Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus

Giuseppe De Giacomo
Dip. di Informatica e Sistemistica,
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma Italy
degiacomo@dis.uniroma1.it

Yves Lespérance
Dept. of Computer Science,
York University
Toronto ON Canada M4N 3M6
lesperan@yorku.ca

Hector J. Levesque
Dept. of Computer Science,
University of Toronto
Toronto ON Canada M5S 1A4
hector@cs.toronto.edu

Abstract

As an alternative to planning, an approach to high-level agent control based on concurrent program execution is considered. A formal definition in the situation calculus of such a programming language is presented and illustrated with a detailed example. The language includes facilities for prioritizing the concurrent execution, interrupting the execution when certain conditions become true, and dealing with exogenous actions. The language differs from other procedural formalisms for concurrency in that the initial state can be incompletely specified and the primitive actions can be user-defined by axioms in the situation calculus.

When it comes to providing high-level control for robots or other agents in dynamic and incompletely known worlds, approaches based on plan synthesis may end up being too demanding computationally in all but simple settings. An alternative approach that is showing promise is that of *high-level program execution* [8]. 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. Previous work on the *Golog* language [8] considered how to reason about actions in programs containing conditionals, iteration, recursion, and non-deterministic operators, where the primitive actions and fluents where characterized by axioms of the situation calculus. In this paper, we explore how to execute programs incorporating a rich account of *concurrency*. The execution task remains the same; what changes is that the programming language, which we call *ConGolog* (for Concurrent Golog), becomes considerably more expressive. One of the nice features of this language is that it allows us to conveniently formulate agent controllers that pursue goal-oriented tasks while concurrently monitoring and reacting to conditions in their environment.

Of course ours is not the first formal model of concurrency. In fact, well developed approaches are available [6, 10, 14][1] and our work inherits many of the intuitions behind them. However, it is distinguished from these in at least two fundamental ways. First, it allows incomplete information about the environment surrounding the program. In contrast to typical computer programs, the initial state of a *ConGolog* program need only be partially specified by a collection of axioms. Second, it allows the primitive actions (elementary instructions) to affect the environment in a complex way. In contrast to typical computer programs whose elementary instructions are simple predefined statements (*e.g.* variable assignments), the primitive actions of a *ConGolog* program are determined by a separate domain-dependent action theory, which specifies the action preconditions and effects, and deals with the frame problem.

The rest of the paper is organized as follows: in Section 1 we very briefly review planning in the situation calculus. In Section 2, we review the *Golog* programming language and present a variant of the original specification of the high-level execution task. In Section 3, we explain informally the sort of concurrency we are concerned with, as well as related notions of priorities and interrupts. The section concludes with changes to the *Golog* specification required to handle concurrency. In Section 4, we present a detailed example of a reactive multi-elevator controller formulated in *ConGolog*. In Section 5, we discuss some of the properties of *ConGolog*, its implementation, and topics for future research.

[1] In [3] a direct use of such approaches to model concurrent (complex) actions in AI is investigated.

1 Situation Calculus

There are a number of ways of making the planning task precise, but perhaps the most appealing is to formulate a specification in terms of a general theory of action. One candidate language for formulating such a theory is the situation calculus [9]. We will not go over the language here except to note the following components: there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation, are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate domain theories which describe how the world changes as the result of the available actions. One possibility is a theory of the following form [12]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [12].
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

For any domain theory of this sort, we have a very clean specification of the planning task, which dates back to the work of Green [4]:

> **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:
>
> $$Axioms \models Legal(\vec{a}, S_0) \wedge \phi(do(\vec{a}, S_0))$$
>
> where $do([a_1, \ldots, a_n], s)$ is an abbreviation for
>
> $$do(a_n, do(a_{n-1}, \ldots, do(a_1, s) \ldots)),$$
>
> and where $Legal([a_1, \ldots, a_n], s)$ stands for
>
> $$Poss(a_1, s) \wedge \ldots \wedge Poss(a_n, do([a_1, \ldots, a_{n-1}], s)).$$

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 state that results from performing the actions in sequence).

