-level program execution* [13]. The idea, roughly, is that instead of searching for a sequence of actions that would take the agent from an initial state to some goal state, the task is to find a sequence of actions that constitutes a legal execution of some high-level non-deterministic program. As in planning, to find such a sequence it is necessary to reason about the preconditions and effects of the actions within the body of the program. However, if the program happens to be almost deterministic, very little searching is required; as more and more non-determinism is included, the search task begins to resemble traditional planning. Thus, in formulating a high-level program, the user gets to control the search effort required. The hope is that in many domains, what an agent needs to do can be conveniently expressed using a suitably rich high-level programming language.[2]

Our proposal for such a language is Golog[13], a logic-programming language whose primitive actions are those of a background domain theory of the form described earlier. It includes the following constructs:

α,	primitive action
$\phi?$,	wait for a condition[3]

[2]This is not to imply that the planning approach or belief-desire-intention models of agents are never useful, quite the opposite. Later on, we will see for instance, that modeling goals would be quite useful in dealing with communication. But the fact remains that these approaches are computationally problematic.

[3]Here, ϕ stands for a situation calculus formula with all situation arguments suppressed; $\phi(s)$ will denote the formula obtained by restoring situation variable s to all fluents appearing in ϕ. Because there are no exogenous actions or concurrent processes in Golog, waiting for ϕ amounts to testing that ϕ holds in the current situation.

$(\sigma_1; \sigma_2)$,	sequence
$(\sigma_1 \mid \sigma_2)$,	nondeterministic choice between actions
$\pi x.\sigma$,	nondeterministic choice of arguments
σ^*,	nondeterministic iteration
if ϕ **then** σ_1 **else** σ_2,	conditional
while ϕ **do** σ,	loop
proc $\beta(\vec{x})\ \sigma$,	procedure definition[4]

Here's a simple example to illustrate some of the more unusual features of the language:

proc REMOVEABLOCK
  πb [ONTABLE(b)?; PICKUP(b); PUTAWAY(b)]
endProc;
REMOVEABLOCK*;
$\neg\exists block$ ONTABLE$(block)$?

Here we first define a procedure to remove a block from the table using the nondeterministic operator π. $\pi x\,[\sigma(x)]$ means nondeterministically pick an individual x, and for that x, perform the program $\sigma(x)$. The wait action ONTABLE(b)? succeeds only if the individual chosen, b, is a block that is on the table. The main part of the program uses the nondeterministic iteration operator; it simply says to execute REMOVEABLOCK zero or more times until the table is clear.

In its most basic form, the high-level program execution task is a special case of the planning task discussed earlier:

> **Program Execution:** Given a domain theory *Axioms* as above, and a program σ, the execution task is to find a sequence of actions $\vec{a}$ such that:
>
> $$\textit{Axioms} \models \textit{Do}(\sigma, S_0, do(\vec{a}, S_0))$$
>
> where $\textit{Do}(\sigma, s, s')$ is an abbreviation for a formula of the situation calculus which says that program σ when executed starting in situation s has s' as a legal terminating situation.

In [13], a simple inductive definition of *Do* was presented, containing rules such as:

$$\textit{Do}([\sigma_1; \sigma_2], s, s') \stackrel{\text{def}}{=} \exists s''.\ \textit{Do}(\sigma_1, s, s'') \wedge \textit{Do}(\sigma_2, s'', s')$$
$$\textit{Do}([\sigma_1 \mid \sigma_2], s, s') \stackrel{\text{def}}{=} \textit{Do}(\sigma_1, s, s') \vee \textit{Do}(\sigma_2, s, s')$$

one for each construct in the language. This kind of semantics is sometimes called *evaluation semantics* [7] since it is based on the complete evaluation of the program.

[4]For space reasons, we ignore these here.

It is difficult to extend this kind of semantics to deal with concurrent actions. Since these are required in multi-agent domains, a more refined kind of semantics was developed in [2]. This kind of semantics is called *computational semantics* [7]. It is based on "single steps" of computation, or *transitions*[5]. A step in the present instance is either a primitive action or testing whether a condition holds in the current situation. Two special predicates are introduced, *Final* and *Trans*, where $\mathit{Final}(\sigma, s)$ is intended to say that program σ may legally terminate in situation s, and where $\mathit{Trans}(\sigma, s, \sigma', s')$ is intended to say that program σ in situation s may legally execute one step, ending in situation s' with program σ' remaining.

Final and *Trans* are characterized by a set of equivalence axioms, each depending on the structure of the first argument. These quantify over programs and so, unlike in [13], it is necessary to encode Golog programs as first-order terms, including introducing constants denoting variables, and so on. This is laborious but quite straightforward [8][6]. Here, details are omitted and we simply use programs within formulas as if they were already first-order terms.

