Non-Terminating Processes in the Situation Calculus

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
degiacomo@dis.uniroma1.it

Eugenia Ternovskaia
Department of Computer Science
University of Toronto
eugenia@cs.toronto.edu

Ray Reiter
Department of Computer Science
University of Toronto
reiter@cs.toronto.edu

1 Introduction

By their very design, many robot control programs are non-terminating. To give a simple example – one we shall use in this paper – an office coffee-delivery robot might be implemented as an infinite loop in which the robot responds to exogenous requests for coffee that are maintained on a queue. Since a future coffee request is always possible, the program never terminates.

As is the case for more conventional programs, we want some reliability assurances for robot controllers. This paper describes the approach being taken by our Cognitive Robotics Group to expressing and proving properties of non-terminating programs expressed in GOLOG, a high level logic programming language for modeling and implementing dynamical systems. The kinds of properties we have in mind are traditional in computer science: liveness, fairness, etc. We differ from the "classical" approaches ([LS87, Cou90, MP95]) for reasons dictated by the following characteristics of GOLOG:

1. To write a GOLOG program, the programmer first axiomatizes the primitive actions of the application domain, using first order logic. These actions may also include exogenous events.
2. Next, she describes, in GOLOG, the complex behaviors her robot is to exhibit in this domain. This GOLOG program is interpreted by means of a formula, this time in second order logic.
3. Finally, a suitable theorem-prover executes the program.

Because these features are all represented in classical (second order) logic, it is natural to express and prove properties of GOLOG programs, including non-terminating ones, in the very same logic. This approach to program proofs has the advantage of logical uniformity and the availability of classical proof theory. It also provides a very rich language with which to express program properties, as we shall see in this paper. Moreover, it provides for proofs of programs with incomplete initial state, the normal situation in robotics where the agent does not have complete information about the world it inhabits. Finally, this approach gracefully accommodates exogenous event occurrences, and proofs of program properties in their presence.

2 Formal Preliminaries

2.1 The Situation Calculus

The situation calculus is a second order language specifically designed for representing dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the *initial situation*, namely the empty history. There is a distinguished binary function symbol do; $do(\alpha, s)$ denotes the successor situation to s resulting from performing the action α. Actions may be parameterized. For example, $put(x, y)$ might stand for the action of putting object x on object y, in which case $do(put(A, B), s)$ denotes that situation resulting from placing A on B when the history is s. Notice that in the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, $do(putdown(A), do(walk(L), do(pickup(A), S_0)))$

is a situation denoting the world history consisting of the sequence of actions [pickup(A), walk(L), put-down(A)]. Notice that the sequence of actions in a history, in the order in which they occur, is obtained from a situation term by reading off the actions from right to left. The situation calculus has a distinguished predicate symbol *Poss*; the intended meaning of $Poss(a, s)$ is that it is possible to perform the action a in situation s.

Relations (functions) whose truth values (function values) vary from situation to situation are called *relational (functional) fluents*. They are denoted by predicate (function) symbols taking a situation term as their last argument. For example, $hasCoffee(p, s)$ is a relational fluent whose intended meaning is that person p has coffee in situation s; $robotLocation(s)$ is a functional fluent denoting the robot's location in situation s.

When formalizing an application domain, one must specify certain axioms:

- *Action precondition axioms*, one for each primitive action. These characterize the relation *Poss*, and give the preconditions for the performance of an action in a situation. In a robot coffee delivery setting, such an axiom might be:

  $$\begin{aligned}&Poss(giveCoffee(person), s) \equiv \\ &\quad holdingCoffee(s) \wedge robotLocation(s) = office(person)\end{aligned}$$

  This says that the preconditions for the robot to give coffee to person p are that the robot is carrying coffee, and the robot's location is p's office.

- *Successor state axioms*, one for each fluent. These capture the causal laws of the domain, together with a solution to the frame problem [Rei91]. For our coffee delivery robot, the following is an example:

  $$\begin{aligned}&Poss(a, s) \supset [holdingCoffee(do(a, s)) \equiv \\ &\quad a = pickupCoffee \vee \\ &\quad holdingCoffee(s) \wedge \neg(\exists person) a = giveCoffee(person)].\end{aligned}$$

  In other words, provided the action a is possible, the robot will be holding a cup of coffee after action a is performed iff a is the action of the robot picking up the coffee, or the robot is already holding coffee and a is not the action of the robot giving that coffee to someone.

- Unique names axioms for the primitive actions, stating that different names for actions denote different actions.

- Axioms describing the initial situation – what is true initially, before any actions have occurred. This is any finite set of sentences which mention no situation term, or only the situation term S_0. Examples of axioms for the initial situation for our coffee delivery example are:

  $\neg(\exists p) hasCoffee(p, S_0)$, $robotLocation(S_0) = CM$.

