ConGolog, a concurrent programming language based on the situation calculus

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. Some mathematical properties of the language are proven, for instance, that the proposed semantics is equivalent to that given earlier for the portion of the language without concurrency.

Keywords: cognitive robotics; reasoning about actions; situation calculus; semantics of programs; concurrency

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, and that at the same time finding a legal execution of that program will be more feasible computationally than the corresponding planning task. 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*) [6], 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, all defined precisely in the language of the situation calculus. But this kind of expressiveness requires considerable mathematical machinery: we need to encode *ConGolog* programs as terms in the situation calculus (which, among other things, requires encoding certain formulas as terms), and we also need to use second-order quantification to deal with iteration and recursive procedures. It is not at all obvious that such complex definitions are well-behaved or even consistent.

Of course ours is not the first formal model of concurrency. In fact, well developed approaches are available [17, 25, 4, 39][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

[1]In [28, 5] a direct use of such approaches to model concurrent (complex) actions in AI is investigated.

determined by a separate domain-dependent action theory, which specifies the action preconditions and effects, and deals with the frame problem. Finally, it might also be noted that the interaction between prioritized concurrency and recursive procedures presents a level of procedural complexity which, as far as we know, has not been dealt with in any previous formal model.

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 such a specification to handle procedures and recursion. Handling the interaction between the very general form of prioritized concurrency allowed in *ConGolog* and recursive procedures will require a quite sophisticated approach. In Section 8 we will show general sufficient conditions that allow us to use a much simplified semantics without loss of generality. In Section 9, we present a Prolog interpreter for *ConGolog* and prove its correctness. In Section 10, we conclude by discussing some of the properties of *ConGolog*, its implementation, and topics for future research.

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* [24], 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,

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

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,

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

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:

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

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 [31]. 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:

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

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. This discussion ignores the ramification and qualification problems; a treatment compatible with the approach described has been proposed by Lin and Reiter [21].

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.
- 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 [21] and Reiter [32].

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
$\phi?$,	wait for a condition[2]
$(\delta_1; \delta_2)$,	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

In the first line, a stands for a situation calculus action where the special situation constant *now* may be used to refer to the current situation (i.e. that where a is to be executed). Similarly, in the line below, ϕ stands for a situation calculus formula where *now* may be used to refer to the current situation, for example $OnTable(block, now)$. $a[s]$ ($\phi[s]$) 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 $OnTable(block)$ instead of $OnTable(block, now)$. In such cases, the situation suppressed version of the action or formula should be understood as an abbreviation for the version with *now*.

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

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:

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

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.

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

In [20], $Do(\delta, s, s')$ was simply viewed as an abbreviation for a formula of the situation calculus. The following inductive definition of *Do* was provided:

1. Primitive actions:

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

2. Wait/test actions:

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

3. Sequence:

$$Do(\delta_1; \delta_2,\ s, s') \stackrel{def}{=} \exists s''.\ Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$$

4. Nondeterministic branch:

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

5. Nondeterministic choice of argument:

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

6. Nondeterministic iteration:

$$Do(\delta^*, s, s') \stackrel{def}{=} \forall P.\{\forall s_1.\, P(s_1, s_1) \wedge \forall s_1, s_2, s_3.[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\,\} \supset\ P(s, s').$$

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

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

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

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

We have left out the expansion for procedures, which is somewhat more complex; see [20] for the details.

4 A Transition Semantics

By using *Do*, programs are assigned a semantics in terms of a relation, denoted by the formulas $Do(\delta, s, s')$, that given a program δ and a situation s, returns a situation s' resulting from executing the program starting in the situation s. Semantics of this form are sometimes called *evaluation semantics* (see [15, 26]), since they are based on the (complete) evaluation the program.

When concurrency is taken into account it is more convenient to adopt semantics of a different form: the so-called *transition semantics* or computation semantics (see again [15, 26]). Transition semantics are based on defining *single steps* of computation in contrast to directly defining complete computations.

In the present case, we are going to define a relation, denoted by the predicate $Trans(\delta, s, \delta', s')$, that associates to a given program δ and situation s, a *new situation* s' that results from executing a primitive action or test action and a *new program* δ' that represents what *remains of the program* after having performed such an action. In other words, *Trans* denotes a *transition* relation between *configurations*. A *configuration* is a pair formed by a program (the part of the initial program that is left to perform) and the a situation (representing the current situation).

We are also going to introduce a predicate $Final(\delta, s)$, meaning that the configuration (δ, s) is a *final* one, that is, where the computation can be considered completed (no program remains to be executed). The final situations reached after a finite number of transitions from a starting situation coincide with those satisfying the *Do* relation. Complete computations are thus defined by repeatedly composing single transitions until a final configuration is reached.

It worth noting that if a program does not terminate, then no final situation will satisfy the *Do* relation (indeed evaluation semantics are typically used for terminating programs), while we can still keep track of the various transitions performed by means of *Trans*. Indeed, nonterminating programs do not need any special treatment within transition semantics, while they typically remain undefined in evaluation semantics.

In general, both evaluation semantics and transition semantics belong to the family of *structural operational semantics* introduced by Plotkin in [27]. Both of these forms of semantics are operational since they do not assign a meaning directly to the programs (as denotational semantics), but instead see programs simply as specifications of computations (or better as syntactic objects that specify the control flow of the computation). They are abstract semantics since, in contrast to *concrete operational semantics*, they do not define a specific machine on which the operations are performed, but instead only define an abstract relation (such as *Do* or *Trans*) which denotes the possible computations (either complete computations for evaluation semantics, or single steps of computations for transition semantics). In addition, both such form of semantics are structural since are are defined on the *structure* of the programs.

4.1 Encoding programs as first-order terms

In the simple semantics using *Do*, it was possible to avoid introducing programs explicitly into the logical language, since $Do(\delta, s, s')$ was only an abbreviation for a formula $\Phi(s, s')$ that did not mention the program δ (or any other programs). This was possible essentially because it was not necessary to quantify over programs.

Basing the semantics on *Trans* however does require quantification over programs. To allow for this, we develop an encoding of programs as first-order terms in the logical language (observe that programs as such, cannot in general be first-order terms, since on one hand, they mention formulas in tests, and on the other, the operator π in $\pi x.\delta$ is a quantifier).

Encoding programs as first-order terms, although it requires some care (e.g. introducing constants denoting variables and defining substitution explicitly in the language), does not pose any major problem[3]. In the following we abstract from the details of the encoding as much as possible, and essentially use programs within formulas as if they were already first-order terms. The full encoding is given in Appendix A.

4.2 *Trans* and *Final*

Let us formally define *Trans* and *Final*, which intuitively specify what are the possible *transitions* between configurations (*Trans*), and when a configuration can be considered final (*Final*).

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 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 does not). 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:
$$Trans(nil, s, \delta', s') \equiv False$$

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

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

[3]Observe that, we assume that formulas that occur in tests never mention programs, so it is impossible to build self-referential sentences.

4. Sequence:

$$\begin{aligned}
&Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\
&\quad \exists \gamma.\delta' = (\gamma; \delta_2) \land Trans(\delta_1, s, \gamma, s') \quad \lor \\
&\quad Final(\delta_1, s) \land Trans(\delta_2, s, \delta', s')
\end{aligned}$$

5. Nondeterministic branch:

$$\begin{aligned}
&Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv \\
&\quad Trans(\delta_1, s, \delta', s') \quad \lor \quad Trans(\delta_2, s, \delta', s')
\end{aligned}$$

6. Nondeterministic choice of argument:

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

7. Iteration:

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

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. The details of how this function is defined are in Appendix A.

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 (see Appendix A for details).

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^v_x, s) can evolve to (δ', s'). Here δ^v_x is the program resulting from δ by substituting v with the variable x.[4]

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).

$\mathit{Final}(\delta, s)$ 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 $\mathit{Final}(\mathit{nil}, s)$, but also $\mathit{Final}(\delta^*, s)$ 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:
$$\mathit{Final}(\mathit{nil}, s) \equiv \mathit{True}$$

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

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

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

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

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

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

[4]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 (see Appendix A).

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.

4.3 *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:

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

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:

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

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'.

For *Golog* programs such a definition for *Do* coincides with the one given in [20]. Formally, we can state the the following result:

Theorem 1: *Let Do_1 be the original definition of Do in [20], presented in Section 3, and Do_2 the new one given above. Then for each* Golog *program δ:*

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

Proof: See Appendix B. □

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

Let us note that a *Trans*-step which brings the state of a computation from one configuration (δ, s) to another (δ', s') need not change the situation part of the configuration, i.e., we may have $s = s'$. In particular, test actions have this property. If we want to abstract from such computation steps that only change the state of the program, we can easily define a new relation, *TransSit*, that skips transitions that do not change the situation:

$$TransSit(\delta, s, \delta', s) \quad \stackrel{def}{=} \quad \forall T'.[\ldots \ \supset \ T'(\delta, s, \delta', s')]$$

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

$$Trans(\delta, s, \delta', s') \wedge s' \neq s \ \supset \ T'(\delta, s, \delta', s')$$
$$Trans(\delta, s, \delta'', s) \wedge T'(\delta'', s, \delta', s') \ \supset \ T'(\delta, s, \delta', s').$$

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.[5]

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 behave in a similar way to the test-and-set atomic instructions used to build semaphores in concurrent programming [1].[6]

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, $<True \rightarrow ringBell>$ at the highest priority would ring a bell and do nothing else. With interrupts, we can easily write controllers that can stop whatever task they are doing to handle various concerns as they arise. They are, dare we say, more reactive.

[5] 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.

[6] In [20], non-synchronized versions of if-then-elses and while-loops are introduced by defining: **if** ϕ **then** δ_1 **else** $\delta_2 \stackrel{def}{=} [(\phi?;\delta_1) \mid (\neg\phi?;\delta_2)]$ and **while** ϕ **do** $\delta \stackrel{def}{=} [(\phi?;\delta)^*;\neg\phi?]$. 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.

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:

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

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

That is ($\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else}\delta_2, s$) 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, ($\textbf{while } \phi \textbf{ do } \delta, s$) can evolve to ($\delta'; \textbf{while } \phi \textbf{ do } \delta, s'$), if $\phi[s]$ holds and (δ, s) can evolve to (δ', s'). ($\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s$) 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, ($\textbf{while } \phi \textbf{ do } \delta, s$) 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:

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

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:

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

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*:

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

To see how this works, first assume that the special fluent $\mathit{Interrupts\_running}$ is identically *True*. When an interrupt $<\phi \rightarrow \delta>$ 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 $\mathit{stop\_interrupts}$, whose only effect is to make $\mathit{Interrupts\_running}$ false. Thus, we imagine that to execute a program δ containing interrupts, we would actually execute the program $\{\mathit{start\_interrupts}\,;\,(\delta \rangle\!\rangle \mathit{stop\_interrupts})\}$ 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:

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

Executing this program involves performing zero, one, or more nondeterministically chosen exogenous events.[7] Then we make the user-specified program δ run concurrently with δ_{EXO}:

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

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

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

5.1 Formal properties of *Trans* and *Final* without procedures

We are going to show that the axioms for *Trans* and *Final* for the whole of *ConGolog* are definitional, in the sense that they completely characterize *Trans* and *Final* for programs without procedures.