2 Golog

As presented in [8], *Golog* is logic-programming language whose primitive actions are those of a background domain theory. It includes the following constructs:

α,	primitive action
$\phi?$,	wait for a condition[2]
$(\sigma_1; \sigma_2)$,	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 $\beta(\vec{x})$ σ,	procedure definition[3]

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

> **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:
>
> $$Axioms \models Do(\sigma, S_0, do(\vec{a}, S_0))$$
>
> where $Do(\sigma, s, s')$ 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 [8], a simple inductive definition of Do was presented, containing rules such as:

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

$$Do([\sigma_1 \mid \sigma_2], s, s') \stackrel{def}{=} Do(\sigma_1, s, s') \vee Do(\sigma_2, s, s')$$

$$Do([\textbf{if } \phi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2], s, s') \stackrel{def}{=} Do([(\phi?; \sigma_1) \mid (\neg\phi?; \sigma_2)], s, s')$$

one for each construct in the language.

The kind of semantics Do associates to programs is sometimes called *evaluation semantics* [5] since it is based on the complete evaluation of the program. With the goal of eventually handling concurrency, it is convenient to give a slightly more refined kind of semantics called *computational semantics* [5], which is based on "single steps" of computation, or *transitions*[4]. A step here is either a primitive action or testing whether a condition holds in the current state. We begin by introducing two special predicates, *Final* and *Trans*, where $Final(\sigma, s)$ is intended to say that program σ may legally terminate in situation s, and where $Trans(\sigma, s, \sigma', s')$ is intended to say that program σ in situation s may legally execute one step, ending in situation s' with program σ' remaining.

[2] 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 state.

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

[4] Both types of semantics belong to the family of structural operational semantics introduced in [11].

Final and *Trans* will be characterized by a set of equivalence axioms, each depending on the structure of the first argument. It will be necessary to quantify over programs and so, unlike in [8], we need to encode *Golog* programs as first-order terms, including introducing constants denoting variables, and so on. This is laborious but quite straightforward [7][5]. We omit all such details here and 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):[6]

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

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

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

It is easy to verify, by induction on the structure of the first argument, the following:

Theorem 1: *For each* Golog *program σ, there exist two situation calculus formulas $\Phi_\sigma(s)$ and $\Psi_\sigma(s, \delta, s')$, not mentioning Final and Trans, such that:*

$$\begin{array}{l}
\Gamma_F, \Gamma_T \models \forall s.Final(\sigma, s) \equiv \Phi_\sigma(s) \\
\Gamma_F, \Gamma_T \models \forall s, \delta, s'.Trans(\sigma, s, \delta, s') \equiv \Psi_\sigma(s, \delta, s')
\end{array}$$

With *Final* and *Trans* in place, we may give a new definition of *Do* as:

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

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

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

where the ellipsis stands for:

$$\begin{array}{l}
\forall s.\ T(\sigma, s, \sigma, s) \quad \wedge \\
\forall s, \delta', s', \delta'', s''.\ T(\sigma, s, \delta', s') \wedge \\
\quad Trans(\delta', s', \delta'', s'') \supset T(\sigma, s, \delta'', s'').
\end{array}$$

In other words, $Do(\sigma, s, s')$ 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'. We then get the following result[7]:

Theorem 2: *The two definitions of Do are equivalent in that for any non-nil* Golog *program σ and situations s and s':*

$$\Gamma_F, \Gamma_T \models Do_1(\sigma, s, s') \equiv Do_2(\sigma, s, s')$$

3 Concurrency

We are now ready to define *ConGolog*, an extended version of *Golog* that incorporates a rich account of concurrency. We say '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, we model concurrent processes 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, we never have more than one primitive action happening at the same time. As discussed in [1, 13], to model actions that intuitively could occur simultaneously, *e.g.* actions of extended duration, we use 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]

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

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

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

[7] See [5] for hints on the proof of this theorem.