  These have the intended reading that initially, no one has coffee, and the robot is located at the coffee machine (CM).

See [LRL$^+$97] for a full description.

2.2 GOLOG

GOLOG [LRL$^+$97] is a situation calculus-based logic programming language that allows for defining complex actions using a repertoire of user specified primitive actions. GOLOG provides the usual kinds of imperative programming language control structures as well as various forms of nondeterminism. Briefly, *GOLOG* programs are formed by using the following constructs:

1. *Primitive actions:* a. Do action a in the current situation. Actually a is a pseudo-action obtained from an action by suppressing the situation argument in each functional fluent. The function $a[s]$ that given a pseudo-action a and a situation s returns the original action (see [LRL$^+$97]).

2. *Test actions:* $\phi?$. Test the truth value of expression ϕ in the current situation. As for primitive actions, ϕ is a pseudo-formula obtained from a situation calculus formula by suppressing all situation arguments. The function $\phi[s]$ that given a pseudo-formula ϕ and a situation s returns the original formula.

3. *Sequence:* $\delta_1; \delta_2$. Execute program δ_1, followed by program δ_2.

4. *Nondeterministic action choice:* $\delta_1 \mid \delta_2$. Execute δ_1 or δ_2.

5. *Nondeterministic choice of arguments:* $(\pi z)\delta$. Nondeterministically pick a value for z, and for that value of z, execute program δ.

6. *Nondeterministic repetition:* δ^*. Execute δ a nondeterministic number of times.

7. *While loops:* **while** ϕ **do** δ **endWhile**, which is expressed as $(\phi?;\delta)^*;\neg\phi?)$.

8. *Conditionals:* **if** ϕ **then** δ_1 **else** δ_2, which is expressed as $(\phi?;\delta_1) \mid (\neg\phi?;\delta_2)$.

9. *Procedures*, including recursion: **proc** $ProcName(\vec{v})$ $\delta_{ProcName}$ **endProc**.

3 Single step semantics for GOLOG

In [LRL+97], GOLOG programs are interpreted by means of a special relation $Do(\delta, s, s')$ that given a (generally nondeterministic) program δ and a situation s returns a possible situation s', resulting by executing δ starting from s. Actually in [LRL+97] the relation Do is not denoted by a predicate, but instead it is defined implicitly by using *macros expansion rules* such as:

$$\begin{array}{l}
Do(\delta_1;\delta_2,s,s') \overset{def}{=} (\exists s'')Do(\delta_1,s,s'') \wedge Do(\delta_2,s'',s') \\
Do(\delta_1 \mid \delta_2,s,s') \overset{def}{=} Do(\delta_1,s,s') \vee Do(\delta_2,s,s') \\
Do(\delta^*,s,s') \overset{def}{=} (\forall P)[\ldots \supset P(s,s')] \\
\text{where} \ldots \text{ stands for the conjunction of:} \\
\quad (\forall s,s)P(s,s) \\
\quad (\forall s,s'',s')P(s,s'') \wedge Do(\delta,s'',s') \supset P(s,s')
\end{array}$$

one for each construct in the language. By using such macro expansions rules the relation $Do(\delta, s, s')$ for the particular program δ is defined by a (generally second order) formula $\Phi_\delta(s, s')$ not mentioning δ at all. This is very convenient, since it completely avoids the introduction of programs into the language (they are used only during the macro expansion process to get the formulas $\Phi_\delta(s, s')$ corresponding to $Do(\delta, s, s')$). Observe however that in this way programs cannot be quantified over, because they are not terms of the language of the situation calculus.

The kind of semantics Do associates to programs, which is based on the complete evaluation of the program, is sometimes called *evaluation semantics* [Hen90]. Such a semantics is not well suited to interpret non-terminating programs, like infinite loops, since for such programs the evaluation can never be completed and a final situation can never be reached.

For non-terminating programs one needs to rely on a semantics that allows for interpreting *segments of program executions.* So we adopt a kind of semantics called *computational semantics* [Hen90], which is based on "single steps" of computation, or *transitions*[1]. A step here is either a primitive or a test action. We begin by introducing two special relations, *Final* and *Trans*. *Final*(δ) is intended to say that program δ is in a final state, i.e. it may legally terminate in the current situation. *Trans*(δ, s, δ', s') is intended to say that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining.

To follow this approach it is necessary to quantify over programs and so, unlike in [LRL+97], we need to encode *GOLOG* programs as first-order terms, including introducing constants denoting variables, and so on. This is laborious but quite straightforward [Lei94][2]. We omit all such details here and simply use programs within formulas as if they were already first-order terms.

Final and *Trans* are denoted by predicates defined inductively on the structure of the first argument. It is convenient to include a special "empty" program ε, denoting that nothing of the program remains to be performed.

[1]Both types of semantics belong to the family of structural operational semantics introduced in [Plo81].

