ConGolog, a concurrent programming language based on the situation calculus: language and implementation.

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

Yves Lespérance
Department of Computer Science
York University
Toronto, ON, Canada M3J 1P3
lesperan@cs.yorku.ca

Hector J. Levesque
Department of Computer Science
University of Toronto
Toronto, ON, Canada M5S 3H5
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 some examples. The language includes facilities for prioritizing the execution of concurrent processes, 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.

1 Introduction

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* [20]. 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 a sequence that constitutes a legal execution of a high-level program, 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 [20] 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*) [7], 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 [16, 23, 4, 35][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 and such changes to the environment can affect the execution of the remainder of the program. 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 2 we briefly review the situation calculus and how it can be used to formulate the planning task. In Section 3, we review the *Golog* programming language and in the following section, we present a variant of the original specification of the high-level execution task. In Section 5, 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 6, we illustrate the use of *ConGolog* by going over several example programs. Then in Section 7, we extend the specification given in Section 5

[1]In [25, 5] a direct use of such approaches to model concurrent (complex) actions in AI is investigated.

to handle procedures and recursion. In Section 8, we present a Prolog interpreter for *ConGolog* and prove its correctness. In Section 9, we conclude by discussing some of the properties of *ConGolog*, its implementation, and topics for future research. Although this paper is self-contained, a companion paper [6] examines the mathematical foundations of *ConGolog* in detail.

2 The Situation Calculus

As mentioned earlier, our high-level programs contain primitive actions and tests that are domain dependent. An interpreter for such programs must reason about the preconditions and effects of actions in the program to find legal executions. So we need a language to specify such domain theories. For this, we use the *situation calculus* [22], a first-order language (with some second-order features) for representing dynamic domains. In this formalism, all changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first-order term called a *situation*. The constant S_0 is used to denote the initial situation, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol *do* and the term $do(a, s)$ denotes the situation resulting from action a being performed in situation s. 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 putting A on B when the world is in situation s. Notice that in the situation calculus, actions are denoted by function symbols, and situations (world histories) are also first-order terms. For example,

$$do(putDown(A), do(walk(P), do(pickUp(A), S_0)))$$

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

$$[pickUp(A), walk(P), putDown(A)].$$

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, $Holding(r, x, s)$ might mean that a robot r is holding an object x in situation s. Functions whose denotations vary from situation to situation are called *functional fluents*. They are denoted by function symbols with an additional situation argument, as in $position(r, s)$, i.e., the position of robot r in situation s.

The actions in a domain are specified by providing certain types of axioms. First, one must state the conditions under which it is physically possible to perform an action by providing a *action precondition axiom*. For this, we use the special predicate $Poss(a, s)$ which represents the fact that primitive action a is physically possible (i.e. executable) in situation s. So for example,

$$Poss(pickup(x), s) \equiv \forall x \neg Holding(x, s) \land NextTo(x, s) \land \neg Heavy(x)$$

says that the action *pickup*(x), i.e. the agent picking up an object x, is possible in situation s if and only if the agent is not already holding something in situation s and is positioned next to x in s and x is not heavy.

Secondly, one must specify how the action affects the state of the world; this is done by providing *effect axioms*. For example,

$$Poss(drop(x), s) \wedge Fragile(x, s) \supset Broken(x, do(drop(x, s)))$$

says that dropping an object x causes it to become broken provided that x is fragile. Effect axioms provide the "causal laws" for the domain of application.

These types of axioms are usually insufficient if one wants to reason about change. One must add *frame axioms* that specify when fluents remain unchanged by actions. For example, dropping an object does not affect the color of things:

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

The frame problem arises because the number of these frame axioms is very large, in general, of the order of $2 \times \mathcal{A} \times \mathcal{F}$, where $\mathcal{A}$ is the number of actions and $\mathcal{F}$ the number of fluents. This complicates the task of axiomatizing a domain and can make theorem proving extremely inefficient.

To deal with the frame problem, we use an approach due to Reiter [28]. The basic idea behind this is to collect all effect axioms about a given fluent and make a completeness assumption, i.e. assume that they specify all of the ways that the value of the fluent may change. A syntactic transformation can then be applied to obtain a *successor state axiom* for the fluent, for example:

$$\begin{aligned} &Poss(a, s) \supset [Broken(x, do(a, s)) \equiv \\ &\qquad a = drop(x) \wedge Fragile(x, s) \vee \\ &\qquad \exists b(a = explode(b) \wedge NextTo(b, x, s)) \vee \\ &\qquad Broken(x, s) \wedge a \neq repair(x)]. \end{aligned}$$

This says that an object x is broken in the situation resulting from action a being performed in s if and only if a is dropping x and x is fragile, or a involves a bomb exploding next to x, or x was already broken in situation s prior to the action and a is not the action of repairing x. This approach yields a solution to the frame problem – a parsimonious representation for the effects of actions. Note that it relies on quantification over actions.[2]

So following this approach, a domain of application will be specified by a theory of the following form:

- Axioms describing the initial situation, S_0.

[2]This discussion ignores the ramification and qualification problems; a treatment compatible with the approach described has been proposed by Lin and Reiter [18].

- 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.
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

The latter foundational axioms include unique names axioms for situations, and an induction axiom. They also introduce the relation $<$ over situations. $s < s'$ holds if and only if s' is the result of some sequence of actions being performed in s, where each action in the sequence is possible in the situation in which it is performed; $s \leq s'$ stands for $s < s' \vee s = s'$. Since the foundational axioms play no special role in this paper, we omit them. For details, and for some of their metamathematical properties, see Lin and Reiter [18] and Reiter [29].

For any domain theory of the sort just described, we have a very clean specification of the planning task, which dates back to the work of Green [13]:

Classical Planning: Given a domain theory $\mathcal{D}$ 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:

$$\mathcal{D} \models Legal(\vec{a}, S_0) \wedge \phi(do(\vec{a}, S_0))$$

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).

3 Golog

As presented in [20], *Golog* is a logic-programming language whose primitive actions are those of a background domain theory. It includes the following constructs (δ, possibly subscripted, ranges over *Golog* programs):

a,	primitive action[3]
$\phi?$,	wait for a condition[4]
$(\delta_1; \delta_2)$,	sequence
$(\delta_1 \mid \delta_2)$,	nondeterministic choice between actions
$\pi v.\delta$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration
$\{\mathbf{proc}\ P_1(\vec{v}_1)\ \delta_1\ \mathbf{end}; \ldots \mathbf{proc}\ P_n(\vec{v}_n)\ \delta_n\ \mathbf{end};\ \delta\}$,	procedures

Let's examine a simple example to see some of the features of the language. Here's a *Golog* program to clear the table in a blocks world:

$$\begin{array}{l} \{\mathbf{proc}\ \mathit{removeAblock} \\ \qquad \pi b\,[\mathit{OnTable}(b, \mathit{now})?; \mathit{pickUp}(b); \mathit{putAway}(b)] \\ \ \mathbf{end}; \\ \ \mathit{removeAblock}^*; \\ \ \neg\exists b\, \mathit{OnTable}(b, \mathit{now})?\ \} \end{array}$$

Here we first define a procedure to remove a block from the table using the nondeterministic choice of argument operator π. $\pi x\,[\delta(x)]$ is executed by nondeterministically picking an individual x, and for that x, performing the program $\delta(x)$. The wait action $\mathit{OnTable}(b, \mathit{now})?$ succeeds only if the individual chosen, b, is a block that is on the table in the current situation. The main part of the program uses the nondeterministic iteration operator; it simply says to execute *removeAblock* zero or more times until the table is clear. Note that *Golog*'s other nondeterministic construct, $(\delta_1 \mid \delta_2)$, allows a choice between two actions; a program of this form can be executed by performing either δ_1 or δ_2.

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 $\mathcal{D}$ as above, and a program δ, the execution task is to find a sequence of actions $\vec{a}$ such that:

$$\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$$

where $Do(\delta, s, s')$ means that program δ when executed starting in situation s has s' as a legal terminating situation.

[3]Here, a stands for a situation calculus action with all situation arguments in its parameters replaced by the special constant *now*. Similarly in the line below ϕ stands for a situation calculus formula with all situation arguments replaced by *now*, for example $\mathit{OnTable}(\mathit{block}, \mathit{now})$. $a[s]$ ($\phi[s]$) will denote the action (formula) obtained by substituting the situation variable s for all occurrences of *now* in functional fluents appearing in a (functional and predicate fluents appearing in ϕ). Moreover when no confusion can arise, we often leave out the *now* argument from fluents altogether; e.g. write $\mathit{OnTable}(\mathit{block})$ instead of $\mathit{OnTable}(\mathit{block}, \mathit{now})$. In such cases, the situation suppressed version of the action or formula should be understood as an abbreviation for the version with *now*.

[4]Because there are no exogenous actions or concurrent processes in *Golog*, waiting for ϕ amounts to testing that ϕ holds in the current state.

Note that since *Golog* programs can be nondeterministic, there may be several terminating situations for the same program and starting situation.

In [20], a simple inductive definition of *Do* was presented, containing rules such as:

$$\begin{array}{rcl}
Do(a,s,s') & \overset{def}{=} & Poss(a[s],s) \wedge s' = do(a[s],s) \\
Do(\delta_1;\delta_2,\ s,s') & \overset{def}{=} & \exists s''.\ Do(\delta_1,s,s'') \wedge Do(\delta_2,s'',s') \\
Do(\delta_1 \mid \delta_2,\ s,s') & \overset{def}{=} & Do(\delta_1,s,s') \vee Do(\delta_2,s,s') \\
Do(\pi x.\delta(x),s,s') & \overset{def}{=} & \exists x\, Do(\delta(x),s,s')
\end{array}$$

one for each construct in the language.

4 A Transition Semantics for *Golog*

The kind of semantics *Do* associates to programs is sometimes called *evaluation semantics* [14] 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* [14], which is based on "single steps" of computation, or *transitions*.[5] 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 $Final(\delta,s)$ is intended to say that program δ may legally terminate in situation s, and where $Trans(\delta,s,\delta',s')$ is intended to say that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining.

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 [20], we need to encode *Golog* programs as first-order terms, including introducing constants denoting variables, and so on. This is laborious but quite straightforward. See the companion paper [6] for details.[6] We omit all such details here and simply use programs within formulas as if they were already first-order terms.

4.1 *Trans* and *Final*

Let us formally define *Trans* and *Final*, which intuitively specify:

- what are the possible *transitions* between configurations (*Trans*).
- when a configuration can be considered final (*Final*).

[5]Both types of semantics belong to the family of structural operational semantics introduced in [24].

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

It is convenient to introduce a special program *nil*, called the *empty program*, to denote the fact that nothing remains to be performed (legal termination). For example, consider a program consisting solely of a primitive action a. If it can be executed (i.e. if the action is possible in the current situation), then after the execution of the action a nothing remains of the program. In this case, we say that the program remaining after the execution of action a is *nil*.

