ConGolog, a concurrent programming language based on the situation calculus: foundations

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. The language includes facilities for prioritizing the concurrent execution, interrupting the execution when certain conditions become true, and dealing with exogenous actions. The language differs from other procedural formalisms for concurrency in that the initial state can be incompletely specified and the primitive actions can be user-defined by axioms in the situation calculus. In a companion paper, a formal definition in the situation calculus of such a programming language is presented and illustrated with detailed examples. In this paper, the mathematical properties of the programming language are explored.

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* [15]. The idea, roughly, is that instead of searching for a sequence of actions that would take the agent from an

initial state to some prespecified goal state, the task is to find a sequence of actions that constitutes a legal execution of some prespecified non-deterministic program. As in planning, to find a sequence that constitutes a legal execution of such a program, it is necessary to reason about the preconditions and effects of the actions within the body of the program. 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.

We argue that a new programming language called *ConGolog* [6] provides just such an expressive formalism for high-level control. *ConGolog* is an extension to the *Golog* programming language [15] that incorporates a rich account of *concurrency*, including prioritized execution, interrupts, and exogenous actions. In the companion paper [5], we explain and motivate *ConGolog* informally, review how the situation calculus and the solution to the frame problem proposed by Reiter [21] can be used to characterize the behavior of the application-dependent primitive actions, specify formally the execution semantics of *ConGolog* programs as axioms of the situation calculus, provide some examples of high-level agent controllers written in *ConGolog*, present a simple interpreter for *ConGolog* written in Prolog, and finally prove the correctness of this interpreter relative to the formal specification.

One of the nicest features of *ConGolog* is that it allows us to easily 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 required considerable mathematical machinery: we needed to encode *ConGolog* programs as terms in the situation calculus (which, among other things, required encoding certain formulas as terms), and we also needed 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.

This paper considers the mathematical foundations of *ConGolog*, and shows that our language specification is indeed mathematically well-behaved. The paper is self-contained in terms of definitions and theorems; motivation and examples, however, are to be found in the companion paper.

Of course ours is not the first formal model of concurrency. In fact, well developed approaches are available [13, 17, 3, 24][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 con-

[1] In [19, 4] a direct use of such approaches to model concurrent (complex) actions in AI is investigated.

trast to typical computer programs whose elementary instructions are simple predefined statements (*e.g.* variable assignments), the primitive actions of a *ConGolog* program are determined by a separate domain-dependent action theory, which specifies the action preconditions and effects, and deals with the frame problem. 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 review the situation calculus and the *Golog* programming language. In Section 3, we briefly explain the sort of concurrency we are concerned with, as well as related notions of priorities and interrupts. In Section 4 we present *ConGolog*'s formal semantics. In Section 5 we extend *ConGolog*'s formal semantics to deal with procedures. 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 6 we will show general sufficient conditions that allow us to use a much simplified semantics without loss of generality. A summary and topics for future research end the paper.

2 The Situation Calculus and Golog

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* [16], a first-order language (with some second-order features) for representing dynamic domains.

2.1 The Situation Calculus

We will not go over the language in details here (see [22]) except to note the following components: there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate domain theories which describe how the world changes as a result of the available actions. One possibility is a theory of the following form [21]:

- Axioms describing the initial situation, S_0.

- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [21].
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

For any domain theory of this sort, we have a very clean specification of the planning task, which dates back to the work of Green [10]:

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

In other words, the task is to find a sequence of actions that is executable (each action is executed in a context where its precondition is satisfied) and that achieves the goal (the goal formula ϕ holds in the final state that results from performing the actions in sequence).

2.2 Golog

As presented in [15], *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[2]
$\phi?$,	wait for a condition[3]
$(\delta_1; \delta_2)$,	sequence
$(\delta_1 \mid \delta_2)$,	nondeterministic choice between actions
$\pi v.\delta$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration
$\{\mathbf{proc}\ P_1(\vec{v}_1)\ \delta_1\ \mathbf{end}; \ldots \mathbf{proc}\ P_n(\vec{v}_n)\ \delta_n\ \mathbf{end};\ \delta\}$,	procedures

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

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

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 $OnTable(b, now)?$ succeeds only if the individual chosen, b, is a block that is on the table. The main part of the program uses the nondeterministic iteration operator; it simply says to execute *removeAblock* zero or more times until the table is clear. 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.