The equivalence axioms Γ_F for *Final* are as follows (universally closing on s):[7]

$$
\begin{aligned}
&\mathit{Final}(\mathit{nil}, s) \Leftrightarrow \mathit{TRUE}\\
&\mathit{Final}(\alpha, s) \Leftrightarrow \mathit{FALSE}\\
&\mathit{Final}(\phi?, s) \Leftrightarrow \mathit{FALSE}\\
&\mathit{Final}([\sigma_1;\sigma_2], s) \;\Leftrightarrow\; \mathit{Final}(\sigma_1, s) \wedge \mathit{Final}(\sigma_2, s)\\
&\mathit{Final}([\sigma_1 \mid \sigma_2], s) \;\Leftrightarrow\; \mathit{Final}(\sigma_1, s) \vee \mathit{Final}(\sigma_2, s)\\
&\mathit{Final}(\pi x.\sigma, s) \Leftrightarrow \exists x.\mathit{Final}(\sigma, s)\\
&\mathit{Final}(\sigma^*, s) \Leftrightarrow \mathit{TRUE}\\
&\mathit{Final}(\textbf{if } \phi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2, s) \Leftrightarrow\\
&\qquad \phi(s) \wedge \mathit{Final}(\sigma_1, s) \;\vee\; \neg\phi(s) \wedge \mathit{Final}(\sigma_2, s)\\
&\mathit{Final}(\textbf{while } \phi \textbf{ do } \sigma, s) \Leftrightarrow \phi(s) \wedge \mathit{Final}(\sigma, s) \;\vee\; \neg\phi(s)
\end{aligned}
$$

The equivalence axioms Γ_T for *Trans* are as follows (universally closing on s, δ, s'):

$$
\begin{aligned}
&\mathit{Trans}(\mathit{nil}, s, \delta, s') \Leftrightarrow \mathit{FALSE}\\
&\mathit{Trans}(\alpha, s, \delta, s') \Leftrightarrow \mathit{Poss}(\alpha, s) \wedge \delta = \mathit{nil} \wedge s' = do(\alpha, s)\\
&\mathit{Trans}(\phi?, s, \delta, s') \;\Leftrightarrow\; \phi(s) \wedge \delta = \mathit{nil} \wedge s' = s\\
&\mathit{Trans}([\sigma_1;\sigma_2], s, \delta, s') \Leftrightarrow\\
&\qquad \mathit{Final}(\sigma_1, s) \wedge \mathit{Trans}(\sigma_2, s, \delta, s') \vee \exists\delta'.\delta = (\delta';\sigma_2) \wedge \mathit{Trans}(\sigma_1, s, \delta', s')\\
&\mathit{Trans}([\sigma_1 \mid \sigma_2], s, \delta, s') \Leftrightarrow \mathit{Trans}(\sigma_1, s, \delta, s') \vee \mathit{Trans}(\sigma_2, s, \delta, s')\\
&\mathit{Trans}(\pi x.\sigma, s, \delta, s') \;\Leftrightarrow\; \exists x.\mathit{Trans}(\sigma, s, \delta, s')\\
&\mathit{Trans}(\sigma^*, s, \delta, s') \Leftrightarrow \exists\delta'.\delta = (\delta';\sigma^*) \wedge \mathit{Trans}(\sigma, s, \delta', s')\\
&\mathit{Trans}(\textbf{if } \phi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2, s, \delta, s') \Leftrightarrow\\
&\qquad \phi(s) \wedge \mathit{Trans}(\sigma_1, s, \delta, s') \quad \vee \neg\phi(s) \wedge \mathit{Trans}(\sigma_2, s, \delta, s')
\end{aligned}
$$

---

[5] Both types of semantics belong to the family of structural operational semantics introduced in [19].

[6] Observe that *Final* and *Trans* cannot occur in tests, hence self-reference is disallowed.

[7] It is convenient to include a special "empty" program *nil*.

$$\begin{aligned}&\mathit{Trans}(\textbf{while}\ \phi\ \textbf{do}\ \sigma, s, \delta, s') \Leftrightarrow\\&\quad \phi(s) \wedge \exists \delta'.\, \delta = (\delta'; \textbf{while}\ \phi\ \textbf{do}\ \sigma) \wedge \mathit{Trans}(\sigma, s, \delta', s')\end{aligned}$$

With *Final* and *Trans* in place, we may define *Do* as:

$$\mathit{Do}(\sigma, s, s') \stackrel{\text{def}}{=} \exists \delta. \mathit{Trans}^*(\sigma, s, \delta, s') \wedge \mathit{Final}(\delta, s')$$

where Trans^* is the transitive closure of *Trans*, defined as the (second-order) situation calculus formula:

$$\mathit{Trans}^*(\sigma, s, \sigma', s') \stackrel{\text{def}}{=} \forall T[\ldots \Rightarrow T(\sigma, s, \sigma', s')]$$

where the ellipsis stands for:

$$\begin{aligned}&\forall s.\, T(\sigma, s, \sigma, s) \quad \wedge\\&\forall s, \delta', s', \delta'', s''.\, T(\sigma, s, \delta', s') \wedge \mathit{Trans}(\delta', s', \delta'', s'') \;\Rightarrow\; T(\sigma, s, \delta'', s'').\end{aligned}$$

In other words, $\mathit{Do}(\sigma, s, s')$ holds if it is possible to repeatedly single-step the program σ, obtaining a program δ and a situation s' such that δ can legally terminate in s'. It can be shown that this definition of *Do* is equivalent to that in [13].

