Proceedings, Eleventh International Conference on Principles of Knowledge Representation and Reasoning (2008)

A Logic for Non-Terminating Golog Programs

Jens Claßen and **Gerhard Lakemeyer**
Department of Computer Science
RWTH Aachen University
52056 Aachen
Germany
⟨classen|gerhard⟩@cs.rwth-aachen.de

Abstract

Typical Golog programs for robot control are non-terminating. Analyzing such programs so far requires meta-theoretic arguments involving complex fix-point constructions. In this paper we propose a logic based on the situation calculus variant $\mathcal{ES}$, which includes elements from branching time, dynamic and process logics and where the meaning of programs is modelled as possibly infinite sequences of actions. We show how properties of non-terminating programs can be formulated in the logic and, for a subset of it, how existing ideas from symbolic model checking in temporal logic can be applied to automatically verify program properties.

Introduction

The action language Golog (Levesque *et al.* 1997), which is based on the situation calculus (McCarthy & Hayes 1969; Reiter 2001), has already been successfully applied to the control of autonomous agents and robots. Usually, such robots fulfill open-ended, non-terminating tasks. Before deploying a control program and actually executing it in the real world, it is often desirable to verify that certain requirements are met. Typical examples are safety, liveness and fairness conditions.

As a simple example (adapted from (De Giacomo, Ternovska, & Reiter 1997)) that we will refer to throughout the paper, consider a mobile robot working in an office environment whose task is to serve coffee to people on request. When a new request arrives before the current one has been served, it is stored in a queue. A possible control program for such a robot might look like this:

loop : **if** $\neg Empty(queue)$
then $(\pi p) selectRequest(p)$;
$pickupCoffee$; $bringCoffee(p)$
else $wait$

Here the robot performs an infinite loop. In each iteration, when the queue is currently not empty, the next request (which comes from person p) is selected to be served, which means that the coffee has to be obtained at the coffee machine and brought to p's office. If there are no requests at all, the robot waits for a short period. Requests, which constitute exogenous actions in this scenario, may arrive at any time. Examples for program properties one may want to verify are: "Every request will eventually be served by the robot;" and "it is possible that no request is ever served."

Popular means for expressing such properties are temporal logics, in particular $\mathcal{LTL}$, $\mathcal{CTL}$ and $\mathcal{CTL}^*$ (Emerson 1990). For these logics, there exists a variety of algorithms (Clarke, Grumberg, & Peled 1999) and tools (Cimatti *et al.* 2002; Holzmann 2003) for checking whether a certain formula is satisfied in a given finite model of a concurrent system. These model checking methods have been successfully applied in practice for the verification of circuit design and communication protocols.

For computational complexity reasons, temporal logics are typically restricted to the propositional case. A Golog domain designer who wants to verify a certain control program is now confronted with the problem that the program and the corresponding background theory has to be abstracted and translated in order to be compatible with the temporal logic and the model checking tool that is to be used. Doing this manually is error-prone. On the other hand, an automated translation or embedding would certainly mean a loss of some of the expressive capabilities, such as first-order quantification. These are however considered desirable features that one would rather not give up, but are the reason why the language was chosen in the first place. It seems more preferable to be able to do the verification in the same formalism that is used for the specification and the actual control of the agent.

De Giacomo, Ternovska and Reiter (1997) show how both the meaning of programs and the properties to be verified can be expressed using fix-point constructions in the style of the μ-calculus, which is known to constitute a superset of the linear and branching time temporal logics named above. Fix points are expressed by means of formulas of second-order logic and one then has to do a manual, meta-theoretic proof of the desired property. There are automated tools that are capable of determining fix points of inductive definitions (Pelov & Ternovska 2005), but they are again often limited to the propositional case.

In this paper, we present a new logic, called $\mathcal{ESG}$, to express and reason about programs and their properties. It is based on the situation calculus variant $\mathcal{ES}$ (Lakemeyer & Levesque 2004) which makes certain restrictions that ease theoretical treatments, but does not yield a loss of expres-

siveness compared to the original situation calculus (Lakemeyer & Levesque 2005). In the non-epistemic case that we will consider here, one of these restrictions is that it assumes a fixed universe of discourse, which allows to treat quantification substitutionally. Another one is that future situations can only be referred to by means of the modal operators $[t]$ ("after the action t") and $\Box$ ("in all future situations").

Here we propose two extensions of the language. For one, we will allow not only atomic actions, but arbitrary Golog programs as arguments of the $[\cdot]$ operator. The formula $[\delta]\alpha$ then means that after executing the program δ, formula α will hold. Programs are composed of atomic actions and tests by means of constructors for non-deterministic choice, finite and infinite iteration, sequential composition and concurrency. In particular, this part of the language resembles dynamic logic (Harel, Kozen, & Tiuryn 2000), though our formalisms is more expressive in terms of modelling arbitrary atomic actions and using first-order quantification. The meaning of programs is defined inside the logic's semantics. We show that in case of terminating programs it is equivalent to the definition of Golog programs in (Lakemeyer & Levesque 2005).

For another, we want to go beyond reasoning about what holds *after* executing a program (which is especially useless in case of non-terminating processes) and consider properties which hold *during* its execution. For this purpose, we introduce formulas of the form $[\![\delta]\!]\alpha$ which expresses that α holds for all possible runs of program δ, where α may contain operators like "until", "eventually" or "globally" known from temporal logics. As an example, assume that δ refers to the above program. Then

$$[\![\delta]\!]\boldsymbol{G}(Occ(requestCoffee(p)) \supset \boldsymbol{F}Occ(selectRequest(p)))$$

states that for all possible executions of δ, it is always ($\boldsymbol{G}$) the case that when a $requestCoffee$ occurs by some person p, then eventually ($\boldsymbol{F}$) that request will be selected to be served.

For the $\mathcal{CTL}$-like fragment of the language we also provide an automated method to verify program properties. Roughly, for a given program, we first construct what we call a characteristic graph, where nodes represent program states and edges are labelled with actions and conditions, under which these action can be taken. Conditions involving temporal operators like "until" are then tested by computing fix points wrt these graphs, using methods from model checking. An adaptation further allows to also check postconditions of terminating programs. Since first-order reasoning is involved in each case, only correctness can be guaranteed but not termination.