[2]We assume that the predicates introduced in this section, including *Final* and *Trans*, cannot occur in tests, hence disallowing self-reference.

The definition of *Final* is as follows:

$(\forall\delta) Final(\delta) \equiv (\forall F)[\ldots \supset F(\delta)]$
where $\ldots$ stands for the conjunction of the universal closure of the following clauses:

$$F(\varepsilon)$$
$$F(\delta_1) \wedge F(\delta_2) \supset F(\delta_1;\delta_2)$$
$$F(\delta_1) \vee F(\delta_2) \supset F(\delta_1 \mid \delta_2)$$
$$F(\delta) \supset F((\pi z)\delta))$$
$$F(\delta^*)$$
$$F(\delta_{ProcName}) \supset F(ProcName(\vec{x}))$$

Observe that being final is a syntactic property of programs: programs of a certain form are considered to be in a final state. Moreover being final does not depend on the objects the program deals with, indeed $Final((\pi z)\delta)$ and $Final(ProcName(\vec{x}))$ depend only on δ and $\delta_{ProcName}$ and not on the particular values of z and $\vec{x}$ respectively. Observe that from the above definition we get that primitive and test actions are never final: for all actions a $Final(a) \equiv$ **False** and for all tests $\phi?$ $Final(\phi?) \equiv$ **False**.

The definition of *Trans* is as follows:[3]

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

$$Poss(a[s], s) \supset T(a, s, \varepsilon, do(a[s], s))$$

$$\phi[s] \supset T(\phi?, s, \varepsilon, s)$$

$$T(\delta_1, s, \delta_1', s') \supset T(\delta_1;\delta_2, s, \delta_1';\delta_2, s')$$
$$Final(\delta_1) \wedge T(\delta_2, s, \delta_2', s') \supset T(\delta_1;\delta_2, s, \delta_2', s')$$

$$T(\delta_1, s, \delta_1', s') \supset T(\delta_1 \mid \delta_2, s, \delta_1', s')$$
$$T(\delta_2, s, \delta_2', s') \supset T(\delta_1 \mid \delta_2, s, \delta_2', s')$$

$$(\exists y) T(\delta^z_y, s, \delta', s') \supset T((\pi z)\delta(x), s, \delta', s')$$

$$T(\delta, s, \delta', s') \supset T(\delta^*, s, \delta';\delta^*, s')$$

$$T((\delta_{ProcName})^{\vec{v}}_{\vec{x}}, s, \delta', s') \supset T(ProcName(\vec{x}), s, \delta', s')$$

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

- (a, s) evolves to $(\varepsilon, do(\alpha[s], s))$, provided $a[s]$ is possible in s. Observe that after having performed a, nothing remains to be performed.
- $(\phi?, s)$ evolves to (ε, s), provided that $\phi[s]$ holds. Otherwise it cannot proceed. Observe that in any case the situation remains unchanged.
- $(\delta_1;\delta_2, s)$ can evolve to $(\delta_1';\delta_2, s')$, provided that (δ_1, s) can evolve to (δ_1', s'). Moreover it can evolve to (δ_2', s'), provided that δ_1 is final and (δ_2, s) can evolve to (δ_2', s').
- $(\delta_1 \mid \delta_2, s)$ can evolve to (δ', s'), provided that either (δ_1, s) or (δ_2, s) can do so.
- $((\pi z)\delta, s)$ can evolve to (δ', s'), provided that there exists a y such that (δ^z_y, s) can evolve to (δ', s') – z is bound by π in $(\pi z)\delta$ and is typically free in δ.
- (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, since δ^* is final.
- $(ProcName(\vec{x}), s)$ can evolve to (δ', s'), provided that the body $\delta_{ProcName}$ of the procedure $ProcName$, with the actual parameters $\vec{x}$ substituted for the formal parameters $\vec{v}$, can do so.

The possible configurations that can be reached by a program δ starting in a situation s are those obtained by repeatly following the transition relation denoted by *Trans* starting from (δ, s), i.e. those in

[3]Here, δ^z_y is the usual notion of substitution, in which the nondeterministic choice operator π is treated like a quantifier.

the reflexive transitive closure of the transition relation. Such a relation is denoted by the "reflexive-transitive closure" of *Trans*, *Trans** defined as:

$(\forall \delta, s, \delta', s')\, Trans^*(\delta, s, \delta', s') \equiv \forall U[\ldots \supset U(\delta, s, \delta', s')]$
where $\ldots$ stands for the conjunction of the universal closure of the following clauses:
$U(\delta, s, \delta, s)$
$U(\delta, s, \delta', s') \wedge Trans(\delta', s', \delta'', s'') \supset U(\delta, s, \delta'', s'')$

Using *Trans** and *Final* we may denote the relation Do as follows:

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

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'. Note that this formulation of Do is equivalent to the one in [LRL$^+$97] (c.f. [Hen90]).