[2]Here, a stands for a situation calculus action with all situation arguments in its parameters replaced by the special constant *now*. Similarly in the line below ϕ stands for a situation calculus formula with all situation arguments replaced by *now*, for example $OnTable(block, now)$. $a[s]$ ($\phi[s]$) 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*.

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

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

In [15], $Do(\delta, s, s')$ 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 [15] for the details.

3 ConGolog

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 the companion paper [5], we discuss 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.[4]

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 the test of the condition ϕ does not involve a transition per se: the evaluation of the condition and the first action of the branch chosen will be executed as a unity. In other words, these

[4] 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.

constructs behave in a similar way to the test-and-set atomic instructions used to build semaphores in concurrent programming [1].[5]

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 $nil \mid \delta \mid (\delta;\delta) \mid (\delta;\delta;\delta) \mid \ldots$ (where nil represents the empty program), 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 the companion paper [5] 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. 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.

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 [11]), 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 [11]). 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

[5] In [15] a non-synchronized version of if-then else and while-loop is 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.

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 [18]. 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[6] (see Appendix A). In the following we abstract from the

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

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 5.

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

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

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

5. Nondeterministic branch:

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

6. Nondeterministic choice of argument:

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

7. Iteration:

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

8. Synchronized if-then-else:

$$\begin{aligned} Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \quad &\equiv \\ \phi[s] \wedge Trans(\delta_1, s, \delta', s') \quad &\vee \\ \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s') \end{aligned}$$

9. Synchronized while:

$$\begin{aligned} Trans(\textbf{while } \phi \textbf{ do } \delta,\ s, \delta', s') \quad &\equiv \\ \exists \gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s') \end{aligned}$$

10. Concurrent execution:

$$\begin{aligned} Trans(\delta_1 \parallel \delta_2, s, \delta', s') \quad &\equiv \\ \exists \gamma.\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \quad &\vee \\ \exists \gamma.\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s') \end{aligned}$$

11. Prioritized concurrent execution:

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

12. Concurrent iteration:

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

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. Observe that in $Trans(a, s, \delta', s')$, a stands for the program term encoding the corresponding situation calculus action, while $Poss$ and do take the latter as argument; we take the function $\cdot[\cdot]$ as mapping the program term a into the corresponding situation calculus action $a[s]$, as well as replacing now by the situation s.

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

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

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

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

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

8. (**if** ϕ **then** δ_1 **else**δ_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.

9. (**while** ϕ **do** δ, s) can evolve to $(\delta'; \textbf{while } \phi \textbf{ do } \delta, s')$, if $\phi[s]$ holds and (δ, s) can evolve to (δ', s').

10. $(\delta_1 \parallel \delta_2, s)$ can evolve to $(\delta_1' \parallel \delta_2, s')$, provided that (δ_1, s) can evolve to (δ_1', s'). Similarly, $(\delta_1 \parallel \delta_2, s)$ can evolve to $(\delta_1 \parallel \delta_2', s')$, provided that (δ_2, s) can evolve to (δ_2', s'). 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.[8]

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

[8] 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

11. $(\delta_1 \gg \delta_2, s)$ can evolve to $(\delta_1' \parallel \delta_2, s')$, provided that (δ_1, s) can evolve to (δ_1', s'). However, $(\delta_1 \parallel \delta_2, s)$ can evolve to $(\delta_1 \parallel \delta_2', s')$ only *if* (δ_1, s) *is blocked (cannot perform any transition)* and (δ_2, s) can evolve to (δ_2', s'). That is the $\delta_1 \gg \delta_2$ construct is identical to $\delta_1 \parallel \delta_2$, 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.