The rest of the paper is organized as follows. In the next section we introduce the logic $\mathcal{ESG}$ and show how it relates to the original $\mathcal{ES}$ and Golog. Then we focus on the verification of properties of non-terminating and terminating programs with underlying basic action theories. After discussing related work we conclude.

The Logic $\mathcal{ESG}$

The language is a second-order modal dialect with equality and sorts of type *object*, *action*. It includes countably infinitely many standard names for each sort. Also included are both fluent and rigid predicate and function symbols, as well as rigid and fluent second-order variables. Fluents vary as the result of actions, but rigids do not. We assume that the fluents include unary predicates *Poss*, Occ, and Exo, whose argument is of type action.

The logical connectives are $\wedge$, $\neg$, $\forall$, together with these modal operators: $\boldsymbol{X}$, $\boldsymbol{U}$, $[\delta]$, and $[\![\delta]\!]$, where δ is a program defined below. Other connectives like $\vee$, $\supset$, $\subset$, $\equiv$, and $\exists$ are used as the usual abbreviations.

Terms and formulas Terms are formed in the usual way. Note that standard names syntactically are treated like ordinary constants. By a *primitive term* we mean one of the form $h(n_1, \ldots, n_k)$ where h is a (fluent or rigid) function symbol and all of the n_i are standard names.

Formulas are divided into two classes, *situation* and *trace* formulas. As their names suggest, situation formulas express properties that hold in a certain situation, while trace formulas describe properties of finite and infinite action sequences. The set of all formulas is defined to be the least set such that for the *situation formulas*:

1. If $t_1, \ldots, t_k$ are terms, and H is a k-ary predicate symbol then $H(t_1, \ldots, t_k)$ is an (atomic) situation formula;
2. If $t_1, \ldots, t_k$ are terms, and V is a k-ary second-order variable, then $V(t_1, \ldots, t_k)$ is an (atomic) situation formula;
3. If t_1 and t_2 are terms, then $(t_1 = t_2)$ is a sit. formula;
4. If δ is a program, α is a situation formula and ϕ is a trace formula, then $[\delta]\alpha$ and $[\![\delta]\!]\phi$ are situation formulas;
5. If α and β are situation formulas, v is a first-order variable, and V is a second-order variable, then the following are also situation formulas: $(\alpha \wedge \beta)$, $\neg\alpha$, $\forall v.\alpha$, $\forall V.\alpha$.

The *trace formulas* are further defined as follows:

1. Every situation formula is a trace formula;
2. If ϕ and ψ are trace formulas, v is a first-order variable and V is a second-order variable, then $\phi \wedge \psi$, $\neg\phi$, $\forall v.\phi$, $\forall V.\phi$, $\boldsymbol{X}\phi$ and $\phi\, \boldsymbol{U}\, \psi$ are also trace formulas.

We read $[\delta]$ as "α holds *after* all possible executions of δ", $[\![\delta]\!]\phi$ as "ϕ holds *for* all possible executions of δ", $Occ(t)$ as "t was the last action", $\boldsymbol{X}\phi$ as "ϕ holds after the next action" and $\phi\, \boldsymbol{U}\, \psi$ as "ϕ will hold until ψ holds". To obtain the duals of $[\delta]\alpha$ and $[\![\delta]\!]\alpha$, we let $\langle\delta\rangle\alpha$ stand for $\neg[\delta]\neg\alpha$ and $\langle\!\langle\delta\rangle\!\rangle\alpha$ for $\neg[\![\delta]\!]\neg\alpha$.

Formulas without free variables are called sentences. A primitive sentence is a predicate whose arguments are standard names. We call a formula without $[\delta]$, $[\![\delta]\!]$, *Poss*, Exo and Occ a *fluent formula*. A formula that does not contain $[\![\delta]\!]$ and where at most $[t]$ appear where the argument t is an atomic action is called a *bounded formula*.

Programs Programs are composed according to the following grammar:

$$\delta ::= t \mid \varphi? \mid (\delta_1; \delta_2) \mid (\delta_1|\delta_2) \mid \pi x.\delta \mid (\delta_1 \| \delta_2) \mid \delta^*$$

Here, t is any (not necessarily ground) term of sort action and φ can be any situation formula of the full In the presented order, the constructs mean a primitive action, a test, sequence of programs, nondeterministic choice between

programs, nondeterministic choice of argument, concurrent[1] execution of programs, and nondeterministic iteration. We remark that, except for procedures, the program constructs essentially correspond to those of ConGolog (De Giacomo, Lespérance, & Levesque 2000). Further control structures can be introduced as abbreviations:[2]

$$\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{def}{=} \varphi?;\delta_1 \mid \neg\varphi?;\delta_2$$

$$\textbf{while } \varphi \textbf{ do } \delta \textbf{ endWhile} \stackrel{def}{=} (\varphi?;\delta)^*;\neg\varphi?$$

$$\textbf{loop } \delta \stackrel{def}{=} \textbf{while } \top \textbf{ do } \delta \textbf{ endWhile}$$

For the purpose of this paper, the **loop** δ construct is particularly important; we will also abbreviate it as δ^ω.

Our language somewhat generalizes classical $\mathcal{CTL}^*$. The usual path quantifiers can be introduced by defining $\boldsymbol{E}\alpha$ as $\langle\!\langle any^\omega \rangle\!\rangle \alpha$ and $\boldsymbol{A}\alpha$ as $[\![any^\omega]\!]\alpha$, where any is shorthand for $\pi a.a$, i.e. it denotes the execution of an arbitrary action. Further we abbreviate $(\top\ \boldsymbol{U}\ \alpha)$ as $\boldsymbol{F}\alpha$ ("eventually") and $\neg\boldsymbol{F}\neg\alpha$ as $\boldsymbol{G}\alpha$ ("always"). To stay compatible with the original $\mathcal{ES}$ from (Lakemeyer & Levesque 2005), we finally understand $\Box\alpha$ as an abbreviation for $\boldsymbol{AG}\alpha$.