$Trans(\delta, s, \delta', s')$ holds if and only if there is a transition from the configuration (δ, s) to the configuration (δ', s'), that is, if by running program δ starting in situation s, one can get to situation s' in one elementary step with the program δ' remaining to be executed. As mentioned, every such elementary step will either be the execution of an atomic action (which changes the current situation) or the execution of a test (which doesn't). As well, if the program is nondeterministic, there may be several transitions that are possible in a configuration. To simplify the discussion, we postpone the introduction of procedures to Section 7.

The predicate *Trans* for programs without procedures is characterized by the following set of axioms $\mathcal{T}$ (here as in the rest of the paper, free variables are assumed to be universally quantified):

1. Empty program:
$$Trans(nil, s, \delta', s') \quad \equiv \quad False$$

2. Primitive actions:
$$\begin{aligned} &Trans(a, s, \delta', s') \quad \equiv \\ &\qquad Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s) \end{aligned}$$

3. Wait/test actions:
$$Trans(\phi?, s, \delta', s') \quad \equiv \quad \phi[s] \wedge \delta' = nil \wedge s' = s$$

4. Sequence:
$$\begin{aligned} &Trans(\delta_1; \delta_2, s, \delta', s') \quad \equiv \\ &\qquad \exists \gamma . \delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \quad \vee \\ &\qquad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \end{aligned}$$

5. Nondeterministic branch:
$$\begin{aligned} &Trans(\delta_1 \mid \delta_2, s, \delta', s') \quad \equiv \\ &\qquad Trans(\delta_1, s, \delta', s') \quad \vee \quad Trans(\delta_2, s, \delta', s') \end{aligned}$$

6. Nondeterministic choice of argument:

$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x. Trans(\delta_x^v, s, \delta', s')$$

7. Iteration:

$$\begin{array}{l} Trans(\delta^*, s, \delta', s') \equiv \\ \quad \exists \gamma.(\delta' = \gamma; \delta^*) \land Trans(\delta, s, \gamma, s') \end{array}$$

The assertions above characterize when a configuration (δ, s) can evolve (in a single step) to a configuration (δ', s'). Intuitively they can be read as follows:

1. (nil, s) cannot evolve to any configuration.

2. (a, s) evolves to $(nil, do(a[s], s))$, provided that $a[s]$ is possible in s. After having performed a, nothing remains to be performed and hence *nil* is returned. Note that in $Trans(a, s, \delta', s')$, a stands for the program term encoding the corresponding situation calculus action, while *Poss* and *do* take the latter as argument; we take the function $\cdot[\cdot]$ as mapping the program term a into the corresponding situation calculus action $a[s]$, as well as replacing *now* by the situation s.

3. $(\phi?, s)$ evolves to (nil, s), provided that $\phi[s]$ holds, otherwise it cannot proceed. Note that the situation remains unchanged. Analogously to the previous case, we take the function $\cdot[\cdot]$ as mapping the program term for condition ϕ into the corresponding situation calculus formulas $\phi[s]$, as well as replacing *now* by the situation s.

4. $(\delta_1; \delta_2, s)$ can evolve to $(\delta_1'; \delta_2, s')$, provided that (δ_1, s) can evolve to (δ_1', s'). Moreover it can also evolve to (δ_2', s'), provided that (δ_1, s) is a final configuration and (δ_2, s) can evolve to (δ_2', s').

5. $(\delta_1|\delta_2, s)$ can evolve to (δ', s'), provided that either (δ_1, s) or (δ_2, s) can do so.

6. $(\pi v.\delta, s)$ can evolve to (δ', s'), provided that there exists an x such that (δ_x^v, s) can evolve to (δ', s'). Here δ_x^v is the program resulting from δ by substituting v with the variable x.[7]

7. (δ^*, s) can evolve to $(\delta'; \delta^*, s')$ provided that (δ, s) can evolve to (δ', s'). Observe that (δ^*, s) can also not evolve at all, (δ^*, s) being final by definition (see below).

[7]To be more precise, v is substituted by a term of the form `nameOf`(x), where `nameOf` is used to convert situation calculus objects/actions into program terms of the corresponding sort.

$Final(\delta, s)$ tells us whether a program δ can be considered to be already in a *final state* (legally terminated) in the situation s. Obviously we have $Final(nil, s)$, but also $Final(\delta^*, s)$ since δ^* requires 0 or more repetitions of δ and so it is possible to not execute δ at all, the program completing immediately.

The predicate *Final* for programs without procedures is characterized by the set of axioms $\mathcal{F}$:

1. Empty program:
$$Final(nil, s) \equiv True$$

2. Primitive action:
$$Final(a, s) \equiv False$$

3. Wait/test action:
$$Final(\phi?, s) \equiv False$$

4. Sequence:
$$\begin{aligned} Final(\delta_1; \delta_2, s) \equiv \\ Final(\delta_1, s) \wedge Final(\delta_2, s) \end{aligned}$$

5. Nondeterministic branch:
$$\begin{aligned} Final(\delta_1 \mid \delta_2, s) \equiv \\ Final(\delta_1, s) \vee Final(\delta_2, s) \end{aligned}$$

6. Nondeterministic choice of argument:
$$Final(\pi v.\delta, s) \equiv \exists x.Final(\delta^v_x, s)$$

7. Iteration:
$$Final(\delta^*, s) \equiv True$$

The assertions above can be read as follows:

1. (nil, s) is a final configuration.

2. (a, s) is not final, indeed the program consisting of the primitive action a cannot be considered completed until it has performed a.

3. $(\phi?, s)$ is not final, indeed the program consisting of the test action $\phi?$ cannot be considered completed until it has performed the test $\phi?$.

4. $(\delta_1; \delta_2, s)$ can be considered completed if both (δ_1, s) and (δ_2, s) are final.

5. $(\delta_1|\delta_2, s)$ can be considered completed if either (δ_1, s) or (δ_2, s) is final.

6. $(\pi v.\delta, s)$ can be considered completed, provided that there exists an x such that (δ^v_x, s) is final, where δ^v_x is obtained from δ by substituting v with x.

7. (δ^*, s) is a final configuration, since by δ^* is allowed to execute 0 times.

In the following we denote by $\mathcal{C}$ be the set of axioms for *Trans* and *Final* plus those needed for the encoding of programs as first-order terms.

4.2 *Trans** and *Do*

The possible configurations that can be reached by a program δ starting in a situation s are those obtained by repeatedly following the transition relation denoted by *Trans* starting from (δ, s), i.e. those in the reflexive transitive closure of the transition relation. Such a relation, denoted by *Trans**, is defined as the (second-order) situation calculus formula:

$$Trans^*(\delta, s, \delta', s') \quad \overset{def}{=} \quad \forall T[\ldots \;\supset\; T(\delta, s, \delta', s')]$$

where ... stands for the conjunction of the universal closure of the following implications:

$$\begin{array}{l} True \;\supset\; T(\delta, s, \delta, s) \\ Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \;\supset\; T(\delta, s, \delta', s') \end{array}$$

Using *Trans** and *Final* we can give a new definition of *Do* as:

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

In other words, $Do(\delta, s, s')$ holds if it is possible to repeatedly single-step the program δ, obtaining a program δ' and a situation s' such that δ' can legally terminate in s'.

It can be shown that for *Golog* programs such a definition for *Do* coincides with the one given in [20]. Indeed in the companion paper [6] the following theorem is proven:

Theorem 1: *Let Do_1 be the original definition of Do in [20], and Do_2 the new one given above. Then for each* Golog *program δ:*

$$\mathcal{C} \quad \models \quad \forall s, s'.\, Do_1(\delta, s, s') \quad \equiv \quad Do_2(\delta, s, s')$$

The theorem also holds for *Golog* programs involving procedures when the treatment in Section 7 is used.

5 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 any given time. This assumption might appear problematic when the domain involves actions with extended duration (e.g. filling a bathtub). In section 6.4, we return to this issue and argue that in fact, there is a straightforward way to handle such cases.

An important concept in understanding concurrent execution is that of a process becoming *blocked*. If a deterministic process δ 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 δ executes next. The net effect is that δ is suspended or blocked, and execution must continue elsewhere.[8]

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

if ϕ **then** δ_1 **else** δ_2,	synchronized conditional
while ϕ **do** δ,	synchronized loop
$(\delta_1 \parallel \delta_2)$,	concurrent execution
$(\delta_1 \rangle\!\rangle\, \delta_2)$,	concurrency with different priorities
$\delta^{\parallel}$,	concurrent iteration
$<\phi \rightarrow \delta>$,	interrupt.

The constructs **if** ϕ **then** δ_1 **else** δ_2 and **while** ϕ **do** δ are the synchronized versions of the usual if-then-else and while-loop. They are synchronized in the sense that testing the condition ϕ does not involve a transition per se: the evaluation of the condition and the first action of the branch chosen are executed as an atomic unit. So these constructs

[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.

behave in a similar way to the test-and-set atomic instructions used to build semaphores in concurrent programming [1].[9]

The construct $(\delta_1 \parallel \delta_2)$ denotes the concurrent execution of the actions δ_1 and δ_2. $(\delta_1 \rangle\!\rangle \delta_2)$ denotes the concurrent execution of the actions δ_1 and δ_2 with δ_1 having higher priority than δ_2. This restricts the possible interleavings of the two processes: δ_2 executes only when δ_1 is either done or blocked. The next construct, $\delta^{\parallel}$, is like nondeterministic iteration, but where the instances of δ are executed concurrently rather than in sequence. Just as δ^* executes with respect to *Do* like $nil \mid \delta \mid (\delta;\delta) \mid (\delta;\delta;\delta) \mid \ldots$, the program $\delta^{\parallel}$ executes with respect to *Do* like $nil \mid \delta \mid (\delta \parallel \delta) \mid (\delta \parallel \delta \parallel \delta) \mid \ldots$. See Section 6.3 for an example of its use.