On the surface, Golog looks a lot like a standard procedural programming language. It is indeed a programming language, but one whose execution, like planning, depends on reasoning about actions. An interpreter for Golog essentially searches for a sequence of primitive actions that can be proven to lead to a final situation of the program. Thus, a crucial part of a Golog program is the *declarative* part: the precondition axioms, the successor state axioms, and the axioms characterizing the initial situation. A Golog program together with the definition of *Do* and some foundational axioms about the situation calculus *is* a formal logical theory about the possible behaviors of an agent in a given environment.

The declarative part of a Golog program is used by the Golog interpreter in two ways. The successor state axioms and the axioms specifying the initial situation are used to evaluate the conditions that appear in the program (wait actions and **if/while** conditions) as the program is interpreted. The action preconditions axioms are used (with the other axioms) to check whether the next primitive action is possible in the situation reached so far. Golog programs are often nondeterministic and a failed precondition or test action causes the interpreter to backtrack and try a different path through the program. For example, given the program $(a; \phi?) \mid (b; c)$, the Golog interpreter might determine that a is possible in the initial situation S_0, but upon noticing that ϕ is false in $\mathit{do}(a, S_0)$, backtrack and return the final situation $\mathit{do}(c, \mathit{do}(b, S_0))$, after confirming that b is possible initially and that c is possible in $\mathit{do}(b, S_0)$.

Thus in a way, the Golog interpreter is automatically maintaining a model of the world state for the programmer using the axioms. If a program is going to maintain a model of its environment, it seems that having it done automatically from declarative specifications is much more convenient and less error prone than having to program such model updating from scratch. The Golog programmer can work at a much higher level of abstraction.

And to reiterate the main idea, Golog aims for a middle ground between run-time planning and explicit programming down to the last detail. It supports search for appropriate actions through nondeterminism as well as explicit programming. Thus for example, the program

$$\textbf{while } \exists b\, \textsc{OnTable}(b) \textbf{ do } \pi b.\, \textsc{remove}(b) \textbf{ endWhile}$$

leaves it to the Golog interpreter to find a legal sequence of actions that clears the table.

5 CONCURRENT ACTIONS AND CONGOLOG

To implement multiple agents in a single program, we need concurrent processes. In [2], an extended version of Golog that incorporates a rich account of concurrency is developped. This extended language is called ConGolog. Let us now review the syntax and semantics of ConGolog (this section is a quasi-verbatim reproduction of part of [2]). The ConGolog account of concurrency said to be 'rich' because it handles:

- concurrent processes with possibly different priorities,
- high-level interrupts,
- arbitrary exogenous actions.

As is commonly done in other areas of computer science, concurrent processes are modeled as interleavings of the primitive actions in the component processes. A concurrent execution of two processes is one where the primitive actions in both processes occur, interleaved in some fashion. So in fact, there is never more than one primitive action happening at the same time. As discussed in [3, 20], to model actions that intuitively could occur simultaneously, *e.g.* actions of extended duration, one may instantaneous start and stop (*i.e.* clipping) actions, where once again interleaving is appropriate.

An important concept in understanding concurrent execution is that of a process becoming *blocked*. If a deterministic process σ is executing, and reaches a point where it is about to do a primitive action a in a situation s but where $Poss(a, s)$ is false (or a wait action $\phi?$, where $\phi(s)$ is false), then the overall execution need not fail as in Golog. In ConGolog, the current interleaving can continue successfully provided that a process other than σ executes next. The net effect is that σ is suspended or blocked, and execution must continue elsewhere.[8]

[8]Just as actions in Golog are external (*e.g.* there is no internal variable assignment), in ConGolog, blocking and unblocking also happen externally, via *Poss* and wait actions. Internal synchronization primitives are easily added.