4 Exogenous actions

Exogenous action are primitive actions that are not under the control of the program. They are executed by other agents in an asynchronous way wrt the program. *Trans* can be easily modified to take into account exogenous actions as well. It suffice to add to the above definition a clause having, as a first approximation, the form:

$$Exo(exo) \wedge Poss(exo, s) \supset T(\delta, s, \delta, do(exo, s))$$

which says that any configuration (δ, s) can evolve, due to the occurrence of an exogenous action *exo*, to $(\delta, do(exo, s))$, where the situation has changed but the program hasn't.

The above clause enables the occurrence of an exogenous action *exo* every time the action preconditions for *exo*, and hence $Poss(exo, s)$, are true. However it is of interest, to restrict further the actual occurrence of *exo* along a sequence of transitions, establishing some sort of *dynamics* for exogenous actions. Such a dynamics has a role similar to that of programs for normal primitive actions although typically it is not strict enough to extract a program that implements it. Rather the dynamics of exogenous actions has to be specified by means of suitable axioms.

A possible way to follow such a strategy is to introduce a special fluent $DynaPoss(exo, s)$ and modify *Trans* by introducing the following refinement of the above clause:

$$Exo(exo) \wedge Poss(exo, s) \wedge DynaPoss(exo, s) \supset T(\delta, s, \delta, do(exo, s)).$$

Then one uses special axioms expressing the dynamics of exogenous actions by specifying in which situations s along a sequence of transitions $DynaPoss(exo, s)$ holds. Such axioms may express sophisticated temporal/dynamic laws and typically they are going to be second order. Observe that *exo* can actually occur only if both $Poss(exo, s)$ and $DynaPoss(exo, s)$ hold in s.

5 Logical representation of inductive definitions and fixpoints

The relations *Trans* and *Final* are defined inductively. *Inductive definitions* [Acz77, Mos74] are broadly used in mathematical logic for defining sets. For the past several years they became popular in computer science [CC92]. A *rule-based inductive definition* is a set $\mathcal{R}$ of rules of the form $\frac{P}{c}$, where P is the set of premises and c is the conclusion, together with a closure condition: a set Z is $\mathcal{R}$-closed if each rule in $\mathcal{R}$ whose premises are in Z also has its conclusion in Z. A set H, *inductively defined by* $\mathcal{R}$, is given by $H = \bigcap\{Z \mid Z \text{ is } \mathcal{R}\text{-closed}\}$ or by $H = \bigcup\{Z \mid Z \text{ is } \mathcal{R}\text{-closed}\}$. The former is called a *positive inductive* definition of H, the latter is called a *negative inductive* or *coinductive* definition of H. Let U be a set. An *operator induced by an inductive definition* is a total mapping $\Gamma : Pow(U) \mapsto Pow(U)$, such that

$$\Gamma(Z) = \{c \in U \mid \exists P \subseteq Z : \frac{P}{c} \in \mathcal{R}\}$$

That is, Γ is a mapping taking sets to sets.

Inductive definitions are strongly related to *fixpoint properties* i.e. properties defined as solutions of recursive equations. Specifically, positive inductive definitions are related to least fixpoints. i.e. minimal solution of the recursive equations, whereas negative inductive definitions are related to greatest fixpoints, i.e. maximal solutions of the recursive equations. Dynamic properties are typically fixpoint properties, expressed as the least or greatest solutions of certain recursive logical equations (e.g. see [Sti96]).

Every property definable as an extreme fixpoint must have, by definition:

- its own construction principle, a recursive equation a fixpoint of which is our property;
- an appropriate induction or coinduction principle to guarantee the minimality or maximality of the solution of the recursive equation.

5.1 Construction principle

To define a set Z, here denoted by a predicate $Z(\vec{x})$, we need to say what its elements are. The *construction principle* tells us how to obtain these elements recursively.

$$(\forall \vec{x}) Z(\vec{x}) \equiv \Phi(Z, \vec{x}) \tag{1}$$

In this case Φ is called a *constructor* for Z. Any solution of this recursive equation is called a *fixpoint* of the operator Φ. The Knaster-Tarski Theorem [Kna28, Tar55] guarantees that if the operator Φ is monotone, the equation (1) has both a least and a greatest solution. A sufficient condition for monotonicity is that all occurrence of Z occur within a even number of negations[4]. This condition is always satisfied in this paper.

5.2 Induction principle: Least fixpoints

To guarantee that Z is the smallest solution, we apply the *induction principle:*[5]

$$(\forall P, \vec{x})\{[(\forall \vec{y})\Phi(P, \vec{y}) \supset P(\vec{y})] \supset [Z(\vec{x}) \supset P(\vec{x})]\} \tag{2}$$

i.e., whatever solution P of the recursive specification we take, Z is included in it.