Finally, $<\phi \rightarrow \delta>$ is an interrupt. It has two parts: a trigger condition ϕ and a body, δ. The idea is that the body δ will execute some number of times. If ϕ never becomes true, δ will not execute at all. If the interrupt gets control from higher priority processes when ϕ is true, then δ will execute. Once it has completed its execution, the interrupt is ready to be triggered again. This means that a high priority interrupt can take complete control of the execution. For example, $<\mathit{True} \rightarrow \mathit{ringBell}>$ at the highest priority would ring a bell and do nothing else. With interrupts, 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 *Trans* and *Final* need to be extended to handle these constructs. (We handle interrupts separately below.) *Trans* and *Final* for synchronized conditionals and loops are defined as follows:

$$\begin{aligned}
&\mathit{Trans}(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s, \delta', s') \quad \equiv \\
&\quad \phi[s] \wedge \mathit{Trans}(\delta_1, s, \delta', s') \quad \vee \quad \neg\phi[s] \wedge \mathit{Trans}(\delta_2, s, \delta', s') \\
&\mathit{Trans}(\mathbf{while}\ \phi\ \mathbf{do}\ \delta,\ s, \delta', s') \quad \equiv \\
&\quad \exists\gamma.(\delta' = \gamma; \mathbf{while}\ \phi\ \mathbf{do}\ \delta) \wedge \phi[s] \wedge \mathit{Trans}(\delta, s, \gamma, s')
\end{aligned}$$

$$\begin{aligned}
&\mathit{Final}(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s) \quad \equiv \\
&\quad \phi[s] \wedge \mathit{Final}(\delta_1, s) \quad \vee \quad \neg\phi[s] \wedge \mathit{Final}(\delta_2, s) \\
&\mathit{Final}(\mathbf{while}\ \phi\ \mathbf{do}\ \delta, s) \quad \equiv \\
&\quad \neg\phi[s] \quad \vee \quad \mathit{Final}(\delta, s)
\end{aligned}$$

That is $(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else} \delta_2, s)$ can evolve to (δ', s'), if either $\phi[s]$ holds and (δ_1, s) can do so, or $\neg\phi[s]$ holds and (δ_2, s) can do so. Similarly, $(\mathbf{while}\ \phi\ \mathbf{do}\ \delta, s)$ can evolve to $(\delta'; \mathbf{while}\ \phi\ \mathbf{do}\ \delta, s')$, if $\phi[s]$ holds and (δ, s) can evolve to (δ', s'). $(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s)$

[9]In [20] a non-synchronized version of if-then else and while-loop is introduced by defining: $\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2 \overset{def}{=} [(\phi?;\delta_1) \mid (\neg\phi?;\delta_2)]$ and $\mathbf{while}\ \phi\ \mathbf{do}\ \delta \overset{def}{=} [(\phi?;\delta)^*;\neg\phi?]$. The synchronized versions of these constructs introduced here behave essentially as the non-synchronized ones in absence of concurrency. However the difference is striking when concurrency is allowed.

can be considered completed, if either $\phi[s]$ holds and (δ_1, s) is final, or if $\neg\phi[s]$ holds and (δ_2, s) is final. Similarly, (**while** ϕ **do** δ, s) can be considered completed if either $\neg\phi[s]$ holds or (δ, s) is final.

For the constructs for concurrency the extension of *Final* is straightforward:

$$\begin{aligned}
\mathit{Final}(\delta_1 \parallel \delta_2, s) &\equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s) \\
\mathit{Final}(\delta_1 \rangle\!\rangle\, \delta_2, s) &\equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s) \\
\mathit{Final}(\delta^{\parallel}, s) &\equiv \mathit{True}.
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{Trans}(\delta_1 \parallel \delta_2, s, \delta', s') \;\equiv \\
&\quad \exists\gamma.\delta' = (\gamma \parallel \delta_2) \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \;\vee \\
&\quad \exists\gamma.\delta' = (\delta_1 \parallel \gamma) \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \\
&\mathit{Trans}(\delta_1 \rangle\!\rangle\, \delta_2, s, \delta', s') \;\equiv \\
&\quad \exists\gamma.\delta' = (\gamma \rangle\!\rangle\, \delta_2) \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \;\vee \\
&\quad \exists\gamma.\delta' = (\delta_1 \rangle\!\rangle\, \gamma) \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.\mathit{Trans}(\delta_1, s, \zeta, s'') \\
&\mathit{Trans}(\delta^{\parallel}, s, \delta', s') \;\equiv \\
&\quad \exists\gamma.\delta' = (\gamma \parallel \delta^{\parallel}) \wedge \mathit{Trans}(\delta, s, \gamma, s')
\end{aligned}$$

In other words, you single step $(\delta_1 \parallel \delta_2)$ by single stepping either δ_1 or δ_2 and leaving the other process unchanged. The $(\delta_1 \rangle\!\rangle\, \delta_2)$ construct is identical, except that you are only allowed to single step δ_2 if there is no legal step for δ_1. This ensures that δ_1 will execute as long as it is possible for it to do so. Finally, you single step $\delta^{\parallel}$ by single stepping δ, and what is left is the remainder of δ as well as $\delta^{\parallel}$ itself. This allows an unbounded number of instances of δ to be running.

Observe that with $(\delta_1 \parallel \delta_2)$, if both δ_1 and δ_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 $\parallel_{\min}$ and $\parallel_{\max}$ that require the amount of interleaving to be minimized or maximized respectively. We omit the details.

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

$$\begin{aligned}
<\phi \to \delta> \overset{def}{=} &\ \textbf{while}\ \mathit{Interrupts\_running}\ \textbf{do} \\
&\quad \textbf{if}\ \phi\ \textbf{then}\ \delta\ \textbf{else}\ \mathit{False}?
\end{aligned}$$

To see how this works, first assume that the special fluent *Interrupts_running* is identically *True*. When an interrupt $<\phi \to \delta>$ gets control, it repeatedly executes δ 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 δ containing interrupts, we would actually execute the program $\{start\_interrupts\,;\, (\delta \rangle\!\rangle\ stop\_interrupts)\}$ which has the effect of stopping all blocked interrupt loops in δ at the lowest priority, *i.e.* when there are no more actions in δ that can be executed.

Finally, let us consider exogenous actions. These 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 define a special program for exogenous events:

$$\delta_{EXO} \stackrel{def}{=} (\pi\, a.\, Exo(a)?; a)^*$$

Executing this program involves performing zero, one, or more nondeterministically chosen exogenous events.[10] Then we make the user-specified program δ run concurrently with δ_{EXO}:

$$\delta \parallel \delta_{EXO}$$

In this way we allow exogenous actions whose preconditions are satisfied to asynchronously occur (outside the control of δ) during the execution of δ.[11]

6 Some Examples

6.1 Two Robots Lifting a Table

Our first example involves a simple case of concurrency: two robots that jointly lift a table. Test actions are used to synchronize the robots' actions so that the table does not tip so much that objects on it fall off. Two instances of the same program are used to control the robots.

- Objects:
  Two agents: $\forall r\, Robot(r) \equiv r = Rob_1 \vee r = Rob_2$.
  Two table ends: $\forall e\, TableEnd(e) \equiv e = End_1 \vee e = End_2$.

[10]Observe the use of π: the program nondeterministically chooses an action a, tests that this a is an exogenous event, and executes it.

[11]We thank David Tremaine for pointing out a problem with an earlier way of handling exogenous events.

- Primitive actions:
  $grab(rob, end)$
  $release(rob, end)$
  $vmove(rob, z)$ move robot arm up or down by z units.

- Primitive fluents:
  $Holding(rob, end, s)$
  $vpos(end, s) = z$ height of the table end

- Initial state:
  $\forall r \forall e \, \neg Holding(r, e, S_0)$
  $\forall e \; vpos(e, S_0) = 0$

- Precondition axioms:
  $Poss(grab(r, e), s) \equiv \forall r^* \, \neg Holding(r^*, e, s) \wedge \forall e^* \, \neg Holding(r, e^*, s)$
  $Poss(release(r, e), s) \equiv Holding(r, e, s)$
  $Poss(vmove(r, z), s) \equiv TRUE$

- Successor state axioms:
  $Holding(r, e, do(a, s)) \equiv$
  $a = grab(r, e) \vee Holding(r, e, s) \wedge a \neq release(r, e)$
  $vpos(e, do(a, s)) = p \equiv$
  $a = vmove(r, z) \wedge Holding(r, e, s) \wedge p = vpos(e, s) + z \vee$
  $a = release(r, e) \wedge p = 0 \vee$
  $p = vpos(e, s) \wedge \neg(\exists z \, a = vmove(r, z) \wedge Holding(r, e, s)) \wedge a \neq$
  $release(r, e)$

The goal here is to get the table up, but to keep it sufficiently level so that nothing falls off. We can define these as follows:

$$TableUp(s) \overset{def}{=} vpos(End_1, s) \geq H \;\wedge\; vpos(End_2, s) \geq H$$
(both ends of the table are higher than some threshold H)

$$Level(s) \overset{def}{=} |vpos(End_1, s) - vpos(End_2, s)| \leq Tol$$
(both ends are at the same height to within a threshold Tol).

So the goal is

$$Goal(s) \overset{def}{=} TableUp(s) \wedge \; \forall s^*. s^* \leq s \supset Level(s^*)$$

and the claim is that this goal can be achieved by having Rob_1 and Rob_2 each concurrently execute the same procedure *ctrl* defined as:

```
proc $ctrl(rob)$
    $\pi\, e[TableEnd(e)?;\ grab(rob, e)]$;
    while $\neg TableUp(now)$ do
        $SafeToLift(rob, now)?$;
        $vmove(rob, Amount)$
end
```

where $Amount$ is some constant such that $0 < Amount < Tol$, and $SafeToLift$ is defined by

$$\begin{aligned}SafeToLift(rob, s) \overset{def}{=}\ & \exists e \exists e^*\, e \neq e^* \wedge TableEnd(e) \wedge TableEnd(e^*) \wedge \\ & Holding(rob, e, s) \wedge vpos(e) \leq vpos(e^*) + Tol - Amount.\end{aligned}$$

Here, we use procedures simply for convenience and the reader can take them as abbreviations. A formal treatment for procedures will be provided in section 7.

So formally, the claim is:[12]

$$\mathcal{C} \cup \mathcal{D} \models \forall s. Do(ctrl(Rob_1) \| ctrl(Rob_2), S_0, s) \supset Goal(s)$$

Here is an informal sketch of a proof. *Do* holds if and only if there is a finite sequence of transitions from the initial configuration $(ctrl(Rob_1) \| ctrl(Rob_2), S_0)$ to a configuration that is *Final*. A program involving two concurrent processes can only get to a *Final* configuration by reaching a configuration that is *Final* for both processes. The processes in our program involve while-loops, which only reach a final configuration when the loop condition becomes is false. So the table must be high enough in the final situation.

It remains to be shown is that the table stayed level. Let v_i stand for the action $vmove(rob_i, Amount)$. Suppose to the contrary that the table went too high on End_1 held by Rob_1, and consider the first configuration where this became true. This situation in this configuration is of the form $do(v_1, s)$ where

$$vpos(End_1, do(v_1, s)) > vpos(End_2, do(v_1, s)) + Tol.$$

However, at some earlier configuration, we had to have $SafeToLift(Rob_1, s')$ with no intervening actions by Rob_1, otherwise the last v_1 would not have been executed. This means that we have

$$vpos(End_1, s') \leq\ vpos(End_2, s') + Tol - Amount.$$

However, if all the actions between s' and s are by Rob_2, since Rob_2 can only increase the value of $vpos(End_2)$, it follows that

$$vpos(End_1, s) \leq\ vpos(End_2, s) + Tol - Amount,$$

that is, that $SafeToLift$ was also true just before the final v_1 action. This contradicts the assumption that v_1 only adds $Amount$ to the value of $vpos(End_1)$.

[12] Actually, proper termination of the program is also guaranteed. However, stating this condition formally, in the case of concurrency, requires additional machinery, since $\exists s Do(ctrl(Rob_1) \| ctrl(Rob_2), S_0, s)$ is too weak.

