GOLOG: A LOGIC PROGRAMMING LANGUAGE FOR DYNAMIC DOMAINS

HECTOR J. LEVESQUE, RAYMOND REITER, YVES LESPÉRANCE,* FANGZHEN LIN,† AND RICHARD B. SCHERL‡

▷ This paper proposes a new logic programming language called GOLOG whose interpreter automatically maintains an explicit representation of the dynamic world being modeled, on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the world. This allows programs to reason about the state of the world and consider the effects of various possible courses of action before committing to a particular behavior. The net effect is that programs may be written at a much higher level of abstraction than is usually possible. The language appears well suited for applications in high level control of robots and industrial processes, intelligent software agents, discrete event simulation, etc. It is based on a formal theory of action specified in an extended version of the situation calculus. A prototype implementation in Prolog has been developed. © Elsevier Science Inc., 1997 ◁

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.

Address correspondence to Department of Computer Science, University of Toronto, Toronto, ON, M5S 1A4 Canada, Email: {hector, reiter, lesperan, fl, scherl}@cs.toronto.edu, http://www.cs.toronto.edu/ ˜cogrobo/.

*Current address: Department of Computer Science, Glendon College, York University, 2275 Bayview Ave., Toronto, ON, Canada M4N 3M6.

†Current address: Department of Computer Science, Hong Kong University of Science and Technology, Kowloon, Hong Kong.

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

Received September 1995; accepted October 1996.

1. INTRODUCTION

Computer systems are often embedded in complex environments with which they interact. In programming such applications, the designer normally has an elaborate mental model of the environment and how the system's action will change the environment's state. Users of the system also have this kind of mental model. Typically, however, the system itself does not maintain an explicit model of the world it is operating in. This can make life difficult both for programmers and users —they may end up having to reconstruct the model being used, as there is no way for the system to explain or justify its behavior. But more importantly, this makes it difficult to reconfigure or extend the system by giving it "high-level" instructions, since it has no understanding at all of what it is doing.[1]

In this paper, we propose a programming language for such systems, whose design is based on a sophisticated logic of action. The interpreter for the language automatically maintains an explicit model of the system's environment and capabilities, which can be queried and reasoned with at run time. This allows complex behaviors to be defined at a much higher level of abstraction than would be possible otherwise. The language appears to be a distinct improvement over current technology for application such as: high-level control of robots and mechanical devices, programming intelligent software agents, modeling and simulation of discrete event systems, etc.

In the next section, we outline the theory of action on which our language is based. Then, we show how complex actions can be defined in the framework and explain how the resulting set of complex action expressions can be viewed as a programming language. In Section 4, we illustrate how our language is used through an example: a simple elevator controller. In the following section, we describe an implementation of the language, and sketch what experimental applications have been developed. Section 6 discusses the main distinguishing characteristics of the language. We conclude by summarizing the main features of our proposal, discussing its limitations, and outlining ongoing work that seeks to address these. The presentation throughout is informal in nature; in a companion paper [14], we explore the more formal aspects of this work.

2. INFORMAL INTRODUCTION TO THE SITUATION CALCULUS

To obtain the benefits mentioned in the Introduction, it is necessary to explicitly model how the world changes as a result of performing actions. There are a variety of ways of doing this, and we use the language of the situation calculus.

2.1. *Intuitive Ontology for the Situation Calculus*

The situation calculus (McCarthy [20]) is a first order language (with, as we shall see later, some second order features) specifically designed 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

[1] A similar view is advanced in Dixon [3].

the *initial situation*, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol *do*; $do(\alpha, s)$ denotes the successor situation to s resulting from performing the action α. Actions may be parameterized. For example, $put(x, y)$ might stand for the action of putting object x on object y, in which case $do(put(A, B), s)$ denotes that situation resulting from placing A on B when the world is in situation s. Notice that in the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, $do(putdown(A), do(walk(L), do(pickup(A), S_0)))$ is a situation denoting the world history consisting of the sequence of actions [pickup(A), walk(L), putdown(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 actions 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, $is\_carrying(robot, p, s)$, meaning that a robot is carrying package p in situation s, is a relational fluent. 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 $loc(robot, s)$, i.e., the robot's location in situation s.

2.2. Axiomatizing Actions and Their Effects in the Situation Calculus

Actions have *preconditions*—necessary and sufficient conditions that characterize when the action is physically possible. For example, in a blocks world, we might have:[2]

$$Poss(pickup(x), s)$$
$$\equiv [(\forall z) \neg holding(z, s)] \wedge nexto(x, s) \wedge \neg heavy(x).$$

World dynamics are specified by *effect axioms*. These describe the effects of a given action on the fluents—the causal laws of the domain. For example, a robot dropping a fragile object causes it to be broken:

$$Poss(drop(r, x), s) \wedge fragile(x, s) \supset broken(x, do(drop(r, x), s)). \tag{2.1}$$

Exploding a bomb next to an object causes it to be broken:

$$Poss(explode(b), s) \wedge nexto(b, x, s) \supset broken(x, do(explode(b), s)). \tag{2.2}$$

A robot repairing an object causes it to be not broken:

$$Poss(repair(r, x), s) \supset \neg broken(x, do(repair(r, x), s)). \tag{2.3}$$

2.3. The Frame Problem

As first observed by McCarthy and Hayes [20], axiomatizing a dynamic world requires more than just action preconditions and effect axioms. So-called *frame axioms* are also necessary. These specify the action *invariants* of the domain,

[2] In formulas, free variables are considered to be universally quantified from the outside. This convention will be followed throughout the paper.

namely, those fluents which remain unaffected by a given action. For example, a robot dropping things does not affect an object's color:

$$Poss(drop(r,x),s) \wedge color(y,s) = c \supset color(y, do(drop(r,x),s)) = c.$$

A frame axiom describing how the fluent *broken* remains unaffected:

$$Poss(drop(r,x),s) \wedge \neg broken(y,s) \wedge [y \neq x \vee \neg fragile(y,s)]$$
$$\supset \neg broken(y, do(drop(r,x),s)).$$

The problem introduced by the need for such frame axioms is that we can expect a vast number of them. Only relatively few actions will affect the truth value of a given fluent; all other actions leave the fluent unchanged. For example, an object's color is not changed by picking things up, opening a door, going for a walk, electing a new prime minister of Canada, etc. This is problematic for the axiomatizer who must think of all these axioms; it is also problematic for the theorem proving system, as it must reason efficiently in the presence of so many frame axioms.

2.3.1. What Counts as a Solution to the Frame Problem? Suppose the person responsible for axiomatizing an application domain has specified all of the causal laws for the world being axiomatized. More precisely, she has succeeded in writing down *all* the effect axioms, i.e., for each fluent F and each action A which can cause F's truth value to change, axioms of the form

$$Poss(A,s) \wedge R(\vec{x},s) \supset (\neg)F(\vec{x}, do(A,s)).$$

Here, R is a first order formula specifying the contextual conditions under which the action A will have its specified effect on F.

A solution to the frame problem is a systematic procedure for generating, from these effect axioms, all the frame axioms. If possible, we also want a *parsimonious* representation for these frame axioms (because in their simplest form, there are too many of them).