The ConGolog language is exactly like Golog except with the following additional constructs:

$$\begin{array}{ll} (\sigma_1 \parallel \sigma_2), & \text{concurrent execution} \\ (\sigma_1 \rangle\!\rangle\, \sigma_2), & \text{concurrency with different priorities} \\ \sigma^{\parallel}, & \text{concurrent iteration} \\ <\phi \rightarrow \sigma>, & \text{interrupt.} \end{array}$$

$(\sigma_1 \parallel \sigma_2)$ denotes the concurrent execution of the actions σ_1 and σ_2. $(\sigma_1 \rangle\!\rangle\, \sigma_2)$ denotes the concurrent execution of the actions σ_1 and σ_2 with σ_1 having higher priority than σ_2. This restricts the possible interleavings of the two processes: σ_2 executes only when σ_1 is either done or blocked. The next construct, $\sigma^{\parallel}$, is like nondeterministic iteration, but where the instances of σ are executed concurrently rather than in sequence. Finally, $<\phi \rightarrow \sigma>$ is an interrupt. It has two parts: a trigger condition ϕ and a body, σ. The idea is that the body σ will execute some number of times. If ϕ never becomes true, σ will not execute at all. If the interrupt gets control from higher priority processes when ϕ is true, then σ will execute. Once it has completed its execution, the interrupt is ready to be triggered again. This means that a high priority interrupt can take complete control of the execution. For example, $<\mathit{TRUE} \rightarrow \mathit{ringBell}>$ at the highest priority would ring a bell and do nothing else. With interrupts, it is easy to write agent programs that are "reactive", that can stop whatever task they are doing to handle various concerns as they arise.

Let us now explain how *Final* and *Trans* are extended to handle these constructs. (Interrupts are handled separately below.) For *Final*, the extension is straightforward:

$$\begin{array}{l} \mathit{Final}([\sigma_1 \parallel \sigma_2], s) \Leftrightarrow \mathit{Final}(\sigma_1, s) \wedge \mathit{Final}(\sigma_2, s) \\ \mathit{Final}([\sigma_1 \rangle\!\rangle\, \sigma_2], s) \Leftrightarrow \mathit{Final}(\sigma_1, s) \wedge \mathit{Final}(\sigma_2, s) \\ \mathit{Final}(\sigma^{\parallel}, s) \Leftrightarrow \mathit{TRUE} \end{array}$$

Observe that the last clause says that it is legal to execute the σ in $\sigma^{\parallel}$ zero times. For *Trans*, one has the following:

$$\begin{array}{l} \mathit{Trans}([\sigma_1 \parallel \sigma_2], s, \delta, s') \Leftrightarrow \\ \quad \exists \delta'.\delta = (\delta' \parallel \sigma_2) \wedge \mathit{Trans}(\sigma_1, s, \delta', s') \vee \delta = (\sigma_1 \parallel \delta') \wedge \\ \mathit{Trans}(\sigma_2, s, \delta', s') \\ \mathit{Trans}([\sigma_1 \rangle\!\rangle\, \sigma_2], s, \delta, s') \Leftrightarrow \\ \quad \exists \delta'.\delta = (\delta' \rangle\!\rangle\, \sigma_2) \wedge \mathit{Trans}(\sigma_1, s, \delta', s') \quad \vee \\ \qquad \delta = (\sigma_1 \rangle\!\rangle\, \delta') \wedge \mathit{Trans}(\sigma_2, s, \delta', s') \wedge \neg \exists \delta'', s''.\mathit{Trans}(\sigma_1, s, \delta'', s'') \\ \mathit{Trans}(\sigma^{\parallel}, s, \delta, s') \Leftrightarrow \exists \delta'.\delta = (\delta' \parallel \sigma^{\parallel}) \wedge \mathit{Trans}(\sigma, s, \delta', s') \end{array}$$

In other words, you single step $(\sigma_1 \parallel \sigma_2)$ by single stepping either σ_1 or σ_2 and leaving the other process unchanged. The $(\sigma_1 \rangle\!\rangle\, \sigma_2)$ construct is identical, except that you are only allowed to single step σ_2 if there is no legal step for σ_1.[9] This

[9] It is true, though not immediately obvious, that *Trans** remains properly defined even with these axioms containing negative occurrences of *Trans*. See [3] for details.

ensures that σ_1 will execute as long as it is possible for it to do so. Finally, you single step $\sigma^{\|}$ by single stepping σ, and what is left is the remainder of σ as well as $\sigma^{\|}$ itself. This allows an unbounded number of instances of σ to be running.

Exogenous actions are primitive actions that may occur without being part of a user-specified program. Let us assume that in the background theory, the user declares using a predicate *Exo* which actions can occur exogenously. Such actions can be handled by modifying the specification of *Trans* for primitive actions and wait actions from Golog as follows:

$$\begin{array}{l} \mathit{Trans}(\alpha, s, \delta, s') \Leftrightarrow \ldots \mathit{as\ before} \ldots \quad \vee \\ \quad \exists a.\mathit{Exo}(a) \wedge \mathit{Poss}(a, s) \wedge \delta = \alpha \wedge s' = \mathit{do}(a, s) \end{array}$$

and similarly for test actions. So while executing a program, exogenous actions whose preconditions are satisfied can occur before any primitive action or while waiting for any condition to become true.