6.2 A Reactive Multi-Elevator Controller

Our next example involves a reactive controller for a bank of elevators; it illustrates the use of interrupts and prioritized concurrency. 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:
  $TooHot(e, s) \overset{def}{=} temp(e, s) > 1$
  $TooCold(e, s) \overset{def}{=} temp(e, s) < -1$

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:
  $\forall a. Exo(a) \quad \equiv \quad a = detectSmoke \vee a = resetAlarm \vee$
  $\quad a = changeTemp(e) \vee \exists n. a = reqElevator(n)$

- Precondition axioms:
  $Poss(goDown(e), s) \quad \equiv \quad floor(e, s) \neq 1$
  $Poss(goUp(e), s) \quad \equiv \quad floor(e, s) \neq 6$
  $Poss(buttonReset(n), s) \quad \equiv \quad True$

$$
\begin{aligned}
&Poss(toggleFan(e), s) \;\equiv\; True\\
&Poss(ringAlarm) \;\equiv\; True\\
&Poss(reqElevator(n), s) \;\equiv\; (1 \le n \le 6) \land \neg ButtonOn(n, s)\\
&Poss(changeTemp, s) \;\equiv\; True\\
&Poss(detectSmoke, s) \;\equiv\; \neg Smoke(s)\\
&Poss(resetAlarm, s) \;\equiv\; Smoke(s)
\end{aligned}
$$

- Successor state axioms:

$$
\begin{aligned}
&Poss(a, s) \supset [floor(e, do(a, s)) = n \;\equiv\\
&\quad (a = goDown(e) \land n = floor(e, s) - 1) \lor\\
&\quad (a = goUp(e) \land n = floor(e, s) + 1) \lor\\
&\quad (n = floor(e, s) \land a \neq goDown(e) \land a \neq goUp(e))]\\
&Poss(a, s) \supset [temp(e, do(a, s)) = t \;\equiv\\
&\quad (a = changeTemp(e) \land FanOn(e, s) \land t = temp(e, s) - 1) \lor\\
&\quad (a = changeTemp(e) \land \neg FanOn(e, s) \land t = temp(e, s) + 1) \lor\\
&\quad (t = temp(e, s) \land a \neq changeTemp(e))]\\
&Poss(a, s) \supset [FanOn(e, do(a, s)) \;\equiv\\
&\quad (a = toggleFan(e) \land \neg FanOn(e, s)) \lor\\
&\quad (a \neq toggleFan(e) \land FanOn(e, s))]\\
&Poss(a, s) \supset [ButtonOn(n, do(a, s)) \;\equiv\\
&\quad a = reqElevator(n) \lor\\
&\quad (ButtonOn(n, s) \land a \neq buttonReset(n))]\\
&Poss(a, s) \supset [Smoke(do(a, s)) \;\equiv\\
&\quad a = detectSmoke \lor\\
&\quad (Smoke(s) \land a \neq resetAlarm)]
\end{aligned}
$$

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 for an elevator e. 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:

```
proc $serveFloor(e, n)$
    while $floor(e) < n$ do $goUp(e)$;
    while $floor(e) > n$ do $goDown(e)$;
    $buttonReset(n)$
end
```

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

As with *Golog*, we try to prove an existential and look at the bindings for the s. They will be of the form $do(\vec{a}, S_0)$ where $\vec{a}$ are the actions to perform. In particular, using this controller program δ, we would get execution traces like

$$\begin{array}{lll} \mathcal{C} \cup \mathcal{D} & \models & Do(\delta \parallel \delta_{EXO}, S_0, do([u, u, b_3, u, u, u, b_6, d, d, d, d, d], S_0)) \\ \mathcal{C} \cup \mathcal{D} & \models & Do(\delta \parallel \delta_{EXO}, S_0, do([u, r_4, u, b_3, u, b_4, u, u, r_2, b_6, d, d, d, d, b_2, d], S_0)) \\ & \ldots & \end{array}$$

where $u = goUp(e)$, $d = goDown(e)$, $b_n = buttonReset(n)$, $r_n = reqElevator(n)$, and $\mathcal{D}$ is the basic action theory specified above. In the first run there were no exogenous actions, while in the second, two elevator requests were made.

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:

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

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

$$\begin{array}{l} < \exists n.EButtonOn(n) \rightarrow \\ \quad \pi n.\{EButtonOn(n)?; serveEFloor(e, n)\} > \end{array}$$

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:

$$\begin{array}{l} < TooHot(e) \wedge \neg FanOn(e) \rightarrow toggleFan(e) > \\ < TooCold(e) \wedge FanOn(e) \rightarrow toggleFan(e) > \end{array}$$

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:

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

Using this controller δ_r, we would get execution traces like

$$\begin{array}{lll}
\mathcal{C} \cup \mathcal{D} & \models & Do(\delta_r \parallel \delta_{EXO}, S_0, do([u,u,b_3,u,u,u,b_6,d,d,d,d,r_5,u,u,u,b_5,d,d,d,d], S_0)) \\
\mathcal{C} \cup \mathcal{D} & \models & Do(\delta_r \parallel \delta_{EXO}, S_0, do([u,u,b_3,u,z,a,a,a,a,h,u,u,b_6,d,d,d,d,d], S_0)) \\
\mathcal{C} \cup \mathcal{D} & \models & Do(\delta_r \parallel \delta_{EXO}, S_0, do([u,t,u,b_3,u,t,f,u,t,t,u,t,b_6,d,t,f,d,t,d,d,d], S_0)) \\
 & \ldots &
\end{array}$$

where $z = detectSmoke$, $a = ringAlarm$, $h = resetAlarm$, $t = changeTemp$, and $f = toggleFan$. In the first run, we see that this controller does handle requests that come in while the elevator is on its way to retire on the bottom floor. The second run illustrates how the controller reacts to smoke being detected by ringing the alarm. The third run shows how the controller reacts immediately to temperature changes while it is serving floors. 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 $(\delta_1 \parallel \delta_2)$, where δ_1 is the above program with e replaced by $Elevator_1$ and δ_2 is the same program with e replaced by $Elevator_2$. This allows the two processes to work completely independently (in terms of priorities).[13] For n elevators, we would use $(\delta_1 \parallel \cdots \parallel \delta_n)$.

[13]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.

6.3 A Client-Server System

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 $\delta^{\|}$ concurrent iteration construct.

Let us give a high-level sketch of how this might be done. Suppose that there is an exogenous action $newClient(cid)$ that occurs when a new client with the ID *cid* first requests service. Also assume that a procedure $serve(cid)$ has been defined, which implements the behavior required for the server for a given client. To set up the system, we run the program:

$$\begin{array}{l}[\pi\, cid.\, acquire(cid); serve(cid)]^{\|};\\ \qquad \neg\exists cid.\, (ClientWaiting(cid))?\end{array}$$

Here, we assume that when the exogenous action $newClient(cid)$ occurs, it makes the fluent $ClientWaiting(cid)$ true. Then, the only way the computation can be completed is by generating an new process that first acquires the client by doing $acquire(cid)$, and then serves it. We formalize this as follows:

$$Poss(acquire(cid), s) \equiv ClientWaiting(cid)$$

$$\begin{array}{l}Poss(a, s) \supset [ClientWaiting(cid, do(a, s)) \equiv\\ \qquad a = newClient(cid) \vee ClientWaiting(cid, s) \wedge a \neq acquire(cid)]\end{array}$$

Then, only a single process can acquire a given client, since *acquire* is only possible when $ClientWaiting(cid)$ is true and performing it makes this fluent false. The whole program can only reach a final configuration if it forks exactly the right number of server processes: at least one for each client because only one server can acquire a client, and no more than one for each client because servers can be activated only if they can acquire a client.

6.4 Actions with Extended Duration

One possible criticism of our approach to concurrency is that it does not work when we consider actions that have extended duration. Consider singing while filling the bathtub with water, for example. If one of the actions involved is “filling the bathtub,” and the other actions are “singing do,” “singing re,” and “singing mi,” say, then there are exactly four possible interleavings,

$$\begin{array}{l}[filling\ ;\ do\ ;\ re\ ;\ mi],\\ [do\ ;\ filling\ ;\ re\ ;\ mi],\\ [do\ ;\ re\ ;\ filling\ ;\ mi],\\ [do\ ;\ re\ ;\ mi\ ;\ filling],\end{array}$$

but none of them capture the idea of singing and filling the tub at the same time. Moreover, the prospect of replacing the filling action by a large number of component actions (that could be interleaved with the singing ones) is even less appealing.

To deal with this type of case, we recommend the following approach (see [30] for a detailed presentation): instead of thinking of filling the bathtub as an *action* or group of actions, think of it as a *state* that an agent could be in, extending possibly over many situations. The idea is that the agent can be in many such states simultaneously, including listening to the radio, walking, and chewing gum. For each such state, we need two primitive actions and a fluent; for the bathtub, they are $startFilling$, which puts the agent into the state, and $endFilling$, which terminates it, as well as the fluent $FillingTub$, which holds in those situations where the agent is filling the tub. Formally, we would express this with a successor state axiom as follows:

$$\begin{array}{l} Poss(a,s) \supset [FillingTub(do(a,s)) \quad \equiv \\ \quad a = startFilling \vee FillingTub(s) \wedge a \neq endFilling]. \end{array}$$

Since the $startFilling$ and $endFilling$ actions can be taken to be instantaneous, the interleaving account is once again plausible. If we define a complex action

$$FillTheTub \stackrel{def}{=} [startFilling\ ;\ endFilling]$$

then, we get these possible interleavings:

$$\begin{array}{l}
[startFilling\ ;\ endFilling\ ;\ do\ ;\ re\ ;\ mi], \\
[startFilling\ ;\ do\ ;\ endFilling\ ;\ re\ ;\ mi], \\
[startFilling\ ;\ do\ ;\ re\ ;\ endFilling\ ;\ mi], \\
[startFilling\ ;\ do\ ;\ re\ ;\ mi\ ;\ endFilling], \\
[do\ ;\ startFilling\ ;\ endFilling\ ;\ re\ ;\ mi], \\
[do\ ;\ startFilling\ ;\ re\ ;\ endFilling\ ;\ mi], \\
[do\ ;\ startFilling\ ;\ re\ ;\ mi\ ;\ endFilling], \\
[do\ ;\ re\ ;\ startFilling\ ;\ endFilling\ ;\ mi], \\
[do\ ;\ re\ ;\ startFilling\ ;\ mi\ ;\ endFilling], \\
[do\ ;\ re\ ;\ mi\ ;\ startFilling\ ;\ endFilling].
\end{array}$$

A better model would be something like

$$FillTheTub \stackrel{def}{=} [startFilling\ ;\ (waterLevel > H)?\ ;\ endFilling]$$

which would rule out interleavings where the filling stops too soon. The most natural way of modeling the water level is as a continuous function of time: $l = L_0 + R \times t$, where L_0 is the initial level, R is the rate of filling (taken to be constant), and t is the elapsed time. One simple way to accommodate this idea within the situation calculus is to assume that every action has a duration $dur(a)$ (which we could also make dependent on the situation the action is performed in). Actions such as $startFilling$ can have duration 0, but there must be some action, if only a $timePasses$, with a non-0 duration. We then describe the $waterLevel$ functional fluent by:

$$Poss(a,s) \supset \; waterLevel(do(a,s)) = waterLevel(s) + waterRate(s) \times dur(a).$$

$$Poss(a,s) \supset \; waterRate(do(a,s)) = \textbf{if } FillingTub(s) \textbf{ then } R \textbf{ else } 0.$$

So as long as a situation is in a filling-the-tub state, the water level rises according to the above equation. In terms of concurrency, the result is that the only allowable interleavings would be those where enough actions of sufficient duration occur between the $startFilling$ and $stopFilling$.