The Semantics

To determine the truth of a sentence, we need a world w which determines the truth values of primitive sentences and co-referring standard names for primitive terms after any sequence of actions. Formally:

> A world $w \in W$ is any function from the primitive sentences and $\mathcal{Z}$ to $\{0,1\}$, and from the primitive terms and $\mathcal{Z}$ to $\mathcal{N}$ (preserving sorts), and satisfying the rigidity constraint: if r is a rigid function or predicate symbol, then $w[r(n_1,\ldots,n_k),z] = w[r(n_1,\ldots,n_k),z']$ for all $z, z' \in \mathcal{Z}$.

Here, $\mathcal{N}$ denotes the set of all standard names and $\mathcal{Z}$ the set of all finite sequences of standard names of sort action, including the empty sequence $\langle\,\rangle$.

The idea of co-referring standard names is extended to arbitrary ground terms as follows. Given a variable-free term t, a world w, and an action sequence z, we define $|t|^z_w$ (read: the co-referring standard name for t given w and z) by:

1. If $t \in \mathcal{N}$, then $|t|^z_w = t$;
2. $|h(t_1,\ldots,t_k)|^z_w = w[h(n_1,\ldots,n_k),z]$, where $n_i = |t_i|^z_w$.

To interpret formulas with free variables, we proceed as follows. First-order variables are handled substitutionally using the standard names. To handle the quantification over second-order variables, we use second-order *variable maps* defined as below:

> The *second-order primitives* are formulas of the form $V(n_1,\ldots,n_k)$ where V is a (fluent or rigid) second-order variable and all of the n_i are standard names. A *variable map* u is a function from second-order primitives and $\mathcal{Z}$ to $\{0,1\}$, satisfying the rigidity constraint: if Q is a rigid second-order variable, then for all z and z' in $\mathcal{Z}$, $u[Q(n_1,\ldots,n_k),z] = u[Q(n_1,\ldots,n_k),z']$.

[1] Like in ConGolog, we understand concurrency as interleaving.

[2] We use $\top$ to denote truth, which can be defined as $\forall x.(x = x)$, and $\bot$ for falsity, i.e. $\neg\top$.

Let u and u' be variable maps, and let V be a second-order variable; we write $u \sim_V u'$ to mean that u and u' agree except perhaps on the second-order primitives involving V.

Now given $w \in W$, we define $w \models \alpha$ for situation formulas α as $w, \langle\,\rangle, u \models \alpha$ for all variable maps u, where for any sequence $z \in \mathcal{Z}$, and variable map u:

1. $w,z,u \models H(t_1,\ldots,t_k)$ iff
   $w[H(n_1,\ldots,n_k),z] = 1$, where $n_i = |t_i|^z_w$;
2. $w,z,u \models V(t_1,\ldots,t_k)$ iff
   $u[V(n_1,\ldots,n_k),z] = 1$, where $n_i = |t_i|^z_w$;
3. $w,z,u \models (t_1 = t_2)$ iff
   n_1 and n_2 are identical, where $n_i = |t_i|^z_w$;
4. $w,z,u \models Occ(t)$ iff $z = z' \cdot n$, where $n = |t|^z_w$;
5. $w,z,u \models \alpha \wedge \beta$ iff $w,z,u \models \alpha$ and $w,z,u \models \beta$;
6. $w,z,u \models \neg\alpha$ iff $w,z,u \not\models \alpha$;
7. $w,z,u \models \forall v.\alpha$ iff $w,z,u \models \alpha^v_n$ for all $n \in \mathcal{N}_v$;
8. $w,z,u \models \forall V.\alpha$ iff $w,z,u' \models \alpha$ for all $u' \sim_V u$;
9. $w,z,u \models [\![\delta]\!]\phi$ iff for all $\tau \in \|\delta\|^w_u(z)$, $w,z,\tau,u \models \phi$;
10. $w,z,u \models [\delta]\alpha$ iff
    for all finite $z' \in \|\delta\|^w_u(z)$, $w, z\cdot z', u \models \alpha$.

$\mathcal{N}_v$ refers to the set of standard names of the same sort as v. α^v_n means α with every free occurrence of v replaced by n. $\|\delta\|^w_u(z)$, which is defined below, maps, given w, u and z, a program to a set of program traces, where a trace can be a finite or infinite sequence of action standard names. Rule 10 only requires that α holds after all finite sequences for $[\delta]\alpha$ to be true. In particular this implies that any formula is vacuously true "after" the execution of a non-terminating program δ^ω. On the other hand, in rule 9, the trace formula ϕ must hold for any trace τ, be it finite or not. The truth of trace formulas is given by:

1. $w,z,\tau,u \models \alpha$ iff
   $w,z,u \models \alpha$, where α is a situation formula;
2. $w,z,\tau,u \models \phi \wedge \psi$ iff $w,z,\tau,u \models \phi$ and $w,z,\tau,u \models \psi$;
3. $w,z,\tau,u \models \neg\phi$ iff $w,z,\tau,u \not\models \phi$;
4. $w,z,\tau,u \models \forall v.\phi$ iff $w,z,\tau,u \models \phi^v_n$ for all $n \in \mathcal{N}_v$;
5. $w,z,\tau,u \models \forall V.\phi$ iff $w,z,\tau,u' \models \phi$ for all $u' \sim_V u$;
6. $w,z,\tau,u \models \boldsymbol{X}\phi$ iff $\tau = n \cdot \tau'$ and $w, z\cdot n, \tau', u \models \phi$;
7. $w,z,\tau,u \models \phi\ \boldsymbol{U}\ \psi$ iff
   there is z' such that $\tau = z' \cdot \tau'$ and $w, z\cdot z', \tau', u \models \psi$
   and for all $z'' \neq z'$ with $z' = z'' \cdot z'''$,
   $w, z\cdot z'', z''' \cdot \tau' \models \phi$.