A set Z satisfying construction principle (1) and induction principle (2) is denoted by $\mu_{P,\vec{y}}\Phi(P, \vec{y})(\vec{x})$, and it is called a *least fixpoint* of an operator $\Phi(P, \vec{y})$. Note that in $\mu_{P,\vec{y}}\Phi(P, \vec{y})(\vec{x})$ the predicate variable P and the individual variables $\vec{y}$ are considered bounded by μ, while the individual variables $\vec{x}$ are free. Another view of $\mu_{P,\vec{y}}\Phi(P, \vec{y})(\vec{x})$ is that $\mu_{P,\vec{y}}\Phi(P, \vec{y})$ is the name of a defined predicate, and $\vec{x}$ are its arguments.

We can rewrite the induction principle (2) in the following way

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

Notice that implication in the opposite direction follows from the construction principle (1). We obtain

$$(\forall \vec{x})\{\mu_{P,\vec{y}}\Phi(P, \vec{y})(\vec{x}) \equiv [(\forall P)[(\forall \vec{y})\Phi(P, \vec{y}) \supset P(\vec{y})] \supset P(\vec{x})]\} \tag{4}$$

The last sentence is often considered as a formal definition of a least fixpoint. Observe that it has exactly the form we have used to define *Trans* and *Final* (as well as $Do(\delta^*, s, s')$ in [LRL$^+$97]).

5.3 Coinduction principle: Greatest fixpoints

To guarantee that Z is the biggest solution of (1), we apply the *coinduction principle:*

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

i.e., whatever solution P of the recursive specification we take, Z includes it.

We can rewrite the coinduction principle (5) in the following way

$$(\forall \vec{x})\{[(\exists P)[(\forall \vec{y})P(\vec{y}) \supset \Phi(P, \vec{y})] \wedge P(\vec{x})] \supset Z(\vec{x})\} \tag{6}$$

An explicit expression for a greatest fixpoint can be obtained in a similar way as was done for a least fixpoint:

$$(\forall \vec{x})\{\nu_{P,\vec{y}}\Phi(P, \vec{y})(\vec{x}) \equiv [(\exists P)[(\forall \vec{y})P(\vec{y}) \supset \Phi(P, \vec{y})] \wedge P(\vec{x})]\} \tag{7}$$

The last sentence can be taken as a definition of a greatest fixpoint.

[4]Interpreting $\Phi \supset \Psi$ as an abbreviation for $\neg\Phi \vee \Psi$

[5]The idea of defining a least fixpoint using two principles, construction and induction, is from [Heh93].

6 Examples of expressible dynamic properties

With *Trans* and *Final* in place a wide variety of dynamic properties can be expressed by relying on second order formulae expressing least and greatest fixpoint properties. In particular properties expressible by logics of programs, such as dynamic logics [KT90], mu-calculus [Par70, Sti96], and temporal logics [Eme96], can be rephrased in our setting. Let us present some examples.

1. The formula:

$$Q_1(\delta_0, s_0) \overset{def}{=} \mu_{P,\delta,s}[\psi(\delta, s) \vee (\exists\delta', s')\, \mathit{Trans}(\delta, s, \delta', s') \wedge P(\delta', s')](\delta_0, s_0)$$

   (where δ_0, s_0 are individual variables) defines a predicate $Q_1(\delta_0, s_0)$ that denotes the smallest set of configurations C_1 such that a configuration (δ, s) belongs to this set (the predicate Q_1 is true on (δ, s)) if and only if either ψ is true on (δ, s) or there exists a configuration (δ', s'), reachable in one step by the relation *Trans*, which also belongs to the set C_1.

   In this way the formula expresses that from each configuration (δ_0, s_0) on which the specified predicate is true, there exists an execution path that eventually reaches a configuration (δ, s) on which ψ is true.

   As a special case, by taking $\psi(\delta, s) \overset{def}{=} \phi(s) \wedge \mathit{Final}(\delta)$ one can express that there exists a terminating execution of program δ_0 starting from situation s_0 such that ϕ is true in the final situation.

2. The formula:

$$Q_2(\delta_0, s_0) \overset{def}{=} \mu_{P,\delta,s}\{\psi(\delta, s) \vee [(\exists\delta', s')\, \mathit{Trans}(\delta, s, \delta', s')] \wedge (\forall\delta', s')\, \mathit{Trans}(\delta, s, \delta', s') \supset P(\delta', s')\}(\delta_0, s_0)$$

   defines a predicate $Q_2(\delta_0, s_0)$ that denotes the smallest set of configurations C_2 such that the predicate is true on configuration (δ, s) if and only if either ψ is true on (δ, s) or there exists a configuration (δ', s') reachable in one step by the relation *Trans*, and on all such configurations the predicate is still true.