12. $(\delta^{\parallel}, s)$ can evolve to $(\delta' \parallel \delta^{\parallel}, s')$, provided that (δ, s) can evolve to (δ', s'). That is 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.

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

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

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

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

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

4. Sequence:
$$Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

5. Nondeterministic branch:
$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

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

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.

7. Iteration:

$$Final(\delta^*, s) \quad \equiv \quad True$$

8. Synchronized if-then-else:

$$Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s) \quad \equiv$$
$$\phi[s] \wedge Final(\delta_1, s) \quad \vee \quad \neg\phi[s] \wedge Final(\delta_2, s)$$

9. Synchronized while:

$$Final(\textbf{while } \phi \textbf{ do } \delta, s) \quad \equiv$$
$$\neg\phi[s] \quad \vee \quad Final(\delta, s)$$

10. Concurrent execution:

$$Final(\delta_1 \parallel \delta_2, s) \quad \equiv$$
$$Final(\delta_1, s) \wedge Final(\delta_2, s)$$

11. Prioritized concurrent execution:

$$Final(\delta_1 \gg \delta_2, s) \quad \equiv$$
$$Final(\delta_1, s) \wedge Final(\delta_2, s)$$

12. Concurrent iteration:

$$Final(\delta^{\parallel}, s) \quad \equiv \quad True$$

The assertions above can be read as follows:

1. (nil, s) is a final configuration.
2. (a, s) is not final, indeed the program consisting of the primitive action a cannot be considered completed until it has performed a.
3. $(\phi?, s)$ is not final, indeed the program consisting of the test action $\phi?$ cannot be considered completed until it has performed the test $\phi?$.
4. $(\delta_1; \delta_2, s)$ can be considered completed if both (δ_1, s) and (δ_2, s) are final.
5. $(\delta_1 | \delta_2, s)$ can be considered completed if either (δ_1, s) or (δ_2, s) is final.
6. $(\pi v.\delta, s)$ can be considered completed, provided that there exists an x such that (δ^v_x, s) is final, where δ^v_x is obtained from δ by substituting v with x.
7. (δ^*, s) is a final configuration, since by δ^* is allowed to execute zero times.

8. (**if** ϕ **then** δ_1 **else** δ_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.

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

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

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

12. $(\delta^{\parallel}, s)$ is a final configuration, since it is legal to execute the δ in $\delta^{\parallel}$ zero times.

In the following, we will denote by $\mathcal{C}$ the set of axioms for *Trans* and *Final* plus the axioms needed for encoding programs as first-order terms (see Appendix A).

4.3 Interrupts

Observe that we didn't define *Trans* and *Final* for interrupts. Indeed, these can be explained using other constructs of *ConGolog*:

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

To see how this works, first assume that the special fluent $Interrupts\_running$ 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 *stop_interrupts*, whose only effect is to make $Interrupts\_running$ false. Thus, we imagine that to execute a program δ containing interrupts, we would actually execute the program $\{start\_interrupts\,;\, (\delta \rangle\rangle stop\_interrupts)\}$ which has the effect of stopping all blocked interrupt loops in δ at the lowest priority, *i.e.* when there are no more actions in δ that can be executed.

4.4 Exogenous events

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 model the occurrence of exogenous events by means of a special program:

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

Executing this program involves performing zero, one, or more nondeterministically chosen exogenous events.[9] Then we make δ_{EXO} run concurrently with the user-specified program δ:

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

In this way we allow exogenous actions whose preconditions are satisfied to asynchronously occur (outside the control of δ) during the execution of δ. See the companion paper [5] for examples.

4.5 *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:

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

Using *Trans** and *Final* we can give a new definition of *Do* as:[10]

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

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 [15].