2.4. A Simple Solution to the Frame Problem

By appealing to earlier ideas of Haas [7], Schubert [29] and Pednault [21], Reiter [23] proposes a simple solution to the frame problem, which we illustrate with an example. Suppose that (2.1), (2.2), and (2.3) are all the effect axioms for the fluent *broken*, i.e., they describe all the ways that an action can change the truth value of *broken*. We can rewrite (2.1) and (2.2) in the logically equivalent form:

$$\begin{aligned} &Poss(a,s) \wedge [(\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \\ &\qquad \vee (\exists b)\{a = explode(b) \wedge nexto(b,x,s)\}] \\ &\supset broken(x, do(a,s)). \end{aligned} \tag{2.4}$$

Similarly, consider the negative effect axiom (2.3) for *broken*; this can be rewritten as:

$$Poss(a,s) \wedge (\exists r)a = repair(r,x) \supset \neg broken(x, do(a,s)). \tag{2.5}$$

In general, we can assume that the effect axioms for a fluent F have been written in the forms:

$$Poss(a,s) \wedge \gamma_F^+(\vec{x},a,s) \supset F(\vec{x}, do(a,s)), \tag{2.6}$$

$$Poss(a,s) \wedge \gamma_F^-(\vec{x},a,s) \supset \neg F(\vec{x}, do(a,s)). \tag{2.7}$$

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 to become true in the successor situation $do(a,s)$; similarly $\gamma_F^-(\vec{x},a,s)$ describes the conditions under which performing a in s results in F becoming false in the successor situation. The solution to the frame problem of [23] rests on a *completeness assumption*, which is that the causal axioms (2.6) and (2.7) characterize all the conditions under which action a can lead to a fluent $F(\vec{x})$ becoming true (respectively, false) in the successor situation. In other words, axioms (2.6) and (2.7) describe all the causal laws affecting the truth values of the fluent F. 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*. Reiter [23] shows how to derive a *successor state axiom* of the following form from the causal axioms (2.6) and (2.7) and the completeness assumption.

Successor State Axiom

$$Poss(a,s) \supset \left[F(\vec{x}, do(a,s)) \equiv \gamma_F^+(\vec{x},a,s) \vee \left(F(\vec{x},s) \wedge \neg\gamma_F^-(\vec{x},a,s)\right)\right].$$

This simple axiom embodies a solution to the frame problem. Notice that this axiom universally quantifies over actions a. In fact, this is one way in which a parsimonious solution to the frame problem is obtained.

Applying this to our example about breaking things, we obtain the following successor state axiom:

$$\begin{aligned} Poss(a,s) \supset [&broken(x, do(a,s)) \\ &\equiv (\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \\ &\vee (\exists b)\{a = explode(b) \wedge nexto(b,x,s)\} \\ &\vee broken(x,s) \wedge \neg(\exists r) a = repair(r,x)]. \end{aligned}$$

It is important to note that the above solution to the frame problem presupposes that there are no *state constraints*, as for example in the blocks world constraint: $(\forall s).on(x,y,s) \supset \neg\, on(y,x,s)$. Such constraints sometimes implicitly contain effect axioms (so-called indirect effects), in which case the above completeness assumption will not be true. The assumption that there are no state constraints in the axiomatization of the domain will be made throughout this paper. In [17, 15], the approach discussed in this section is extended to deal with state constraints, by compiling their effects into the successor state axioms.

2.5. Axiomatizing an Application Domain in the Situation Calculus

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

- Action precondition axioms, one or each primitive action.
- Successor state axioms, one for each fluent.

- Unique names axioms for the primitive actions.
- Axioms describing the initial solution—what is true initially, before any actions have occurred. This is any finite set of sentences which mention no situation term, or only the situation term S_0.
- Foundational, domain independent axioms for the situation calculus. These include unique names axioms for situations, and an induction axiom. Since these play no special role in this paper, we omit them. For details, and for their metamathematical properties, see Lin and Reiter [17] and Reiter [24].

3. COMPLEX ACTIONS, PROCEDURES, AND GOLOG

The previous section outlines a situation calculus-based approach for representing, and reasoning about, simple actions. It fails to address the problem of expressing, and reasoning with, complex actions and procedures, for example:

- **if** *car_in_driveway* **then** *drive* **else** *walk* **endIf**
- **while** $(\exists block)$ *ontable*$(block)$ **do** *remove_a_block* **endWhile**
- **proc** *remove_a_block* $(\pi x)[$*pickup*(x); *putaway*$(x)]$ **endProc**

Here, we have introduced a procedure declaration (*remove_a_block*), and also the nondeterministic operator π; $(\pi x)[\delta(x)]$ means nondeterministically pick an individual x, and for that x, perform $\delta(x)$. We shall see later that this kind of nondeterminism is very useful for robotics and similar applications.

3.1. Complex Actions and Procedures in the Situation Calculus

Our approach will be to define complex action expressions using some additional extralogical symbols (e.g., **while**, **if**, etc.) which act as *abbreviations* for logical expressions in the language of the situation calculus. These extralogical expressions should be thought of as *macros* which expand into genuine formulas of the situation calculus. So below, we define the abbreviation $Do(\delta, s, s')$, where δ is a complex action expression; intuitively, $Do(\delta, s, s')$ will hold whenever the situation s' is a terminating situation of an execution of complex action δ starting in situation s. Note that our complex actions may be nondeterministic, that is, may have several different executions terminating in different situations.

Do is defined intuitively on the structure of its first argument as follows:

1. *Primitive actions*:

$$Do(a, s, s') \overset{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s).$$

By the notation $a[s]$ we mean the result of restoring the situation arguments to any functional fluents mentioned by the action term a (see the next item immediately below). For example, if a is *read*(*favorite_book*(*John*)), and if *favorite_book* is a functional fluent (which means that its value is situation dependent) then $a[s]$ is *read*(*favorite_book*(*John*, s)).

2. *Test actions*:

$$Do(\phi?, s, s') \overset{def}{=} \phi[s] \wedge s = s'.$$

Here, ϕ is a pseudo-fluent expression (not a situation calculus formula) which stands for a formula in the language of the situation calculus, but with all situation arguments suppressed. $\phi[s]$ denotes the situation calculus formula obtained from ϕ by restoring situation variable s as the suppressed situation argument for all fluent names (relational and functional) mentioned in ϕ.

Examples: If ϕ is

$$(\forall x).ontable(x) \wedge \neg on(x, A),$$

then $\phi[s]$ stands for

$$(\forall x).ontable(x, s) \wedge \neg on(x, A, s).$$

If ϕ is

$$(\exists x) on(x, favorite\_block(Mary)),$$

then $\phi[s]$ stands for

$$(\exists x) on(x, favorite\_block(Mary, s), s).$$

3. *Sequence*:

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

4. *Nondeterministic choice of two actions*:

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

5. *Nondeterministic choice of action arguments*:

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

6. *Nondeterministic iteration*: Execute δ zero or more times.

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

In other words, doing action δ zero or more times takes you from s to s' iff (s, s') is in every set (and therefore, the smallest set) such that:

(a) (s_1, s_1) is in the set for all situations s_1.

(b) Whenever (s_1, s_2) is in the set, and doing δ in situation s_2 takes you to situation s_3, then (s_1, s_3) is in the set.

The above definition of nondeterministic iteration utilizes the standard second order way of expressing this set. Some appeal to second order logic appears necessary here because transitive closure is not first order definable, and nondeterministic iteration appeals to this closure.