   In this way the formula expresses that from each configuration (δ_0, s_0) on which the specified predicate is true, all execution paths eventually reach a configuration (δ, s) on which ψ is true.

3. The formula:

$$Q_3(\delta_0, s_0) \overset{def}{=} \nu_{P,\delta,s}[\psi(\delta, s) \wedge (\exists\delta', s')\, \mathit{Trans}(\delta, s, \delta', s') \wedge P(\delta', s')](\delta_0, s_0)$$

   defines a predicate $Q_3(\delta_0, s_0)$ that denotes the greatest set of configurations C_3 such that the predicate is true on configuration (δ, s) if and only if both ψ is true on (δ, s) and the predicate is still true on at least one configuration (δ', s') reachable in one step by the relation *Trans*.

   In this way the formula expresses that from each configuration (δ_0, s_0) on which the specified predicate is true, there exists a non-terminating execution path along which ψ is always true.

   As a special case, by $\psi(\delta, s) \overset{def}{=} \mathbf{True}$, one can express that there exists a non-terminating execution path.

4. The formula:

$$Q_4(\delta_0, s_0) \overset{def}{=} \nu_{P,\delta,s}[\psi(\delta, s) \wedge (\forall\delta', s')\, \mathit{Trans}(\delta, s, \delta', s') \supset P(\delta', s')](\delta_0, s_0)$$

   defines a predicate that denotes the greatest set of configurations C_4 such that the predicate is true on configuration (δ, s) if and only if both ψ is true on (δ, s) and the predicate is still true on each configuration (δ', s') reachable in one step by the relation *Trans*.

   In this way the formula expresses that from each configuration (δ_0, s_0) on which the specified predicate is true, along all execution paths ψ is always true.

   As a special case, by $\psi(\delta, s) \overset{def}{=} \neg\mathit{Final}(\delta) \wedge (\exists\delta', s')\, \mathit{Trans}(\delta, s, \delta', s')$, one can express that all execution paths are non-terminating and no final state is ever reached.

7 Example: A Coffee Delivery Robot

Here, we describe a robot whose task is to deliver coffee in an office environment. The robot can carry just one cup of coffee at a time, and there is a central coffee machine from which it gets the coffee. The robot receives *asynchronous* requests for coffee from employees. These requests are put in a queue. The robot continuously takes the first request from the queue and serves coffee to the specified person. The use of the queue guarantees that all requests will in fact be served (implementing a *fair* serving policy).

7.1 Representation of the queue

As usual, to define an abstract data type we need to specify the *domain of its values*, and its *functions and predicates.*

The domain of values for queues is constructed inductively from the constant nil and the functor $cons(\cdot,\cdot)$ as follows:[6]

$$\begin{array}{l}(\forall q) IsQueue(q) \equiv (\forall Q)[\ldots \supset Q(q)] \\ \text{where } \ldots \text{ stands for the conjunction of:} \\ \quad Q(nil) \\ \quad (\forall f, r) Q(r) \supset Q(cons(f, r))\end{array}$$

The functions and predicates for queues are the usual $first(\cdot)$, $dequeue(\cdot)$, $enqueue(\cdot,\cdot)$ and $isEmpty(\cdot)$. They are defined in our setting as follows:

$$(\forall f, r) first(cons(f, r)) = f \qquad \texttt{(unspecified for } nil\texttt{)}$$

$$(\forall f, r) dequeue(cons(f, r)) = r \qquad \texttt{(unspecified for } nil\texttt{)}$$

$$\begin{array}{l}(\forall p) enqueue(nil, p) = cons(p, nil) \\ (\forall p, f, r) enqueue(cons(f, r), p) = cons(f, enqueue(r, p))\end{array}$$

$$(\forall q) isEmpty(q) \equiv (q = nil)$$

To these we add the function $length(\cdot)$ that returns the length of the queue, and the predicate $isFull(\cdot)$ since we are going to need queues of a bounded length.

$$\begin{array}{l}length(nil) = 0 \\ (\forall f, r) length(cons(f, r)) = 1 + length(r)\end{array}$$

$$(\forall q) isFull(q) \equiv (length(q) = 100)$$

We enforce unique name assumption for terms built from nil and $cons(\cdot,\cdot)$, but obviously not for those built with the functions $dequeue(\cdot)$, $enqueue(\cdot,\cdot)$ and $length(\cdot)$.

7.2 Formalization of the Example

Primitive Actions:

- $requestCoffee(person)$. A request for coffee is received from the employee $person$. This action is an *exogenous* one, i.e. an action not under the control of the robot. $(\forall p) Exo(requestCoffee(p))$ holds.
- $selectRequest(person)$. The first request in the queue is selected, and the employee $person$ that made that request will be served.
- $pickupCoffee$. The robot picks up a cup of coffee from the coffee machine.
- $giveCoffee(person)$. The robot gives a cup of coffee to $person$.
- $startGo(loc_1, loc_2)$. The robot starts to go from location loc_1 to loc_2.
- $endGo(loc_1, loc_2)$. The robot ends its process of going from location loc_1 to loc_2.