Lemma 1: *For any* ConGolog *program term $\delta(\vec{x})$ containing only variables $\vec{x}$ of sort object or action, there exist two formulas $\Phi(\vec{x}, s, \delta, s')$ and $\Psi(\vec{x}, s)$, where $\vec{x}, s, \delta', s'$ and $\vec{x}, s$ are the only free variables in Φ and in Ψ respectively, that do not mention Final and Trans, and are such that:*

$$\mathcal{C} \models \forall \vec{x}, s, \delta', s'.\ Trans(\delta(\vec{x}), s, \delta', s') \equiv \Phi(\vec{x}, s, \delta', s') \quad (1)$$

$$\mathcal{C} \models \forall \vec{x}, s.\ Final(\delta(\vec{x}), s) \equiv \Psi(\vec{x}, s) \quad (2)$$

Proof: For both (1) and (2), the proof is similar; it is done by induction on the program structure considering as base cases programs of the form *nil*, a, and $\phi?$. Base cases: the thesis is an immediate consequence of the axioms of *Trans* and *Final* since the right-hand side of the equivalences does not mention *Trans* and *Final*. Inductive cases: by inspection, all the axioms have on the right-hand side simpler program terms, which contain only variables of sort object or action, as the first argument to *Trans* and *Final*, hence the thesis is a straightforward consequence of the inductive hypothesis. □

It follows from the lemma that the axioms in $\mathcal{T}$ and $\mathcal{F}$, together with the axioms for encoding of programs as first-order terms, completely determine the interpretation of the predicates *Trans* and *Final* on the basis of the interpretation of the other predicates. That is $\mathcal{T}$ and $\mathcal{F}$ *implicitly define* the predicates *Trans* and *Final*. Formally, we have the following theorem:

Theorem 2: *There are no pair of models of $\mathcal{C}$ that differ only in the interpretation of the predicates Trans and Final.*

Proof: By contradiction. Suppose that there are two models M_1 and M_2 of $\mathcal{C}$ that agree in the interpretation of all non-logical symbols (constant, function, predicates) other than either *Trans* or *Final*. Let's say that they disagree on *Trans*, i.e. there is a tuple of domain values $(\hat{\delta}, \hat{s}, \hat{\delta}', \hat{s}')$ such that $(\hat{\delta}, \hat{s}, \hat{\delta}', \hat{s}') \in Trans^{M_1}$ and $(\hat{\delta}, \hat{s}, \hat{\delta}', \hat{s}') \notin Trans^{M_2}$. Considering the structure of the sort *programs* (see Appendix A), we have that for every value of the domain of sort *programs* $\hat{\delta}$ there is a program term $\delta(\vec{x})$, containing only variables $\vec{x}$ of sort object or action, such that for some assignment σ to $\vec{x}$, $\delta^{M_1,\sigma} = \delta^{M_2,\sigma} = \hat{\delta}$. Now let us consider three variables s, δ', s' and an assignment σ' such that $\sigma'(\vec{x}) = \sigma(\vec{x})$, $\sigma'(s) = \hat{s}$,

$\sigma'(\delta') = \hat{\delta}'$, and $\sigma'(s') = \hat{s}'$. By Lemma 1, there exists a formula Φ such that neither *Trans* nor *Final* occurs in Φ and:

$$M_i, \sigma' \models \mathit{Trans}(\delta, s, \delta', s') \text{ iff } M_i, \sigma' \models \Phi(\vec{x}, s, \delta', s') \qquad i = 1, 2.$$

Since, $M_1, \sigma' \models \Phi(\vec{x}, s, \delta', s')$ iff $M_2, \sigma' \models \Phi(\vec{x}, s, \delta', s')$, we get a contradiction. □

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: $\forall r.\, \mathit{Robot}(r) \equiv r = \mathit{Rob}_1 \vee r = \mathit{Rob}_2$.
  Two table ends: $\forall e.\, \mathit{TableEnd}(e) \equiv e = \mathit{End}_1 \vee e = \mathit{End}_2$.

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

- 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:
  $Holding(r, e, do(a, s)) \equiv$
  $\quad a = grab(r, e) \vee Holding(r, e, s) \wedge a \neq release(r, e)$
  $vpos(e, do(a, s)) = p \equiv$
  $\quad \exists r, z.(a = vmove(r, z) \wedge Holding(r, e, s) \wedge p = vpos(e, s) + z) \vee$

$$\exists r.\, a = release(r,e) \wedge p = 0 \vee$$
$$p = vpos(e,s) \wedge \neg\exists r, z.(a = vmove(r,z) \wedge Holding(r,e,s)) \wedge$$
$$\neg\exists r.\, a = release(r,e)$$

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:

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

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

So the goal is

$$Goal(s) \stackrel{def}{=} \; TableUp(s) \wedge \; \forall s'.s' \leq s \supset Level(s')$$

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

$$SafeToLift(rob,s) \stackrel{def}{=} \; \exists e, e'.\; e \neq e' \;\wedge\; TableEnd(e) \;\wedge\; TableEnd(e') \wedge$$
$$Holding(rob,e,s) \;\wedge\; vpos(e) \leq vpos(e') + Tol - Amount.$$

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:[8]

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

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 $Final$

[8]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.

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 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 $vpos(End_1)$.

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:
  $TooHot(e, s) \stackrel{def}{=} temp(e, s) > 1$
  $TooCold(e, s) \stackrel{def}{=} temp(e, s) < -1$

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:
  $Poss(goDown(e), s) \quad \equiv \quad floor(e, s) \neq 1$
  $Poss(goUp(e), s) \quad \equiv \quad floor(e, s) \neq 6$
  $Poss(buttonReset(n), s) \quad \equiv \quad True$
  $Poss(toggleFan(e), s) \quad \equiv \quad True$
  $Poss(ringAlarm) \quad \equiv \quad True$
  $Poss(reqElevator(n), s) \quad \equiv \quad (1 \leq n \leq 6) \wedge \neg ButtonOn(n, s)$
  $Poss(changeTemp, s) \quad \equiv \quad True$
  $Poss(detectSmoke, s) \quad \equiv \quad \neg Smoke(s)$
  $Poss(resetAlarm, s) \quad \equiv \quad Smoke(s)$

- Successor state axioms:
  $floor(e, do(a, s)) = n \quad \equiv$
  $\quad (a = goDown(e) \wedge n = floor(e, s) - 1) \vee$
  $\quad (a = goUp(e) \wedge n = floor(e, s) + 1) \vee$
  $\quad (n = floor(e, s) \wedge a \neq goDown(e) \wedge a \neq goUp(e))$
  $temp(e, do(a, s)) = t \quad \equiv$
  $\quad (a = changeTemp(e) \wedge FanOn(e, s) \wedge t = temp(e, s) - 1) \vee$
  $\quad (a = changeTemp(e) \wedge \neg FanOn(e, s) \wedge t = temp(e, s) + 1) \vee$
  $\quad (t = temp(e, s) \wedge a \neq changeTemp(e))$
  $FanOn(e, do(a, s)) \quad \equiv$
  $\quad (a = toggleFan(e) \wedge \neg FanOn(e, s)) \vee$

$$
\begin{aligned}
&\quad (FanOn(e,s) \wedge a \neq toggleFan(e))\\
&ButtonOn(n, do(a,s)) \quad \equiv\\
&\quad a = reqElevator(n) \vee\\
&\quad (ButtonOn(n,s) \wedge a \neq buttonReset(n))\\
&Smoke(do(a,s)) \quad \equiv\\
&\quad a = detectSmoke \vee\\
&\quad (Smoke(s) \wedge a \neq resetAlarm)
\end{aligned}
$$

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:

$$
\begin{aligned}
&\textbf{while } \exists n.ButtonOn(n) \textbf{ do}\\
&\quad \pi n.\{BestButton(n)?; serveFloor(e,n)\};\\
&\textbf{while } floor(e) \neq 1 \textbf{ do } goDown(e)
\end{aligned}
$$

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:

$$
\begin{aligned}
&\textbf{proc } serveFloor(e,n)\\
&\quad \textbf{while } floor(e) < n \textbf{ do } goUp(e);\\
&\quad \textbf{while } floor(e) > n \textbf{ do } goDown(e);\\
&\quad buttonReset(n)\\
&\textbf{end}
\end{aligned}
$$

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

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

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:

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

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

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

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:

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

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:

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

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).[9] For n elevators, we would use $(\delta_1 \parallel \cdots \parallel \delta_n)$.

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 $\delta^{\parallel}$ 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

[9]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.

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:

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

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 a new process that first acquires the client by doing $acquire(cid)$, and then serves it. We formalize this as follows:

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

$$ClientWaiting(cid, do(a, s)) \equiv\\ a = newClient(cid) \vee ClientWaiting(cid, s) \wedge a \neq acquire(cid)]$$

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 a server can only acquire one 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,

$$[filling\ ;\ do\ ;\ re\ ;\ mi],\\ [do\ ;\ filling\ ;\ re\ ;\ mi],\\ [do\ ;\ re\ ;\ filling\ ;\ mi],\\ [do\ ;\ re\ ;\ mi\ ;\ filling],$$

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 [33] 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:

$$\begin{aligned}&FillingTub(do(a,s)) \equiv \\ &\quad a = startFilling \vee FillingTub(s) \wedge a \neq endFilling.\end{aligned}$$

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

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

and run it concurrently with the singing, 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:

$$waterLevel(do(a,s)) = waterLevel(s) + waterRate(s) \times dur(a)$$

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

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 Extending the Transition Semantics to Procedures

We now extend the transition semantics introduced above to deal with procedures. Because a recursive procedure may do an arbitrary number of procedure calls before it performs a primitive action or test, and such procedure calls are not viewed as transitions, we must use a second-order definition of *Trans* and *Final*. In doing so, great care has to be put in understanding the interaction between recursive procedures and the very general form of prioritized concurrency allowed in *ConGolog*

Let **proc** $P_1(\vec{v}_1)\delta_1$ **end**; ... ; **proc** $P_n(\vec{v}_n)\delta_n$ **end** be a collection of procedure definitions. We call such a collection an *environment* and denote it by Env. In a procedure definition **proc** $P_i(\vec{v}_i)\delta_i$ **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.

- $[Env : P(\vec{t})]$, 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 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).[10] *Trans* is defined as follows:

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

where $\ldots$ stands for the conjunction of $\mathcal{T}_T^{Trans}$ – i.e. the set of axioms $\mathcal{T}$ 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 parameters $\vec{v}$ 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 $\ldots$ 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 assertions:

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

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 actual parameters are first evaluated in the current situation, and then are substituted for the formal parameters in the procedure bodies[11], thus yielding *call-by-value* parameter passing.

The following example program δ_{StSc} illustrates *ConGolog*'s static scoping:

[10] 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.

[11] 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 (see Appendix A).

$$
\begin{array}{l}
\{\ \textbf{proc}\ P_1()\\
\quad\ a\\
\quad \textbf{end};\\
\quad \textbf{proc}\ P_2()\\
\quad\ P_1()\\
\quad \textbf{end};\\
\quad \textbf{proc}\ P_3()\\
\quad \{\ \textbf{proc}\ P_1()\\
\qquad\quad b\\
\qquad \textbf{end};\\
\qquad P_2(); P_1()\\
\quad \}\\
\quad \textbf{end};\\
\quad P_3()\\
\}
\end{array}
$$

One can show that for this program, the sequence of atomic actions performed will be a followed by b (assuming that both a and b are always possible):

$$
\begin{array}{l}
\forall s.[Poss(a,s) \land Poss(b,s)] \supset\\
\forall s,s'.[Do(\delta_{StSc},s,s') \equiv s' = do(b,do(a,s))]
\end{array}
$$

To see this consider the following. Let

$$
\begin{array}{rl}
Env_1 \stackrel{def}{=} & \textbf{proc}\ P_1()\ a\ \textbf{end};\\
& \textbf{proc}\ P_2()\ P_1()\ \textbf{end};\\
& \textbf{proc}\ P_3()\ \{Env_2; P_2(); P_1()\}\ \textbf{end};\\
\\
Env_2 \stackrel{def}{=} & \textbf{proc}\ P_1()\ b\ \textbf{end};
\end{array}
$$

Then it is easy to see that:

$$
\begin{array}{l}
Trans(\delta_{StSc},s,\delta',s')\\
\qquad \equiv Trans(\{Env_1; P_3()\},s,\delta',s')\\
\qquad \equiv Trans([Env_1 : P_3()],s,\delta',s')\\
\qquad \equiv Trans(\{Env_1; \{Env_2; P_2(); P_1()\}\},s,\delta',s')\\
\qquad \equiv Trans(\{Env_2; [Env_1 : P_2()]; P_1()\},s,\delta',s')\\
\qquad \equiv Trans([Env_1 : P_2()]; [Env_2 : P_1()],s,\delta',s')\\
\qquad \equiv Trans(\{Env_1; P_1()\}; [Env_2 : P_1()],s,\delta',s')\\
\qquad \equiv Trans([Env_1 : P_1()]; [Env_2 : P_1()],s,\delta',s')\\
\qquad \equiv Trans(a; [Env_2 : P_1()],s,\delta',s')\\
\qquad \equiv Poss(a,s) \land s' = do(a,s) \land \delta' = (nil; [Env_2 : P_1()]).
\end{array}
$$

Similarly, one can show that: $Trans([Env_2 : P_1()], do(a,s), nil, do(b, do(a,s)))$ and $Final(nil, do(b, do(a,s)))$, which yields the thesis.