Finally, regarding interrupts, the approach taken is define these using other constructs of ConGolog:

$$\begin{array}{l} <\phi \rightarrow \sigma> \stackrel{\text{def}}{=} \textbf{while } Interrupts\_running \textbf{ do} \\ \qquad\qquad\qquad\qquad \textbf{if } \phi \textbf{ then } \sigma \textbf{ else } \mathit{FALSE}? \end{array}$$

To see how this works, first assume that the special fluent $Interrupts\_running$ is always true. When an interrupt $<\phi \rightarrow \sigma>$ gets control, it repeatedly executes σ until ϕ becomes false, at which point it blocks, releasing control to anyone else able to execute. Note that according to the above definition of *Trans*, no transition occurs between the test condition in a while-loop or an if-then-else and the body. In effect, if ϕ becomes false, the process blocks right at the beginning of the loop, until some other action makes ϕ true and resumes the loop. To actually terminate the loop, a a special primitive action $stop_interrupts$is used; its only effect is to make $Interrupts\_running$ false. Thus, to execute a program σ containing interrupts, one would actually execute the program

$$\{start\_interrupts\ ;\ (\sigma \rangle\!\rangle\ stop\_interrupts)\}$$

which has the effect of stopping all blocked interrupt loops in σ at the lowest priority, *i.e.* when there are no more actions in σ that can be executed.

6 COMMUNICATION IN CONGOLOG

Multi-agent applications usually require some kind of inter-agent communication facility. A popular choice is the KQML communication language [4] 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 define our own simple communication toolkit. 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. In a given situation, each agent is taken to have a set of messages it has received and not yet processed; this is modeled using the predicate fluent $\mathrm{MSGRCVD}(agt, sender, msgId, msg, s)$, meaning that in situation s, agt has received a message msg with message ID $msgId$ from $sender$ (which it has yet to process). Note that this is more general than a simple queue; the agent need not process the messages in the order in which they arrive. We assume that message IDs are generated using a global message counter represented by the functional fluent $\mathrm{MSGCTR}(s)$ (it is straightforward to generalize this to use agent-relative IDs). There are three primitive action types that operate on these fluents:

- $\mathrm{SENDMSG}(agt, recipient, msg)$: agt sends message msg to $recipient$; the current value of the message counter is used as message ID and the message is added to the set of messages $recipient$ has received and not yet processed; the value of the message counter is also incremented;
- $\mathrm{SENSEMSGS}(agt)$: agt senses what messages he has received and not yet processed; and
- $\mathrm{RMVMSG}(agt, msgId)$: agt removes the message with ID $msgId$ from his set of messages received and not yet processed.

The preconditions of these actions are as follows: $\mathrm{RMVMSG}(agt, msgId)$ is possible in s iff agt has received a message with the given ID and not yet processed it in s:

$$Poss(\mathrm{RMVMSG}(agt, msgId), s) \Leftrightarrow \\ \exists sender, msg\ \mathrm{MSGRCVD}(agt, sender, msgId, msg, s).$$

$\mathrm{SENDMSG}(agt, rcpt, msg)$ and $\mathrm{SENSEMSG}(agt)$ 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 MSGRCVD and MSGCTR fluents:

$$\begin{array}{l} Poss(act, s) \Rightarrow \\ [\mathrm{MSGRCVD}(agt, sndr, mId, m, do(act, s)) \Leftrightarrow \\ \ \exists m(act = \mathrm{SENDMSG}(sndr, agt, m) \wedge \mathrm{MSGCTR}(s) = mId \\ \ \ \vee \mathrm{MSGRCVD}(agt, sndr, mId, m, s) \wedge act \neq \mathrm{RMVMSG}(agt, mId)]. \end{array}$$

$$\begin{array}{l} Poss(act, s) \Rightarrow \\ [\mathrm{MSGCTR}(do(act, s)) = n \Leftrightarrow \\ \ \exists agt, agt', m(act = \mathrm{SENDMSG}(agt, agt', m) \wedge \mathrm{MSGCTR}(s) = n - 1 \\ \ \ \vee \mathrm{MSGCTR}(s) = n \wedge \neg\exists agt, agt', m\, act = \mathrm{SENDMSG}(agt, agt', m)]. \end{array}$$

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

proc $\text{INFORM}(agt, agt', \phi)$
$\quad$ **Know**$(agt, \phi)?;\ \text{SENDMSG}(agt, agt', \ulcorner\text{INFORM}(\phi)\urcorner)$
end

proc $\text{INFORMWHETHER}(agt, agt', \phi)$
$\quad \text{INFORM}(agt, agt', \phi) \mid \text{INFORM}(agt, agt', \neg\phi)$
$\quad\ \mid \text{INFORM}(agt, agt', \neg\textbf{KWhether}(agt, \phi))$
end

proc $\text{REQUEST}(agt, agt', \sigma)$
$\quad \text{SENDMSG}(agt, agt', \ulcorner\text{REQUEST}(\sigma)\urcorner)$
end

proc $\text{QUERYWHETHER}(agt, agt', \phi)$
$\quad \text{REQUEST}(agt, agt', \text{INFORMWHETHER}(agt', agt, \phi))$
end

Note that the above definitions use quotation.