Conditionals and white-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:

The difficulty with giving a situation calculus semantics for recursive procedure calls using macro expansion is that there is no straightforward way to macro expand a procedure body when that body includes a recursive call to itself.

1. We begin with an auxiliary macro definition: For any predicate symbol P of arity $n+2$, taking a pair of situation arguments:

$$Do(P(t_1,\dots,t_n),s,s') \overset{def}{=} P(t_1[s],\dots,t_n[s],s,s').$$

In what follows, expressions of the form $P(t_1,\dots,t_n)$ occurring in programs will serve as procedure calls, and we will understand $Do(P(t_1,\dots,t_n),s,s')$ to mean that executing the procedure P on actual parameters $t_1,\dots,t_n$ causes a transition from situation s to s'. Notice that in the macro expansion, the actual parameters (t_i) are first evaluated with respect to the current situation s $(t_i[s])$ before passing them to the procedure P, so the procedure mechanism we are defining is *call by value*. Because we now want to include procedure calls among our actions, we extend the definition of complex actions to consist of any expression that can be constructed from primitive actions and procedure calls using the complex action constructors of 1–6 above.

2. Next, we define a situation calculus semantics for *programs* involving (recursive) procedures. We suppose, in the standard block-structured programming style, that a GOLOG program consists of a sequence of declarations of procedures $P_1,\dots,P_n$, with formal parameters $\vec{v}_1,\dots,\vec{v}_n$ and procedure bodies $\delta_1,\dots,\delta_n$ respectively, followed by a main program body δ_0. Here, $\delta_1,\dots,\delta_n,\delta_0$ are complex actions, extended by actions for procedure calls, as described in 1 above. So a GOLOG program will have the form:

$$\textbf{proc } P_1(\vec{v}_1)\,\delta_1 \textbf{ endProc};\dots;\textbf{ proc } P_n(\vec{v}_n)\,\delta_n \textbf{ endProc};\ \delta_0.$$

We define the result of evaluating a program of this form as follows:

$$Do(\{\textbf{proc } P_1(\vec{v}_1)\,\delta_1 \textbf{ endProc};\dots;\textbf{ proc } P_n(\vec{v}_n)\,\delta_n \textbf{ endProc};\ \delta_0\},s,s')$$
$$\overset{def}{=} (\forall P_1,\dots,P_n).\left[\bigwedge_{i=1}^{n}(\forall s_1,s_2,\vec{v}_i).Do(\delta_i,s_1,s_2)\supset Do(P_i(\vec{v}_i),s_1,s_2)\right]$$
$$\supset Do(\delta_0,s,s').$$

This is the situation calculus definition corresponding to the more usual Scott-Strachey *least fixed-point* definition in standard programming language semantics (Stoy [32]).[3]

Examples:

1. Given that *down* means move an elevator down one floor, define $d(n)$, meaning move an elevator down n floors.

$$\textbf{proc } d(n)\,(n=0)?\,|\,d(n-1);\ down \textbf{ endProc}.$$

[3] By using programs as above within the bodies of other procedures, we obtain the tree-structured nesting of procedures typical of Algol-like languages. Moreover, we get the lexical scoping rules of these languages for free from our use of the quantifiers in the definition of *Do*.

2. Parking an elevator on the ground floor:

   proc $park\ (\pi\, m)[atfloor(m)?; d(m)]$ **endProc**.

3. Define *above* to be the test action which is the transitive closure of *on*.

   proc $above(x,y)(x=y)?|(\pi\, z)[on(x,z)?; above(z,y)]$ **endProc**.

4. *clean* means put away all the blocks into the box.

   proc $clean\ (\forall x)[block(x) \supset in(x, Box)]?|$

   $(\pi\, x)[(\forall y) \neg on(y,x)?; put(x, Box)]; clean$ **endProc**.

5. A GOLOG blocks world program consisting of three procedure declarations devoted to creating towers of blocks, and a main program which makes a seven block tower, while ensuring that block A is clear in the final situation.

```
proc maketower(n)        % Make a tower of n blocks.
  (πx, m)[tower(x, m) ?;  % tower(x, m) means that there is a tower
                          % of m blocks, whose top block is x.
   if m ≤ n then stack(x, n − m)
     else unstack(x, m − n)
   endIf]
endProc;

proc stack(x, n)     % Place n blocks on the tower whose top block is x.
  n = 0? | (π y)[put(y, x); stack(y, n − 1)]
endProc;

proc unstack(x, n)     % Remove n blocks from the tower
                       % whose top block is x.
  n = 0? | (π y)[on(x, y)?; movetotable(x); unstack(y, n − 1)]
endProc;
% main: create a seven block tower, with A clear at the end.
maketower(7); ¬(∃x)on(x, A)?
```

Except for procedures, this formalization draws considerably from dynamic logic [5]. In effect, it reifies as situations in the object language of the situation calculus, the possible worlds with which the semantics of dynamics logic is defined. For a more technical treatment of this macro approach to complex actions, see Levesque, Lin, and Reiter [14].

3.2. *Why Macros?*

Programs and complex actions "macro expand" to (sometimes second order) formulas of the situation calculus; *complex behaviors are described by situation calculus formulas*. But why do we treat these as macros rather than as first class objects (terms) in the language of the situation calculus? To see why, consider the complex action

while $[(\exists block)\, ontable(block)]$ **do** *remove_a_block* **endWhile**.

Now ask what kind of thing is *ontable*(*block*)? It is not a fluent, since fluents take situations as arguments. But it is meant to stand for a fluent since the expression *ontable*(*block*) will be evaluated with respect to the current situation of the execution of the **while**-loop. To see what must happen if we avoid the macro approach, suppose we treat complex actions as genuine first order terms in the language of the situation calculus.

- We must augment this language with new distinguished function symbols ?, ;, |, π, and perhaps *while*, *if_then_else*.
- Moreover, since a **while**-loop is now a first order term, the p in *while*(p, a) must be a first order term also. But p can be any "formula" standing for a situation calculus formula, e.g., *ontable*(*block*), $(\exists x, y).ontable(x) \wedge \neg red(x) \vee on(x, y)$.
- So we must introduce new function symbols into the language; $\widehat{on}$, $\widehat{ontable}$, *and*, *or*, *exists*, *not* etc. (We need $\widehat{on}$ to distinguish it from the fluent *on*.) Now these "formulas" look like genuine terms:

  $$\widehat{ontable}(block),$$

  $$exists\Big(X, exists\Big(Y, or\Big(and\Big(\widehat{ontable}(X), not\big(\widehat{red}(X)\big)\Big), \widehat{on}(X, y)\Big)\Big)\Big).$$

  Notice that X and Y here must be constants. In other words, we must *reify* fluents and formulas about fluents whose situation arguments have been suppressed. This makes the resulting first order language much more complicated.
- Even worse, we must *axiomatize* the correspondence between these reified formulas and the actual situation calculus formulas they stand for. In the axioms for *Do*, such reified formulas get evaluated as

  $$Do(p?, s, s') \equiv apply(p, s) \wedge s = s'.$$

  Here, *apply*(p, s) is true iff the reified formula p, with its situation argument s restored (so that it becomes a genuine situation calculus formula), is true. So we have to axiomatize *apply*. These axioms are schemas over fluents F and reified formulas p, p_1, p_2 and the quantified "variables" X of these expressions.

  $$apply\big(\hat{F}(t_1, \ldots, t_n), s\big) \equiv F(apply1(t_1, s), \ldots, apply1(t_n, s), s),$$

  where *apply*1 restores situation arguments to functional fluents. Also needed are:

  $$apply(and(p_1, p_2), s) \equiv apply(p_1, s) \wedge apply(p_2, s),$$

  $$apply(or(p_1, p_2), s) \equiv apply(p_1, s) \vee apply(p_2, s),$$

  etc.