When Σ is a set of sentences and α is a sentence, we write $\Sigma \models \alpha$ (read: Σ logically entails α) to mean that for every w, if $w \models \alpha'$ for every $\alpha' \in \Sigma$, then $w \models \alpha$. Finally, we write $\models \alpha$ (read: α is valid) to mean $\{\} \models \alpha$. As a notational convention, we will in the following always use (possibly with sub- or superscripts) z for finite sequences, π for infinite ones and τ for arbitrary traces.

The Meaning of Programs The program semantics we present here is an adaptation of the single step semantics of (De Giacomo, Lespérance, & Levesque 2000). A key difference is that we do not define the meaning of programs axiomatically, but inside the logic. We also treat tests differently in that we do not view them as transitions, but rather as conditions under which a transition (which is always a physical action in our framework) may be taken or under which the run of a program may terminate.

A central concept is that of a *configuration*, which we denote as a pair (δ, z), where δ is a program (intuitively what remains to be executed) and z a sequence of actions (that have been already performed). Program configurations can be *final*, which means that the run may successfully terminate in that particular situation, or they can make certain *transitions* to other configurations, each of which involves performing some physical action.

Formally, the set of final configurations $\mathcal{F}^{w,u}$ is the smallest set such that for all δ, δ_1, δ_2, situation formulas φ and finite z:

1. $(\varphi?, z) \in \mathcal{F}^{w,u}$ if $w, z, u \models \varphi$;
2. $(\delta_1; \delta_2, z) \in \mathcal{F}^{w,u}$ if $(\delta_1, z) \in \mathcal{F}^{w,u}$ and $(\delta_2, z) \in \mathcal{F}^{w,u}$;
3. $(\delta_1|\delta_2, z) \in \mathcal{F}^{w,u}$ if $(\delta_1, z) \in \mathcal{F}^{w,u}$ or $(\delta_2, z) \in \mathcal{F}^{w,u}$;
4. $(\pi x.\delta, z) \in \mathcal{F}^{w,u}$ if $(\delta^x_n, z) \in \mathcal{F}^{w,u}$ for some $n \in \mathcal{N}_x$;
5. $(\delta^*, z) \in \mathcal{F}^{w,u}$;
6. $(\delta_1 \| \delta_2, z) \in \mathcal{F}^{w,u}$ if$(\delta_1, z) \in \mathcal{F}^{w,u}$ and $(\delta_2, z) \in \mathcal{F}^{w,u}$.

Thus, a configuration $(\varphi?, z)$ whose remaining program is a test is final wrt w and u if the formula φ holds at w, z, u. From the above it also follows that $(t, z) \notin \mathcal{F}^{w,u}$ for atomic t, i.e. if some action t remains to be done, the configuration cannot be final. Further, sequences are only final when the involved subprograms are both final etc. The transition relation among configurations is given as follows:

1. $(t, z) \underset{w,u}{\rightarrow} (nil, z \cdot n)$ if $n = |t|^z_w$;
2. $(\delta_1; \delta_2, z) \underset{w,u}{\rightarrow} (\gamma; \delta_2, z \cdot n)$ if $(\delta_1, z) \underset{w,u}{\rightarrow} (\gamma, z \cdot n)$;
3. $(\delta_1; \delta_2, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$
   if $(\delta_1, z) \in \mathcal{F}^{w,u}$ and $(\delta_2, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$;
4. $(\delta_1|\delta_2, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$
   if $(\delta_1, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$ or $(\delta_2, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$;
5. $(\pi x.\delta, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$
   if $(\delta^x_{n'}, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$ for some $n' \in \mathcal{N}_x$;
6. $(\delta^*, z) \underset{w,u}{\rightarrow} (\gamma; \delta^*, z \cdot n)$ if $(\delta, z) \underset{w,u}{\rightarrow} (\gamma, z \cdot n)$;
7. $(\delta_1 \| \delta_2, z) \underset{w,u}{\rightarrow} (\delta' \| \delta_2, z \cdot n)$ if $(\delta_1, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$;
8. $(\delta_1 \| \delta_2, z) \underset{w,u}{\rightarrow} (\delta_1 \| \delta', z \cdot n)$ if $(\delta_2, z) \underset{w,u}{\rightarrow} (\delta', z \cdot n)$.

Above, nil denotes the empty program and should be read as an abbreviation for $\top?$. (t, z) may therefore successfully terminate after performing physical action n, the latter being the action standard name by which t is interpreted in w at z. A sequence of programs can either make a transition in the first subprogram or a transition in the second subprogram, provided that the first one is final etc.

We can now define the set $\|\delta\|^w_u(z)$ of execution traces of a program δ, given w, u, z. In general this set may contain both finite and infinite sequences of action standard names. If $\underset{w,u}{\overset{*}{\rightarrow}}$ is the reflexive transitive closure of $\underset{w,u}{\rightarrow}$, $\|\delta\|^w_u(z)$ is

$$\{z' \mid (\delta, z) \underset{w,u}{\overset{*}{\rightarrow}} (\delta', z \cdot z') \text{ and } (\delta', z \cdot z') \in \mathcal{F}^{w,u}\} \cup$$
$$\{\pi \mid \text{for all } z' \in Pre(\pi),\ (\delta, z) \underset{w,u}{\overset{*}{\rightarrow}} (\delta', z \cdot z') \text{ and } (\delta', z \cdot z') \notin \mathcal{F}^{w,u}\},$$

where $Pre(\pi)$ is the set of all prefixes of π. A finite execution trace therefore is given by repeatedly following the transition relation and ending up in a terminating configuration; infinite runs never visit final configurations.

$\mathcal{ES}$ is part of $\mathcal{ESG}$

The language presented in the previous section is in fact a superset of the non-epistemic part of $\mathcal{ES}$ as presented in (Lakemeyer & Levesque 2005):

Theorem 1 *Let α be a sentence of $\mathcal{ES}$ without epistemic operators. Then* $\models_{\mathcal{ES}} \alpha$ *iff* $\models_{\mathcal{ESG}} \alpha$.