[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.

$(\sigma_1 \parallel \sigma_2)$, concurrent execution
$(\sigma_1 \gg \sigma_2)$, concurrency with different priorities
$\sigma^{\parallel}$, concurrent iteration
$<\phi \rightarrow \sigma>$, interrupt.

$(\sigma_1 \parallel \sigma_2)$ denotes the concurrent execution of the actions σ_1 and σ_2. $(\sigma_1 \gg \sigma_2)$ 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, $<TRUE \rightarrow ringBell>$ at the highest priority would ring a bell and do nothing else. With interrupts, we can easily write controllers that can stop whatever task they are doing to handle various concerns as they arise. They are, dare we say, more reactive.

We now show how *Final* and *Trans* need to be extended to handle these constructs. (We handle interrupts separately below.) For *Final*, the extension is straightforward:

$$Final([\sigma_1 \parallel \sigma_2], s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$$
$$Final([\sigma_1 \gg \sigma_2], s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$$
$$Final(\sigma^{\parallel}, s) \equiv TRUE$$

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

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

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 $(\sigma_1 \gg \sigma_2)$ 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 ensures that σ_1 will execute as long as it is possible for it to do so. Finally, you single step $\sigma^{\parallel}$ by single stepping σ, and what is left is the remainder of σ as well as $\sigma^{\parallel}$ itself. This allows an unbounded number of instances of σ to be running.

[9] It is true, though not immediately obvious, that *Trans** remains properly defined even with these axioms containing negative occurrences of *Trans*. See [1] for details.

Observe that with $(\sigma_1 \parallel \sigma_2)$, if both σ_1 and σ_2 are always able to execute, the amount of interleaving between them is left completely open. It is legal to execute one of them completely before even starting the other, and it also legal to switch back and forth after each primitive or wait action. It is not hard to define, however, new concurrency constructs $\|_{\min}$ and $\|_{\max}$ that require the amount of interleaving to be minimized or maximized respectively. We omit the details.

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

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

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, it turns out that these can be explained using other constructs of *ConGolog*:

$$<\phi \rightarrow \sigma> \stackrel{def}{=} \textbf{while } Interrupts\_running \textbf{ do}$$
$$\textbf{if } \phi \textbf{ then } \sigma \textbf{ else } FALSE?$$

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, we use a special primitive action *stop_interrupts*, whose only effect is to make *Interrupts_running* false. Thus, we imagine that to execute a program σ containing interrupts, we would actually execute the program $\{start\_interrupts\,;\, (\sigma \gg stop\_interrupts)\}$ 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.

4 A reactive multi-elevator controller

We illustrate the use of the concurrency primitives using a reactive elevator controller example. The example will use the following terms (where e stands for an elevator):

- ordinary primitive actions:
  $goDown(e)$ move elevator down one floor
  $goUp(e)$ move elevator up one floor
  $buttonReset(n)$ turn off call button of floor n
  $toggleFan(e)$ change the state of elevator fan
  $ringAlarm$ ring the smoke alarm

- exogenous primitive actions:
  $reqElevator(n)$ call button on floor n is pushed
  $changeTemp(e)$ the elevator temperature changes
  $detectSmoke$ the smoke detector first senses smoke
  $resetAlarm$ the smoke alarm is reset
- primitive fluents:
  $floor(e,s) = n$ the elevator is on floor n, $1 \leq n \leq 6$
  $temp(e,s) = t$ the elevator temperature is t
  $FanOn(e,s)$ the elevator fan is on
  $ButtonOn(n,s)$ call button on floor n is on
  $Smoke(s)$ smoke has been detected
- defined fluents:
  $TooHot(e,s) \stackrel{def}{=} temp(e,s) > 3$
  $TooCold(e,s) \stackrel{def}{=} temp(e,s) < -3$