Formally, we can state the the following result:

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

[10] Equivalently we can define *Do* directly as:

$$Do(\delta, s, s') \stackrel{def}{=} \forall D.[\ldots \supset D(\delta, s, s')]$$

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

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

Theorem 1: *Let Do_1 be the original definition of Do in [15], presented in Section 2, 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. □

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:

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

4.6 Formal properties

We are going to show that the axioms for *Trans* and *Final* 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:*

$$\mathcal{C} \quad \models \quad \forall \vec{x}, s, \delta', s'.\ Trans(\delta(\vec{x}), s, \delta', s') \ \equiv \ \Phi(\vec{x}, s, \delta', s') \tag{1}$$
$$\mathcal{C} \quad \models \quad \forall \vec{x}, s.\ Final(\delta(\vec{x}), s) \ \equiv \ \Psi(\vec{x}, s) \tag{2}$$

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:

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

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. □

5 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 $\{$**proc** $P_1()$ a **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 already been contextualized: the environment in which the definition of P is to be looked for is Env.

We define the semantics of *ConGolog* programs with procedures by defining both *Trans* and *Final* by a second-order formula (instead of a set of axioms).[11] *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:

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

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 $\delta_{P\vec{t}[s]}^{\vec{v}_P}$ denotes the body of the procedure P in Env with formal parameter $\vec{v}$ substituted by the actual parameters $\vec{t}$ evaluated in the current situation.

Similarly, *Final* is defined as follows:

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

where $\ldots$ 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:

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

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 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

[11] 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.

the formal parameters in the procedure bodies[12], thus yielding *call-by-value* parameter passing.

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

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

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):

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

To see this consider the following. Let

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

$$
Env_2 \stackrel{def}{=}\ \textbf{proc}\ P_1()\ b\ \textbf{end};
$$

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

Then it is easy to see that:

$$
\begin{aligned}
&\mathit{Trans}(\delta_{StSc}, s, \delta', s') \\
&\quad \equiv \mathit{Trans}(\{Env_1; P_3()\}, s, \delta', s') \\
&\quad \equiv \mathit{Trans}([Env_1 : P_3()], s, \delta', s') \\
&\quad \equiv \mathit{Trans}(\{Env_1; \{Env_2; P_2(); P_1()\}\}, s, \delta', s') \\
&\quad \equiv \mathit{Trans}(\{Env_2; [Env_1 : P_2()]; P_1()\}, s, \delta', s') \\
&\quad \equiv \mathit{Trans}([Env_1 : P_2()]; [Env_2 : P_1()], s, \delta', s') \\
&\quad \equiv \mathit{Trans}(\{Env_1; P_1()\}; [Env_2 : P_1()], s, \delta', s') \\
&\quad \equiv \mathit{Trans}([Env_1 : P_1()]; [Env_2 : P_1()], s, \delta', s') \\
&\quad \equiv \mathit{Trans}(a; [Env_2 : P_1()], s, \delta', s') \\
&\quad \equiv \mathit{Poss}(a, s) \wedge s' = do(a, s) \wedge \delta' = (nil; [Env_2 : P_1()]).
\end{aligned}
$$

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

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

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

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.

5.1 Formal properties

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

$$\begin{aligned}
&Trans(\delta, s, \delta', s') \quad \equiv \\
&\quad \forall T.[\forall \delta_1, s_1, \delta_2, s_2.\, \Phi_{Trans}(T, \delta_1, s_1, \delta_2, s_2) \equiv T(\delta_1, s_1, \delta_2, s_2)] \\
&\qquad \supset \; T(\delta, s, \delta', s')
\end{aligned}$$

$$\begin{aligned}
&Final(\delta, s, \delta', s') \quad \equiv \\
&\quad \forall F.[\forall \delta_1, s_1.\, \Phi_{Final}(F, \delta_1, s_1) \equiv F(\delta_1, s_1)] \\
&\qquad \supset \; F(\delta, s)
\end{aligned}$$

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, 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 [22]). 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.:

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

Formally we can state the following theorem:

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

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

$$\begin{aligned}
&\forall P.[\forall \delta_1, s_1.\, \Phi_{Final}(P, \delta_1, s_1) \equiv P(\delta_1, s_1)] \;\supset \\
&\quad \forall \delta, s.\, Final(\delta, s, \delta', s') \;\supset\; P(\delta, s)
\end{aligned}$$

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

By definition we have:

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

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

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

So moving the quantifiers around we get:

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

and hence the thesis. □

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*. □

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 [25]. □

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

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

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.