[6] Equivalently, $(\forall q_0) IsQueue(q_0) \equiv \mu_{Q,q}[q = nil \vee (\exists f, r) q = cons(f, r) \wedge Q(r)](q_0)$.

Fluents:

- $queue(s)$. A functional fluent denoting the queue of requests in situation s.
- $robotLocation(s)$. A functional fluent denoting the robot's location in situation s.
- $hasCoffee(person, s)$. $person$ has coffee in s.
- $going(loc_1, loc_2, s)$. In situation s, the robot is going from loc_1 to loc_2.
- $holdingCoffee(s)$. In situation s, the robot is holding a cup of coffee.

Situation Independent Predicates and Functions:

- $office(person)$. Denotes the office of $person$.
- CM. Constant denoting coffee machine's location.
- Sue, $Mary$, $Bill$, Joe. Constants denoting people.

Primitive Action Preconditions:

$$Poss(requestCoffee(p), s) \equiv \neg isFull(queue(s))$$

$$Poss(selectRequest(p), s) \equiv \neg isEmpty(queue(s)) \wedge p = first(queue(s))$$

$$Poss(pickupCoffee, s) \equiv \neg holdingCoffee(s) \wedge robotLocation(s) = CM$$

$$Poss(giveCoffee(person), s) \equiv holdingCoffee(s) \wedge robotLocation(s) = office(person)$$

$$Poss(startGo(loc_1, loc_2), s) \equiv \neg(\exists l, l')going(l, l', s) \wedge loc_1 \neq loc_2 \wedge robotLocation(s) = loc_1$$

$$Poss(endGo(loc_1, loc_2), s) \equiv going(loc_1, loc_2, s).$$

Successor State Axioms:

$$\begin{aligned}&Poss(a, s) \supset [queue(do(a, s)) = q \equiv\\ &\quad (\exists p)a = requestCoffee(p) \wedge q = enqueue(queue(s), p) \vee\\ &\quad (\exists p)a = selectRequest(p) \wedge q = dequeue(queue(s), p) \vee\\ &\quad (\forall p)a \neq requestCoffee(p) \wedge a \neq selectRequest(p) \wedge q = queue(s)]\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset [hasCoffee(person, do(a, s)) \equiv\\ &a = giveCoffee(person) \vee hasCoffee(person, s)]\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset [robotLocation(do(a, s)) = loc \equiv\\ &\quad (\exists loc')a = endGo(loc', loc) \vee\\ &\quad robotLocation(s) = loc \wedge \neg(\exists loc', loc'')a = endGo(loc', loc'')]\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset [going(l, l', do(a, s)) \equiv\\ &\quad a = startGo(l, l') \vee\\ &\quad going(l, l', s) \wedge a \neq endGo(l, l')]\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset [holdingCoffee(do(a, s)) \equiv\\ &\quad a = pickupCoffee \vee\\ &\quad holdingCoffee(s) \wedge \neg(\exists person)a = giveCoffee(person)].\end{aligned}$$

Additional Axioms: [7]

$$(\forall s)IsQueue(queue(s)) \qquad \texttt{(the values of } queue(\cdot) \texttt{ are queues)}$$

Unique names axioms stating that the following terms, together with those formed from nil and $cons(\cdot, \cdot)$ (see above), are pairwise unequal:

$$Sue,\ Mary,\ Bill,\ Joe,\ CM,\ office(Sue),$$
$$office(Mary),\ office(Bill),\ office(Joe).$$

[7]The first axiom is not strictly necessary, we add it for sake of clarity.

Initial Situation:

$$robotLocation(S_0) = CM \wedge \neg holdingCoffee(S_0) \wedge \neg(\exists l, l')going(l, l', S_0) \wedge \neg(\exists p)hasCoffee(p, S_0) \wedge queue(S_0) = nil$$

Robot's GOLOG Program: The robot execute the program $DeliverCoffee$ defined as follows (note the suppressed situation argument in primitive and test actions):

```
proc $DeliverCoffee$
    while True do
        if $\neg isEmpty(queue)$
            then $(\pi p) selectRequest(p); ServeCoffee(p)$
            else True?     (skip)
    endWhile
endProc

proc $ServeCoffee(p)$
    $Goto(CM)$;
    $pickupCoffee$;
    $Goto(office(p))$;
    $giveCoffee(p)$
endProc

proc $Goto(loc)$
    $startGo(robotLocation, loc)$;
    $endGo(robotLocation, loc)$
endProc
```