All of this would result in a much more complex theory. To avoid this technical clutter, we have chosen to take the above macro route in defining complex actions, and to see just how far we can push this idea. As we shall see, it is possible to develop a very rich theory of actions this way.

3.3. Programs as Macros: What Price Do We Pay?

By opting to define programs as macros, we obtain a much simpler theory than if we were to reify these actions. The price we pay for this is a less expressive formalism. For example, we cannot *quantify* over complex actions, since these are not objects in the language of the situation calculus. This means, for example, that we cannot synthesize programs using conventional theorem proving techniques, as in Manna and Waldinger [19]. In their approach to program synthesis, one would obtain a program satisfying the goal formula *Goal* as a side effect of proving the following entailment:

$$Axioms \models (\exists \delta, s).Do(\delta, S_0, s) \wedge Goal(s).$$

Here, *Axioms* are those described in Section 2.5. But the program to be synthesized is being existentially quantified in the theorem, so that this theorem cannot even be expressed in our language.

On the other hand, many other program properties are, in principle, provable with our formalism. Moreover, doing so is (conceptually) straightforward precisely because program executions are formulas of the situation calculus.

1. *Correctness*: To show that, whenever program δ terminates, it leads to a world situation satisfying property P:

   $$Axioms \models (\forall s).Do(\delta, S_0, s) \supset P(s).$$

   Or, the stronger

   $$Axioms \models (\forall s_0, s).Do(\delta, s_0, s) \supset P(s).$$

2. *Termination*: To show that program δ terminates:

   $$Axioms \models (\exists s) Do(\delta, S_0, s).$$

   Or, the stronger

   $$Axioms \models (\forall s_0)(\exists s) Do(\delta, s_0, s).$$

In other words, our macro account is well-suited to applications where a program δ is *given*, and the job is to prove it has some property. As we will see, the main property we have been concerned with is execution: given δ and an initial situation, find a terminating situation for δ, if one exists. To do so, we prove the termination of δ as above, and then extract from the proof a binding for the terminating situation.

3.4. GOLOG

The program and complex action expressions defined above can be viewed as a programming language whose semantics is defined via macro-expansion into sentences of the situation calculus. We call this language GOLOG, for "alGOl in LOGic." GOLOG appears to offer significant advantages over current tools for applications in dynamic domains like the high-level programming of robots and software agents, process control, discrete event simulation, etc. In the next section, we present a simple example.

4. AN ELEVATOR CONTROLLER IN GOLOG

Here we show how to axiomatize the primitive actions and fluents for a simple elevator, and we write a GOLOG program to control this elevator.

Primitive actions:

- *up*(n)—Move the elevator up to floor n.
- *down*(n)—Move the elevator down to floor n.
- *turnoff*(n)—Turn off call button n.
- *open*—Open the elevator door.
- *close*—Close the elevator door.

Fluents:

- $current\_floor(s) = n$—In situation s, the elevator is at floor n.
- $on(n, s)$—In situation s, call button n is on.
- $next\_floor(n, s)$—In situation s, the next floor to be served is n.

Primitive action preconditions:

$$Poss(up(n), s) \equiv current\_floor(s) < n.$$
$$Poss(down(n), s) \equiv current\_floor(s) > n.$$
$$Poss(open, s) \equiv true.$$
$$Poss(close, s) \equiv true.$$
$$Poss(turnoff(n), s) \equiv on(n, s).$$

Successor state axioms:

$$\begin{aligned} Poss(a, s) \supset [&current\_floor(do(a, s)) = m \\ &\equiv \{a = up(m) \lor a = down(m) \\ &\quad \lor \\ &\quad current\_loor(s) \\ &\quad = m \land \neg(\exists n)a = up(n) \land \neg(\exists n)a = down(n)\}]. \end{aligned}$$
$$Poss(a, s) \supset [on(m, do(a, s)) \equiv on(m, s) \land a \neq turnoff(m)].$$

A defined fluent.

$$\begin{aligned} &next\_floor(n, s) \equiv on(n, s) \\ &\quad \land (\forall m).on(m, s) \supset |m - current\_floor(s)| \geq |n - current\_floor(s)|. \end{aligned}$$

This defines the next floor to be served as a nearest floor to the one where the elevator happens to be.

The GOLOG procedures:

proc *serve*(n) *go_floor*(n); *turnoff*(n); *open*; *close* **endProc**.

proc *go_floor*(n) $(current\_floor = n)?|up(n)|down(n)$ **endProc**.

proc *serve_a_floor* $(\pi\, n)[next\_floor(n)?;\ serve\ (n)]$ **endProc**.

proc *control* [**while** $(\exists n)on(n)$ **do** *serve_a_floor* **endWhile**]; *park* **endProc**.

proc *park* **if** $current\_floor = 0$ **then** *open* **else** *down*(0); *open* **endIf endProc**.

Initial situation:

$$current\_floor(S_0) = 4, \qquad on(b, S_0) \equiv b = 3 \vee b = 5.$$

Notice that this last axioms specifies that, initially, buttons 3 and 5 are on, and moreover no other buttons are on. In other words, we have complete information initially about which call buttons are on. It is this completeness property of the initial situation which will justify, in part, the Prolog implementation described below in Section 5.

Running the program:

This is a theorem proving task; we need to establish the following entailment:

$$Axioms \models (\exists s) Do(control, S_0, s).$$[4]

Here, *Axioms* are those of Section 2.5. Notice especially what this entailment says, and why it makes sense.

- Although the expression $Do(control, S_0, s)$ looks like an atomic formula, *Do* is a macro not a predicate, and the expression stands for a much longer second order situation calculus sentence. This will mention only the primitive actions *up*, *down*, *turnoff*, *open*, *close* and the fluents *current_floor*, *on*, *next_floor*, as well as the distinguished situation calculus symbols *do*, S_0, *Poss*.
- Because this macro-expanded sentence is legitimate situation calculus, it makes sense to seek a proof of it from *Axioms*, which characterize the fluents and actions of this elevator world.

A successful "execution" of the program, i.e., a successful proof, might return the following binding for s:

$$s = do\Big(open, do\Big(down(0), do\Big(close, do\Big(open, do\Big(turnoff(5),$$
$$do\big(up(5), do\big(close, do\big(open, do\big(turnoff(3), do\big(down(3), S_0\big)\big)\big)\big)\big)\Big)\Big)\Big)\Big)\Big).$$

Such a binding represents an execution trace of the GOLOG program for the given description of the initial situation. This trace, namely, the action sequence

$$[down(3), turnoff(3), open, close, up(5), turnoff(5), open, close, down(0), open],$$

would next be passed to the elevator's execution module for controlling it in the physical world.

As one can see from the example, GOLOG is a logic programming language in the following sense:

1. Its interpreter is a general-purpose theorem prover. In its most general form, this must be a theorem prover for second order logic; in practice (see Section 6 below, and Levesque, Lin, and Reiter [14]), first order logic is sufficient for most purposes.
2. Like Prolog, GOLOG programs are executed for their side effects, namely, to obtain bindings for the existentially quantified variables of the theorem.