The theorem, which is proved in the appendix, essentially tells us that our $[\delta]$ operator, when restricted to atomic actions t, corresponds to Lakemeyer and Levesque's $[t]$ operator. Furthermore our definition of $\Box\alpha$ as an abbreviation for $\boldsymbol{AG}\alpha$ correctly captures their semantics for this construct.

Lakemeyer and Levesque provide a macro-based definition of the semantics of terminating Golog programs which they prove to be equivalent to the one for the classical situation calculus:

Definition 2 (The Do Semantics) *Let α, φ be formulas and δ, δ' be Golog programs.*

1. $Do(t, \alpha) \overset{def}{=} (Poss(t) \wedge [t]\alpha)$;
2. $Do((\varphi)?, \alpha) \overset{def}{=} (\varphi \wedge \alpha)$;
3. $Do(\delta; \delta', \alpha) \overset{def}{=} Do(\delta, Do(\delta', \alpha))$;
4. $Do(\delta|\delta', \alpha) \overset{def}{=} (Do(\delta, \alpha) \vee Do(\delta', \alpha))$;
5. $Do(\pi x.\delta, \alpha) \overset{def}{=} \exists x.Do(\delta, \alpha)$;
6. $Do(\delta^*, \alpha) \overset{def}{=}$
   $\forall P.\{\Box(\alpha \supset P) \wedge \Box(Do(\delta, P) \supset P)\} \supset P$.

$Do(\delta, \alpha)$ therefore can be expanded to a formula that states under which condition there is a successfully terminating execution of δ after which α will hold. In our logic, this fact can simply be expressed by a formula:

Theorem 3 *Let δ be a program without $\|$ and α be a formula. Let δ^p be δ with every atomic action t replaced by $Poss(t)?; t$. Then* $\models Do(\delta, \alpha) \equiv \langle\delta^p\rangle\alpha$.

Basic Action Theories and Regression

Since we established the correspondence to the original $\mathcal{ES}$, we can make use of all results related to it, in particular concerning action theories and regression.

Basic Action Theories A basic action theory (BAT) is used to describe the specifics of a certain dynamic application domain. In $\mathcal{ES}$ and therefore also $\mathcal{ESG}$, they can be defined similar to Reiter-style BATs for the situation calculus. A set of sentences Σ is a *basic action theory* iff it only mentions the fluents in a given finite set $\mathcal{F}$ and is of the form $\Sigma = \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}}$ where Σ_0, the initial database, is a finite set of fluent sentences and Σ_{pre} is a precondition axiom of the form[3] $\Box \mathit{Poss}(a) \equiv \pi$, with π being a fluent formula whose only free variable is a. Σ_{post} is a finite set of successor state axioms (SSAs)[4]

$$\Box[a]F(\vec{x}) \equiv \gamma_F \text{ and } \Box[a]f(\vec{x}) = y \equiv \gamma_f$$

for each relational fluent $F \in \mathcal{F} \setminus \{\mathit{Poss}\}$ and each functional fluent $f \in \mathcal{F}$, incorporating Reiter's (2001) solution to the frame problem. γ_F has to be a fluent formula with free variables $\vec{x}$, and γ_f one with free variables among $\vec{x}$ and y.

In the example case of the coffee delivery robot, $\mathcal{F}$ consists of two fluents: $\mathit{HoldingCoffee}$ is relational and holds when the robot is currently holding coffee; queue is functional and contains the current queue of coffee requests. Further we have five primitive actions: $\mathit{requestCoffee}(p)$ is the exogenous action (not under the robot's control) of person p sending a request for coffee. $\mathit{selectRequest}(p)$ means the next request (which came from person p) from the queue is selected by the robot to be served. The robot can also go and get coffee from the coffee machine ($\mathit{pickupCoffee}$) and bring it to p ($\mathit{bringCoffee}(p)$). wait finally means to do nothing for a while.

A possible initial database expresses that the robot initially is not holding coffee and the request queue is empty:

$$\Sigma_0 = \{\neg \mathit{HoldingCoffee},\ \mathit{Empty}(\mathit{queue})\}$$

The preconditions Σ_{pre} are given by:

$$\begin{aligned}\Box\mathit{Poss}(a) \equiv\ & a = \mathit{wait} \vee \\ & \exists p.\ a = \mathit{requestCoffee}(p) \wedge \neg \mathit{Full}(\mathit{queue}) \vee \\ & \exists p.\ a = \mathit{selectRequest}(p) \wedge \mathit{IsFirst}(\mathit{queue}, p) \vee \\ & \quad a = \mathit{pickupCoffee} \wedge \neg \mathit{HoldingCoffee} \vee \\ & \exists p.\ a = \mathit{bringCoffee}(p) \wedge \mathit{HoldingCoffee}\end{aligned}$$

Here, $\mathit{requestCoffee}(p)$ is only possible when the queue is not already full (assuming a fixed limit), $\mathit{pickupCoffee}$ only when the robot is not holding coffee etc. Σ_{post} finally contains the following successor state axioms:

$$\begin{aligned}&\Box[a]\mathit{HoldingCoffee} \equiv a = \mathit{pickupCoffee} \vee \\ &\quad \mathit{HoldingCoffee} \wedge \neg\exists p.\ a = \mathit{bringCoffee}(p), \\ &\Box[a]\mathit{queue} = y \equiv \\ &\quad \exists p.\ a = \mathit{requestCoffee}(p) \wedge \mathit{Enqueue}(\mathit{queue}, p, y) \vee \\ &\quad \exists p.\ a = \mathit{selectRequest}(p) \wedge \mathit{Dequeue}(\mathit{queue}, p, y) \vee \\ &\quad \mathit{queue} = y \wedge \neg\exists p\ (a = \mathit{requestCoffee}(p) \vee \\ &\qquad\qquad a = \mathit{selectRequest}(p))\end{aligned}$$