6 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.

6.1 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 $Rank(n, \delta, s)$ iff:

$$Rank(n, nil, s) \equiv True$$

[13] In fact syntactically monotonic.

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

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

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

6.2 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]

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

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

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:

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

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:

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

6.3 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 [15] satisfy such hypothesis, except for the procedure d at page 9 whose definition is reported below (n is a natural number):

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

However, the variants

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

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

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

do satisfy the hypothesis.

7 Conclusion

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 programming language ends 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*.

With all of this procedural richness (nondeterminism, concurrency, recursive procedures, priorities), however, it is important not to lose sight of the basic 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 its *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" supporting concurrency.

Standard semantic accounts of programming languages 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 or what their effects will be. Standard semantic accounts do not attempt to characterize dynamic properties of the external environment in which a program is executed [8].

As mentioned in the introduction, an important motivation for the development of *ConGolog* is the need for "reactive" intelligent agent programs. In the companion paper [5], a comparison is made between *ConGolog* and agent architectures with similar goals such as IRMA [2] and PRS [20], as well as related agent programming languages such as AGENT-0 [23], Concurrent MetateM [9], and 3APL [12].

A prototype implementation of *ConGolog* in Prolog has been developed. Indeed, a simple if somewhat inefficient interpreter can be lifted directly from the axioms for *Final*, *Trans*, and *Do* introduced above.[15] For example, for $(\delta_1 \rangle\!\rangle\ \delta_2)$, we would have the following two Prolog clauses for *Trans*:

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

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

This implementation requires that the program's precondition axioms, successor state axioms, and axioms about the initial state be expressible as Prolog clauses. This is a limitation of the implementation, not the theory. Further details on the implementation as well as on a number of *ConGolog* applications can be found in the companion paper [5].

From a more theoretical point of view, there remain, however, many areas for future research. Among them, we mention: 1) incorporating sensing actions, that is, actions whose effect is not to change the world so much as to provide information to be used by the agent at runtime; 2) handling non-termination, that is, developing accounts of program correctness (fairness, liveness *etc.*) appropriate for controllers expected to operate indefinitely; 3) incorporating utilities, so that nondeterministic choices in execution can be made to maximize the expected benefit. Regarding (1), in [7], we adapt the transition semantics developed in this paper so that execution can be interleaved with program interpretation in order to accommodate sensing actions.

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

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

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

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

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

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

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

[19] 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

[20] 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.

[21] 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.

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

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

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

[25] 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 [14] 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 $\mathtt{succ} : \mathit{Idx} \rightarrow \mathit{Idx}$. 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 $\ldots$ 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 $\mathtt{nameOf}_{\mathit{Sort}} : \mathit{Sort} \rightarrow \mathit{PseudoSort}$ for $\mathit{Sort} = \mathit{Obj}, \mathit{Act}$. We use the notation $[\![x]\!]$ to denote $\mathtt{nameOf}_{\mathit{Sort}}(x)$, leaving *Sort* implicit.
- A function $\mathtt{var}_{\mathit{Sort}} : \mathit{Idx} \rightarrow \mathit{PseudoSort}$ 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 $\mathtt{z}_i$ (or just $\mathtt{x}, \mathtt{y}, \mathtt{z}$) to denote $\mathtt{var}_{\mathit{Sort}}(i)$, leaving *Sort* implicit.