Our next example illustrates *ConGolog*'s call-by-value parameter passing:

$$
\begin{array}{l}
\{\ \textbf{proc}\ P(n) \\
\quad \textbf{if}\ (n = 1)\ \textbf{then}\ nil \\
\quad\quad\quad\quad\quad\quad \textbf{else}\ goDown; P(n-1) \\
\quad \textbf{end}; \\
\quad P(floor) \\
\}
\end{array}
$$

Intuitively, this program is intended to bring an elevator down to the bottom floor of a building. If we run the program starting in situation S_0, the procedure call $P(floor)$ invokes P with the value of the functional fluent $floor$ in S_0, i.e. P is called with $floor[S_0]$, the floor the elevator is on in S_0, as actual parameter. If *ConGolog* used call-by-name parameter passing, P would be invoked with the term "$floor$" as actual parameter, and the elevator would only go halfway to the bottom floor. Indeed at each iteration of the procedure the call $P(n-1)$ would be evaluated by textually replacing n by $floor$, which at that moment has already decreased by 1.

As mentioned earlier, 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 $\{\textbf{proc}\ P()\ P()\ \textbf{end}; P()\}$ does not have any transitions, but it is not final for any situation s.[12]

7.1 Formal properties of *Trans* and *Final* with procedures

We observe that the second-order definitions of *Trans* and *Final* can easily be put in the following form:

$$
\begin{array}{l}
\mathit{Trans}(\delta, s, \delta', s') \equiv \\
\quad \forall T.[\forall \delta_1, s_1, \delta_2, s_2.\ \Phi_{\mathit{Trans}}(T, \delta_1, s_1, \delta_2, s_2) \equiv T(\delta_1, s_1, \delta_2, s_2)] \\
\quad\quad \supset\ T(\delta, s, \delta', s')
\end{array}
$$

$$
\begin{array}{l}
\mathit{Final}(\delta, s, \delta', s') \equiv \\
\quad \forall F.[\forall \delta_1, s_1.\ \Phi_{\mathit{Final}}(F, \delta_1, s_1) \equiv F(\delta_1, s_1)] \\
\quad\quad \supset\ F(\delta, s)
\end{array}
$$

where Φ_{Trans} and Φ_{Final} are obtained by rewriting each of the assertions in the definition of *Trans* and *Final* so that only variables appear in the left-hand part of the equations,

[12] Note that both *Golog* and *ConGolog* do not allow for boolean procedures to be used in tests. Introducing such kind of procedures requires particular care to avoid counterintuitive implications.

i.e.:

$$T(\delta, s, \delta', s') \equiv \phi_t(T, \delta, s, \delta', s') \qquad\qquad F(\delta, s) \equiv \phi_f(F, \delta, s)$$

and then getting the disjunction of all right-hand sides, which are mutually exclusive since each of them deals with programs of a specific form.

From such definitions, natural "induction principles" emerge (cf. the discussion on extracting induction principles from inductive definitions in [34]). These are principles saying that to prove that a property P holds for instances of *Trans* and *Final*, it suffices to prove that the property P is closed under the assertions in the definition of *Trans* and *Final*, i.e.:

$$\begin{array}{rcl}\Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) & \equiv & P(\delta_1, s_1, \delta_2, s_2)\\ \Phi_{Final}(P, \delta_1, s_1) & \equiv & P(\delta_1, s_1)\end{array}$$

Formally we can state the following theorem:

Theorem 3: *The following sentences are consequences of the second-order definitions of Trans and Final respectively:*

$$\begin{array}{l}\forall P.[\forall \delta_1, s_1, \delta_2, s_2.\ \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)] \ \supset\\ \qquad \forall \delta, s, \delta', s'.\ Trans(\delta, s, \delta', s') \ \supset \ P(\delta, s, \delta', s')\end{array}$$

$$\begin{array}{l}\forall P.[\forall \delta_1, s_1.\ \Phi_{Final}(P, \delta_1, s_1) \equiv P(\delta_1, s_1)] \ \supset\\ \qquad \forall \delta, s.\ Final(\delta, s, \delta', s') \ \supset \ P(\delta, s)\end{array}$$

Proof: We prove only the first sentence. The proof of the second sentence is analogous.

By definition we have:

$$\begin{array}{l}\forall \delta, s, \delta', s'.\ Trans(\delta, s, \delta', s') \ \equiv\\ \qquad \forall P.[\forall \delta_1, s_1, \delta_2, s_2.\ \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)]\\ \qquad\quad \supset \ P(\delta, s, \delta', s')\end{array}$$

By considering the only-if part of the above equivalence, we get:

$$\begin{array}{l}\forall \delta, s, \delta', s'.\ Trans(\delta, s, \delta', s') \ \wedge\\ \qquad \forall P.[\forall \delta_1, s_1, \delta_2, s_2.\ \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)]\\ \qquad\quad \supset \ P(\delta, s, \delta', s')\end{array}$$

So moving the quantifiers around we get:

$$\begin{array}{l}\forall P.[\forall \delta_1, s_1, \delta_2, s_2.\ \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)] \ \wedge\\ \qquad \forall \delta, s, \delta', s'.\ Trans(\delta, s, \delta', s')\\ \qquad\quad \supset \ P(\delta, s, \delta', s')\end{array}$$

and hence the thesis. $\Box$

These induction principles allow us to prove that *Trans* and *Final* for programs with procedures can be considered an extension of those for programs without procedures.

Theorem 4: *With respect to* ConGolog *programs without procedures, Trans and Final introduced above are* equivalent *to the versions introduced in Section 4.*

Proof: Let us denote *Trans* defined by the second-order sentence as $Trans_{SOL}$ and *Trans* implicitly defined through axioms in Section 4 as $Trans_{FOL}$. Since procedures are not considered we can drop, without loss of generality, the assertions for $\{Env; \delta\}$ and $[Env : P(\vec{t})]$ in the definition of $Trans_{SOL}$. Then:

- $Trans_{SOL}(\delta, s, \delta', s') \supset Trans_{FOL}(\delta, s, \delta', s')$, is proven simply by noting that $Trans_{FOL}$ satisfies (is closed under) the assertions in the definition of $Trans_{SOL}$, and then using Theorem 3.

- $Trans_{FOL}(\delta, s, \delta', s') \supset Trans_{SOL}(\delta, s, \delta', s')$, is proven by induction on the structure of δ considering as base cases *nil*, a, and $\phi?$, and then applying the induction argument.

Similarly for *Final.* $\square$

It is interesting to examine whether *Trans* and *Final* introduced above are themselves closed under the assertions in their definitions. For *Final* a positive answer can be established:

Theorem 5: *The following sentence is a consequence of the second-order definition of Final:*

$$\Phi_{Final}(Final(\delta, s), \delta, s) \quad \equiv \quad Final(\delta, s).$$

Proof: Observe that Φ_{Final} is *monotonic*[13], i.e.:

$$\forall Z_1, Z_2.[\forall \delta, s.Z_1(\delta, s) \supset Z_2(\delta, s)] \supset [\forall \delta, s.\Phi_{Final}(Z_1, \delta, s) \supset \Phi_{Final}(Z_2, \delta, s).]$$

Hence the thesis is a direct consequence of the Tarski-Knaster fixpoint theorem [40]. $\square$

For *Trans* an analogous result does not hold in general. Indeed consider the following program δ_q:

$$\begin{array}{l} \{\ \textbf{proc}\ Q() \\ \quad\quad Q() \rangle\!\rangle\ a \\ \quad \textbf{end}; \\ \quad Q() \\ \} \end{array}$$

Observe that the definition of *Trans* implies that $Trans(\delta_q, s, \delta', s') \equiv False$. Hence if *Trans* was closed under Φ_{Trans}, then we would have $Trans(\delta_q \rangle\!\rangle\ a, s, \delta', s') \equiv Trans(a, s, \delta', s')$, which would imply that $Trans(\delta_q, s, \delta', s') \equiv Trans(a, s, \delta', s')$. Contradiction.

Obviously there are several classes of *ConGolog* programs that are closed under Φ_{Trans}. For instance, if we disallow prioritized concurrency in procedures we get one such class. Another such class is that obtained by allowing prioritized concurrency to appear only in non-recursive procedures. Yet another quite general class is immediately obtainable from what is discussed next.

[13] In fact syntactically monotonic.

8 First-order *Trans* and *Final* for Procedures

In this section we investigate conditions that allow us to replace the second-order definitions of *Trans* and *Final* for programs with procedures by the first-order definitions, as in the case where procedures are not allowed.

8.1 Guarded configurations

We define a quite general condition on configurations (pairs of programs and situations) that guarantees the possibility of using first-order axioms for *Trans* and *Final* for procedures as well. To this end we introduce a notion of "configuration rank". Intuitively, a configuration is of rank n if and only if makes at most n (recursive) procedure calls before trying to make an actual program step (either an atomic action or a test).

We define the rank of a configuration inductively. A configuration is of rank n denoted by $Rank(n, \delta, s)$ if and only if:

$$
\begin{array}{rcl}
Rank(n, nil, s) & \equiv & True \\
Rank(n, a, s) & \equiv & True \\
Rank(n, \phi?, s) & \equiv & True \\
Rank(n, \delta_1; \delta_2, s) & \equiv & Rank(n, \delta_1, s) \wedge (Final(\delta_1, s) \supset Rank(n, \delta_2, s)) \\
Rank(n, \delta_1 \mid \delta_2, s) & \equiv & Rank(n, \delta_1, s) \wedge Rank(n, \delta_2, s) \\
Rank(n, \pi v.\delta, s) & \equiv & \forall x.Rank(n, \delta^v_x, s) \\
Rank(n, \delta^*, s) & \equiv & Rank(n, \delta, s) \\
Rank(n, \mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s) & \equiv & \phi[s] \wedge Rank(n, \delta_1, s)\ \vee\ \neg\phi[s] \wedge Rank(n, \delta_2, s) \\
Rank(n, \mathbf{while}\ \phi\ \mathbf{do}\ \delta, s) & \equiv & \phi[s] \supset Rank(n, \delta, s) \\
Rank(n, \delta_1 \parallel \delta_2, s) & \equiv & Rank(n, \delta_1, s) \wedge Rank(n, \delta_2, s) \\
Rank(n, \delta_1 \rangle\!\rangle\ \delta_2, s) & \equiv & Rank(n, \delta_1, s) \wedge \\
 & & ((\neg\exists\delta'_1, s'.\, Trans(\delta_1, s, \delta'_1, s')) \supset Rank(n, \delta_2, s)) \\
Rank(n, \delta^{\parallel}, s) & \equiv & Rank(n, \delta, s) \\
Rank(n, \{Env; \delta\}, s) & \equiv & Rank(n, \delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s) \\
Rank(n, [Env : P(\vec{t})], s) & \equiv & Rank(n-1, \{Env; \delta_P{}^{\vec{v}_P}_{\vec{t}[s]}\}, s)
\end{array}
$$

A configuration (δ, s) is *guarded* if and only if it is of rank n for some n:

$$Guarded(\delta, s) \overset{def}{=} \exists n.\, Rank(n, \delta, s)$$

8.2 First-order *Trans* and *Final* for procedures

For guarded configurations, we do not need to use the second-order definitions of *Trans* and *Final* when dealing with procedures. Instead we can use the first-order axioms in

Section 4 together with the following:[14]

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

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

Let us call $Trans_{FOL}$ and $Final_{FOL}$ the predicates determined by the first-order axioms and $Trans_{SOL}$ and $Final_{SOL}$ the original predicates determined by the second-order definition for procedures. We can prove the following result:

Theorem 6:

$$
\begin{aligned}
&Guarded(\delta, s) \supset \\
&\quad \forall \delta', s'. Trans_{SOL}(\delta, s, \delta', s') \equiv Trans_{FOL}(\delta, s, \delta', s') \\
&Guarded(\delta, s) \supset \\
&\quad Final_{SOL}(\delta, s) \equiv Final_{FOL}(\delta, s).
\end{aligned}
$$

Proof:(*outline*) By induction on the rank of the configuration (δ, s). For rank 0 the thesis is trivial. For rank $n+1$, we assume that the thesis holds for all configurations of rank n, and show the thesis by induction on the structure of the program considering *nil*, a, $\phi?$ and $[Env : P(\vec{t})]$ as base cases. □

A configuration (δ, s) has a *guarded evolution*, if and only if:

$$
\begin{aligned}
&GuardedEvol(\delta, s) \stackrel{def}{=} \\
&\quad \forall \delta', s'. Trans^*_{SOL}(\delta, s, \delta', s') \supset Guarded(\delta', s')
\end{aligned}
$$

For configurations with guarded evolution we have the following easy consequences:

$$
\begin{aligned}
&GuardedEvol(\delta, s) \supset \\
&\quad \forall \delta', s'. Trans^*_{SOL}(\delta, s, \delta', s') \equiv Trans^*_{FOL}(\delta, s, \delta', s') \\
&GuardedEvol(\delta, s) \supset \\
&\quad \forall s'. Do_{SOL}(\delta, s, s') \equiv Do_{FOL}(\delta, s, s')
\end{aligned}
$$

[14]The form of these axioms is exactly that of the conditions on the predicate variables T and F in the second-order definitions.