For a queue with size limit k, we use a simple encoding that represents the queue's state by a term of the form $\mathit{list}(p_1, \ldots, p_k)$, where empty positions are represented by having $p_i = e$, e being a distinguished standard name. Above we made use of these abbreviations:

$$\begin{aligned}&\mathit{IsFirst}(q, p) \stackrel{def}{=} (p \neq e) \wedge \\ &\quad \exists p_2 \ldots \exists p_k.\ q = \mathit{list}(p, p_2, \ldots, p_k), \\ &\mathit{Empty}(q) \stackrel{def}{=} q = \mathit{list}(e, \ldots, e), \\ &\mathit{Full}(q) \stackrel{def}{=} \exists x_1 \ldots \exists x_k.\ \textstyle\bigwedge_{i=1}^{k}(x_i \neq e) \wedge \\ &\quad\quad q = \mathit{list}(x_1, \ldots, x_k), \\ &\mathit{Enqueue}(q_o, p, q_n) \stackrel{def}{=} (p \neq e) \wedge \\ &\quad \textstyle\bigvee_{i=0}^{k-1} \exists x_1 \ldots \exists x_i.\ \bigwedge_{j=1}^{i}(x_i \neq e) \wedge \\ &\quad\quad q_o = \mathit{list}(x_1, \ldots, x_i, e, \ldots, e) \wedge \\ &\quad\quad q_n = \mathit{list}(x_1, \ldots, x_i, p, e, \ldots, e), \\ &\mathit{Dequeue}(q_o, p, q_n) \stackrel{def}{=} (p \neq e) \wedge \exists x_2 \ldots \exists x_k. \wedge \\ &\quad q_o = \mathit{list}(p, x_2, \ldots, x_k) \wedge \\ &\quad q_n = \mathit{list}(x_2, \ldots, x_{k-1}, e)\end{aligned}$$

Exogenous Actions To model actions that are not under the agent's control, we include an additional axiom Σ_{exo} in the BAT. It has the form $\Box \mathit{Exo}(a) \equiv \chi$, where χ is a fluent formula whose only free variable is a. It defines the necessary and sufficient conditions under which a is exogenous. In case of the coffee delivery robot, χ is $\exists p.\ a = \mathit{requestCoffee}(p)$. We then have a process $\delta_{EXO} = (\pi a.\mathit{Exo}(a)?; a)^{\omega}$ that constantly executes exogenous actions and use $(\delta_{ctrl} \| \delta_{EXO})$ instead of the actual control program δ_{ctrl}.

Regression For such BATs and formulas that only contain $[t]$ constructs for atomic actions t, (Lakemeyer & Levesque 2004) introduce an $\mathcal{ES}$ equivalent of Reiter's regression operator. The idea behind regression is that whenever we encounter a subformula of the form $[t]F(\vec{x})$, we may substitute it by γ_F, the right-hand side of the successor state axiom of F. This is sound in the sense that the axiom defines the two expressions to be equivalent. The result of the substitution will be true in exactly the same worlds satisfying the action theory Σ as the original one, but contains one less modal operator $[t]$. Similarly, $[t]\mathit{Occ}(t')$ is replaced by $(t = t')$ and $\mathit{Poss}(t)$ and $\mathit{Exo}(t)$ by the right-hand sides of the corresponding axiom. Iteratively applying such substitution steps, we end up with a fluent formula that describes exactly the conditions on the initial situation under which the original, non-static formula holds. We have an adapted version of Lakemeyer and Levesque's regression theorem:

Theorem 4 *Let Σ be a BAT and let α be a bounded sentence. Then $\mathcal{R}[\alpha]$, the regression of α, is a fluent sentence and $\Sigma \models \alpha$ iff $\Sigma_0 \models \mathcal{R}[\alpha]$.*

Program Verification

We now have everything in hand to reason about programs properties. Typically, the verification of a program δ, given a BAT Σ, amounts to checking whether $\Sigma \models (\neg)\langle\langle\delta\rangle\rangle\varphi$ (for non-terminating δ) and $\Sigma \models (\neg)\langle\delta\rangle\alpha$ (for terminating δ).

[3] Free variables are understood as universally quantified from the outside; $\Box$ has lower syntactic precedence than the logical connectives, i.e. $\Box\mathit{Poss}(a) \equiv \pi$ stands for $\forall a.\Box(\mathit{Poss}(a) \equiv \pi)$.

[4] The $[t]$ construct has higher precedence than the logical connectives. So $\Box[a]F(\vec{x}) \equiv \gamma_F$ abbreviates $\forall a.\Box(([a]F(\vec{x})) \equiv \gamma_F)$.

Verifying Non-Terminating Programs

As a motivation, consider again the example control program δ_{coffee} that was presented in the introduction and let δ in the following stand for $(\delta_{coffee} \| \delta_{EXO})^p$. Then indeed the BAT presented in the last section entails these two sentences:

$$[\![\delta]\!]\boldsymbol{G}(Occ(requestCoffee(p)) \supset \boldsymbol{F}Occ(selectRequest(p)))$$
$$\langle\!\langle\delta\rangle\!\rangle\boldsymbol{G}\neg\exists p(Occ(selectRequest(p)))$$

Both properties that we mentioned in the introduction therefore actually hold: whenever a request for coffee occurs, it will eventually be served and it might happen that no request is ever served. Because of the former, the latter can only happen when there are no requests at all.