- A function $\mathtt{f} : \mathit{PseudoSort}_1 \times \ldots \times \mathit{PseudoSort}_n \rightarrow \mathit{PseudoSort}_{n+1}$ for each fluent or nonfluent function f of sort $\mathit{Sort}_1 \times \ldots \times \mathit{Sort}_n \rightarrow \mathit{Sort}_{n+1}$ 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:

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

where $\ldots$ stands for the universal closure of

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

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

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

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

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

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[16]

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

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

[16] 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 $\mathtt{decode}(x, s)$. Such a function is characterized by the following assertions:

$$\mathtt{decode}(Now, s) = s$$
$$\mathtt{decode}(\mathtt{nameOf}(x), s) = x$$
$$\mathtt{decode}(\mathtt{f}(x_1 \ldots, x_n), s) = f(\mathtt{decode}(x_1, s), \ldots, \mathtt{decode}(x_n, s)) \quad \text{(for each } \mathtt{f})$$

A.3 Sort *PseudoForm*

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

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

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

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

where $\ldots$ stands for the universal closure of

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

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

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

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

$$g(x_1, \ldots, x_n) \neq g'(y_1, \ldots, y_m)$$
$$g(x_1, \ldots, x_n) = g(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \ldots \wedge x_n = y_n$$

Next we formalize the notion of substitution. We introduce the function $\mathtt{sub} : PseudoSort \times PseudoSort \times PseudoSort' \rightarrow PseudoSort'$ 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:

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

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

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

where y in the last equation is any variable that does not appear in ρ.

A.4 Sorts *PROG* and *ENV*

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

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

- A function $\texttt{choice} : PROG \times PROG \to PROG$. We use the notation $\delta_1 \mid \delta_2$ to denote $\texttt{choice}(\delta_1, \delta_2)$.
- A function $\texttt{iter} : PROG \to PROG$. We use the notation δ^* to denote $\texttt{iter}(\delta)$.
- Two functions $\texttt{pick}_{Sort} : PseudoSort \times PROG \to PROG$, where $PseudoSort$ is either $PseudoObj$ or $PseudoAct$. We use the notation $\pi \mathsf{z}_i.\delta$ to denote $\texttt{pick}_{Sort}(\texttt{var}_{Sort}(i), \delta)$, leaving $Sort$ implicit.
- A function $\texttt{if} : PseudoForm \times PROG \times PROG \to PROG$. We use the notation $\texttt{if}\ \rho\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2$ to denote $\texttt{if}(\rho, \delta_1, \delta_2)$.
- A function $\texttt{while} : PseudoForm \times PROG \to PROG$. We use the notation $\texttt{while}\ \rho\ \texttt{do}\ \delta$ to denote $\texttt{while}(\rho, \delta)$.
- A function $\texttt{conc} : PROG \times PROG \to PROG$. We use the notation $\delta_1 \parallel \delta_2$ to denote $\texttt{conc}(\delta_1, \delta_2)$.
- A function $\texttt{prconc} : PROG \times PROG \to PROG$. We use the notation $\delta_1 \gg \delta_2$ to denote $\texttt{prconc}(\delta_1, \delta_2)$.
- A function $\texttt{iterconc} : PROG \to PROG$. We use the notation $\delta^{\parallel}$ to denote $\texttt{iterconc}(\delta)$.

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

- A finite number of functions $\mathtt{P} : PseudoSort_1 \times \ldots \times PseudoSort_n \to PROG$, where $PseudoSort_i$ is either $PseudoObj$ or $PseudoAct$. These functions are going to be used as procedure calls.
- A function $\texttt{proc} : PROG \times PROG \to PROG$. 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 $\texttt{proc}\ \mathtt{P}(\mathsf{z}_1, \ldots, \mathsf{z}_n)\ \delta\ \texttt{end}$ to denote $\texttt{proc}(\mathtt{P}(\mathsf{z}_1, \ldots, \mathsf{z}_n), \delta)$.
- A constant $\varepsilon : ENV$, denoting the *empty environment*.
- A function $\texttt{addproc} : ENV \times PROG \to ENV$. We will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \texttt{proc}\ \mathtt{P}(\vec{\mathsf{z}})\ \delta\ \texttt{end}$ to denote $\texttt{addproc}(\mathcal{E}, \texttt{proc}\ \mathtt{P}(\vec{\mathsf{z}})\ \delta\ \texttt{end})$.
- A function $\texttt{pblock} : ENV \times PROG \to PROG$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\texttt{pblock}(\mathcal{E}, \delta)$.
- A function $\texttt{c\_call} : ENV \times PROG \to PROG$. 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 $\texttt{c\_call}(\mathcal{E}, \mathtt{P}(\vec{t}))$.