8.3 Sufficient condition for guarded evolutions

Theorem 7: *If all procedures P with environment Env in a program δ are such that*

$$\forall \vec{t}, s. Guarded([Env : P(\vec{t})], s)$$

then we have:

$$\forall s. GuardedEvol(\delta, s).$$

Proof:(*outline*) By induction on the number of transitions. For 0 transitions, we get the thesis by induction on the structure of the program (considering $nil, a, \phi?$ and $[Env : P(\vec{t})]$ as base cases). For $k+1$ transitions, we assume the thesis holds for k transitions, and we prove by induction on the structure of the program (again considering $nil, a, \phi?$ and $[Env : P(\vec{t})]$ as base cases) that making a further transition from the program resulting from the k transitions still preserves the thesis. □

It is easy to verify that non-recursive procedures, as well as procedures whose body starts with an atomic action or a wait action, trivially satisfy the hypothesis of the theorem. Observe also that all procedures in [20] satisfy such hypothesis, except for the procedure d at page 9 whose definition is reported below (n is a natural number):

$$\textbf{proc}\, d(n)\ (n = 0?) \mid d(n-1); goDown\, \textbf{end}$$

However, the variants

$$\textbf{proc}\, d(n)\ (n = 0?) \mid goDown; d(n-1)\, \textbf{end}$$

$$\textbf{proc}\, d(n)\ (n = 0?) \mid (n > 0)?; d(n-1); goDown\, \textbf{end}$$

$$\textbf{proc}\, d(n)\, \textbf{if}\, (n = 0)\, \textbf{then}\, nil\, \textbf{else}\, (d(n-1); goDown)\, \textbf{end}$$

do satisfy the hypothesis.

9 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.[15] 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*

[15] 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.

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.
- `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, 34].

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 [34] */

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 [34] */

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 amounts 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:

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

   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 [34].