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 situation s, every INFORM message sent to $rcpt$ by $sender$ was truthfully sent:

$$\text{SINCERE}(sender, rcpt, s) \stackrel{\text{def}}{=}$$
$$\forall s'[do(\text{SENDMSG}(sender, rcpt, \ulcorner\text{INFORM}(\phi)\urcorner), s') \leq s \Rightarrow \textbf{Know}(sender, \phi, s')].$$

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 he had no messages initially and that the sender has been sincere with him over that period:

Proposition

$$\textbf{Know}(rcpt, \neg\exists sndr, mId, m\ \text{MSGRCVD}(rcpt, sndr, mId, m, now), S_0) \wedge$$
$$\textbf{Know}(rcpt, \text{SINCERE}(sndr, rcpt, now),$$
$$\quad do(\text{SENSEMSGS}(rcpt), do(\text{SENDMSG}(sndr, rcpt, \ulcorner\text{INFORM}(\phi)\urcorner), S_0))) \Rightarrow$$
$$\textbf{Know}(rcpt, \exists s'[s' \leq now \wedge \textbf{Know}(sndr, \phi, s')],$$
$$\quad do(\text{SENSEMSGS}(rcpt), do(\text{SENDMSG}(sndr, rcpt, \ulcorner\text{INFORM}(\phi)\urcorner), S_0)))$$

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 [27, 28] 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.

7 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 (2). For RMVFROMSCHED, 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:

$$\begin{aligned}
&\mathit{Poss}(\textsc{rmvFromSched}(agt, user, period), s) \Leftrightarrow \\
&\quad agt = \textsc{scheduleManager}(user) \wedge \\
&\quad \exists activ, org\ \textsc{Schedule}(user, period, activ, org, s)
\end{aligned}$$

The effects of these actions on the SCHEDULE fluent are captured in the successor state axiom given earlier (2).

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 two abbreviations. First, it uses an iteration construct **for** $x : \phi(x)[\delta(x)]$ that performs $\delta(x)$ for all x's such that $\phi(x)$ holds (at the beginning of the loop).[10] Secondly, it uses the abbreviation:

$$\begin{aligned}
&\textsc{AgreedToMeet}(participant, requester, period, organizer, s) \stackrel{\text{def}}{=} \exists s'(\\
&\mathit{do}(\textsc{addToSchedule}(\textsc{smp}, participant, period, \textsc{meeting}, organizer), s') \leq s \\
&\wedge\ \mathbf{Know}(\textsc{smp}, \\
&\quad \exists mId\ \textsc{MsgRcvd}(\textsc{smp}, requester, mId, \ulcorner\textsc{request}(\textsc{smp}, \\
&\qquad \textsc{addToSchedule}(\textsc{smp}, participant, period, \textsc{meeting}, organizer))\urcorner, \\
&\qquad \mathit{now}), \\
&\quad s')),
\end{aligned}$$

where $\textsc{smp} \stackrel{\text{def}}{=} \textsc{scheduleManager}(participant)$.

[10] **for** $x : \phi(x)[\delta(x)]$ is defined as:

[**proc** $P(Q)$ /* where P is a new predicate variable */
  if $\exists y\, Q(y)$ **then** $\pi\, y, R\, [Q(y) \wedge \forall z(R(z) \Leftrightarrow Q(z) \wedge z \neq y)?; \delta(y); P(R)]$ **endIf**
endProc;
$\pi\, Q\, [\forall z(Q(z) \Leftrightarrow \phi(z))?; P(Q)]$]

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

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

Thus, we take the participant to have agreed to a meeting request iff at some prior situation, the participant's schedule manager added the meeting to his/her schedule while knowing that it had received the request.[11]

Meanwhile, users' "schedule manager" agents run the procedure in figure 2[12]. Because these agents are "event-driven", we program them as a set of interrupts running concurrently. Interrupts handling "more urgent" events are assigned a higher priority. For instance in the example, requests to remove an activity from the schedule are handled at the highest priority in order to minimize the chance of scheduling conflicts. Note that the procedure given 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

[11] This is quite a simplistic way of modeling agreement to a request. We should for instance, talk about the most recent instance of the request.