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

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

where $\ldots$ stands for

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

Observe that procedures $\texttt{P}$ 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:

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

where $\ldots$ stands for the universal closure of

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

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

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 $\mathtt{P}$ 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 \gg \delta_2)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} & = & (\delta_1)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]} \gg (\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})]} & = & \begin{cases} \{\mathcal{E}';\delta\} & \text{if procedure } \mathtt{P} \text{ is (re)defined in } \mathcal{E}' \\ \{\mathcal{E}';(\delta)^{\mathtt{P}(\vec{x})}_{[\mathcal{E}:\mathtt{P}(\vec{x})]}\} & \text{otherwise} \end{cases} \\
([\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 $\mathtt{P}$ 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 8: *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. □

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:

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

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

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

Theorem 9: *With respect to* Golog *programs, the definitions above are* equivalent *to the ones given in Section 5 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 [25]. If

$$S(\vec{x}) \equiv \forall Z.[[\forall \vec{y}.\Phi(Z,\vec{y}) \supset Z(\vec{y})] \supset Z(\vec{x})] \tag{3}$$

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[17]

$$S(\vec{x}) \equiv \Phi(S,\vec{x}) \tag{4}$$

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

Now it is easy to see that the above definition of *Trans* and *Final* can be rewritten as (3) 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 (5). Once in this form it is easy to see that for *Golog* programs they are equivalent to those introduced in Section 5. □

B.2 Do_1 is equivalent to Do_2

Let Do_1 be the original definition of *Do* in [15] 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 2: *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 3: *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.

[17] In fact, (4) 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

$$\Phi = [\bigwedge_{i=1}^{n} \forall \vec{x}, s, s'.Do_1(\delta_i^{\vec{v}_i}_{\vec{x}}, s, s') \supset P_i(\vec{x}, s, s')]. \tag{6}$$

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

$$M \models \forall P_1, \ldots, P_n.\ [\Phi \supset P_i(\vec{x}, s, s')] \tag{7}$$

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 $\delta_1 = \{Env; P_i(\vec{x})\}$[18], $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n, s_n)$ 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 (6).

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 4: *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 5: *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.

[18] To be more precise, the variables x_i in $P_i(\vec{x})$ should be read as `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 4, 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:

$$M \models \forall P_1, \ldots, P_n.\ [\Psi \supset Do_1(\delta_i{}^{\vec{v_i}}_{\vec{t}[s]}, s, s')] \quad (8)$$

$$M \models \exists P_1, \ldots, P_n.\ [\Psi \wedge \neg P_i(\vec{t}[s]), s, s')]. \quad (9)$$

where $\Psi = [\bigwedge_{i=1}^{n} \forall \vec{x}_i, s, s'. Do_1(\delta_{i\,\vec{x}_i}^{\vec{v}_i}, s, s') \supset P_i(\vec{x}_i, s, s')]$. Then by (9) 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 $M, \sigma \models \neg Do_1(\delta_{\vec{t}[s]}^{\vec{v}_i}, s, s')$, which contradicts (8).

6. Implication for programs within an environment.

   We have to show

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

   It suffices to prove that:

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

   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 6: *For every model M of $\mathcal{C}$, if there exist $\delta_1, s_1 \ldots \delta_n, s_n$ such that $\delta_1 = \delta$, $s_1 = s$, $s_n = s'$, $M \models Final(\delta_n, s_n)$ 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 4. If $n > 1$, then by induction hypothesis $M \models Do_1(\delta_2, s_2, s')$, hence by applying Lemma 5, 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 3 and Lemma 2; $\Leftarrow$ by Lemma 2 and Lemma 6. □