Dynamics of Exogenous Actions: Along all possible evolutions of any program δ_0, starting from S_0, into any configuration, in a finite number of transitions, a situation s is reached where somebody may request coffee ($DynaPoss$ holds) (provided that it is possible to request coffee, i.e. that also $Poss$ holds):

$$(\forall \delta_0, \delta, s)\, Trans^*(\delta_0, S_0, \delta, s) \supset ExoLaws(\delta, s)$$

$$ExoLaws(\delta_1, s_1) \overset{def}{=} \mu_{E,\delta,s}\{[(\exists p)DynaPoss(requestCoffee(p), s)] \vee [(\forall \delta', s')Trans(\delta, s, \delta', s') \supset E(\delta', s')]\}(\delta_1, s_1)$$

7.3 Reasoning

Next we show some dynamic properties of the overall system (the program plus the exogenous actions). First it is easy to see, from its structure, that the program $DeliverCoffee$ will never reach a final configuration:

$$(\forall \delta, s)\, Trans^*(DeliverCoffee, S_0, \delta, s) \supset \neg Final(\delta).$$

A more complex property that is possible to show is the following: every request for coffee sooner or later will be served. Formally, the *fairness* property $Fair(DeliverCoffee, S_0)$ holds, where:

$$Fair(\delta_0, s_0) \overset{def}{=} (\forall p, \delta, s)Trans^*(\delta_0, s_0, \delta, do(requestCoffee(p), s)) \supset EventuallyServed(p, \delta, do(requestCoffee(p), s))$$

and

$$EventuallyServed(p, \delta_1, s_1) \overset{def}{=} \mu_{P,\delta,s}\{[(\exists s'')s = do(selectRequest(p), s'')] \vee [((\exists \delta'.s')Trans(\delta, s, \delta', s')) \wedge (\forall \delta', s')Trans(\delta, s, \delta', s') \supset P(\delta', s')]\}(\delta_1, s_1)$$

It is also possible to show that there exists an (infinite) execution path where no coffee is ever served:

$$PossiblyAlwaysIdle(DeliverCoffee, S_0)$$

where

$$PossiblyAlwaysIdle(\delta_0, s_0) \overset{def}{=} \nu_{A,\delta,s}\{[(\forall p, s'')(s \neq do(selectRequest(p), s''))] \wedge [(\exists \delta', s')Trans(\delta, s, \delta', s') \wedge A(\delta', s')]\}(\delta_0, s_0).$$

However, by the fairness property above, this means that no requests for coffee were made along that execution path.

8 Conclusion and further work

In this paper we have given an account of non-terminating programs in the Situation Calculus. The framework obtained is quite powerful. It allows the specification of the dynamic system by modeling one agent with a program, and external events by suitable dynamic laws (extensions to multiple agents are also possible, see [DGLL97] for hints). Observe that although related this framework is more general than that typically considered in program verification, where exogenous actions that are specified by dynamic laws (axioms) are not allowed. There are many directions for further research. Among these we mention the development of systematic techniques for verification, such as suitable induction principles.

References

[Acz77] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. Elsevier, 1977.

[CC92] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 83–94, New York, U.S.A., 1992. ACM Press.

[Cou90] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 841–989. 1990.

[DGLL97] G. De Giacomo, H. J. Levesque, and Y. Lespérance. Reasoning about concurrent executions, prioritized interrupts, and exogenous actions in the situation calculus. Submitted, 1997.

[Eme96] E. A. Emerson. Automated temporal reasoning about reactive systems. In *Logics for Concurrency: Structure versus Automata*, number 1043 in Lecture Notes in Computer Science, pages 41–101. Springer-Verlag, 1996.

[Heh93] E.C.R. Hehner. *A practical theory of programming.* Springer-Verlag, 1993.

[Hen90] M. Hennessy. *The Semantics of Programming Languages.* John Wiley & Sons, 1990.

[Kna28] B. Knaster. Un thèoréme sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.

[KT90] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 790–840. 1990.

[Lei94] D. Leivant. Higher order logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 229–321. Clarendon Press, 1994.

[LRL$^+$97] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: a logic programming language for dynamic domains. *J. of Logic Programming*, 1997. To appear.

[LS87] J. Loeckx and K. Sieber. *Foundation of Program Verification.* Teubner-Wiley, New York, 1987.

[Mos74] Y.N. Moschovakis. *Elementary Induction on Abstract Structures.* Amsterdam, North Holland, 1974.

[MP95] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Vol. 1–2.* Springer-Verlag, 1992–1995.

[Par70] D. Park. Fixpoint induction and proofs of program properties. In *Machine Intelligence*, volume 5, pages 59–78. Edinburgh University Press, 1970.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Dept. Aarhus Univ. Denmark, 1981.

[Rei91] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Sti96] C. Stirling. Modal and temporal logics for processes. In *Logics for Concurrency: Structure versus Automata*, number 1043 in Lecture Notes in Computer Science, pages 149–237. Springer-Verlag, 1996.

[Tar55] B. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.