[4] Strictly speaking, we must prove the sentence $(\exists s)Do(\Pi; control, S_0, s)$ where Π is the sequence of procedure declarations just given. The call to *control* in this sentence serves as the main program. See the definition of GOLOG programs and their semantics in Section 3.1 above.

```
:- op(950, xfy, [#]). /* Nondeterministic action choice.*/

do([],S,S).        /* This clause and the next are for sequences */
do([E|L],S,S1) :- do(E,S,S2), do(L,S2,S1).
do(?(P),S,S) :- holds(P,S).
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
do(if(P,E1,E2),S,S1) :- do([?(P),E1] # [?(neg(P)),E2],S,S1).
do(star(E),S,S1) :- do([] # [E,star(E)],S,S1).
do(while(P,E),S,S1):- do([star([?(P),E]),?(neg(P))],S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).

/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */
sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

holds(and(P1,P2),S) :- holds(P1,S), holds(P2,S).
holds(or(P1,P2),S) :- holds(P1,S); holds(P2,S).
holds(neg(P),S) :- not holds(P,S).   /* Negation by failure */
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).
```

FIGURE 1. A GOLOG interpreter in CProlog.

5. IMPLEMENTATION AND EXPERIMENTATION

In this section, we discuss an implementation of the GOLOG language in Prolog. We begin by presenting a very simple version of this interpreter. We then show how the elevator example above would be written for this interpreter and some execution traces. We conclude by listing some of the applications currently being investigated in GOLOG.

5.1. An Interpreter

Given that the execution of GOLOG involves a finding a proof in second-order logic, it is perhaps somewhat surprising how easy it is to write a GOLOG interpreter. Figure 1 shows the entire program in CProlog.

The do predicate here takes 3 arguments: a GOLOG action expression, and terms standing for the initial and final situations. Normally, a query will be of the form do(e, **s0**, S), so that an answer will be a binding for the final situation S. In this implementation, a legal GOLOG action expression e is one of the following:

- $[e_1, \ldots, e_n]$, sequence.
- ?(p), where p is a condition (see below).
- $e_1 \# e_2$, nondeterministic choice of e_1 or e_2.
- if(p, e_1, e_2), conditional.

- star(e), nondeterministic repetition.
- while(p, e), iteration.
- pi(v, e) nondeterministic assignment, where v is an atom (standing for a GOLOG variable) and e is a GOLOG action expression that uses v.
- a, where a is the name of a user-declared primitive action or defined procedure (see below).

A condition p in the above is either a fluent or an expression of the form and(p_1, p_2), or(p_1, p_2), neg(p), or some(v, p), where v is an atom and p is a condition using v. In evaluating these conditions, the interpreter uses negation as failure to handle neg, and consults the user-supplied holds predicate to determine which fluents are true.

In this implementation, a GOLOG application (like the elevator, below) is expected to have the following parts:

1. A collection of clauses of the form primitive_action(*act*), declaring each primitive action.
2. A collection of clauses of the form **proc**(*name*, *body*) declaring each defined procedure (which can be recursive). The *body* here can be legal GOLOG action expression.
3. A collection of clauses which define the predicate poss(*act*, *situation*) for every primitive action and situation. Typically, this requires one clause per action, using a variable to range over all situations.
4. A collection of clauses which define the predicate holds(*fluent*, *situation*) for every fluent and situation. Normally, this is done in two parts:
   (a) A collection of clauses defining holds(*fluent*, **s0**), characterizing which fluents are true in the initial situation. The clauses need not be atomic, and can involve arbitrary Prolog computation for determining entailments of the initial database. We make the usual Prolog closed world assumption on this database.
   (b) A collection of clauses defining holds(*fluent*, do(*act*, *situation*)) for every combination of fluent, primitive action, and situation. Typically, this is done with a single clause for each fluent, with variables for the actions and situations. This amounts to writing the successor state axiom for the fluent.

While this interpreter might appear intuitively to be doing the right thing, at least in cases where the closed world assumption (CWA) is made, it turns out to be non-trivial to state precisely in what sense it is correct. On the one hand, we have the specification of *Do* as a formula in second order logic, and on the other, we have the above do predicate, characterized by a set of Horn clauses. The exact correspondence between the two depends on a number of factors, and we do not intend to discuss them here. For a formal statement and proof of correctness of this interpreter, the interested reader should consult the companion paper [14].

Given the simplicity of the characterization of the do predicate (in first-order Horn clauses), and the complexity of the formula that results from *Do* (in second-order logic), a reasonable question to ask is why we even bother with the latter. The answer is that the definition of do is too weak: it is sufficient for finding a terminating situation (when it exists) given an initial one,[5] but it cannot be used to show non-termination. Consider the program $\delta = [a^*; (x \neq x)?]$. For this program, we have that $\neg Do(\delta, s, s')$ is entailed for any s and s'; the do predicate, however, would simply run forever.

On the other hand, the semantics of Prolog is often formulated in terms of minimal models which, in the case of simple logic programs like the above interpreter, have a number of desirable features. Could we not use these ideas instead of second-order quantification to characterize GOLOG program execution? The answer is that we could, but only when the set of axioms characterizing the initial situation S_0 can be made part of a logic program. Our specification of *Do*, on the other hand, is fully general: it does exactly the right thing even when the axioms describing the initial situation contain disjunctions, existential quantifications, and so on. The semantics of logic programs can perhaps be generalized to accommodate such axioms, but it is not clear that the resulting specification would be much simpler than ours.

We emphasize that the above interpreter relies on the standard Prolog CWA that the initial database—the facts true in the initial situation S_0—is complete. This was the case for the logical specification of the elevator example of Section 4. For many applications, this is a reasonable assumption. For many others this is unrealistic, for example in a robotics setting in which the environment is not completely known to the robot. In such cases, a more general GOLOG interpreter is necessary. Such an interpreter might still make use of Prolog's backchaining mechanism to reduce queries about the current situation to queries about the initial situation. In other words, *regression*-based query evaluation (Waldinger [34], Pednault [21], Reiter [23]) can be implemented using Prolog. However, answering the regressed queries in the initial situation would require, in general, the full power of a first order theorem prover.

5.2. *The Elevator Example*

In Figure 2, we present clauses defining the previously discussed elevator example, and in Figure 3, we show some queries to the interpreter for this program.

In the first query, we ask the interpreter to pick a floor and turn off its call button. The answers show that there are only two ways to do this: either turn off floor 3 or turn off floor 5.

In the second query, we ask the interpreter to either turn off a call button or to go to a floor that satisfies the test **next_floor**. Since this predicate has been defined to hold only of those floors whose button is on, this gives us four choices: turn off floor 3 or 5, or go to floor 3 or 5.

In the final query, we call the main elevator controller, **control**, to serve all floors and then park the elevator. There are only two ways of doing this: serve floor 3 then 5 then park, or serve floor 5 then 3 then park. Note that we have not attempted to prune the backtracking to avoid duplicate answers.