We begin with the following basic action theory for the above primitive actions and fluents:

- initial state:
  $floor(e,S_0) = 1 \quad \neg FanOn(S_0) \quad temp(e,S_0) = 0$
  $ButtonOn(3,S_0) \quad ButtonOn(6,S_0)$
- exogenous actions:
  $\forall a.Exo(a) \equiv a = detectSmoke \vee a = resetAlarm \vee$
  $a = changeTemp(e) \vee \exists n.a = reqElevator(n)$
- precondition axioms:
  $Poss(goDown(e),s) \equiv floor(e,s) \neq 1$
  $Poss(goUp(e),s) \equiv floor(e,s) \neq 6$
  $Poss(buttonReset(n),s) \equiv TRUE$
  $Poss(toggleFan(e),s) \equiv TRUE$
  $Poss(ringAlarm) \equiv TRUE$
  $Poss(reqElevator(n),s) \equiv (1 \leq n \leq 6) \wedge$
  $\neg ButtonOn(n,s)$
  $Poss(changeTemp,s) \equiv TRUE$
  $Poss(detectSmoke,s) \equiv \neg Smoke(s)$
  $Poss(resetAlarm,s) \equiv Smoke(s)$
- successor state axioms:
  $Poss(a,s) \supset [floor(e,do(a,s)) = n \equiv$
  $(a = goDown(e) \wedge n = floor(e,s) - 1) \vee$
  $(a = goUp(e) \wedge n = floor(e,s) + 1) \vee$
  $(n = floor(e,s) \wedge a \neq goDown(e) \wedge$
  $a \neq goUp(e))]$
  $Poss(a,s) \supset [temp(e,do(a,s)) = t \equiv$
  $(a = changeTemp(e) \wedge FanOn(e,s) \wedge$
  $t = temp(e,s) - 1) \vee$
  $(a = changeTemp(e) \wedge \neg FanOn(e,s) \wedge$
  $t = temp(e,s) + 1) \vee$
  $(t = temp(e,s) \wedge a \neq changeTemp(e))]$
  $Poss(a,s) \supset [FanOn(e,do(a,s)) \equiv$
  $(a = toggleFan(e) \wedge \neg FanOn(e,s)) \vee$
  $(a \neq toggleFan(e) \wedge FanOn(e,s))]$
  $Poss(a,s) \supset [ButtonOn(n,do(a,s)) \equiv$
  $a = reqElevator(n) \vee$
  $(ButtonOn(n,s) \wedge a \neq buttonReset(n))]$
  $Poss(a,s) \supset [Smoke(do(a,s)) \equiv$
  $a = detectSmoke \vee$
  $(Smoke(s) \wedge a \neq resetAlarm)]$

Note that many fluents are affected by both exogenous and programmed actions. For instance, the fluent $ButtonOn$ is made true by the exogenous action $reqElevator$ (*i.e.* someone calls for an elevator) and made false by the programmed action $buttonReset$ (*i.e.* when an elevator serves a floor).

Now we are ready to consider a basic elevator controller. It might be defined by something like:

while $\exists n.ButtonOn(n)$ **do**
  $\pi n.\{BestButton(n)?; serveFloor(e,n)\};$
while $floor(e) \neq 1$ **do** $goDown(e)$

The fluent $BestButton$ would be defined to select among all buttons that are currently on, the one that will be served next. For example, it might choose the button that has been on the longest. For our purposes, we can take it to be any $ButtonOn$. The procedure $serveFloor(e,n)$ would consist of the actions the elevator would take to serve the request from floor n. For our purposes, we can use:

$serveFloor(e,n) \stackrel{def}{=}$
  while $floor(e) < n$ **do** $goUp(e);$
  while $floor(e) > n$ **do** $goDown(e);$
  $buttonReset(n)$

We have not bothered formalizing the opening and closing of doors, or other nasty complications like passengers.