Of course, this model of the continuous process of water entering the bathtub does not allow us to predict the eventual outcome, for example, the water overflowing if a tap is not turned off, *etc.* A more complex program, typically involving interrupts, would be required, so that suitable "trajectory altering" actions are triggered under the appropriate conditions.

7 Procedures

In this section we introduce procedures. This will require us to adopt a second-order definition of *Trans* and *Final*. Formal details can be found in [6].[14]

Let $\textbf{proc } P_1(\vec{v}_1)\delta_1 \textbf{ end}; \ldots; \textbf{proc } P_n(\vec{v}_n)\delta_n \textbf{ end}$ be a collection of procedure definitions. We call such a collection an *environment* and denote it by Env. In a procedure definition $\textbf{proc } P_i(\vec{v}_i)\delta_i \textbf{ end}$, P_i is the name of the i-th procedure in Env; $\vec{v}_i$ are its formal parameters; and δ_i is the procedure body, which is a *ConGolog* program possibly including both *procedure calls* and new procedure definitions. We use *call-by-value* as the parameter passing mechanism, and *lexical (or static) scope* as the scoping rule.

Formally we introduce three program constructs:

- $P(\vec{t})$ where P is a procedure name and $\vec{t}$ actual parameters associated to the procedure P; as usual we replace the situation argument in the terms constituting $\vec{t}$ by now. $P(\vec{t})$ denotes a procedure call, which invokes procedure P on the actual parameters $\vec{t}$ evaluated in the current situation.

- $\{Env; \delta\}$, where Env is an environment and δ is a program extended with procedures calls. $\{Env; \delta\}$ binds procedures calls in δ to the definitions given in Env. The usual notion of free and bound apply, so for e.g. in $\{\textbf{proc } P_1() \; a \textbf{ end}; P_2(); P_1()\}$, P_1 is bound but P_2 is free.

[14]Obviously, with respect to *ConGolog* programs without procedures, *Trans* and *Final* introduced here are equivalent to the versions introduced in Section 4, see [6].

- $[Env : P(\vec{t})]$, where Env is an environment, P a procedure name and $\vec{t}$ actual parameters associated to the procedure P. $[Env : P(\vec{t})]$ denotes a procedure call that has already been contextualized: the environment in which the definition of P is to be looked for is Env.

We define the semantics of *ConGolog* programs with procedures by defining both *Trans* and *Final* by a second-order formula (instead of a set of axioms).[15] *Trans* is defined as follows:

$$Trans(\delta, s, \delta', s') \equiv \forall T.[\ \ldots \quad \supset \quad T(\delta, s, \delta', s')]$$

where ... stands for the conjunction of $\mathcal{T}_T^{Trans}$ – i.e. the set of axioms $\mathcal{T}$ from Sections 4 and 5 modulo textual substitution of *Trans* with T – and (the universal closure of) the following two assertions:

$$\begin{array}{rcl} T(\{Env; \delta\}, s, \delta', s') & \equiv & T(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s, \delta', s') \\ T([Env : P(\vec{t})], s, \delta', s') & \equiv & T(\{Env; \delta_P{}^{\vec{v}_P}_{\vec{t}[s]}\}, s, \delta', s') \end{array}$$

where $\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}$ denotes the program δ with all procedures bound by Env and free in δ replaced by their contextualized version (this gives us the lexical scope), and where $\delta_P{}^{\vec{v}_P}_{\vec{t}[s]}$ denotes the body of the procedure P in Env with formal parameter $\vec{v}$ substituted by the actual parameters $\vec{t}$ evaluated in the current situation.

Similarly, *Final* is defined as follows:

$$Final(\delta, s) \equiv \forall F.[\ \ldots \quad \supset \quad F(\delta, s)]$$

where ... stands for the conjunction of $\mathcal{F}_F^{Final}$ – i.e. the set of axioms $\mathcal{F}$ modulo textual substitution of *Final* with F – and (the universal closure of) the following assertion:

$$\begin{array}{rcl} F(\{Env; \delta\}, s) & \equiv & F(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s) \\ F([Env : P(\vec{t})], s) & \equiv & F(\{Env; \delta_P{}^{\vec{v}_P}_{\vec{t}[s]}\}, s) \end{array}$$

Note that no assertions for (uncontextualized) procedure calls are present in the definitions of *Trans* and *Final*. Indeed a procedure call which cannot be bound to a procedure definition neither can do transitions nor can be considered successfully completed.

Observe also the two uses of substitution to deal with procedure calls. When a program with an associated environment is executed, for all procedure calls bound by Env, we simultaneously substitute the corresponding procedure calls, contextualized by the *environment of the procedure* in order to deal with further procedure calls according to the *static scope* rules. Then when a (contextualized) procedure is actually executed, the

[15]For compatibility with the formalization in Section 4, we treat *Trans* and *Final* as predicates, although it is clear that they could be understood as abbreviations for the second-order formulas.

actual parameters are first evaluated in the current situation, and then are substituted for the formal parameters in the procedure bodies[16], thus yielding *call-by-value* parameter passing. See [6] for some examples and additional discussion.

The need for a second-order definition of $\mathit{Trans}(\delta, s, \delta', s')$ and $\mathit{Final}(\delta, s)$ when procedures are introduced comes from recursive procedures. The second-order definition allows us to assign a formal semantics to every such procedure, including viciously circular ones. The definition of *Trans* disallows the execution of such ill-formed procedures. At the same time the definition of *Final* considers them not to have completed (non-final). For example, the program $\{\mathbf{proc}\ P()\ P()\ \mathbf{end}; P()\}$ does not have any transitions, but it is not final for any situation s.

8 Implementation

Despite the fact that in defining the semantics of *ConGolog* we resorted to first- and second-order logic, it is possible to come up with a simple implementation of the *ConGolog* language in Prolog.

In this section, we present a *ConGolog* interpreter in Prolog which is lifted directly from the definition of *Final*, *Trans*, and *Do* introduced above.[17] This interpreter requires that the program's precondition axioms, successor state axioms, and axioms about the initial situation be expressible as Prolog clauses. In particular, the usual *closed world assumption* (CWA) is made on the initial situation. Note that this is a limitation of this particular implementation, not the theory.

Prolog terms representing *ConGolog* programs are as follows:

- `nil`, empty program.
- `act(`a`)`, atomic action, where a is an action term with the situation arguments replaced by the constant `now`.
- `test(`c`)`, wait/test, where c is a condition described below.
- `seq(`p_1`,`p_2`)`, sequence.
- `choice(`p_1`,`p_2`)`, nondeterministic branch.
- `pick(`v`,`p`)`, nondeterministic choice of argument, where v is a Prolog constant (atom), standing for a *ConGolog* variable, and p a program-term that uses v.

[16]To be more precise, every formal parameter v is substituted by a term of the form `nameOf(`$t[s]$`)`, where again `nameOf` is used to convert situation calculus objects/actions into program terms of the corresponding sort.

[17]Exogenous actions can be generated by simulating them probabilistically, by asking the user at runtime when they should occur, or by monitoring the environment in which the program is running.

- `iter(`p`)`, nondeterministic iteration.
- `if(`c`,`p_1`,`p_2`)`, if-then-else, with p_1 the then-branch and p_2 the else-branch.
- `while(`c`,`p`)`, while-do.
- `conc(`p_1`,`p_2`)`, concurrency.
- `prconc(`p_1`,`p_2`)`, prioritized concurrency.
- `iterconc(`p`)`, iterated concurrency.
- `pcall(`$pArgs$`)`, procedure call, with $pArgs$ the procedure name and arguments.

A condition c in the above is either a Prolog-term representing an atomic formula/fluent with the situation arguments replaced by `now` or an expression of the form `and(`c_1`,`c_2`)`, `or(`c_1`,`c_2`)`, `neg(`c`)`, `all(`v`,`c`)`, or `some(`v`,`c`)`, with the obvious intended meaning. In `all(`v`,`c`)` and `some(`v`,`c`)`, v is an Prolog constant, standing for a logical variable, and c a condition using v.

The Prolog predicate `trans`/4, `final`/2, `trans*`/4 and `do`/3 implement respectively the predicate *Trans*, *Final*, *Trans** and *Do*.

The Prolog predicate `holds/2` is used to evaluate conditions in tests, while-loops and if-then-else's in *ConGolog* programs. As well, the Prolog predicate `sub`/4 implements the substitution so that `sub`(x, y, t, t') means that $t' = t^x_y$. The definition of these two Prolog predicates is taken from [20, 31].

The following is the Prolog code.

```
/*********************************************************************/
/*                 Trans-based ConGolog Interpreter                  */
/*********************************************************************/

/* trans(Prog,Sit,Prog_r,Sit_r) */

trans(act(A),S,nil,do(AS,S)) :- sub(now,S,A,AS), poss(AS,S).

trans(test(C),S,nil,S) :- holds(C,S).

trans(seq(P1,P2),S,P2r,Sr) :- final(P1,S),trans(P2,S,P2r,Sr).
trans(seq(P1,P2),S,seq(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).

trans(choice(P1,P2),S,Pr,Sr) :- trans(P1,S,Pr,Sr) ; trans(P2,S,Pr,Sr).

trans(pick(V,P),S,Pr,Sr) :- sub(V,_,P,PP), trans(PP,S,Pr,Sr).
trans(iter(P),S,seq(PP,iter(P)),Sr) :- trans(P,S,PP,Sr).

trans(if(C,P1,P2),S,Pr,Sr) :- holds(C,S),trans(P1,S,Pr,Sr) ;
                              holds(neg(C),S),trans(P2,S,Pr,Sr).

trans(while(C,P),S,seq(PP,while(C,P)),Sr) :- holds(C,S),trans(P,S,PP,Sr).

trans(conc(P1,P2),S,conc(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).
trans(conc(P1,P2),S,conc(P1,P2r),Sr) :- trans(P2,S,P2r,Sr).

trans(prconc(P1,P2),S,prconc(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).
trans(prconc(P1,P2),S,prconc(P1,P2r),Sr) :- not trans(P1,S,_,_),trans(P2,S,P2r,Sr).

trans(iterconc(P),S,conc(PP,iterconc(P)),Sr) :- trans(P,S,PP,Sr).

trans(pcall(P_Args),S,Pr,Sr) :- sub(now,S,P_Args,P_ArgsS),
                                proc(P_ArgsS,P), trans(P,S,Pr,Sr).


/* final(Prog,Sit) */

final(nil,S).

final(seq(P1,P2),S) :- final(P1,S),final(P2,S).

final(choice(P1,P2),S) :- final(P1,S) ; final(P2,S).

final(pick(V,P),S) :- sub(V,_,P,PP), final(PP,S).

final(iter(P),S).

final(if(C,P1,P2),S) :- holds(C,S),final(P1,S) ;
                        holds(neg(C),S),final(P2,S).

final(while(C,P),S) :- holds(neg(C),S) ; final(P,S).

final(conc(P1,P2),S) :- final(P1,S),final(P2,S).

final(prconc(P1,P2),S) :- final(P1,S),final(P2,S).

final(iterconc(P),S).

final(pcall(P_Args)) :- sub(now,S,P_Args,P_ArgsS),
                        proc(P_ArgsS,P),final(P,S).
```

```
/* trans*(Prog,Sit,Prog_r,Sit_r) */

trans*(P,S,P,S).
trans*(P,S,Pr,Sr) :- trans(P,S,PP,SS), trans*(PP,SS,Pr,Sr).


/* do(Prog,Sit,Sit_r) */

do(P,S,Sr) :- trans*(P,S,Pr,Sr),final(Pr,Sr).


/* holds(Cond,Sit): as defined in [32] */

holds(and(F1,F2),S) :- holds(F1,S), holds(F2,S).
holds(or(F1,F2),S) :- holds(F1,S); holds(F2,S).
holds(all(V,F),S) :- holds(neg(some(V,neg(F))),S).
holds(some(V,F),S) :- sub(V,_,F,Fr), holds(Fr,S).
holds(neg(neg(F)),S) :- holds(F,S).
holds(neg(and(F1,F2)),S) :- holds(or(neg(F1),neg(F2)),S).
holds(neg(or(F1,F2)),S) :- holds(and(neg(F1),neg(F2)),S).
holds(neg(all(V,F)),S) :- holds(some(V,neg(F)),S).
holds(neg(some(V,F)),S) :- not holds(some(V,F),S).  /* Negation by failure */
holds(P_Xs,S) :-
   P_Xs\=and(_,_),P_Xs\=or(_,_),P_Xs\=neg(_),P_Xs\=all(_,_),P_Xs\=some(_._),
   sub(now,S,P_Xs,P_XsS), P_XsS.
holds(neg(P_Xs),S) :-
   P_Xs\=and(_,_),P_Xs\=or(_,_),P_Xs\=neg(_),P_Xs\=all(_,_),P_Xs\=some(_._),
   sub(now,S,P_Xs,P_XsS), not P_XsS.               /* Negation by failure */


/* sub(Const,Var,Term1,Term2): as defined in [32] */

sub(X,Y,T,Tr) :- var(T), Tr = T.
sub(X,Y,T,Tr) :- not var(T), T = X, Tr = Y.
sub(X,Y,T,Tr) :- T \= X, T =..[F|Ts], sub_list(X,Y,Ts,Trs), Tr =..[F|Trs].
sub_list(X,Y,[],[]).
sub_list(X,Y,[T|Ts],[Tr|Trs]) :- sub(X,Y,T,Tr), sub_list(X,Y,Ts,Trs).
```