[5] This needs to be hedged: the Prolog interpreter is sufficient only if we assume a breadth-first execution strategy. Otherwise, GOLOG programs like *park* in Section 3.1, which terminate according to *Do*, could cause do to run forever.

```
/* Primitive control actions */

primitive_action(turnoff(N)). /* Turn off call button N. */
primitive_action(open).       /* Open the elevator door. */
primitive_action(close).      /* Close the elevator door. */
primitive_action(up(N)).      /* Move the elevator up to floor N.*/
primitive_action(down(N)).    /* Move the elevator down to floor N.*/

/* Definitions of Complex Control Actions  */

proc(go_floor(N), ?(current_floor(N)) # up(N) # down(N)).
proc(serve(N), [go_floor(N), turnoff(N), open, close]).
proc(serve_a_floor, pi(n, [?(next_floor(n)), serve(n)])).
proc(park, if(current_floor(0), open, [down(0), open])).

/* control is the main loop. So long as there is an active call button,
   it serves one floor. When all buttons are off, it parks the elevator.   */

proc(control, [while(some(n, on(n)), serve_a_floor), park]).

/* Preconditions for Primitive Actions */

poss(up(N),S) :- holds(current_floor(M),S), M < N.
poss(down(N),S) :- holds(current_floor(M),S), M > N.
poss(open,S).
poss(close,S).
poss(turnoff(N),S) :- holds(on(N),S).

/* Successor state axioms for primitive fluents. */

holds(current_floor(M),do(E,S)) :- E = up(M) ; E = down(M) ;
              not E = up(N), not E = down(N), holds(current_floor(M),S).

holds(on(M),do(E,S)) :- holds(on(M),S), not E = turnoff(M).

/* Initial situation. Call buttons: 3 and 5. The elevator is at floor 4. */

holds(on(3),s0).   holds(on(5),s0).   holds(current_floor(4),s0).

/* next_floor(N) determines which of the active call buttons should be served
   next. Here, we simply choose an arbitrary active call button.   */

holds(next_floor(N),S) :- holds(on(N),S).
```

FIGURE 2. The elevator controller.

5.3. Experimentation

The actual implementation of GOLOG we have been using at the University of Toronto is in Quintus Prolog and incorporates a number of additional features for debugging and for efficiency beyond those of the simple interpreter presented here.

For example, one serious limitation of the style of interpreter presented here is the following: determining if some condition (like **current_floor(0)**) holds in a

```
?- do(pi(n,[?(on(n)),turnoff(n)]),s0,S).

S = do(turnoff(3),s0) ;

S = do(turnoff(5),s0) ;

no

- ----------------------------------------

?- do(pi(n, turnoff(n) # ([?(next_floor(n)),go_floor(n)])),s0,S).

S = do(turnoff(3),s0) ;

S = do(turnoff(5),s0) ;

S = do(down(3),s0) ;

S = do(up(5),s0) ;

no

- ----------------------------------------

?- do(control,s0,S).

S = do(open,do(down(0),do(close,do(open,do(turnoff(5),do(up(5),do(close,
do(open,do(turnoff(3),do(down(3),s0)))))))))) ;

S = do(open,do(down(0),do(close,do(open,do(turnoff(3),do(down(3),do(close,
do(open,do(turnoff(5),do(up(5),s0)))))))))) ;

S = do(open,do(down(0),do(close,do(open,do(turnoff(5),do(up(5),do(close,
do(open,do(turnoff(3),do(down(3),s0)))))))))) ;

S = do(open,do(down(0),do(close,do(open,do(turnoff(3),do(down(3),do(close,
do(open,do(turnoff(5),do(up(5),s0)))))))))) ;

no
```

FIGURE 3. Running the elevator program.

situation involves looking at what actions led to that situation, and unwinding these actions all the way back to the initial situation. This process is called *regression* in the AI planning literature. Doing this repeatedly with very long sequences of actions can take considerable time. Moreover, the Prolog terms representing situations that are far removed from the initial situation end up being gigantic.

However, it is possible in many cases to *progress* the initial database to handle this (Lin and Reiter [16, 18]). The idea is that the interpreter periodically "rolls the initial database forward" in response to the actions generated thus far during the evaluation of the program. This progressed database becomes the new initial database for the purposes of the continuing evaluation of the program. In this way, the interpreter maintains a database of just the current value of all fluents, and the distance from the initial situation is no longer a problem.

To evaluate our interpreter and the entire GOLOG framework, we have been experimenting with various types of applications. The most advanced involves a robotics application—mail delivery in an office environment [9]. The high-level controller of the robot programmed in GOLOG is interfaced to an existing robotics package that supports path planning and local navigation. The system currently works in simulation mode; experiments with a real robot have begun in collaboration with the robotics group at the University of Bonn.

Another application involves tools for home banking [27]. In this case, a number of software agents written in GOLOG 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.), communicating over TCP/IP.

CONGOLOG, a version of the language supporting concurrency (including interrupts, priorities, and support for exogenous actions) is also being implemented, and experiments with various applications (meeting scheduling, multi-elevator coordination) are under way.

6. DISCUSSION

GOLOG is designed as a logic programming language for dynamic domains. As its full name (alGOl in LOGic) implies, GOLOG attempts to blend ALGOL programming style into logic. It borrows from ALGOL many well-known, and well-studied programming constructs such as sequence, conditionals, recursive procedures and loops.

However, unlike ALGOL and most other conventional programming languages, programs in GOLOG 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 commands which merely change machine states (e.g., assignments to registers). Furthermore, these primitives are formulated by axioms in first-order logic so their effects can be formally reasoned about. This feature of GOLOG supports the specification of dynamic systems at the right level of abstraction.

More importantly, GOLOG 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 situation of the world, and a GOLOG program specifying the behavior of the agents in the system. Executing a program amounts to finding a ground situation term σ such that

$$Axioms \models Do(program, S_0, \sigma).$$

This is done by trying to prove

$$Axioms \models (\exists s) Do(program, S_0, s),$$

and if a (constructive) proof is obtained, such a ground term

$$do(a_n, \dots do(a_2, do(a_1, S_0)) \dots)$$

is obtained as a binding for the variable s. Then the sequence of actions $[a_1, a_2, \dots, a_n]$ is sent to the primitive action execution module. This looks very much like logic programming languages such as Prolog. However, unlike such general purpose logic programming languages, GOLOG is designed specifically for specifying agents' behaviors and for modeling dynamic systems. In particular, in GOLOG, actions play a fundamental role.

There is a body of literature related to the GOLOG project:

1. Dixon's Amala [3]. Amala is a programming language in a conventional imperative style. It is designed after the observation that the semantics of embedded programs should reflect the assumptions about the environment as directly as possible. This is similar to our concern that language primitives should be user-defined, at a high level of abstraction. However, while GOLOG requires these primitives be formally specified within the language, Amala does not. One consequence of this is that programs in GOLOG can be executed by a theorem power, but not those in Amala.
2. Classical AI planning work (Green [6] and Fikes and Nilsson [4]). Like classical AI planning, GOLOG requires primitives and their effects to be formally specified. The major difference is that GOLOG 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. In our elevator example, it was left to the GOLOG interpreter to find a legal sequence of actions to serve all active call buttons. But we can go well beyond this. As an extreme case, the program

   $$\textbf{while}\, \neg Goal\ \textbf{do}\ (\pi\, a)[Appropriate(a)?;\ a]\,\textbf{endWhile},$$

   repeatedly selects an appropriate action and performs it until some goal is achieved. Finding a legal sequence of actions in this case is simply a reformulation of the planning problem.
3. Situated automata [26]. GOLOG shares with situated automata the same philosophy of designing agents using a high level language, and then compiling the high-level programs into low-level ones that can be immediately executed. In the framework considered here, the low-level programs are simply sequences of primitive actions. In [13], we also consider cases involving sensing (see below) where no such sequence exists, and it is necessary to compile to low-level programs containing loops and conditionals.
4. Shoham's AGENT-0 programming language [31]. This 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 GOLOG 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.