9.1 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 `val(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
```

9.2 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.[16]

[16] UNA is already enforced for programs, see Appendix A.

- The predicate sub/4 correctly implements substitution for both programs and formulas.
- The predicate holds/2 satisfies the following properties:
  1. If a goal $\texttt{holds}(\phi, s)$, 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 $\texttt{holds}(\phi, s)$, 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 $\texttt{poss}(a, s)$, 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 $\texttt{poss}(a, s)$, with free variables only on object terms and action terms, finitely fails, then $\mathcal{D} \models \forall\neg Poss(a, s)$.
- The Prolog interpreter flounders (and hence does not return) on goals of the form $\texttt{not trans}(\delta, s, \_, \_)$[17] with non-ground δ and s.[18]

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 [34]).

Theorem 8: *Under the hypotheses above the following holds:*

1. *If a goal* $\texttt{do}(\delta, s, s')$*, 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* $\texttt{do}(\delta, s, s')$*, 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.

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

[18] 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 $\pi z.(\phi(z)?; (\delta_1(z) \rangle\!\rangle \delta_2(z)))$.

Without procedures

Theorem 8 is an easy consequence of Lemma 2 and Lemma 3 below.

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

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

  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* $\mathcal{C} \cup \mathcal{D} \models \forall\, \mathit{Trans}(\delta, s, \delta', s')\theta$, *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* $\mathcal{C} \cup \mathcal{D} \models \forall\neg\, \mathit{Trans}(\delta, s, \delta', s')$.

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

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

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 [23] 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 $\mathcal{C} \cup \mathcal{D} \models \forall\, \mathit{Trans}(\delta_1, s, \delta_1', s')\theta_1$, 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 $\mathcal{C} \cup \mathcal{D} \models \forall\neg\, \mathit{Trans}(\delta_1, s, \delta_1', s_1')$, $\mathcal{C} \cup \mathcal{D} \models \forall\, \mathit{Trans}(\delta_2, s, \delta_2', s')\theta_2$, and $s'\theta_2$ and $\delta_2'\theta_2$ may contain free variables only on object terms and action terms. Considering

$$\begin{aligned} \mathit{Trans}(\delta_1 \rangle\!\rangle\, \delta_2, s, \delta', s') \;\; &\equiv \\ &\exists\gamma.\delta' = (\gamma \rangle\!\rangle\, \delta_2) \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \;\vee \\ &\exists\gamma.\delta' = (\delta_1 \rangle\!\rangle\, \gamma) \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.\mathit{Trans}(\delta_1, s, \zeta, s'') \end{aligned} \tag{3}$$

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 $\mathcal{C} \cup \mathcal{D} \models \forall\neg\, \mathit{Trans}(\delta_1, s, \delta_1', s') \wedge \delta' = (\delta_1' \rangle\!\rangle\, \delta_2))$; (ii) either $\mathtt{trans}(\delta_1, s, \_, \_)$ succeeds, hence $\mathcal{C} \cup \mathcal{D} \models$

$\exists \delta'_1, s'_1. \mathit{Trans}(\delta_1, s, \delta'_1, s'_1)$, 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 $\mathcal{C} \cup \mathcal{D} \models \forall \neg \mathit{Trans}(\delta_2, s, \delta'_2, s') \wedge \delta' = (\delta_1 \rangle\!\rangle\ \delta'_2))$. Considering (3) and the UNA for object, actions, and program terms, we get the thesis. □

Lemma 3: *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* $\mathcal{C} \cup \mathcal{D} \models \forall \mathit{Trans}^*(\delta, s, \delta', s')\theta$, *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* $\mathcal{C} \cup \mathcal{D} \models \forall \neg \mathit{Trans}^*(\delta, s, \delta', s')$.

Proof: Using Lemma 2. *Success.* Then there exists a successful SLDNF-derivation [23] . 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 [23] 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 2.

We now prove Lemma 2 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 $\texttt{trans}(P(\vec{t}), s_1, \delta_2, s_2)$ ($\texttt{final}(P(\vec{t}), s_1)$). 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) [23].

- Induction on the number of selected literals of the form $\texttt{trans}(P(\vec{t}), s_1, \delta_2, s_2)$ ($\texttt{final}(P(\vec{t}), s_1)$) occurring in a successful SLDNF-derivation, for success. Induction on the maximal number of selected literals of the form $\texttt{trans}(P(\vec{t}), s_1, \delta_2, s_2)$ ($\texttt{final}(P(\vec{t}), s_1)$) 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. However, *Trans*, which is *not* closed under the constraints for T in its definition, does not satisfy the assumption, in general. However, we get the desired result by noticing that the equivalences assumed for *Trans* form a *conservative extension* (see e.g. [37]) 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 $Trans(\delta, s, \delta', s')$ holds for the first-order version of *Trans*, it must also hold for the second-order version.

10 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 [8].

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 [30] (PRS), and agent programming languages that are to some extent based on this kind of architectures, such as AGENT-0 [38], AgentSpeak(L) [29], and 3APL [16]. 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 more complex execution models.

Other programming languages share features with *ConGolog*. The agent programming language Concurrent MetateM [11] supports concurrency and uses a temporal logic to specify the behavior of agents. Bonner and Kifer [3] have proposed a logical formalism to specify concurrent database transactions. Also related are concurrent constraint languages such as CCP [35] and HCC [14], which support incompletely specified information states and concurrency. But unlike *ConGolog*, these languages generally restrict the kinds of constraints allowed in order to make entailment easy to compute. In *ConGolog*, the action theory is what determines how how states are updated. Also in constraint languages, control seems somewhat deemphasized. van Eijk et al.'s [10] have proposed an agent language partly inspired from CCP.

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 to *progress* the initial situation theory to a new initial situation theory representing the state of affairs after the sequence of actions.[19] Subsequent queries can then be efficiently evaluated 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.* [19] have implemented a "reactive" high-level control module for a mobile robot in *ConGolog*. The

[19] 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 is always first-order representable and can be computed efficiently; see [22] for details.

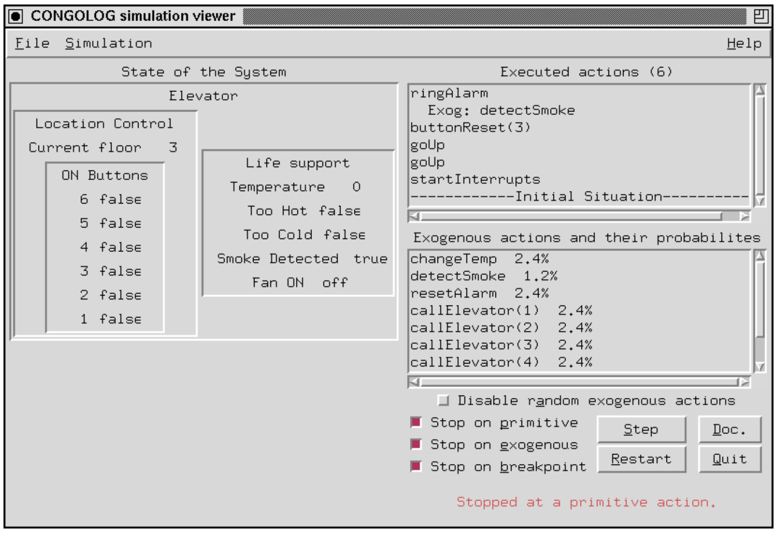

Figure 1: The *ConGolog* toolkit's graphical viewer.

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 [36]. 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 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 [36]. *ConGolog*-based tools for specifying and verifying complex multiagent systems are being investigated.

Finally, in [7], 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 seen 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 into complex actions for high-level agent control. The semantics of the resulting language end up deriving directly from that of the underlying primitive actions. In this sense, the solution to the frame problem provided by successor state axioms for primitive actions is extended to cover the complex actions of *ConGolog*. So *ConGolog* can be viewed as an action theory (that supports complex actions), as a specification language, and as an implementation language, and has been used in all three ways.

There are, however, many areas for future research. Among them, we mention: 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; and also incorporating utilities, so that nondeterministic choices in execution can be made to maximize the expected benefit.

Acknowledgments

This research has been funded by the National Science and Engineering Research Council of Canada. We thank Koen Hindriks for interesting discussions about this work and for pointing out an error in an earlier version regarding situations other than *now* appearing in primitive actions and tests. We also thank David Tremaine for alerting us to a problem with an earlier way of handling exogenous events in the semantics. We thank David McAllester for pointing out the subtleties that arise in extending *Golog* and *ConGolog* with boolean procedures. Finally, we thank the referees for their comments.

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] X. J. Chen and G. De Giacomo. Reasoning about nondeterministic and concurrent actions: A process algebra approach. *Artificial Intelligence*, **1071**, pages 63–98, 1999.

[6] 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.

[7] 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.

[8] M. Dixon. *Embedded Computation and the Semantics of Programs.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, 1991. Also appeared as Xerox PARC Technical Report SSL-91-1.

[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] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with Continuous Change. *Science of Computer Programming*, **30**, 3–50, 1998.

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

[16] 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.

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

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

[19] 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.

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

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

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

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

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

[26] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, 1992.

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

[28] 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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

[35] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245, 1990.

[36] 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.

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

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

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

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

A Appendix: Programs as Terms

In this section, we develop an encoding of programs as first-order terms. Although some care is required (e.g. introducing constants denoting variables and defining substitution explicitly in the language), this does not pose any major problem; see [18] for an introduction to problems and techniques in this area.

We add to the sorts *Sit*, *Obj* and *Act* of the Situation Calculus, the following new sorts: *Idx*, *PseudoSit*, *PseudoAct*, *PseudoObj*, *PseudoForm*, *ENV*, and *PROG*.

Intuitively, elements of *Idx* denote natural numbers, and are used for building indexing functions. Elements of *PseudoAct*, *PseudoObj*, *PseudoSit* and *PseudoForm* are syntactic devices to denote respectively actions, objects, situations and formulas within programs. Elements of *ENV* denote environments, i.e sets of procedure definitions. And finally, elements of *PROG* denote programs, which are considered as simply syntactic objects.

A.1 Sort *Idx*

We introduce the constant 0 of sort *Idx*, and a function $\mathtt{succ} : \mathit{Idx} \to \mathit{Idx}$. For them we enforce the following unique name axioms:

$$\begin{array}{l} \mathtt{succ}(i) \neq 0 \\ \mathtt{succ}(i) = \mathtt{succ}(i') \supset i = i' \end{array}$$

We define the predicate $\mathtt{Idx} : \mathit{Idx}$ as:

$$\mathtt{Idx}(i) \quad \equiv \quad \forall X.[\ \ldots \supset X(i)]$$

where ... stands for the universal closure of

$$\begin{array}{lcl} X(0) & & \\ X(i) & \supset & X(\mathtt{succ}(i)) \end{array}$$

Finally we assume the following domain closure axiom for sort *Idx*:

$$\forall i.\mathtt{Idx}(i)$$

A.2 Sorts *PseudoSit*, *PseudoObj*, *PseudoAct*

The languages of *PseudoSit*, *PseudoObj* and *PseudoAct* are as follows:

- A constant $\mathit{Now} : \mathit{PseudoSit}$.
- A function $\mathtt{nameOf}_{\mathit{Sort}} : \mathit{Sort} \to \mathit{PseudoSort}$ for $\mathit{Sort} = \mathit{Obj}, \mathit{Act}$. We use the notation $[\![x]\!]$ to denote $\mathtt{nameOf}_{\mathit{Sort}}(x)$, leaving *Sort* implicit.
- A function $\mathtt{var}_{\mathit{Sort}} : \mathit{Idx} \to \mathit{PseudoSort}$ for $\mathit{Sort} = \mathit{Obj}, \mathit{Act}$. We call terms of the form $\mathtt{var}_{\mathit{Sort}}(i)$ *pseudo-variables* and we use the notation z_i (or just $\mathsf{x}, \mathsf{y}, \mathsf{z}$) to denote $\mathtt{var}_{\mathit{Sort}}(i)$, leaving *Sort* implicit.