In this implementation a *ConGolog* application is expected to have the following parts:

1. A collection of clauses which together define which fluents are true in the initial situation `s0`. The clauses need not to be atomic, and can involve arbitrary amount

of computation for determining entailments in the initial database.

2. A collection of clauses which together define the predicate $Poss(a, s)$ for every action a and situation s. Typically, this requires one clause per action, using a variable to range over all situations.

3. A collection of clauses which together define the successor state axioms for each fluent. Typically, this requires one clause per fluent, with variables for actions and situations.

4. A collection of facts defining *ConGolog* procedures. In particular for each procedure p occurring in the program we have a fact of the form:

   $$\texttt{proc}(p(X_1, \ldots, X_n), \mathit{body})$$

   In such facts: (i) formal parameters are represented as Prolog variables so as to use Prolog built-in unification mechanism instead of a substitution procedure; (ii) in the body *body* the only variables that can occur are those representing the formal parameters $X_1, \ldots, X_n$. For simplicity, we do not consider nested procedures in the above implementation.

Expressing action theories as Prolog clauses places a number of restrictions on the action theories that are representable. These restrictions force the closed world assumption (Prolog CWA) on the initial situation and the unique name assumption (UNA) on both actions and objects. For an in-depth study on action theories expressible as Prolog clauses, we refer to [31].

8.1 Example

Below, we give an implementation in Prolog of the two robots lifting a table scenario discussed in subsection 6.1. The code is written as close to the specification as possible. The inability of Prolog to define directly the functional fluent $vpos(e, s)$ is resolved by introducing a predicate `val/2` such that $\texttt{val}(\texttt{vpos}(e, s), v)$ stands for $vpos(e, s) = v$.

```
/*********************************************************************/
/*               Two Robots Lifting a Table Example                  */
/*********************************************************************/

/* Precondition axioms */

poss(grab(Rob,E),S) :- not holding(_,E,S), not holding(Rob,_,S).
poss(release(Rob,E),S) :- holding(Rob,E,S).
poss(vmove(Rob,Amount),S) :- true.
```

```
/* Succ state axioms */

val(vpos(E,do(A,S)),V) :-
   (A=vmove(Rob,Amount), holding(Rob,E,S), val(vpos(E,S),V1), V is V1+Amount) ;
   (A=release(Rob,E), V=0) ;
   (val(vpos(E,S),V), not((A=vmove(Rob,Amount), holding(Rob,E,S))),
                      A\=release(Rob,E)).

holding(Rob,E,do(A,S)) :-
   A=grab(Rob,E) ; (holding(Rob,E,S), A\=release(Rob,E)).

/* Defined Fluents */

tableUp(S) :- val(vpos(end1,S),V1), V1 >= 3, val(vpos(end2,S),V2), V2 >= 3.

safeToLift(Rob,Amount,Tol,S) :-
   tableEnd(E1), tableEnd(E2), E2\=E1, holding(Rob,E1,S),
   val(vpos(E1,S),V1), val(vpos(E2,S),V2), V1 =< V2+Tol-Amount.

/* Initial state */

val(vpos(end1,s0),0).       /* plus by CWA:           */
val(vpos(end2,s0),0).       /*                        */
tableEnd(end1).             /* not holding(rob1,_,s0) */
tableEnd(end2).             /* not holding(rob2,_,s0) */

/* Control procedures  */

proc(ctrl(Rob,Amount,Tol),
  seq(pick(e,seq(test(tableEnd(e)),act(grab(Rob,e)))),
    while(neg(tableUp(now)),
       seq(test(safeToLift(Rob,Amount,Tol,now)),
                  act(vmove(Rob,Amount)))))).

proc(jointLiftTable,
        conc(pcall(ctrl(rob1,1,2)), pcall(ctrl(rob2,1,2)))).
```

Below we show a few final situations returned by the interpreter for the above example (note that the interpreter does not filter out identical situations).

```
?- do(pcall(jointLiftTable),s0,S).

S = do(vmove(rob2, 1), do(vmove(rob1, 1), do(vmove(rob2, 1), do(vmove(rob1, 1),
 do(vmove(rob2, 1), do(grab(rob2, end2), do(vmove(rob1, 1), do(vmove(rob1, 1),
do(grab(rob1, end1), s0))))))))) ;

S = do(vmove(rob2, 1), do(vmove(rob1, 1), do(vmove(rob2, 1), do(vmove(rob1, 1),
 do(vmove(rob2, 1), do(grab(rob2, end2), do(vmove(rob1, 1), do(vmove(rob1, 1),
do(grab(rob1, end1), s0))))))))) ;

S = do(vmove(rob1, 1), do(vmove(rob2, 1), do(vmove(rob2, 1), do(vmove(rob1, 1),
 do(vmove(rob2, 1), do(grab(rob2, end2), do(vmove(rob1, 1), do(vmove(rob1, 1),
do(grab(rob1, end1), s0)))))))))

Yes
```

8.2 Correctness of the Prolog implementation

In this section we prove the correctness of the interpreter presented above under suitable assumptions. Let $\mathcal{C}$ be the set of axioms for *Trans*, *Final*, and *Do* plus those needed for the encoding of programs as first-order terms, and $\mathcal{D}$ the domain theory. To keep notation simple we denote the condition corresponding to a situation calculus formula ϕ with the situation argument replaced by *now*, simply by ϕ. Similarly for Prolog terms corresponding to actions and programs.

Our proof of correctness relies on the following assumptions:

- The domain theory $\mathcal{D}$ enforces the unique name assumption (UNA) on both actions and objects.[18]
- The predicate `sub`/4 correctly implements substitution for both programs and formulas.
- The predicate `holds`/2 satisfies the following properties:
  1. If a goal `holds`(ϕ, s), with free variables only on object terms and action terms, succeeds with computed answer θ, then $\mathcal{D} \models \forall \phi[s]\theta$ (by $\forall\psi$, we mean the universal closure of ψ).
  2. If a goal `holds`(ϕ, s), with free variables only on object terms and action terms, finitely fails, then $\mathcal{D} \models \forall \neg\phi[s]$.
- The predicate `poss`/2 satisfies the following properties:
  1. If a goal `poss`(a, s), with free variables only on object terms and action terms, succeeds with computed answer θ then $\mathcal{D} \models \forall Poss(a, s)\theta$.
  2. If a goal `poss`(a, s), with free variables only on object terms and action terms, finitely fails, then $\mathcal{D} \models \forall \neg Poss(a, s)$.

[18]UNA is already enforced for programs, see [6].

- The Prolog interpreter flounders (and hence does not return) on goals of the form `not trans`$(\delta, s, \_, \_)$[19] with non-ground δ and s.[20]

Observe that the hypotheses required for `sub`/4, `holds`/2 and `poss`/2 do hold when these predicates are defined as above and run by an interpreter that flounders on non-ground negative goals (see [31]).

Theorem 2: *Under the hypotheses above the following holds:*

1. *If a goal* `do`(δ, s, s')*, where* δ *and* s *may contain variables only on object terms and action terms, succeeds with computed answer* θ*, then* $\mathcal{C} \cup \mathcal{D} \models \forall Do(\delta, s, s')\theta$*, moreover* $s'\theta$ *may contain free variables only on object terms and action terms.*

2. *If a goal* `do`(δ, s, s')*, where* δ *and* s *may contain variables only on object terms and action terms, finitely fails, then* $\mathcal{C} \cup \mathcal{D} \models \forall \neg Do(\delta, s, s')$*.*

To make the arguments more apparent we will first prove the theorem without considering procedures. Then we show how introducing procedures affects the proof.

Without procedures

Theorem 2 is an easy consequence of Lemma 1 and Lemma 2 below.

Lemma 1: *Under the hypotheses above the following holds:*

- *The predicate* `trans`/4 *satisfies the following properties:*

  1. *If a goal* `trans`(δ, s, δ', s')*, where* δ *and* s *may contain variables only on object terms and action terms, succeeds with computed answer* θ*, then* $\mathcal{C} \cup \mathcal{D} \models \forall Trans(\delta, s, \delta', s')\theta$*, moreover* $\delta'\theta$ *and* $s'\theta$ *may contain free variables only on object terms and action terms.*
  2. *If a goal* `trans`(δ, s, δ', s')*, where* δ *and* s *may contain variables only on object terms and action terms, finitely fails, then* $\mathcal{C} \cup \mathcal{D} \models \forall \neg Trans(\delta, s, \delta', s')$*.*

- *The predicate* `final`/2 *satisfies the following properties:*

  1. *If a goal* `final`(δ, s)*, where* δ *and* s *may contain variables only on object terms and action terms, succeeds with computed answer* θ*, then* $\mathcal{C} \cup \mathcal{D} \models \forall Final(\delta, s)\theta$*.*
  2. *If a goal* `final`(δ, s)*, where* δ *and* s *may contain variables only on object terms and action terms, finitely fails, then* $\mathcal{C} \cup \mathcal{D} \models \forall \neg Final(\delta, s)$*.*

[19] From a formal point of view `not trans`$(\delta, s, \_, \_)$ is a shorthand for `not aux`(δ, s) with `aux`/2 defined as `aux`(δ, s) `:-` `trans`$(\delta, s, \_, \_)$.