Using this controller σ, we would get execution traces like

$$Axioms \models Do(\sigma, S_0, do([u, u, r_3, u, u, u, r_6, d, d, d, d, d], S_0))$$

where $u = goUp(e)$, $d = goDown(e)$, $r_n = buttonReset(n)$. In this particular run, there were no exogenous actions.

This controller does have a big drawback, however: if no buttons are on, the first loop terminates, the elevator returns to the first floor and stops, even if buttons are pushed on its way down. It would be better to structure it as two interrupts:

$< \exists n.ButtonOn(n) \rightarrow$
  $\pi n.\{BestButton(n)?; serveFloor(e,n)\} >$
$< floor(e) \neq 1 \rightarrow goDown(e) >$

with the second at lower priority. So if no buttons are on, and you're not on the first floor, go down a floor, and reconsider; if at any point buttons are pushed exogenously, pick one and serve that floor, before checking again. Thus, the elevator only quits when it is on the first floor with no buttons on.

With this scheme, it is easy to handle emergency or high-priority requests. We would add

$< \exists n.EButtonOn(n) \rightarrow$
  $\pi n.\{EButtonOn(n)?; serveEFloor(e,n)\} >$

as an interrupt with a higher priority than the other two (assuming suitable additional actions and fluents).

To deal with the fan, we can add two new interrupts:

$< TooHot(e) \wedge \neg FanOn(e) \rightarrow toggleFan(e) >$
$< TooCold(e) \wedge FanOn(e) \rightarrow toggleFan(e) >$

These should both be executed at the very *highest* priority. In that case, while serving a floor, whatever that amounts to, if the temperature ever becomes too hot, the fan will be turned on before continuing, and similarly if it ever becomes too cold. Note that if we did not check for the state of the fan, this interrupt would loop repeatedly, never releasing control to lower priority processes.

Finally, imagine that we would like to ring a bell if smoke is detected, and disrupt normal service until the smoke alarm is reset exogenously. To do so, we add the interrupt:

$$< Smoke \rightarrow ringAlarm >$$

with a priority that is less than the emergency button, but higher than normal service. Once this interrupt is triggered, the elevator will stop and ring the bell repeatedly. It will handle the fan and serve emergency requests, however.

Putting all this together, we get the following controller:

$$\begin{array}{l}(< TooHot(e) \wedge \neg FanOn(e) \rightarrow toggleFan(e) > \parallel \\ < TooCold(e) \wedge FanOn(e) \rightarrow toggleFan(e) >) \rangle\!\rangle \\ < \exists n.EButtonOn(n) \rightarrow \\ \quad \pi n.\{EButtonOn(n)?; serveEFloor(e,n)\} > \rangle\!\rangle \\ < Smoke \rightarrow ringAlarm > \rangle\!\rangle \\ < \exists n.ButtonOn(n) \rightarrow \\ \quad \pi n.\{BestButton(n)?; serveFloor(e,n)\} > \rangle\!\rangle \\ < floor(e) \neq 1 \rightarrow goDown(e) >\end{array}$$

Note that this elevator controller uses 5 different levels of priority. It could have been programmed in *Golog* without interrupts, but the code would have been a lot messier.

Now let us suppose that we would like to write a controller that handles two independent elevators. In *ConGolog*, this can be done very elegantly using $(\sigma_1 \parallel \sigma_2)$, where σ_1 is the above program with e replaced by $elevator_1$ and σ_2 is the same program with e replaced by $elevator_2$. This allows the two processes to work completely independently (in terms of priorities)[10]. For n elevators, we would use $(\sigma_1 \parallel \cdots \parallel \sigma_n)$. In some applications, it is useful to have an *unbounded* number of instances of a process running concurrently. For example in an FTP server, we may want an instance of a manager process for each active FTP session. This can be programmed using the $\sigma^{\parallel}$ concurrent iteration construct.