- A function $\mathtt{f} : \mathit{PseudoSort}_1 \times \ldots \times \mathit{PseudoSort}_n \to \mathit{PseudoSort}_{n+1}$ for each fluent or nonfluent function f of sort $\mathit{Sort}_1 \times \ldots \times \mathit{Sort}_n \to \mathit{Sort}_{n+1}$ with $\mathit{Sort}_i = \mathit{Obj}, \mathit{Act}, \mathit{Sit}$ in the original language (note that if $n = 0$ then f is a constant).

We define the predicates $\mathtt{PseudoSit} : \mathit{PseudoSit}$, $\mathtt{PseudoObj} : \mathit{PseudoObj}$ and $\mathtt{PseudoAct} : \mathit{PseudoAct}$ respectively as:

$$\begin{array}{rcl} \mathtt{PseudoSit}(x) & \equiv & \forall P_{\mathit{Sit}}, P_{\mathit{Obj}}, P_{\mathit{Act}}.[\ \ldots \supset P_{\mathit{Sit}}(x)] \\ \mathtt{PseudoObj}(x) & \equiv & \forall P_{\mathit{Sit}}, P_{\mathit{Obj}}, P_{\mathit{Act}}.[\ \ldots \supset P_{\mathit{Obj}}(x)] \\ \mathtt{PseudoAct}(x) & \equiv & \forall P_{\mathit{Sit}}, P_{\mathit{Obj}}, P_{\mathit{Act}}.[\ \ldots \supset P_{\mathit{Act}}(x)] \end{array}$$

where ... stands for the universal closure of

$$\begin{array}{rcll} & & P_{\mathit{Sit}}(\mathit{Now}) & \\ & & P_{\mathit{Sort}}(\mathtt{nameOf}_{\mathit{Sort}}(x)) & \text{for } \mathit{Sort} = \mathit{Obj}, \mathit{Act} \\ & & P_{\mathit{Sort}}(\mathtt{z}_i) & \text{for } \mathit{Sort} = \mathit{Obj}, \mathit{Act} \\ P_{\mathit{Sort}}(x_1) \land \ldots \land P_{\mathit{Sort}}(x_n) & \supset & P_{\mathit{Sort}}(\mathtt{f}(x_1 \ldots, x_n)) & \text{(for each } \mathtt{f}) \end{array}$$

We assume the following domain closure axioms for the sorts *PseudoSit*, *PseudoObj* and *PseudoAct*:

$$\begin{array}{l} \forall x.\mathtt{PseudoSit}(x) \\ \forall x.\mathtt{PseudoObj}(x) \\ \forall x.\mathtt{PseudoAct}(x) \end{array}$$

We also enforce unique name axioms for them, that is, for all functions g, g' of any arity (including constants) introduced above:

$$\begin{array}{l} g(x_1, \ldots, x_n) \neq g'(y_1, \ldots, y_m) \\ g(x_1, \ldots, x_n) = g(y_1, \ldots, y_n) \supset x_1 = y_1 \land \ldots \land x_n = y_n \end{array}$$

Observe that the unique name axioms impose that $\mathtt{nameOf}(x) = \mathtt{nameOf}(y) \supset x = y$ but do not say anything on domain elements denoted by x and y since these are elements of *Act* or *Obj*.

Next we want to relate pseudo-situations, pseudo-objects and pseudo-actions to real situations, object and actions. In fact we do not want to relate all terms of sort *PseudoObj* and *PseudoAct* to real object and actions, but just the "closed" ones, i.e. those in which no pseudo variable $\mathtt{z}_i$ occur. To formalize the notion of *closedness*, we introduce the predicate $\mathtt{Closed} : \mathit{PseudoSort}$ for $\mathit{Sort} = \mathit{Sit}, \mathit{Obj}, \mathit{Act}$, characterized by the following assertions[20]

$$\begin{array}{rcll} & & \mathtt{Closed}(\mathit{Now}) & \\ & & \mathtt{Closed}(\mathtt{nameOf}(x)) & \\ & & \neg\mathtt{Closed}(\mathtt{z}_i) & \\ \mathtt{Closed}(\mathtt{f}(x_1, \ldots, x_n)) & \equiv & \mathtt{Closed}(x_1) \land \ldots \land \mathtt{Closed}(x_n) & \text{for each } \mathtt{f} \end{array}$$

Closed terms of sort *PseudoObj* and *PseudoAct* are related to real objects and actions by means of the function $\mathtt{decode} : (\mathit{PseudoSort} \times \mathit{Sit} \to \mathit{Sort})$ for $\mathit{Sort} = \mathit{Sit}, \mathit{Obj}, \mathit{Act}$. We use the

[20] We say the following theory is "characterizing" since it is complete, in the sense that it partitions the elements in *PseudoSort* into those that are *closed* and those that are not.

notation $x[s]$ to denote $\texttt{decode}(x, s)$. Such a function is characterized by the following assertions:

$$\begin{array}{l}\texttt{decode}(Now, s) = s \\ \texttt{decode}(\texttt{nameOf}(x), s) = x \\ \texttt{decode}(\texttt{f}(x_1 \ldots, x_n), s) = f(\texttt{decode}(x_1, s), \ldots, \texttt{decode}(x_n, s)) \quad \text{(for each } \texttt{f}\text{)}\end{array}$$

A.3 Sort *PseudoForm*

Next we introduce *pseudo-formulas* used in tests. Specifically, we introduce:

- A function $\texttt{p} : PseudoSort_1 \times \ldots \times PseudoSort_n \to PseudoForm$ for each nonfluent/fluent predicate p in the original language (not including the new the predicates introduced in this section).
- A function $\texttt{and} : PseudoForm \times PseudoForm \to PseudoForm$. We use the notation $\rho_1 \wedge \rho_2$ to denote $\texttt{and}(\rho_1, \rho_2)$.
- A function $\texttt{not} : PseudoForm \to PseudoForm$. We use the notation $\neg\rho$ to denote $\texttt{not}(\rho)$.
- A function $\texttt{some}_{Sort} : PseudoSort \times PseudoForm \to PseudoForm$, for $PseudoSort = PseudoObj, PseudoAct$. We use the notation $\exists \texttt{z}_i.\rho$ to denote $\texttt{some}(\texttt{var}_{Sort}(i), \rho)$, leaving *Sort* implicit.

We define the predicate $\texttt{PseudoForm} : PseudoForm$ as:

$$\texttt{PseudoForm}(\rho) \quad \equiv \quad \forall P_{Form}.[\ \ldots\ \supset P_{Form}(\rho)]$$

where ... stands for the universal closure of

$$\begin{array}{rcl} P_{Form}(\texttt{p}(x_1, \ldots, x_n)) & & \text{(for each } \texttt{p}\text{)} \\ P_{Form}(\rho_1) \wedge P_{Form}(\rho_2) & \supset & P_{Form}(\rho_1 \wedge \rho_2) \\ P_{Form}(\rho) & \supset & P_{Form}(\neg\rho) \\ P_{Form}(\rho) & \supset & P_{Form}(\exists \texttt{z}_i.\rho) \end{array}$$

We assume the following domain closure axiom for the sort *PseudoForm*:

$$\forall \rho.\texttt{PseudoForm}(\rho).$$

We also enforce unique name axioms for pseudo-formulas, that is, for all functions g, g' of any arity introduced above:

$$\begin{array}{l} g(x_1, \ldots, x_n) \neq g'(y_1, \ldots, y_m) \\ g(x_1, \ldots, x_n) = g(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \ldots \wedge x_n = y_n \end{array}$$

Next we formalize the notion of substitution. We introduce the function $\texttt{sub} : PseudoSort \times PseudoSort \times PseudoSort' \to PseudoSort'$ for $Sort = Obj, Act$ and $Sort' = Sit, Obj, Act$. We use the notation t^x_y to denote $\mathtt{sub}(x,y,t)$. Such a function is characterized by the following assertions:

$$
\begin{array}{l}
Now^x_y = Now \\
\mathtt{nameOf}(t)^x_y = \mathtt{nameOf}(t) \\
\mathtt{z}_i{}^{\mathtt{z}_i}_y = y \\
x \neq \mathtt{z}_i \supset \mathtt{z}_i{}^x_y = \mathtt{z}_i \\
\mathtt{f}(t_1,\ldots,t_n)^x_y = \mathtt{f}(t_1{}^x_y,\ldots,t_n{}^x_y) \quad \text{(for each } \mathtt{f}\text{)}
\end{array}
$$

We extend the function $\mathtt{sub}$ to pseudo-formulas (as third argument) as follows:

$$
\begin{array}{l}
\mathtt{p}(t_1,\ldots,t_n)^x_y = \mathtt{p}(t_1{}^x_y,\ldots,t_n{}^x_y) \quad \text{(for each } \mathtt{p}\text{)} \\
(\rho_1 \wedge \rho_2)^x_y = (\rho_1)^x_y \wedge (\rho_2)^x_y \\
(\neg\rho)^x_y = \neg(\rho)^x_y \\
(\exists \mathtt{z}_i.\rho)^{\mathtt{z}_i}_y = \exists \mathtt{z}_i.\rho \\
x \neq \mathtt{z}_i \supset (\exists \mathtt{z}_i.\rho)^x_y = \exists \mathtt{z}_i.(\rho^x_y)
\end{array}
$$

Next we extend the predicate $\mathtt{Closed}$ to pseudo-formulas in a natural way:

$$
\begin{array}{rcl}
\mathtt{Closed}(\mathtt{p}(x_1,\ldots,x_n)) & \equiv & \mathtt{Closed}(x_1) \wedge \ldots \wedge \mathtt{Closed}(x_n) \quad \text{for each } \mathtt{p} \\
\mathtt{Closed}(\rho_1 \wedge \rho_2) & \equiv & \mathtt{Closed}(\rho_1) \wedge \mathtt{Closed}(\rho_2) \\
\mathtt{Closed}(\neg\rho) & \equiv & \mathtt{Closed}(\rho_1) \\
\mathtt{Closed}(\exists \mathtt{z}_i.\rho) & \equiv & \forall y.\mathtt{Closed}(\rho^{\mathtt{z}_i}_{\mathtt{nameOf}(y)})
\end{array}
$$

We relate *closed* pseudo-formulas to real formulas by introducing a predicate $\mathtt{Holds} : \mathit{PseudoForm} \times \mathit{Sit}$, characterized by the following assertions:

$$
\begin{array}{rcl}
\mathtt{Holds}(\mathtt{p}(x_1,\ldots,x_n),s) & \equiv & p(\mathtt{decode}(x_1,s),\ldots,\mathtt{decode}(x_n,s)) \quad \text{(for each } \mathtt{p}\text{)} \\
\mathtt{Holds}(\rho_1 \wedge \rho_2, s) & \equiv & \mathtt{Holds}(\rho_1,s) \wedge \mathtt{Holds}(\rho_2,s) \\
\mathtt{Holds}(\neg\rho, s) & \equiv & \neg\mathtt{Holds}(\rho,s) \\
\mathtt{Holds}(\exists \mathtt{z}.\rho, s) & \equiv & \exists y.\mathtt{Holds}(\rho^{\mathtt{z}}_{\mathtt{nameOf}(y)}, s)
\end{array}
$$

where y in the last equation is any variable that does not appear in ρ. We use the notation $\phi[s]$ to denote $\mathtt{Holds}(\phi,s)$.

A.4 Sorts *PROG* and *ENV*

Now we are ready to introduce *programs*. Specifically, we introduce:

- A constant $nil : PROG$.
- A function $\mathtt{act} : PseudoAct \to PROG$. As notation we write simply a to denote $\mathtt{act}(a)$ when confusion cannot arise.
- A function $\mathtt{test} : PseudoForm \to PROG$. We use the notation $\rho?$ to denote $\mathtt{test}(\rho)$.
- A function $\mathtt{seq} : PROG \times PROG \to PROG$. We use the notation $\delta_1;\delta_2$ to denote $\mathtt{seq}(\delta_1,\delta_2)$.

- A function $\mathtt{choice} : \mathit{PROG} \times \mathit{PROG} \to \mathit{PROG}$. We use the notation $\delta_1 \mid \delta_2$ to denote $\mathtt{choice}(\delta_1, \delta_2)$.

- A function $\mathtt{iter} : \mathit{PROG} \to \mathit{PROG}$. We use the notation δ^* to denote $\mathtt{iter}(\delta)$.

- Two functions $\mathtt{pick}_{Sort} : \mathit{PseudoSort} \times \mathit{PROG} \to \mathit{PROG}$, where *PseudoSort* is either *PseudoObj* or *PseudoAct*. We use the notation $\pi \mathsf{z}_i.\delta$ to denote $\mathtt{pick}_{Sort}(\mathtt{var}_{Sort}(i), \delta)$, leaving *Sort* implicit.

- A function $\mathtt{if} : \mathit{PseudoForm} \times \mathit{PROG} \times \mathit{PROG} \to \mathit{PROG}$. We use the notation $\mathtt{if}\, \rho\, \mathtt{then}$ $\delta_1\ \mathtt{else}\ \delta_2$ to denote $\mathtt{if}(\rho, \delta_1, \delta_2)$.

- A function $\mathtt{while} : \mathit{PseudoForm} \times \mathit{PROG} \to \mathit{PROG}$. We use the notation $\mathtt{while}\, \rho\ \mathtt{do}\ \delta$ to denote $\mathtt{while}(\rho, \delta)$.

- A function $\mathtt{conc} : \mathit{PROG} \times \mathit{PROG} \to \mathit{PROG}$. We use the notation $\delta_1 \parallel \delta_2$ to denote $\mathtt{conc}(\delta_1, \delta_2)$.

- A function $\mathtt{prconc} : \mathit{PROG} \times \mathit{PROG} \to \mathit{PROG}$. We use the notation $\delta_1 \rangle\!\rangle\ \delta_2$ to denote $\mathtt{prconc}(\delta_1, \delta_2)$.

- A function $\mathtt{iterconc} : \mathit{PROG} \to \mathit{PROG}$. We use the notation $\delta^{\parallel}$ to denote $\mathtt{iterconc}(\delta)$.

To deal with procedures we need to introduce the notion of environment together with that of program. We introduce:

- A finite number of functions $\mathtt{P} : \mathit{PseudoSort}_1 \times \ldots \times \mathit{PseudoSort}_n \to \mathit{PROG}$, where $\mathit{PseudoSort}_i$ is either *PseudoObj* or *PseudoAct*. These functions are going to be used as procedure calls.

- A function $\mathtt{proc} : \mathit{PROG} \times \mathit{PROG} \to \mathit{PROG}$. This function is used to build procedure definitions and so we will force the first argument to have the form $\mathtt{P}(\mathsf{z}_{\mathsf{i}_1}, \ldots, \mathsf{z}_{\mathsf{i}_n})$, where $\mathsf{z}_1 \ldots \mathsf{z}_n$ are used to denote the formal parameters of the defined procedure. We use the notation $\mathtt{proc}\ \mathtt{P}(\mathsf{z}_1, \ldots, \mathsf{z}_n)\ \delta\ \mathtt{end}$ to denote $\mathtt{proc}(\mathtt{P}(\mathsf{z}_1, \ldots, \mathsf{z}_n), \delta)$.

- A constant $\varepsilon : \mathit{ENV}$, denoting the *empty environment*.

- A function $\mathtt{addproc} : \mathit{ENV} \times \mathit{PROG} \to \mathit{ENV}$. We will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \mathtt{proc}\ \mathtt{P}(\vec{\mathsf{z}})\ \delta\ \mathtt{end}$ to denote $\mathtt{addproc}(\mathcal{E}, \mathtt{proc}\ \mathtt{P}(\vec{\mathsf{z}})\ \delta\ \mathtt{end})$.

- A function $\mathtt{pblock} : \mathit{ENV} \times \mathit{PROG} \to \mathit{PROG}$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\mathtt{pblock}(\mathcal{E}, \delta)$.

- A function $\mathtt{c\_call} : \mathit{ENV} \times \mathit{PROG} \to \mathit{PROG}$. We will restrict the programs allowed to appear as the second argument to procedure calls only. We use the notation $[\mathcal{E} : \mathtt{P}(\vec{t})]$ to denote $\mathtt{c\_call}(\mathcal{E}, \mathtt{P}(\vec{t}))$.

We next introduce a predicate $\texttt{defined} : PROG \times ENV$ meaning that a procedure is defined in an environment. It is specified as:

$$\texttt{defined}(c, \mathcal{E}) \equiv \forall D.[\ \ldots \supset D(c, \mathcal{E})]$$

where ... stands for

$$\begin{array}{l} D(\mathtt{P}(\vec{x}), \varepsilon; \texttt{proc P}(\vec{y})\ \delta\ \texttt{end}) \\ D(c, \mathcal{E}') \ \supset \ D(c, \mathcal{E}'; d) \end{array}$$

Observe that procedures $\mathtt{P}$ are only defined in an environment $\mathcal{E}$, and that the parameters the procedure is applied to do not play any role in determining whether the procedure is defined.

Now we define the predicate $\texttt{Prog} : PROG$ and the predicate $\texttt{Env} : ENV$ as:

$$\begin{array}{rcl} \texttt{Prog}(\delta) & \equiv & \forall P_{PROG}, P_{ENV}.[\ \ldots \supset P_{PROG}(\delta)] \\ \texttt{Env}(\mathcal{E}) & \equiv & \forall P_{PROG}, P_{ENV}.[\ \ldots \supset P_{ENV}(\mathcal{E})] \end{array}$$

where ... stands for the universal closure of