[12] Here we use the abbreviation $\langle\ \vec{x}\ :\ \phi \rightarrow \sigma\ \rangle \stackrel{\text{def}}{=} \langle\ \exists \vec{x}\, \phi \rightarrow \pi\vec{x}.[\phi?; \sigma]\ \rangle$.

proc $\text{MANAGESCHEDULE}(agt, user)$
$\quad \text{SENSEMSGS}(agt);$
$\quad (\,\langle\; requester, msgId, period\;:$
$\qquad \text{MSGRCVD}(agt, requester, msgId,$
$\qquad\qquad\qquad ⌜\text{REQUEST}(\text{RMVFROMSCHEDULE}(agt, user, period))⌝) \rightarrow [$
$\qquad$ **if** $agt = \text{SCHEDULEMANAGER}(user) \wedge$
$\qquad\quad \exists activ\; \text{SCHEDULE}(user, period, activ, \text{OWNER}(requester))$
$\qquad\quad$ **then** $\text{RMVFROMSCHEDULE}(agt, user, period)$ **endIf**;
$\qquad \text{RMVMSG}(agt);$
$\qquad \text{SENSEMSGS}(agt)]\;\rangle$
$\quad \rangle\!\rangle$
$\quad \langle\; requester, msgId, period, activ, organizer\;:$
$\qquad \text{MSGRCVD}(agt, requester, msgId, ⌜\text{REQUEST}(\text{ADDTOSCHEDULE}($
$\qquad\quad agt, user, period, activ, organizer))⌝) \rightarrow [$
$\qquad$ **if** $\text{PERMITTEDTOADDTOSCHED}(agt, organizer) \wedge$
$\qquad\quad Poss(\text{ADDTOSCHEDULE}(agt, user, period, activ, organizer))$
$\qquad\quad$ **then** $\text{ADDTOSCHEDULE}(agt, user, period, activ, organizer)$ **endIf**;
$\qquad \text{RMVMSG}(agt);$
$\qquad \text{SENSEMSGS}(agt)]\;\rangle$
$\quad \|$
$\quad \langle\; queryer, msgId, p\;:$
$\qquad \text{MSGRCVD}(agt, queryer, msgId, ⌜\text{QUERYWHETHER}(p)⌝) \rightarrow [$
$\qquad \text{INFORMWHETHER}(agt, queryer, p);$
$\qquad \text{RMVMSG}(agt);$
$\qquad \text{SENSEMSGS}(agt)]\;\rangle$
$\quad \|$
$\quad \langle\; \exists informer, msgId, p \text{MSGRCVD}(agt, informer, msgId, ⌜\text{INFORM}(p)⌝) \rightarrow [$
$\qquad$ % if message is $\text{INFORM}(p)$, nothing to do
$\qquad \text{RMVMSG}(agt);$
$\qquad \text{SENSEMSGS}(agt)]\;\rangle$
$\quad \rangle\!\rangle$
$\quad \langle$ **True** $\rightarrow [\qquad$ % if no new messages
$\qquad \text{SENSEMSGS}(agt)]\;\rangle)$
endProc

Figure 2. Procedure run by the “schedule manager” agents.

program to the ConGolog interpreter:

$$\begin{array}{l}\text{MANAGESCHEDULE}(\text{SM}_1, \text{USER}_1) \parallel \\ \text{MANAGESCHEDULEG}(\text{SM}_2, \text{USER}_2) \parallel \\ \text{MANAGESCHEDULE}(\text{SM}_3, \text{USER}_3) \parallel \\ \text{SCHEDULEMEETING}(\text{MS}_1, \text{USER}_1, \{\text{USER}_1, \text{USER}_3\}, \text{NOON}) \parallel \\ \text{SCHEDULEMEETING}(\text{MS}_2, \text{USER}_2, \{\text{USER}_2, \text{USER}_3\}, \text{NOON}).\end{array}$$

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.

8 IMPLEMENTATION AND EXPERIMENTATION

Prototype interpreters have been implemented in Prolog for both Golog [13] and ConGolog [3]. The implementations require that the program's precondition axioms, successor state axioms, and axioms about the initial situation be expressible as Prolog clauses. This is a limitation of the implementations, not the theory.

For programs that are embedded in real environments and run for long periods or perform sensing, it is often advantageous to interleave the interpretation of the program with its execution. In the current implementation, whenever the interpreter reaches a sensing action, it commits to the primitive actions generated so far and executes them, performs the sensing action, and generates exogenous actions to appropriately update the values of the sensed fluent. One can also add directives to programs to force the interpreter to commit when it gets to that point in the program. As well, whenever the interpreter commits and executes part of the program, it rolls its database forward to reflect the execution of the actions, and the situation reached behaves like a new initial situation [14, 16].[13]

Note however, that commiting to a sequence of action as soon as a sensing action is reached could lead to problems when the program to be executed is nondeterministic. Perhaps we are on a branch that does not lead to a final situation. To avoid this, we need to lookahead over sensing actions and generate a kind of conditional plan that is guaranteed to lead to a final situation no matter how the sensing turns out. A prototype interpreter that does this kind of lookahead has been implemented. The account of planning in the presence of sensing developed in [12] clarifies these issues. A general account of when an agent *knows how* to execute a program (i.e., of the knowledge preconditions of actions) has also been developed [10]. There are still some discrepancies between the Golog implementation and our theory of

[13]To evaluate whether a condition holds in a given situation, Golog regresses the condition to the initial situation and then uses the axioms about the initial situation to evaluate the regressed condition. This becomes less efficient as the number of action grows. After a while it becomes necessary to roll the database forward.

agency in the way knowledge, sensing, exogenous events, and the relation between planning and execution are treated. We are working to bridge this gap.