Finally, if it is desirable to have the elevator continue working indefinitely, we can do so by adding an interrupt:

$$< TRUE \rightarrow wait >$$

at the lowest possible priority, where *wait* is a no-op in terms of fluents. So if everything else is satisfied, the elevator simply waits until a higher priority interrupt is triggered exogenously. Such programs never terminate, so semantics based on *Do* cannot be used, but their behavior can nonetheless be specified using *Trans* [1].

5 Discussion

With all of this procedural richness, it is important not to lose sight of the logical framework. *ConGolog* is indeed a programming language, but one whose execution, like planning, depends on reasoning about actions. Thus, a crucial part of a *ConGolog* program is the *declarative* part: the precondition axioms, the successor state axioms, and the axioms characterizing the initial state. This is central to how the language differs from superficially similar "procedural languages". A *ConGolog* 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[11]. And this theory must be used explicitly by a *ConGolog* interpreter.

We have developed a prototype *ConGolog* interpreter in Prolog (see [1]). Indeed, a simple if somewhat inefficient interpreter can be lifted directly from *Final*, *Trans*, and *Do* introduced above[12]. For example, for $(\sigma_1 \rangle\!\rangle \sigma_2)$, we would have the following two Prolog clauses for *Trans*:

```
trans(prioConc(Sigma1,Sigma2),S1,
      prioConc(Delta,Sigma2),S2) :-
   trans(Sigma1,S1,Delta,S2).
trans(prioConc(Sigma1,Sigma2),S1,
      prioConc(Sigma1,Delta),S2) :-
   trans(Sigma2,S1,Delta,S2),
   not trans(Sigma1,S1,_,_).
```

Our implementation requires that the program's precondition axioms, successor state axioms, and axioms about the initial state be expressible as Prolog clauses. This is a limitation of the implementation, not the theory.

In summary, we have shown how, given a basic action theory describing an initial state and the preconditions and effects of a collection of primitive actions, it is possible to combine these in complex ways appropriate for providing high-level control. The semantics of these complex actions ends up deriving directly from that of the underlying primitive actions. In this sense, we inherit the solution to the frame problem provided by successor state axioms for primitive actions.

There are, however, many areas for future research. Among them, we mention: 1) incorporating sensing actions, that is, actions whose effect is not to change the world so much as to provide information to be used by the agent at runtime; 2) handling non-termination, that is, developing accounts of program correctness (fairness, liveness *etc.*) appropriate for controllers expected to operate indefinitely.

[10] Of course, when an elevator is requested on some floor, both elevators may decide to serve it. It is easy to program a better strategy that coordinates the elevators: when an elevator decides to serve a floor, it immediately makes a fluent true for that floor, and the other elevator will not serve a floor for which that fluent is already true.

[11] Although with a different emphasis, this approach is shared by [2] where a logical formalism is proposed for concurrent database transactions.

[12] Exogenous actions can be simulated by generating them probabilistically or by asking the user at runtime when they should occur.

References

[1] A longer version of this paper, in preparation.

[2] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Proc. ICDT'95*, 1995.

[3] G. De Giacomo and X. Chen. Reasoning about nondeterministic and concurrent actions: A process algebra approach. In *Proc. AAAI'96*, pages 658–663, 1996.

[4] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In *Machine Intelligence*, volume 4, pages 183–205. Edinburgh University Press, 1969.

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

[6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall Int., 1985.

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

[8] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. To appear in the *Journal of Logic Programming*, 1996.

[9] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, vol. 4, Edinburgh University Press, 1969.

[10] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

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

[13] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. KR'96*, pages 2–13, 1996.

[14] C. Stirling. Modal and temporal logics for processes. In *Logics for Concurrency: Structure versus Automata*, number 1043 in LNCS, pages 149–237. Springer-Verlag, 1996.