   A number of other groups are also developing formalisms for the specification of artificial agents. See [35] for a detailed survey of this research.
5. Transaction logic (Bonner and Kifer [2]). This is a new logic for defining complex database transactions, and like GOLOG provides a rich repertoire of operators for defining new transactions in terms of old. These include sequence, nondeterministic choice, conditionals and iteration. The Bonner-Kifer approach focuses on the definition of complex transactions in terms of

elementary updates. On the assumption that these elementary updates successfully address the frame problem, any complex update defined in terms of these elementary ones will inherit a correct solution to the frame problem. Unfortunately, Bonner and Kifer do not address the frame problem for these elementary updates; this task is left to the person specifying the database.

6. The strategies of McCarthy and Hayes [20]. This is a surprisingly early proposal for representing complex actions (called *strategies*) in the situation calculus. McCarthy and Hayes even appeal to an Algol-like language for representing their strategies, and they include a mechanism for returning symbolic execution traces, as sequences of actions, of these strategies. Moreover, they sketch a method for proving properties of strategies. While McCarthy and Hayes provide no formal development of their proposal, it nevertheless anticipates much of the spirit and technical content of our GOLOG project.

The version of GOLOG presented here omits some important considerations. The following is a partial list:

1. Sensing and knowledge. When modeling an autonomous agent, it is necessary to consider the agent's *perceptual actions*, e.g., acts of seeing, hearing, etc. Unlike ordinary actions that affect the environment, perceptual actions affect an agent's mental state, i.e., its state of *knowledge*. Scherl and Levesque [28] provide a situation calculus account of knowledge, and within this setting, show how to solve the frame problem for perceptual actions.
2. Sensing and knowing how. In the presence of sensing actions, the method described above for executing a GOLOG program is no longer adequate. For example, suppose the sensing action SENSE_P reads the truth value of P, and the primitives a and b are always possible. Then the following program $\mathscr{P}$ is perfectly reasonable:

   SENSE_P; **if** P **then** a **else** b **endIf**

   and should be executable with respect to any initial situation. However, it is not the case that

   $$\mathit{Axioms} \models Do(\mathscr{P}, S_0, \sigma)$$

   for any ground situation term σ. That is, at compile time, the agent does not know the truth value of P and therefore does not know the exact sequence of primitive actions that corresponds to the execution of this program. We have considered several possible solutions to this problem. See [11, 13].
3. Exogenous actions. We have assumed that all events of importance are under the agent's control. That is why, in the elevator example, we did not include a primitive action *turnon*(n), meaning push call button n. Such an action can occur at any time, and is not under the elevator's control. *turnon*(n) is an example of an *exogenous* action. Other such examples are actions under nature's control—it starts to rain, a falling ball bounces on reaching the floor. In writing an elevator or robot controller, one would not include exogenous actions as part of the program, because the robot is in no position to cause such actions to happen.

4. Concurrency and reactivity. Once we allow for exogenous events, it becomes very useful to write programs which monitor certain conditions, and take appropriate actions when they become true. For example, in the middle of serving a floor, smoke might be detected by the elevator, in which case, normal operation should be suspended, and an alarm should be sounded until the alarm is reset. As mentioned earlier, we are investigating a concurrent version of GOLOG where a number of complex actions of this sort can be executed concurrently (at different priorities). We believe that this form of concurrency allows a much more natural specification of controllers that need to quickly react to their environment while following predetermined plans.
5. Continuous processes. It is widely believed that, by virtue of its reliance on discrete situations, the situation calculus cannot represent continuous processes and their evolution in time, like an object falling under the influence of gravity. However, as shown by Pinto [22] and also by Ternovskaia [33], one can view a process as a fluent—$falling(s)$—which becomes true at the time t that the instantaneous action $start\_falling(t)$ occurs, and becomes false at the time t of occurrence of the instantaneous action $end\_falling(t)$. One can then write axioms that describe the evolution in time of the falling object. Reiter [25] gives a situation calculus account of such natural events whose behaviors are described by known laws of physics. This means that one can write GOLOG simulators of such dynamical systems [8]. Moreover, although we have not yet explored this possibility, the GOLOG programmer can now write robot controllers which allow a robot to exploit such naturally occurring exogenous events in its environment.

7. CONCLUSIONS

GOLOG is a logic programming language for implementing applications in dynamic domains like robotics, process control, intelligent software agents, discrete event simulation, etc. Its basis is a formal theory of actions specified in an extended version of the situation calculus.

GOLOG has a number of novel features, both as a programming language, and as an implementation tool for dynamic modeling.

1. Formally, a GOLOG program is a macro which expands during the evaluation of the program to a (usually second order) sentence in the situation calculus. This sentence mentions only the primitive, user defined actions and fluents. The theorem proving task in the evaluation of the program is to prove this sentence relative to a background axiomatization consisting of the foundational axioms of the situation calculus, the action precondition axioms for the primitive actions, the successor state axioms for the fluents, and the axioms describing the initial situation.
2. GOLOG programs are normally evaluated to obtain a binding for the existentially quantified situation variable in the top-level call $(\exists s)Do(program, S_0, s)$. The binding so obtained by a successful proof is a symbolic trace of the program's execution, and denotes that sequence of actions which is to be performed in the external world. At this point, the entire GOLOG computa-

tion has been performed off-line. To effect an actual change in the world, this program trace must be passed to an execution module which knows how to physically perform the sequence of primitive actions in the trace.

3. Because a GOLOG program macro-expands to a situation calculus sentence, we can prove properties of this program (termination, correctness, etc.) directly within the situation calculus.
4. Unlike conventional programming languages, whose primitive instruction set is fixed in advance (assignments to variables, pointer-changing, etc.), and whose primitive function and predicate set is also predefined (values and types of program variables, etc.), GOLOG primitive actions and fluents are user defined by action precondition and successor state axioms. In the simulation of dynamic systems, this facility allows the programmer to specify his primitives in accordance with the naturally occurring events in the world he is modeling. This, in turn, allows programs to be written at a very high level of abstraction, without concern for how the system's primitive architecture is actually implemented.
5. The GOLOG programmer can define complex action *schemas*—advice to a robot about how to achieve certain effects—*without specifying in detail how to perform these actions*. It becomes the theorem prover's responsibility to figure out one or more detailed *executable* sequences of primitive actions which will achieve the desired effects.

   $$\textbf{while}[(\exists block)ontable(block)]\ \textbf{do}\ (\pi\, b)remove(b)\ \textbf{endWhile},$$

   is such an action schema; it does not specify any particular sequence in which the blocks are to be removed. Similarly, the elevator program does not specify in which order the floors are to be served. On this view of describing complex behaviors, the GOLOG programmer specifies a skeleton plan; the evaluator uses deduction, in the context of a specific initial world situation, to fill in the details. Thus GOLOG allows the programmer to strike a compromise between the often computationally infeasible classical planning task, in which a plan must be deduced entirely from scratch, and detailed programming, in which every little step must be specified.