[20] This form of floundering arises for example when we expand π in programs of the form $\pi z.(\delta_1(z) \rangle\!\rangle \delta_2(z))$. Notably it does not arise for their variants $\pi z.(\phi(z)?; (\delta_1(z) \rangle\!\rangle \delta_2(z)))$.

Proof: First we observe that since we are not considering procedures, *Trans* and *Final* satisfy the axioms $\mathcal{T}$ and $\mathcal{F}$ from Sections 4 and 5. We prove simultaneously 1 and 2 for both `trans`/4 and `final`/2 by induction on the program δ. Here we show only the case $\delta = \delta_1 \rangle\!\rangle\, \delta_2$ for `trans`/4.

Success. If $\mathtt{trans}(\delta_1 \rangle\!\rangle\, \delta_2, s, \delta', s')$ succeeds with computed answer θ, then: either (i) $\mathtt{trans}(\delta_1, s, \delta_1', s')$ succeeds with computed answer θ_1, and $\theta = \theta'\theta_1$ where $\theta' = mgu(\delta', \delta_1' \rangle\!\rangle\, \delta_2)$ is the most general unifier [21] between δ' and $\delta_1' \rangle\!\rangle\, \delta_2$; or (ii) $\mathtt{trans}(\delta_1, s, \_, \_)$ finitely fails and $\mathtt{trans}(\delta_2, s, \delta_2', s')$ succeeds with computed answer θ_2 and $\theta = mgu(\delta', \delta_1 \rangle\!\rangle\, \delta_2')\theta_2$. In case (i) by the induction hypothesis $\mathcal{C} \cup \mathcal{D} \models \forall\, Trans(\delta_1, s, \delta_1', s')\theta_1$, and $s'\theta_1$ and $\delta_1'\theta_1$ may contain free variables only on object terms and action terms. In case (ii) by the induction hypothesis $\mathcal{C} \cup \mathcal{D} \models \forall \neg\, Trans(\delta_1, s, \delta_1', s_1')$, $\mathcal{C} \cup \mathcal{D} \models \forall\, Trans(\delta_2, s, \delta_2', s')\theta_2$, and $s'\theta_2$ and $\delta_2'\theta_2$ may contain free variables only on object terms and action terms. Considering

$$\begin{aligned} & Trans(\delta_1 \rangle\!\rangle\, \delta_2, s, \delta', s') \;\equiv \\ & \quad \exists\gamma.\delta' = (\gamma \rangle\!\rangle\, \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \;\vee \\ & \quad \exists\gamma.\delta' = (\delta_1 \rangle\!\rangle\, \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''. Trans(\delta_1, s, \zeta, s'') \end{aligned} \tag{1}$$

and how θ is defined in both cases, we get the thesis.

Failure. If $\mathtt{trans}(\delta_1 \rangle\!\rangle\, \delta_2, s, \delta', s')$ finitely fails, then: (i) for all δ_1' such that δ' unifies with $\delta_1' \rangle\!\rangle\, \delta_2$, $\mathtt{trans}(\delta_1, s, \delta_1', s')$ finitely fails, hence by the induction hypothesis $\mathcal{C} \cup \mathcal{D} \models \forall\neg\, Trans(\delta_1, s, \delta_1', s') \wedge \delta' = (\delta_1' \rangle\!\rangle\, \delta_2))$; (ii) either $\mathtt{trans}(\delta_1, s, \_, \_)$ succeeds, hence $\mathcal{C} \cup \mathcal{D} \models \exists\delta_1', s_1'\, Trans(\delta_1, s, \delta_1', s_1')$, or for all δ_2' such that δ' unifies with $\delta_1 \rangle\!\rangle\, \delta_2'$, $\mathtt{trans}(\delta_2, s, \delta_2', s')$ finitely fails, hence by the induction hypothesis $\mathcal{C} \cup \mathcal{D} \models \forall\neg\, Trans(\delta_2, s, \delta_2', s') \wedge \delta' = (\delta_1 \rangle\!\rangle\, \delta_2'))$. Considering (1) and the UNA for object, actions, and program terms, we get the thesis. □

Lemma 2: *Under the hypotheses above the following holds:*

1. *If a goal* $\mathtt{trans*}(\delta, s, \delta', s')$, *where* δ *and* s *may contain variables only on object terms and action terms, succeeds with computed answer* θ, *then* $\mathcal{C} \cup \mathcal{D} \models \forall\, Trans^*(\delta, s, \delta', s')\theta$, *moreover* $\delta'\theta$ *and* $s'\theta$ *may contain free variables only on object terms and action terms.*

2. *If a goal* $\mathtt{trans*}(\delta, s, \delta', s')$, *where* δ *and* s *may contain variables only on object terms and action terms, finitely fails, then* $\mathcal{C} \cup \mathcal{D} \models \forall\neg\, Trans^*(\delta, s, \delta', s')$.

Proof: Using Lemma 1. *Success.* Then there exists a successful SLDNF-derivation [21] . Such a derivation must contain a finite number k of selected literals of the form $\mathtt{trans*}(\delta_1, s_1, \delta_2, s_2)$. The thesis is proven by induction on such a number k. *Failure.* Then there exists a finitely failed SLDNF-tree [21] formed by failed SLDNF-derivations each of which contains a finite number of selected literals of the form $\mathtt{trans*}(\delta_1, s_1, \delta_2, s_2)$. The thesis is proven by induction on the maximal number of selected literals of the form $\mathtt{trans*}(\delta_1, s_1, \delta_2, s_2)$ contained in the SLDNF-derivations forming the tree. □

With procedures

Since we do not have nested procedures in the Prolog implementation, we can avoid carrying around the procedure environment. Hence we can simplify the constraints on procedures in the definition of *Trans* and *Final* from Section 7 to respectively:

$$\begin{array}{lcl} T(P(\vec{t}), s, \delta', s') & \equiv & T(\delta_P{}^{\vec{v}_P}_{\vec{t}[s]}, s, \delta', s') \\ F(P(\vec{t}), s) & \equiv & F(\delta_P{}^{\vec{v}_P}_{\vec{t}[s]}, s) \end{array}$$

To prove the soundness of the interpreter in presence of procedures, we need only redo the proof of Lemma 1.

We now prove Lemma 1 as follows. Assume, for the moment, that *Trans* and *Final* satisfy the axioms $\mathcal{T}$ and $\mathcal{F}$ from Sections 4 and 5 plus the following ones:

$$\begin{array}{lcl} \mathit{Trans}(P(\vec{t}), s, \delta', s') & \equiv & \mathit{Trans}(\delta_P{}^{\vec{v}_P}_{\vec{t}[s]}, s, \delta', s') \\ \mathit{Final}(P(\vec{t}), s) & \equiv & \mathit{Final}(\delta_P{}^{\vec{v}_P}_{\vec{t}[s]}, s) \end{array}$$

Then we follow the line of the proof given above. However we need to deal with the additional complication that due to procedure expansions the program now does not get always simpler anymore. To this end, we observe that every terminating SLDNF-derivation contains a finite number of selected literals of the form $\mathtt{trans}(P(\vec{t}), s_1, \delta_2, s_2)$ $(\mathtt{final}(P(\vec{t}), s_1))$. Hence we can prove the lemma using the following three nested inductions:

- Induction on the rank of successful SLDNF-derivations/finitely failed SLDNF-trees (i.e., the depth of nesting of auxiliary finitely failed SLDNF-trees) [21].
- Induction on the number of selected literals of the form $\mathtt{trans}(P(\vec{t}), s_1, \delta_2, s_2)$ $(\mathtt{final}(P(\vec{t}), s_1))$ occurring in a successful SLDNF-derivation, for success. Induction on the maximal number of selected literals of the form $\mathtt{trans}(P(\vec{t}), s_1, \delta_2, s_2)$ $(\mathtt{final}(P(\vec{t}), s_1))$ contained in the SLDNF-derivations forming the finitely failed SLDNF-tree, for failure.
- Induction on the structure of the program.

Now we come back to the assumption we made above for *Trans* and *Final*. In fact *Final*, being closed under the constraints on F in its definition, does actually satisfy the axioms $\mathcal{F}$ from Sections 4 and 5 as well as the one above [6]. However, *Trans*, which is *not* closed under the constraints for T in its definition, does not satisfy the assumption in general [6]. However, we get the desired result by noticing that the equivalences assumed for *Trans* form a *conservative extension* (see e.g. [33]) of domain theory $\mathcal{D}$ plus the axioms needed for the encoding of programs as first-order terms, and appealing to the following general result:

Proposition 1: *Let* Γ *be a consistent theory,* $\Gamma \cup \{\Phi\}$ *a conservative extension of* Γ *where* Φ *is a closed first-order formula, and* P *a predicate occurring in* Φ *but not in* Γ*. Then for any tuple of terms* $\vec{t}$*:*

1. $\Gamma \cup \{\Phi\} \models \forall P(\vec{t})$ *implies* $\Gamma \models \forall(\forall Z.[\Phi^P_Z \supset Z(\vec{t})])$
2. $\Gamma \cup \{\Phi\} \models \forall \neg P(\vec{t})$ *implies* $\Gamma \models \forall(\neg\forall Z.[\Phi^P_Z \supset Z(\vec{t})])$

Proof: (1) by contradiction. Suppose there exists a model M of Γ and variable assignment σ with $\sigma(Z) = \mathcal{R}$ for some relation $\mathcal{R}$, such that $M, \sigma \models \Phi^P_Z$ but $M, \sigma \not\models Z(\vec{t})$. Now consider the model M' of Γ obtained from M by changing the interpretation of P to $P^{M'} = \mathcal{R}$. Then $M' \models \Phi$ and $M', \sigma \not\models P(\vec{t})$, which contradicts $\Gamma \cup \{\Phi\} \models \forall P(\vec{t})$. (2) by contradiction. Suppose exists a model M of Γ and a variable assignment σ such that $M, \sigma \models \forall Z.[\Phi^P_Z \supset Z(\vec{t})]$. Then for every variable assignment σ' obtained from σ by putting $\sigma(Z) = \mathcal{Q}$ if $M, \sigma' \models \Phi^P_Z$ then $M, \sigma' \models Z(\vec{t})$. Let M' be an expansion of M such that $M' \models \Phi$. Then for $\mathcal{Q} = P^{M'}$ we have $M, \sigma' \models Z(\vec{t})$, i.e., $M', \sigma \models P(\vec{t})$, which contradicts $\Gamma \cup \{\Phi\} \models \forall \neg P(\vec{t})$. $\square$

Intuitively, Proposition 1 says that when we constrain a relation P by a first-order statement, then every tuple that is forced to be "in" or "out" of the relation, will also be similarly "in" or "out" of the relation obtained by the second-order version of the statement. Thus if *Trans*(δ, s, δ', s') holds for the first-order version of *Trans*, it must also hold for the second-order version.