$$\begin{array}{rcl} P_{PROG}(nil) & & \\ P_{PROG}(\texttt{act}(a)) & & (a \text{ pseudo-action}) \\ P_{PROG}(\rho?) & & (\rho \text{ pseudo-formula}) \\ P_{PROG}(\delta_1) \wedge P_{PROG}(\delta_2) & \supset & P_{PROG}(\delta_1; \delta_2) \\ P_{PROG}(\delta_1) \wedge P_{PROG}(\delta_2) & \supset & P_{PROG}(\delta_1 \mid \delta_2) \\ P_{PROG}(\delta) & \supset & P_{PROG}(\delta^*) \\ P_{PROG}(\delta) & \supset & P_{PROG}(\pi \mathtt{z}_i.\delta) \\ P_{PROG}(\delta_1) \wedge P_{PROG}(\delta_2) & \supset & P_{PROG}(\texttt{if}\ \rho\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2) \\ P_{PROG}(\delta) & \supset & P_{PROG}(\texttt{while}\ \rho\ \texttt{do}\ \delta) \\ P_{PROG}(\delta_1) \wedge P_{PROG}(\delta_2) & \supset & P_{PROG}(\delta_1 \parallel \delta_2) \\ P_{PROG}(\delta_1) \wedge P_{PROG}(\delta_2) & \supset & P_{PROG}(\delta_1 \rangle\!\rangle\ \delta_2) \\ P_{PROG}(\delta) & \supset & P_{PROG}(\delta^{\parallel}) \\ P_{PROG}(\mathtt{P}(x_1, \ldots, x_n)) & & (\text{for each } \mathtt{P}) \\ P_{ENV}(\mathcal{E}) \wedge P_{PROG}(\delta) & \supset & P_{PROG}(\{\mathcal{E}; \delta\}) \\ P_{ENV}(\mathcal{E}) \wedge \texttt{defined}(\mathtt{P}(\vec{z}), \mathcal{E}) & \supset & P_{PROG}([\mathcal{E} : \mathtt{P}(x_1, \ldots, x_n)]) \end{array}$$

$$\begin{array}{l} P_{ENV}(\varepsilon) \\ P_{ENV}(\mathcal{E}) \wedge P_{PROG}(\delta) \wedge \neg\texttt{defined}(\mathtt{P}(\vec{z}), \mathcal{E}) \wedge (\bigwedge_{h,k=1}^{n} \mathtt{z}_{i_h} \neq \mathtt{z}_{i_k}) \supset \\ \qquad\qquad P_{ENV}(\mathcal{E}; \texttt{proc P}(\mathtt{z}_{i_1}, \ldots, \mathtt{z}_{i_n})\ \delta\ \texttt{end}) \end{array}$$

We assume the following domain closure axioms for the sorts $PROG$ and ENV:

$$\forall \delta.\texttt{Prog}(\delta) \qquad\qquad \forall \mathcal{E}.\texttt{Env}(\mathcal{E}).$$

We also enforce unique name axioms for programs and environments, that is for all functions g, g' of any arity introduced above:

$$\begin{array}{l} g(x_1, \ldots, x_n) \neq g'(y_1, \ldots, y_m) \\ g(x_1, \ldots, x_n) = g(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \ldots \wedge x_n = y_n \end{array}$$

We extend the predicate Closed to $PROG$ by induction on the structure of the program terms in the obvious way so as to consider *closed*, programs in which all occurrences of pseudo-variables $\mathtt{z}_i$ are bound either by π, or by being a formal parameter of a procedure. *Only closed programs are considered legal.*

We introduce the function $\mathtt{resolve} : ENV \times PROG \times PROG \to PROG$, to be used to associate to procedure calls the environment to be used to resolve them. Namely, given the procedure P defined in the environment $\mathcal{E}$, $\mathtt{resolve}(\mathcal{E},\mathtt{P}(\vec{t}),\delta)$ denoted by $(\delta)^{\mathtt{P}(\vec{t})}_{[\mathcal{E}:\mathtt{P}(\vec{t})]}$, suitably replaces $\mathtt{P}(\vec{t})$ by $\mathtt{c\_call}(\mathcal{E},\mathtt{P}(\vec{t}))$ in order to obtain static scope for procedures. It is obvious how the function can be extended to resolve whole sets of procedure calls whose procedures are defined in the environment $\mathcal{E}$. Formally this function satisfies the following assertions:

$$
\begin{array}{rcl}
(nil)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & nil \\
(a)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & a \\
(\rho?)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & \rho? \\
(\delta_1;\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & (\delta_1)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]};(\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\delta_1 \mid \delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & (\delta_1)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \mid (\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\pi \mathtt{z}_i.\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & \pi \mathtt{z}_i.(\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\delta^*)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & ((\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]})^* \\
(\mathtt{if}\,\rho\,\mathtt{then}\,\delta_1\,\mathtt{else}\,\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & \mathtt{if}\,\rho\,\mathtt{then}\,(\delta_1)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]}\,\mathtt{else}\,(\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\mathtt{while}\,\rho\,\mathtt{do}\,\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & \mathtt{while}\,\rho\,\mathtt{do}\,(\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\delta_1 \parallel \delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & (\delta_1)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \parallel (\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\delta_1 \rangle\!\rangle\, \delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & (\delta_1)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \rangle\!\rangle\, (\delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \\
(\delta^{\parallel})^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & ((\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]})^{\parallel} \\
(\mathtt{P}(\vec{x}))^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & [\mathcal{E} : \mathtt{P}(\vec{x})] \\
(\mathtt{Q}(\vec{t}))^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & \mathtt{Q}(\vec{t}) \quad \text{for any procedure call } \mathtt{Q}(\vec{t}) \text{ different from } \mathtt{P}(\vec{x}) \\
(\{\mathcal{E}';\delta\})^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & \left\{ \begin{array}{ll} \{\mathcal{E}';\delta\} & \text{if procedure P is (re)defined in } \mathcal{E}' \\ \{\mathcal{E}';(\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]}\} & \text{otherwise} \end{array} \right. \\
([\mathcal{E}' : \mathtt{Q}(\vec{t})])^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & [\mathcal{E}' : \mathtt{Q}(\vec{t})] \quad \text{for every procedure call } \mathtt{Q}(\vec{t}) \text{ and environment } \mathcal{E}'
\end{array}
$$

Finally, we extend the function sub to $PROG$ (as third argument) again by induction on the structure of program terms in the natural way considering π as a binding construct for pseudo-variables and without doing any substitutions into environments. sub is used for substituting formal parameters with actual parameters in contextualized procedure calls, as well as to deal with π. We also introduce a function $\mathtt{c\_body} : PROG \times ENV \to PROG$ to be used to return the body of the procedures. Namely, $\mathtt{c\_body}(\mathtt{P}(\vec{x}),\mathcal{E})$ returns the body of the procedure P in $\mathcal{E}$ with the formal parameters substituted by the actual parameters $\vec{x}$. Formally this function satisfies the following assertions:

$$
\begin{array}{l}
\mathtt{c\_body}(\mathtt{P}(\vec{x}),\mathcal{E};\mathtt{proc}\ \mathtt{P}(\vec{y})\ \delta\ \mathtt{end}) = \delta^{\vec{y}}_{\vec{x}} \\
\mathtt{c\_body}(\mathtt{P}(\vec{x}),\mathcal{E};\mathtt{proc}\ \mathtt{Q}(\vec{y})\ \delta\ \mathtt{end}) = \mathtt{c\_body}(\mathtt{P}(\vec{x}),\mathcal{E}) \quad (\mathtt{Q} \neq \mathtt{P})
\end{array}
$$

A.5 Consistency preservation

The encoding presented here preserves consistency as stated by the following theorem.

Theorem 9: *Let $\mathcal{H}$ be the axioms defining the encoding above. Then every model of an action theory $\mathcal{D}$ (involving sorts Sit, Act and Obj) can be extended to a model of $\mathcal{H} \cup \mathcal{D}$ (involving the additional sorts Idx, PseudoSit, PseudoAct, PseudoObj, PseudoForm, ENV and PROG).*

Proof: It suffices to observe that for each new sort $(Idx, \ldots, PROG)$ $\mathcal{H}$ contains:

- A second-order axiom that explicitly defines a predicate which inductively characterizes the elements of the sort.
- An axiom that closes the domain of the new sort with respect to the characterizing predicate.
- Unique name axioms that extend the interpretation of $=$ to the new sort by induction on the structure of the elements (as imposed by the characterizing axiom).
- Axioms that characterize predicates and functions, such as `Closed`, `decode`, `sub`, `Holds`, etc., by induction on the structure of the elements of the sort.

Hence, given a model M of the action theory $\mathcal{D}$, it is straightforward to introduce domains for the new sorts that satisfy the characterizing predicate, the domain closure axioms, and the unique name axioms for the sort, by proceeding by induction on the structure of the elements forced by the characterizing predicate, and then establishing the extension of the newly defined predicates/functions for the sort. $\square$

B Appendix: Proof of Theorem 1 – Equivalence between the *Do*'s for *Golog* programs

In this section, we prove Theorem 1, i.e. the equivalence of the original definition of *Do* and the new one given in this paper, in the more general language which includes procedures. To simplify the presentation of the proof, we use the same symbols to denote terms and elements of the domain of interpretation; the meaning will be clear from the context.

B.1 Alternative definitions of *Trans* and *Final*

For proving the following results, it is convenient to reformulate the definitions of *Trans* and *Final*:

- $Trans(\delta, s, \delta', s') \equiv \forall T.[\ \ldots\ \supset T(\delta, s, \delta', s')]$, where $\ldots$ stands for the conjunction of the universal closure of the following implications:

$$
\begin{array}{rcl}
Poss(a[s], s) & \supset & T(a, s, nil, do(a[s], s)) \\
\phi[s] & \supset & T(\phi?, s, nil, s) \\
T(\delta, s, \delta', s') & \supset & T(\delta; \gamma, s, \delta'; \gamma, s') \\
Final(\gamma, s) \wedge T(\delta, s, \delta', s') & \supset & T(\gamma; \delta, s, \delta', s') \\
T(\delta, s, \delta', s') & \supset & T(\delta \mid \gamma, s, \delta', s') \\
T(\delta, s, \delta', s') & \supset & T(\gamma \mid \delta, s, \delta', s') \\
T(\delta^v_x, s, \delta', s') & \supset & T(\pi v.\delta, s, \delta', s') \\
T(\delta, s, \delta', s') & \supset & T(\delta^*, s, \delta'; \delta^*, s') \\
T(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s, \delta', s') & \supset & T(\{Env; \delta\}, s, \delta', s') \\
T(\{Env; \delta_P{}^{\vec{v}_P}_{\vec{t}[s]}\}, s, \delta', s') & \supset & T([Env : P(\vec{t})], s, \delta', s')
\end{array}
$$

- $Final(\delta, s) \equiv \forall F.[\ \ldots\ \supset F(\delta, s)]$, where $\ldots$ stands for the conjunction of the universal closure of the following implications:

$$
\begin{array}{rcl}
True & \supset & F(nil, s) \\
F(\delta, s) \wedge F(\gamma, s) & \supset & F(\delta; \gamma, s) \\
F(\delta, s) & \supset & F(\delta \mid \gamma, s) \\
F(\delta, s) & \supset & F(\gamma \mid \delta, s) \\
F(\delta^v_x, s) & \supset & F(\pi v.\delta, s) \\
True & \supset & F(\delta^*, s) \\
F(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s) & \supset & F(\{Env; \delta\}, s) \\
F(\{Env; \delta_P{}^{\vec{v}_P}_{\vec{t}[s]}\}, s) & \supset & F([Env : P(\vec{t})], s)
\end{array}
$$

Theorem 10: *With respect to* Golog *programs, the definitions above are* equivalent *to the ones given in Section 7 of the paper.*

Proof: To prove this equivalence, consider first the following general results, which are a direct consequence of the Tarski-Knaster fixpoint theorem [40]. If

$$S(\vec{x}) \quad \equiv \quad \forall Z.[[\forall \vec{y}.\Phi(Z,\vec{y})\supset Z(\vec{y})]\supset Z(\vec{x})] \tag{4}$$

and $\Phi(Z,\vec{y})$ is monotonic (i.e. $\forall Z_1, Z_2.[\forall \vec{y}.Z_1(\vec{y})\supset Z_2(\vec{y})] \supset [\forall \vec{y}.\Phi(Z_1,\vec{y})\supset\Phi(Z_2,\vec{y})]$), then we get the following consequences[21]

$$S(\vec{x}) \quad \equiv \quad \Phi(S,\vec{x}) \tag{5}$$
$$S(\vec{x}) \quad \equiv \quad \forall Z.[[\forall \vec{y}.Z(\vec{y}) \equiv \Phi(Z,\vec{y})]\supset Z(\vec{x})]. \tag{6}$$

Now it is easy to see that the above definition of *Trans* and *Final* can be rewritten as (4) and that the resulting Φ is indeed monotonic (in particular it is syntactically monotonic since the predicate variables T and F do not occur in the scope of any negation). Thus, by the Tarski-Knaster fixpoint theorem, the above definitions can be rewritten in the form of (6). Once in this form it is easy to see that for *Golog* programs they are equivalent to those introduced in Section 7. □

B.2 Do_1 is equivalent to Do_2

Let Do_1 be the original definition of Do in [20] extended with $Do_1(nil,s,s') \stackrel{def}{=} s' = s$ and $Do([Env : P(\vec{t})],s,s') \stackrel{def}{=} Do(\{Env; P(\vec{t})\},s,s')$, and Do_2 the new definition in terms of *Trans* and *Final*. Also, we do not allow procedure calls for which no procedure definitions are given.

Lemma 4 : *For every model M of $\mathcal{C}$, there exist $\delta_1, s_1 \ldots \delta_n, s_n$ such that $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_{i+1})$ for $i = 1,\ldots,n-1$ if and only if $M \models Trans^*(\delta_1,s_1,\delta_n,s_n)$.*

Proof: $\Rightarrow$ By induction on n. If $n = 1$, then $M \models Trans^*(\delta_1,s_1,\delta_1,s_1)$ by definition of $Trans^*$. If $n > 1$, then by induction hypothesis $M \models Trans^*(\delta_2,s_2,\delta_n,s_n)$, and since $M \models Trans(\delta_1,s_1,\delta_2,s_2)$, we get $M \models Trans^*(\delta_1,s_1,\delta_n,s_n)$ by definition of $Trans^*$.