Instead of proving such properties manually, automated verification is of course preferable. We will present an algorithm that can do so for a restricted $\mathcal{CTL}$-like subset of $\mathcal{ESG}$. More precisely, we say that $\varphi \in \mathcal{ESG}_{\mathcal{CTL}}$ if it is built according to the following grammar:

$$\varphi ::= (t_1 = t_2) \mid F(\vec{t}) \mid Occ(t) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \mid \langle\!\langle\delta\rangle\!\rangle \boldsymbol{X}\varphi \mid \langle\!\langle\delta\rangle\!\rangle \varphi \,\boldsymbol{U}\, \varphi \mid \langle\!\langle\delta\rangle\!\rangle \boldsymbol{G}\varphi$$

As in $\mathcal{CTL}$, only using existential path quantifiers is no real restriction, since the following equivalences are valid:

$$\models [\![\delta]\!]\boldsymbol{X}\varphi \equiv \neg\langle\!\langle\delta\rangle\!\rangle\boldsymbol{X}\neg\varphi$$
$$\models [\![\delta]\!]\boldsymbol{G}\varphi \equiv \neg\langle\!\langle\delta\rangle\!\rangle(\top\,\boldsymbol{U}\,\neg\varphi)$$
$$\models [\![\delta]\!](\phi\,\boldsymbol{U}\,\psi) \equiv \neg\langle\!\langle\delta\rangle\!\rangle(\neg\psi\,\boldsymbol{U}\,(\neg\phi\wedge\neg\psi)) \wedge \neg\langle\!\langle\delta\rangle\!\rangle\boldsymbol{G}\neg\psi$$

The second example property above is in $\mathcal{ESG}_{\mathcal{CTL}}$, while the first one is not. We further require that programs have the form ${\delta_1}^\omega \| \cdots \| {\delta_k}^\omega$, which is rather typical for non-terminating robot control programs and therefore not a rigorous restriction. The example δ is of this form, including the part that encodes exogenous actions.

Characteristic Graphs A central idea of our algorithm is that we encode the space of reachable program configurations by what we call a *characteristic graph* of a program. The nodes V in such a graph are tuples of the form $\langle\delta', \phi\rangle$, which intuitively denote the remaining program of the current run and the condition under which execution may terminate there. The initial node is denoted by v_0. Edges in E are labeled with tuples $\pi\vec{x} : t/\psi$, where $\vec{x}$ is a list of variables (if it is empty, we omit the leading π), t is an action term and ψ is a formula (which we omit when it is $\top$). Intuitively, this means when one wants to take action t, one has to choose instantiations for the $\vec{x}$ and ψ must hold.

The rather lengthy formal definition of characteristic graphs is presented in the appendix. To get an intuition, consider Figure 1, which depicts the characteristic graph for $\delta_{coffee} \| \delta_{EXO}$, where

$$v_0 = \langle \delta_{coffee} \parallel \delta_{EXO}, \bot\rangle$$
$$v_1 = \langle (pickupCoffee; bringCoffee(p); \delta_{coffee}) \parallel \delta_{EXO}, \bot\rangle$$
$$v_2 = \langle (bringCoffee(p); \delta_{coffee}) \parallel \delta_{EXO}, \bot\rangle$$

The fact that the program is non-terminating is reflected in the cyclic structure of the graph and the fact that the termination condition in each node is $\bot$. In the initial node, depending on whether $Empty(queue)$ holds or not, either $wait$ or the $selectRequest$ is the only applicable (non-exogenous) action. In any configuration, an exogenous action (here: an incoming request) may occur, hence the reflexive edges $\pi a : a/Exo(a)$ at each node. Note that the preconditions of actions are omitted here; the corresponding graph for the above δ is obtained by simply conjoining the condition ϕ in each transition $\pi\vec{x} : t/\phi$ with $Poss(t)$.

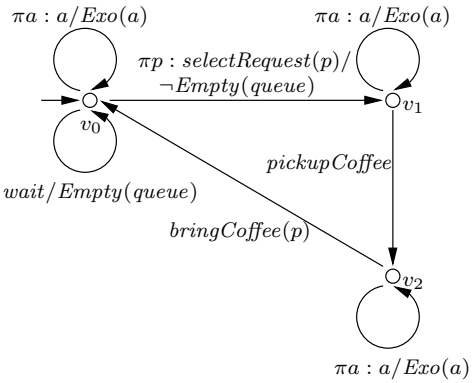

Figure 1: Characteristic Graph for the Coffee Example

The Algorithm To check a formula of $\mathcal{ESG}_{\mathcal{CTL}}$ against some BAT Σ, we transform it as follows:

$$\mathcal{C}[(t_1 = t_2)] = (t_1 = t_2); \qquad \mathcal{C}[F(\vec{t})] = F(\vec{t});$$
$$\mathcal{C}[\varphi_1 \wedge \varphi_2] = \mathcal{C}[\varphi_1] \wedge \mathcal{C}[\varphi_2]; \qquad \mathcal{C}[\neg\varphi] = \neg\mathcal{C}[\varphi];$$
$$\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle\boldsymbol{G}\varphi] = \textsc{CheckEG}[\delta, \varphi]; \qquad \mathcal{C}[\exists x.\varphi] = \exists x.\mathcal{C}[\varphi];$$
$$\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle\boldsymbol{X}\varphi] = \textsc{CheckEX}[\delta, \varphi]; \qquad \mathcal{C}[Occ(t)] = Occ(t);$$
$$\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle\varphi_1\,\boldsymbol{U}\,\varphi_2] = \textsc{CheckEU}[\delta, \varphi_1, \varphi_2].$$

In each of the cases with path quantifiers, we first determine the characteristic graph of δ. The procedures for these cases operate on sets of labels of the form $\langle v, \psi\rangle$, where $v = \langle\delta', \phi\rangle$ is a node in the graph and ψ is a formula, which should be regarded as a representation for all the infinitely many configurations (δ', z) where ψ holds. Let us first consider the procedure for "next":

Procedure 1 $\textsc{CheckEX}[\delta, \varphi]$

$X := \textsc{Pre}[\mathcal{G}_\delta, \textsc{Label}[\mathcal{G}_\delta, \mathcal{C}[\varphi]]]$;
return $\textsc{InitLabel}[\mathcal{G}_\delta, X]$

$\textsc{Label}[\mathcal{G}, \alpha]$ means the set where all vertices of $\mathcal{G}$ are labelled only with α, i.e.

$$\textsc{Label}[\langle V, E, v_0\rangle, \alpha] = \{\langle v, \alpha\rangle \mid v \in V\}.$$

The preimage of a set of labels is determined by traversing the edges in $\mathcal{G}$ backwards and regressing the formula in the each label, conjoined with the transition condition:

$$\textsc{Pre}[\langle V, E, v_0\rangle, S] =$$
$$\{\langle v', \mathcal{R}[\exists\vec{x}.\phi \wedge [t]\psi]\rangle \mid v' \overset{\pi\vec{x}:t/\phi}{\rightarrow} v \in E,\ \langle v, \psi\rangle \in S\}$$

Finally, given a set of labels, we need a function that extracts the information about the initial situation, following the convention that an empty disjunction is understood as $\bot$:

$$\textsc{InitLabel}[\langle V, E, v_0\rangle, S] = \bigvee\{\psi \mid \langle v_0, \psi\rangle \in S\}$$

Therefore, to check whether there is some trace such that φ holds at the next situation, we recursively transform φ, label all nodes with the result and compute the corresponding preimage using regression. The labels that are then at the initial node encode the conditions under which there is a trace with the desired property.