9 Discussion

With all of this procedural richness (nondeterminism, concurrency, recursive procedures, priorities, etc.), 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. And this theory must be used explicitly by a *ConGolog* interpreter.

In contrast, an interpreter for an ordinary procedural language does not use its semantics explicitly. Standard semantic accounts of programming languages also require the initial state to be completely specified; our account does not; an agent may have to act without knowing everything about its environment. Our account accommodates domain-dependent primitive actions and allows the interactions between the agent and its environment to be modeled – actions may change the environment in a way that affects what actions can later occur.

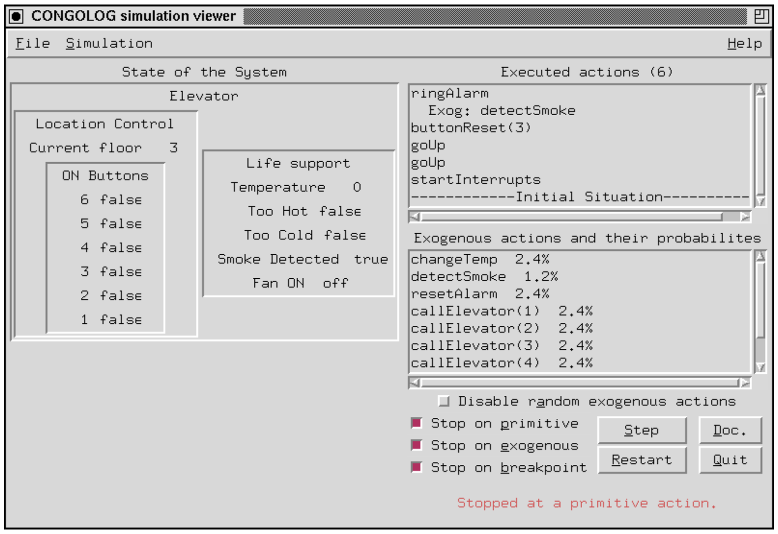
As mentioned, an important motivation for the development of *ConGolog* is the need for tools to implement intelligent agent programs that are "reactive" in the sense that they reconsider their plans in response to significant changes in their environment. Thus, our work is related to earlier research on resource-bounded deliberative architectures such as [2] (IRMA) and [27] (PRS), and agent programming languages that are to some extent based on this kind of architectures, such as AGENT-0 [34], AgentSpeak(L) [26], and 3APL [15]. One difference is that in *ConGolog*, domain dynamics are specified declaratively and the specification is used automatically in program execution; there is no need to program the updating of a world model when actions are performed. On the other hand, plan selection or generation is not specified using rules; it must be coded up in the program; this produces more complex programs, but there is perhaps less overhead. Finally, agents programmed in *ConGolog* can be understood as executing programs, albeit in a smart way; they have a simple operational semantics; architectures like IRMA and PRS, and languages like AGENT-0, AgentSpeak(L), and 3APL have much more complex execution models.

There has been much work on agent programming languages recently and several proposed languages share features with *ConGolog*. Concurrent MetateM [11] supports concurrency and uses a temporal logic to specify the behavior of agents. The rule-based languages AgentSpeak(L) [26] and 3APL [15] as well as van Eijk et al.'s [10] concurrent-constraint-based language all come with formal semantics specified using transition systems. There are also similarities with the work of Bonner and Kifer [3], where a logical formalism is proposed for concurrent database transactions. But it is difficult to make detailed comparisons between these languages which draw on different formalisms and focus on different issues.

The simple Prolog implementation of the *ConGolog* interpreter described in section 8 is at the core of a toolkit we have developed for implementing *ConGolog* applications. The interpreter in the toolkit is very similar to the one described, but uses a more convenient syntax, performs some error detection, and has tracing facilities for debugging.

The toolkit also includes a module for *progressing* the initial state database. To understand the role of this component, first note that the basic method used by our implementation of action theories for determining whether a condition holds in a given situation (i.e. evaluate $\texttt{holds}(\phi, do(a_1, \ldots, do(a_n, S_0)\ldots)$ is to perform *regression* on the condition to obtain a new condition that only mentions the initial situation and then query the initial situation database to determine whether the new condition holds. But regressing the condition all the way back to the initial situation can be quite inefficient when the program has been running for a while and many actions have been performed. If the program is willing to commit to a particular sequence of actions, it is possible *progress* the initial situation theory to a new initial situation theory representing the state of affairs after the sequence of actions [19].[21] Subsequent queries can then be efficiently evaluated

[21] In general, the progression of an initial situation database may not be first-order representable; but when the initial situation is completely known (as we are assuming in this implementation), its progression



```
State of the System
Elevator
  Location Control
  Current floor   3
    ON Buttons
      6 false
      5 false
      4 false
      3 false
      2 false
      1 false
  Life support
  Temperature    0
  Too Hot false
  Too Cold false
  Smoke Detected  true
  Fan ON   off

Executed actions (6)
ringAlarm
  Exog: detectSmoke
buttonReset(3)
goUp
goUp
startInterrupts
------------Initial Situation----------

Exogenous actions and their probabilites
changeTemp  2.4%
detectSmoke  1.2%
resetAlarm  2.4%
callElevator(1)  2.4%
callElevator(2)  2.4%
callElevator(3)  2.4%
callElevator(4)  2.4%

☐ Disable random exogenous actions
■ Stop on primitive
■ Stop on exogenous
■ Stop on breakpoint

Stopped at a primitive action.
```

Figure 1: The *ConGolog* toolkit's graphical viewer.

with respect to this new initial situation database. The progression module performs this updating of the initial situation database.

The toolkit also includes a graphical viewer (see figure 1) for debugging *ConGolog* programs and delivering process modeling applications. The tool, which is implemented in Tcl/Tk, displays the sequence of actions performed by the *ConGolog* program and the value of the fluents in the resulting situation (or any situation along the path). The program can be stepped through and exogenous events can be generated either manually or at random according to a given distribution. The manner in which state information is displayed can be specified easily and customized as required.

Finally, a high-level Golog Domain Specification language (GDL) similar to Gelfond and Lifschitz's $\mathcal{A}$ [12] has also been developed. The toolkit includes a GDL compiler that takes a domain specification in GDL, generates successor state axioms for it, and then produces a Prolog implementation of the resulting domain theory.

ConGolog has already been used in various applications. Lespérance *et al.* [17] have implemented a "reactive" high-level control module for a mobile robot in *ConGolog*. The robot performs a mail-delivery task. The *ConGolog* control program involves a set of prioritized interrupts that react to events such as the robot arriving to a customer's mailbox or failing to get to a mailbox due to obstacles, as well as new shipment orders with varying degrees of urgency being received. The *ConGolog* controller was interfaced to navigation software and successfully tested on a RWI B12 mobile robot.

Work has also been done on using *ConGolog* to model multiagent systems [32]. In this case, the domain theory includes fluents that model the beliefs and goals of the system's agents (this is done by adapting a possible-world semantics of such mental states to the

is always first-order representable and can be computed efficiently; see [19] for details.

situation calculus). A *ConGolog* program is used to specify the complex behavior of the agents in such a system. A simple multiagent meeting scheduling example is specified in [32]. *ConGolog*-based tools for specifying and verifying complex multiagent systems are being investigated.

Finally, in [8], the transition semantics developed in this paper is adapted so that execution can be interleaved with program interpretation in order to accommodate 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.

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. Let us mention one: handling non-termination, that is, developing accounts of program correctness (fairness, liveness *etc.*) appropriate for controllers expected to operate indefinitely as in [9], but without giving up the agent's control over nondeterministic choices that characterizes the *Do*-based semantics for terminating programs.

References

[1] G. R. Andrews, and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, **15:1**, 3–43, 1983.

[2] M. E. Bratman, D.J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, **4**, 349–355, 1988.

[3] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 142–156, Bonn, Germany, Sept. 1996.

[4] J. De Bakker and E. De Vink. *Control Flow Semantics.* MIT Press, 1996.

[5] G. De Giacomo and X. Chen. Reasoning about nondeterministic and concurrent actions: A process algebra approach. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 658–663, 1996.

[6] G. De Giacomo, Y. Lespérance, and H. J. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus: foundations. 1998. Submitted.

[7] G. De Giacomo, Y. Lespérance, and H. J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 1221-1226, 1997.

[8] G. De Giacomo and H. J. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. In *Cognitive Robotics – Papers from the 1998 AAAI Fall Symposium*, pages 28–34, Orlando, FL, October, 1998, Technical Report FS-98-02, AAAI Press.

[9] G. De Giacomo, E. Ternovskaia and R. Reiter. Non-terminating processes in the situation calculus. In *Proceedings of the AAAI'97 Workshop on Robots, Softbots, Immobots: Theories of Action, Planning and Control.* Providence, Rhode Island, July 1997.

[10] R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Information-passing and belief revision in multi-agent systems. In *Proceedings of ATAL'98*, J. P. Müller, M. P. Singh, and A. S. Rao, editors, pages 75–89, Paris, 1998.

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

[12] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, **17**, 301–327, 1993.

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

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

[15] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. A formal semantics for an abstract agent programming language. In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, *Proceedings of ATAL'97*, LNAI 1365, pages 215-229, Springer-Verlag, 1998.

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

[17] Y. Lespérance, M. Jenkin, and K. Tam. Reactivity in a logic-based robot programming framework. In *Cognitive Robotics – Papers from the 1998 AAAI Fall Symposium,* pages 98–105, Orlando, FL, October, 1998, Technical Report FS-98-02, AAAI Press.

[18] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation*, **4**(5), 655–678, 1994.

[19] F. Lin and R. Reiter. How to progress a database. *Artificial Intelligence*, **92**, 131–167, 1997.

[20] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. In *Journal of Logic Programming*, 31, 59–84, 1997.

[21] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

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

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

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

[25] D. Pym, L. Pryor, D. Murphy. Processes for plan-execution. In *Proceedings of the 14th Workshop of the UK Planning and Scheduling Special Interest Group*, 1995

[26] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, W. Van der Velde and J. W. Perram, editors, LNAI 1038, pages 42–55, Springer-Verlag, 1996.

[27] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, B. Nebel, C. Rich, and W. Swartout, editors, pages 439–449, Morgan Kaufmann Publishing, 1992.

[28] 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.

[29] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, **64**, 337–351, 1993.

[30] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 2–13, Morgan Kaufmann Publishing, 1996.

[31] R. Reiter. *Knowledge in Action: Logical Foundation for Describing and Implementing Dynamical Systems.* In preparation.

[32] S. Shapiro, Y. Lespérance, and H.J. Levesque. Specifying communicative multiagent systems. In W. Wobcke, M. Pagnucco, and C. Zhang *Agents and Multi-Agent Systems – Formalisms, Methodologies, and Applications*, LNAI 1441, pages 1–14, Springer-Verlag, 1998.

[33] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[34] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, **60**, 51–92, 1993.

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

[36] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics,***5**, 285–309, 1955.