$\Leftarrow$ Let $\mathcal{R}$ be the relation formed by the tuples $(\delta_1,s_1,\delta_n,s_n)$ such that there exist $\delta_1,s_1\ldots\delta_n,s_n$ and $M \models Trans(\delta_i,s_i,\delta_{i+1},s_{i+1})$ for $i = 1,\ldots,n-1$. It is easy to verify that (i) for all δ,s, $(\delta,s,\delta,s) \in \mathcal{R}$; (ii) for all $\delta,s,\delta',s',\delta'',s''$, $M \models Trans(\delta,s,\delta',s')$ and $(\delta',s',\delta'',s'') \in \mathcal{R}$ implies $(\delta,s,\delta'',s'') \in \mathcal{R}$. □

Lemma 5: *For every model M of $\mathcal{C}$, $M \models Do_1(\delta,s,s')$ implies that there exist $\delta_1,s_1\ldots\delta_n,s_n$ such that $\delta_1 = \delta$, $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n,s_n)$, and $M \models Trans(\delta_i,s_i,\delta_{i+1},s_{i+1})$ for $i = 1,\ldots,n-1$.*

Proof: We prove the lemma by induction on the structure of the program. We only give details for the most significant cases.

[21] In fact, (5) is only mentioned in passing and not used in the proof.

1. a (atomic action). $M \models Do_1(a, s, s')$ iff $M \models Poss(a[s], s)$ and $s' = do(a[s], s)$. Then $M \models Trans(a, s, nil, do(a[s], s))$, and hence the thesis.

2. $\delta;\gamma$ (sequence). $M \models Do_1(\delta;\gamma, s, s')$ iff $M \models Do_1(\delta, s, s'')$ and $M \models Do_1(\gamma, s'', s')$.

   Then by induction hypothesis: (i) there exist $\delta_1, s_1 \ldots, \delta_k, s_k$ such that $\delta_1 = \delta$, $s_1 = s$, $s_k = s''$, $M \models Final(\delta_k, s_k)$ and $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_i)$ for $i = 1, \ldots, k-1$; (ii) there exist $\gamma_k, s_k \ldots, \gamma_n, s_n$ such that $\gamma_1 = \gamma$, $s_k = s''$, $s_n = s'$, $M \models Final(\gamma_n, s_n)$ and $M \models Trans(\gamma_i, s_i, \gamma_{i+1}, s_i)$ for $i = k, \ldots, n-1$.

   Since $Trans$ itself is closed under the assertions in its definition we have that: $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_{i+1})$ implies $M \models Trans(\delta_i;\gamma, s_i, \delta_{i+1};\gamma, s_{i+1})$. Moreover $M \models Final(\delta_k, s_k)$ and $M \models Trans(\gamma_k, s_k, \gamma_{k+1}, s_{k+1})$ implies $M \models Trans(\delta_k;\gamma_k, s_k, \gamma_{k+1}, s_{k+1})$. Similarly in the case $k = n$ we have that, since $Final$ is also closed under the assertions in its definition $M \models Final(\delta_k, s_k)$ and $M \models Final(\gamma_k, s_k)$ implies $M \models Final(\delta_k;\gamma_k, s_k)$. Hence the thesis.

3. δ^* (iteration). $M \models Do_1(\delta^*, s, s')$ iff $M \models \forall P.[\ \ldots\ \supset P(s, s')]$ where $\ldots$ stand for the following two assertions: (i) $\forall s.P(s, s)$; (ii) $\forall s, s', s''.Do_1(\delta, s, s'') \wedge P(s'', s') \supset P(s, s')$.

   Consider the relation $\mathcal{Q}$ defined as the set of pairs (s, s') such that: there exist $\delta_1, s_1 \ldots, \delta_n, s_n$ with $\delta_1 = \delta^*$, $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n, s_n)$ and $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_i)$ for $i = 1, \ldots, n-1$. To prove the thesis, it is sufficient to show that $\mathcal{Q}$ satisfies the two assertions (i) and (ii).

   - (i) Let $\delta_1 = \delta_n = \delta^*$, $s_1 = s_n = s$; since $M \models Final(\delta^*, s)$, it follows that for all s, $(s, s) \in \mathcal{Q}$.
   - (ii) By the first induction hypothesis (the induction on the structure of the program): $M \models Do_1(\delta, s, s'')$ implies that there exist $\delta_1, s_1 \ldots, \delta_k, s_k$ such that $\delta_1 = \delta$, $s_1 = s$, $s_k = s''$, $M \models Final(\delta_k, s_k)$ and $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_{i+1})$ for $i = 1, \ldots, k-1$. This implies that $M \models Trans(\delta_i;\delta^*, s_i, \delta_{i+1};\delta^*, s_{i+1})$ for $i : 2, \ldots, k-1$. Moreover, we must also have $M \models Trans(\delta^*, s_1, \delta_2;\delta^*, s_2)$.

     By the second induction hypothesis (rule induction for P), we can assume that there exist $\gamma_k, s_k \ldots, \gamma_n, s_n$ such that $\gamma_k = \delta^*$, $s_k = s''$, $s_n = s'$, $M \models Final(\gamma_n, s_n)$ and $M \models Trans(\gamma_i, s_i, \gamma_{i+1}, s_{i+1})$ for $i = k, \ldots, n-1$.

     Now observe that $Final(\delta_k, s_k)$ and $Trans(\gamma_k, s_k, \gamma_{k+1}, s_{k+1})$ implies that $Trans(\delta_k;\gamma_k, s_k, \gamma_{k+1}, s_{k+1})$. Thus, we get that (ii) holds for $\mathcal{Q}$.

   Hence the thesis.

4. $\{Env;\delta\}$ (procedures). $M \models Do_1(\{Env;\delta\}, s, s')$ iff

$$M \models \forall P_1, \ldots, P_n.\ [\Phi\ \supset\ Do_1(\delta, s, s')]$$

   where

$$\Phi = [\bigwedge_{i=1}^{n} \forall \vec{x}, s, s'.Do_1(\delta_i{}^{\vec{v}_i}_{\vec{x}}, s, s') \supset P_i(\vec{x}, s, s')]. \tag{7}$$

To get the thesis, it suffices to prove it for the case:

$$M \models \forall P_1, \ldots, P_n.\ [\Phi\ \supset\ P_i(\vec{x}, s, s')] \tag{8}$$

and then apply the induction argument on the structure of the program considering as base cases *nil*, a, $\phi?$, and $P(\vec{t})$.

Consider the relations $\mathcal{Q}_i$ defined as the set of tuples $(\vec{x}, s, s')$ such that there exist $\delta_1, s_1 \ldots, \delta_n, s_n$ with $\delta_1 = \{Env; P_i(\vec{x})\}$[22], $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n, s_n)$ and $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_i)$ for $i = 1, \ldots, n-1$. To prove the thesis it is sufficient to show that each $\mathcal{Q}_i$ satisfies (is closed under) the assertion (7).

Recall that $Do_1(P_i(\vec{x})), s, s') \stackrel{def}{=} P_i(\vec{x}, s, s')$ where P_i is a free predicate variable. This means that for any variable assignment σ, $M, \sigma^{P_1,\ldots,P_n}_{\mathcal{Q}_1,\ldots,\mathcal{Q}_n} \models Do_1(P_i(\vec{x}), s, s')$ implies $(\vec{x}, s, s') \in \mathcal{Q}_i$, i.e., there exist $\delta_1, s_1 \ldots, \delta_n, s_n$ with $\delta_1 = \{Env; P_i(\vec{x})\}$, $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n, s_n)$ and $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_i)$ for $i = 1, \ldots, n-1$. Hence by induction on the structure of the program, considering as base cases *nil*, a, $\phi?$ and $P(\vec{t})$, we have that $M, \sigma^{P_1,\ldots,P_n}_{\mathcal{Q}_1,\ldots,\mathcal{Q}_n} \models Do_1({\delta_i}^{\vec{v}_i}_{\vec{x}}, s, s')$ implies that there exist $\delta_1, s_1 \ldots, \delta_n, s_n$ with $\delta_1 = \{Env; {\delta_i}^{\vec{v}_i}_{\vec{x}}\}$, $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n, s_n)$ and $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_i)$ for $i = 1, \ldots, n-1$. Now considering that $M \models Trans(\{Env; {\delta_i}^{\vec{v}_i}_{\vec{x}}\}, s_1, \delta_2, s_2)$ implies $M \models Trans([Env : P_i(\vec{x})], s_1, \delta_2, s_2)$ implies $M \models Trans(\{Env; P_i(\vec{x})\}, s_1, \delta_2, s_2)$, we get that $(\vec{x}, s, s') \in \mathcal{Q}_i$.

□

Lemma 6: *For all* Golog *programs δ and situations s:*

$$Final(\delta, s) \supset Do_1(\delta, s, s)$$

Proof: It is easy to show that $Do_1(\delta, s, s)$ is closed with respect to the implications in the inductive definition of *Final*. □

Lemma 7: *For all* Golog *programs δ, δ' and situations s, s':*

$$Trans(\delta, s, \delta', s') \wedge Do_1(\delta', s', s'') \supset Do_1(\delta, s, s'').$$

Proof: The property we want to prove can be rewritten as follows:

$$Trans(\delta, s, \delta', s') \supset \Phi(\delta, s, \delta', s')$$

with

$$\Phi(\delta, s, \delta', s') \stackrel{def}{=} \forall s''. Do_1(\delta', s', s'') \supset Do_1(\delta, s, s'').$$

Hence it is sufficient to show that Φ is closed under the implications that inductively define *Trans*. Again, we only give details for the most significant cases.

[22] To be more precise, the variables x_i in $P_i(\vec{x})$ should be read as $\mathtt{nameOf}(x_i)$ thus converting situation calculus objects/actions variables into suitable program terms (see appendix A).

1. Implication for primitive actions. We show that $Poss(a[s], s) \supset \Phi(a[s], s, nil, do(a[s], s))$ i.e.:

$$Poss(a[s], s) \supset \forall s''. Do_1(nil, do(a[s], s), s'') \supset Do_1(a, s, s'').$$

Since $Do_1(nil, s, s') \stackrel{def}{=} s' = s$, this reduces to $Poss(a[s], s) \supset Do_1(a, s, do(a, s))$, which holds by the definition of Do_1.

2. First implication for sequences. We have to show $\Phi(\delta, s, \delta', s') \supset \Phi(\delta; \gamma, s, \delta', s')$, i.e.:

$$\forall s''.[Do_1(\delta', s', s'') \supset Do_1(\delta, s, s'')] \supset \forall s''. Do_1(\delta'; \gamma, s', s'') \supset Do_1(\delta; \gamma, s, s'').$$

By contradiction. Suppose that there is a model M such that $M \models \forall s''. Do_1(\delta', s', s'') \supset Do_1(\delta, s, s'')$, and $M \models Do_1(\delta'; \gamma, s', s_c)$ and $M \models \neg Do_1(\delta; \gamma, s, s_c)$ for some s_c. This means that $M \models Do_1(\delta', s', s_t) \wedge Do_1(\gamma, s_t, s_c)$ for some s_t, but $M \models \forall t. \neg Do_1(\delta, s, t) \vee \neg Do_1(\gamma, t, s_c)$. Since $M \models Do_1(\delta', s', s_t)$ implies $M \models Do_1(\delta, s, s_t)$, we have a contradiction.

3. Second implication for sequences. We have to show $Final(\delta, s) \wedge \Phi(\gamma, s, \gamma', s') \supset \Phi(\delta; \gamma, s, \gamma', s')$, i.e.:

$$Final(\delta, s) \wedge \forall s''.[Do_1(\gamma', s', s'') \supset Do_1(\gamma, s, s'')] \supset \forall s''. Do_1(\gamma', s', s'') \supset Do_1(\delta; \gamma, s, s'').$$

By contradiction. Suppose that there is a model M such that $M \models Final(\delta, s)$, $M \models \forall s''. Do_1(\gamma', s', s'') \supset Do_1(\gamma, s, s'')$, and $M \models Do_1(\gamma', s', s_c)$ – thus $M \models Do_1(\gamma, s, s_c)$ – and $M \models \neg Do_1(\delta; \gamma, s, s_c)$ for some s_c. The latter means that $M \models \forall t. \neg Do_1(\delta, s, t) \vee \neg Do_1(\gamma, t, s_c)$. Since $M \models Final(\delta, s)$ implies $M \models Do_1(\delta, s, s)$ by lemma 6, then $M \models \neg Do_1(\gamma, s, s_c)$, contradiction.

4. Implication for iteration. We have to show $\Phi(\delta, s, \delta', s') \supset \Phi(\delta^*, s, \delta'; \delta^*, s')$, i.e.:

$$\forall s''.[Do_1(\delta', s', s'') \supset Do_1(\delta, s, s'')] \supset \forall s''. Do_1(\delta'; \delta^*, s', s'') \supset Do_1(\delta^*, s, s'').$$

By contradiction. Suppose that there is a model M such that $M \models \forall s''. Do_1(\delta', s', s'') \supset Do_1(\delta, s, s'')$, and $M \models Do_1(\delta'; \delta^*, s', s_c)$ and $M \models \neg Do_1(\delta^*, s, s_c)$ for some s_c. Since $M \models Do_1(\delta'; \delta^*, s', s_c)$ implies $M \models Do_1(\delta', s', s_t)$ – thus $M \models Do_1(\delta, s, s_t)$ – and $M \models Do_1(\delta^*, s_t, s_c)$, and $M \models Do_1(\delta, s, s_t)$ and $M \models Do_1(\delta^*, s_t, s_c)$ imply $M \models Do_1(\delta^*, s, s_c)$, contradiction.

5. Implication for contextualized procedure calls. We have to show that

$$\Phi(\{Env; \delta_i{}^{\vec{v_i}}_{\vec{t}[s]}\}, s, \delta', s') \;\supset\; \Phi([Env : P_i(\vec{t})], s, \delta', s')$$

It suffices to prove that:

$$Do_1(\{Env; \delta_i{}^{\vec{v_i}}_{\vec{t}[s]}\}, s, s') \;\supset\; Do_1([Env : P_i(\vec{t})], s, s').$$

We proceed by contradiction. Suppose that there exists an model M such that $M \models Do_1(\{Env; \delta_i{}^{\vec{v_i}}_{\vec{t}[s]}\}, s, s')$ and $M \models \neg Do_1([Env; P_i(\vec{t})], s, s')$, for some $\vec{t}$, s and s'. That is:

$$M \models \forall P_1, \ldots, P_n.\ [\Psi \supset Do_1(\delta_i{}^{\vec{v_i}}_{\vec{t}[s]}, s, s')] \tag{9}$$

$$M \models \exists P_1, \ldots, P_n.\ [\Psi \wedge \neg P_i(\vec{t}[s]), s, s')]. \tag{10}$$

where $\Psi = [\bigwedge_{i=1}^{n} \forall \vec{x}_i, s, s'. Do_1(\delta_i{}^{\vec{v_i}}_{\vec{x}_i}, s, s') \supset P_i(\vec{x}_i, s, s')]$. Then by (10) there exists a variable assignment such that $M, \sigma \models \Psi$ and $M, \sigma \models \neg P_i(\vec{t}[s], s, s')$, which implies $M, \sigma \models \neg Do_1(\delta^{\vec{v_i}}_{\vec{t}[s]}, s, s')$, which contradicts (9).

6. Implication for programs within an environment. We have to show

$$\Phi(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s, \delta', s') \supset \Phi(\{Env; \delta\}, s, \delta', s').$$

It suffices to prove that:

$$Do_1(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s, s') \supset Do_1(\{Env; \delta\}, s, s')$$

This can be done by induction on the structure of the program δ considering *nil*, a, $\phi?$, and $[Env' : P(\vec{t})]$ as base cases (such programs do not make use of Env).

□

Lemma 8: *For every model M of $\mathcal{C}$, if there exist $\delta_1, s_1 \ldots \delta_n, s_n$ such that $\delta_1 = \delta$, $s_1 = s$, $s_n = s'$, $M \models$ Final(δ_n, s_n) and $M \models$ Trans$(\delta_i, s_i, \delta_{i+1}, s_{i+1})$ for $i = 1, \ldots, n-1$, then $M \models Do_1(\delta, s, s')$.*

Proof: By induction on n. If $n = 1$, then *Final*$(\delta, s) \supset Do_1(\delta, s, s)$ by lemma 6. If $n > 1$, then by induction hypothesis $M \models Do_1(\delta_2, s_2, s')$, hence by applying Lemma 7, we get the thesis. □

With these lemmas in place we can finally prove the wanted result:

Theorem 1: For each *Golog* program δ:

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

Proof: $\Rightarrow$ by Lemma 5 and Lemma 4; $\Leftarrow$ by Lemma 4 and Lemma 8. □