There are several limitations to the version of GOLOG that has been presented here. The implementation only works with completely known initial situations. Adapting GOLOG to work with non-Prolog theories in the initial situation will require some effort (see [16] for ideas on this). Handling sensing actions requires the system's knowledge state to be modeled explicitly [28] and complicates the representation and updating of the world model. Exogenous events also affect the picture as the system may no longer know what the actual history is. In many domains, it is also necessary to deal with sensor noise and "control error" (see [1] for some initial results).

We are also developing an extended version of the language called CONGOLOG that supports concurrent processes, interrupts, and differing priorities on processes (based on an interleaving semantics for concurrent processes) [12]. Techniques for representing and reasoning about continuous processes (e.g., filling a bathtub) are also under investigation [25]. Finally, work is also in progress on a multi-agent distributed version of CONGOLOG for agent-oriented programming applications, which will support distinct world models for each agent and a library of high-level communication actions [10]. Notions like ability, goals, commitments, and rational choice become important in such domains and we are extending our model to deal with them [30].

REFERENCES

1. Bacchus, F., Halpern, J. Y., and Levesque, H. J., Reasoning about Noisy Sensors in the Situation Calculus, in: C. S. Mellish (ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montréal, Aug. 1995, Morgan Kaufmann Publishing, pp. 1933–1940.
2. Bonner, A. and Kifer, M., An Overview of Transaction Logic, *Theoretical Computer Science* 133:205–265 (1994).
3. Dixon, M., *Embedded Computation and the Semantics of Programs*, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA, 1991. Also appeared as Xerox PARC Technical Report SSL-91-1.
4. Fikes, R. E. and Nilsson, N. J., STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence* 2(3/4):189–208 (1971).
5. Goldblatt, R., *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.
6. Green, C. C., Theorem Proving by Resolution as a Basis for Question-Answering Systems, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, American Elsevier, New York, 1969, pp. 183–205.
7. Haas, A. R., The Case for Domain-Specific Frame Axioms, in: F. M. Brown (ed.), *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, Lawrence, KA, April 1987, Morgan Kaufmann Publishing, pp. 343–348.
8. Kelley, T. G., Modeling Complex Systems in the Situation Calculus: A Case Study Using the Dagstuhl Steam Boiler Problem, in: L. C. Aiello, J. Doyle, and S. C. Shapiro (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference* (*KR'96*), Morgan Kaufmann Publishers, San Francisco, CA, pp. 26–37.
9. Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Scherl, R. B., A Logical Approach to High-Level Robot Programming—A Progress Report, in: B. Kuipers (ed.), *Control of the Physical World by Intelligent Agents, Papers from the 1994 AAAI Fall Symposium*, New Orleans, LA, Nov. 1994, pp. 109–119.
10. Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Scherl, R. B., Foundations of a Logical Approach to Agent Programming, in: M. Wooldridge, J. P. Müller, and M. Tambe (eds.), *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, pp. 331–346.
11. Lespérance, Y., Levesque, H. J., Lin, F., and Scherl, R. B., Ability and Knowing How in the Situation Calculus, Unpublished manuscript, 1997.
12. Levesque, H. J., Concurrency in the Situation Calculus, Unpublished manuscript, 1997.
13. Levesque, H. J., What Is Planning in the Presence of Sensing?, in: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, Aug. 4–8, 1996, AAAI Press/MIT Press, pp. 1139–1146.
14. Levesque, H. J., Lin, F., and Reiter, R., Defining Complex Actions in the Situation Calculus, Technical Report, Department of Computer Science, University of Toronto, 1997, to appear.
15. Lin, F., Embracing Causality in Specifying the Indirect Effects of Actions, in: C. S. Mellish (ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montréal, Aug. 1995, Morgan Kaufmann Publishing, pp. 1933–1940.
16. Lin, F. and Reiter, R., How to Progress a Database (and Why) I. Logical Foundations, in: J. Doyle, E. Sandewall, and P. Torasso (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, Bonn, Germany, 1994, Morgan Kaufmann Publishing, pp. 425–436.

17. Lin, F. and Reiter, R., State Constraints Revisited, *Journal of Logic and Computation* 4(5):655–678 (1994).
18. Lin, F. and Reiter, R., How to Progress a Database II: The STRIPS Connection, in: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Aug. 20–25, 1995, pp. 2001–2007.
19. Manna, Z. and Waldinger, R., How to Clear a Bock: A Theory of Plans, *Journal of Automated Reasoning* 3:343–377 (1987).
20. McCarthy, J. and Hayes, P., Some Philosophical Problems from the Standpoint of Artificial Intelligence, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, Edinburgh University Press, Edinburgh, Scotland, 1969, pp. 463–502.
21. Pednault, Edwin P. D., ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus, in: R. J. Brachman, H. J. Levesque, and R. Reiter (eds.), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, ON, May 1989, Morgan Kaufmann Publishing, pp. 324–332.
22. Pinto, J. A., *Temporal Reasoning in the Situation Calculus*, Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, ON, Feb. 1994. Available as Technical Report KRR-TR-94-1.
23. Reiter, R., The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression, in: V. Lifschitz (ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, San Diego, CA, 1991, pp. 359–380.
24. Reiter, R., Proving Properties of States in the Situation Calculus, *Artificial Intelligence* 337–351 (Dec. 1993).
25. Reiter, R., Natural Actions, Concurrency and Continuous Time in the Situation Calculus, in: L. C. Aiello, J. Doyle, and S. C. Shapiro (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference* (*KR'96*), 1996, Morgan Kaufmann Publishers, San Francisco, CA, pp. 2–13.
26. Rosenschein, S. J. and Kaelbling, L. P., The Synthesis of Digital Machines with Provable Epistemic Properties, in: J. Y. Halpern (ed.), *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, Morgan Kaufmann Publishers, Monterey, CA, 1986, pp. 83–98.
27. Ruman, S., Golog as an Agent-Programming Language: Experiments in Developing Banking Applications, Master's Thesis, Department of Computer Science, University of Toronto, Toronto, ON, 1996.
28. Scherl, R. B. and Levesque, H. J., The Frame Problem and Knowledge-Producing Actions, in: *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC, July 1993, AAAI Press/MIT Press, pp. 689–695.
29. Schubert, L. K., 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 (eds.), *Knowledge Representation and Defeasible Reasoning*, Kluwer Academic Press, Boston, MA, 1990, pp. 23–67.
30. Shapiro, S., Lespérance, Y., and Levesque, H. J., 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, Nov. 1995, pp. 117–122.
31. Shoham, Y., Agent-Oriented Programming, *Artificial Intelligence* 60(1):51–92 (1993).
32. Stoy, J. E., *Denotational Semantics*, MIT Press, 1977.
33. Ternovskaia, E., Interval Situation Calculus, in: *Proceedings of ECAI'94 Workshop W5 on Logic and Change*, Amsterdam, Aug. 8–12, 1994, pp. 153–164.
34. Waldinger, R., Achieving Several Goals Simultaneously, in: E. Elcock and D. Michie (eds.), *Machine Intelligence 8*, Ellis Horwood, Edinburgh, Scotland, 1977, pp. 94–136.
35. Wooldrige, M. J. and Jennings, N. R., Intelligent Agents: Theory and Practice, *Knowledge Engineering Review* 10(2) (1995).