Experiments in the use of Golog and ConGolog to develop various applications have been conducted. Our longest running application project is in the area of robotics. High-level controllers have been programmed in Golog and ConGolog to get a robot to perform mail delivery in an office environment [9]. These have been used to drive RWI-B21 robots at the University of Toronto and the University of Bonn and a Nomad200 robot at York University.

Our first multi-agent application involved a personal banking assistant system [11, 25]. Users can perform transactions (in a simulated financial environment) using the system and have it monitor their financial situation for particular conditions and take action when they arise, either by notifying them or by performing transactions on their behalf. The system was implemented as a collection of Golog agents that communicate using TCP/IP.

A version of the meeting scheduling application described in this paper has also been implemented. The agents are realized as a set of processes in a single ConGolog program. Associated tcl/tk processes are used to implement the agents' user interfaces. The behavior of the implemented meeting scheduler agent is more sophisticated than that of the simple meeting scheduler in the previous section. When a meeting request is rejected by one of the participants' agent, the meeting scheduler agent will collect information about the schedule of all users involved in the conflict and plan a set of rescheduling actions that would resolve it; if a plan is found, the meeting scheduler will then request the agents involved to actually perform the rescheduling actions.

Another project under way involves developing ConGolog-based tools for modeling business and organizational processes [32]. In contrast to the operational view of conventional modeling tools, ConGolog takes a logical view of processes. This should prove advantageous when it comes to modeling system behavior under incompletely known conditions and proving properties about the system. So far we have used ConGolog to model a simple mail order business, as well as a section of a nuclear power plant; in the latter we model potential faults that can occur and how the operators deal with them.

9 DISCUSSION

One project that is closely related to ours is work on the AGENT-0 programming language [29]. 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 [5]. 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 [30], 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 [24], which allows knowledge to be attributed to agents without any commitment to a symbolic architecture.

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 agent design tools that are based on a solid theoretical foundation. We think the framework we have developed 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 supporting deliberation in the presence of sensing and exogenous actions, as well as the interleaving of deliberation with execution. We are also looking at mechanisms to facilitate the use of Golog nondeterminism for planning. Also under investigation are issues such as handling uncertainty and belief revision, as well as agent-relative (indexical) representations for robotics. A version of Golog that supports temporal constraints is also under development.

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 actions 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 [27, 28, 10], 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 hope to examine 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] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. To appear in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, August,, 1997.

[3] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus (extended version). In preparation, 1997.

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

[5] 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.

[6] C.C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 183–205. American Elsevier, New York, 1969.

[7] M. Hennessy. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.

[8] D. Leivant. Higher order logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 229–321. Clarendon Press, 1994.

[9] 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.

[10] Yves Lespérance, Hector J. Levesque, Fangzhen Lin, and Richard B. Scherl. Ability and knowing how in the situation calculus. In preparation, 1997.

[11] Yves Lespérance, Hector J. Levesque, and Shane J. Ruman. An experiment in using Golog to build a personal banking assistant. In Lawrence Cavedon, Anand Rao, and Wayne Wobcke, editors, *Intelligent Agent Systems: Theoretical and Practical Issues (Based on a Workshop Held at PRICAI '96 Cairns, Australia, August 1996),*, volume 1209 of *LNAI*, pages 27–43. Springer-Verlag, 1997.

[12] Hector J. Levesque. What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1139–1146, Portland, OR, August 1996.

[13] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(59–84), 1997.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Dept. Aarhus Univ. Denmark, 1981.

[20] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. of the 5th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 2–13, 1996.

[21] 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.

[22] Raymond Reiter. Proving properties of states in the situation salculus. *Artificial Intelligence*, pages 337–351, December 1993.

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

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

[25] Shane J. Ruman. GOLOG as an agent-programming language: Experiments in developing banking applications. Master's thesis, Department of Computer Science, University of Toronto, 1996.

[26] 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.

[27] 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*, pages 117–122, Cambridge, MA, November 1995.

[28] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. Specifying communicative multi-agent systems with ConGolog. In *Working Notes of the AAAI Fall 1997 Symposium on Communicative Action in Humans and Machines*, Cambridge, MA, November 1997. To appear.

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

[30] 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.

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

[32] Eric K.S. Yu, John Mylopoulos, and Yves Lespérance. AI models for business process reengineering. *IEEE Expert*, 11:16–23, August 1996.