The procedures for "until" and "always" are presented below. They transform the set of labels iteratively until it converges. In case of "until", the idea is to start with $\mathcal{C}[\psi]$ labels and in each iteration, go one transition backwards to a configuration where $\mathcal{C}[\phi]$ must hold. Similarly, for "always", we have an initial labelling with $\mathcal{C}[\varphi]$ and iteratively conjoin the labels from the last iteration with its preimage.

Procedure 2 CHECKEU$[\delta, \phi, \psi]$

$X := \textsc{Label}[\mathcal{G}_\delta, \mathcal{C}[\psi]]; \quad X_O := \textsc{Label}[\mathcal{G}_\delta, \top];$
while $X \not\equiv X_O$ **do**
  $X_O := X;$
  $X := X \cup \textsc{And}(\textsc{Label}[\mathcal{G}_\delta, \mathcal{C}[\phi]], \textsc{Pre}[\mathcal{G}_\delta, X]);$
end while
return $\textsc{InitLabel}[\mathcal{G}, X]$

Procedure 3 CHECKEG$[\delta, \varphi]$

$X := \textsc{Label}[\mathcal{G}_\delta, \mathcal{C}[\varphi]]; \quad X_O := \textsc{Label}[\mathcal{G}_\delta, \bot];$
while $X \not\equiv X_O$ **do**
  $X_O := X;$
  $X := \textsc{And}(X, \textsc{Pre}[\mathcal{G}_\delta, X]);$
end while
return $\textsc{InitLabel}[\mathcal{G}_\delta, X]$

The conjunction of two sets of labels is as expected:

$$\textsc{And}(S_1, S_2) = \{\langle v, \psi_1 \wedge \psi_2\rangle \mid \langle v, \psi_1\rangle \in S_1, \langle v, \psi_2\rangle \in S_2\}$$

The condition "$X \not\equiv X_O$" is violated when each label $\langle v, \psi\rangle \in X$ has a counterpart $\langle v, \psi'\rangle \in X_O$ such that $\Sigma_{\text{UNA}} \models \psi \equiv \psi'$, where Σ_{UNA} are unique names axioms for actions. Since we allow first-order quantification, such tests for equivalence are in general undecidable. In practice it is however often sufficient to do simple syntactical simplifications (e.g. replace $\phi \wedge \phi$ by ϕ) in order to detect convergence, as we will later see in the example.

Theorem 5 *Let $\varphi \in \mathcal{ESG}_{\mathcal{CTL}}$. If the computation of $\mathcal{C}[\varphi]$ wrt Σ terminates, it is a fluent formula and*

$$\Sigma \cup \Sigma_{UNA} \models \varphi \quad \textit{iff} \quad \Sigma_0 \cup \Sigma_{UNA} \models \mathcal{C}[\varphi].$$

As a remark, our procedures CHECKEU$[\delta, \phi, \psi]$ and CHECKEG$[\delta, \varphi]$ correspond to the iterative approximation of least and greatest fix points in Kleene's constructive proof of the Tarski-Knaster Theorem, which guarantees that these fix points exist. Of course since our formalism allows infinite state spaces, they need not be finite like in propositional $\mathcal{CTL}$, i.e. the computation is not guaranteed to terminate.

Example We can apply the algorithm in order to check $\langle\!\langle\delta\rangle\!\rangle \boldsymbol{G} \neg\exists p\ (Occ(selectRequest(p)))$ against the example BAT, where δ denotes $(\delta_{coffee} \parallel \delta_{EXO})^p$ and we have a small queue of size $k = 2$. Let $rC(p)$ stand for $requestCoffee(p)$, $sR(p)$ for $selectRequest(p)$ and ψ for $\neg\exists p(Occ(sR(p)))$. The initial set of labels in CHECKEG$[\delta, \psi]$ then is

$$X = \{\langle v_0, \psi\rangle,\ \langle v_1, \psi\rangle,\ \langle v_2, \psi\rangle\}$$

Starting the first iteration of the loop, we have to determine the preimage of the current X. $\textsc{Pre}[\mathcal{G}_\delta, X]$ consists of seven different labels, three of those for v_0, corresponding to the edges leaving that node:

$\langle v_0, \mathcal{R}[\exists a.Exo(a) \wedge \mathit{Poss}(a) \wedge [a]\psi]\rangle$
$\langle v_0, \mathcal{R}[\mathit{Empty}(queue) \wedge \mathit{Poss}(wait) \wedge [wait]\psi]\rangle$
$\langle v_0, \mathcal{R}[\exists p.\neg\mathit{Empty}(queue) \wedge \mathit{Poss}(sR(p)) \wedge [sR(p)]\psi]\rangle$

When applying regression and using the unique names assumption for actions, we obtain:

$\langle v_0, \exists a.\exists p(a = rC(p)) \wedge \neg\mathit{Full}(queue) \wedge \neg\exists p(a = sR(p))\rangle$
$\langle v_0, \mathit{Empty}(queue) \wedge \top \wedge \top\rangle$
$\langle v_0, \exists p.\neg\mathit{Empty}(queue) \wedge \neg\mathit{Empty}(queue) \wedge \bot\rangle$

These can be further simplified to

$$\langle v_0, \neg\mathit{Full}(queue)\rangle,\ \langle v_0, \mathit{Empty}(queue)\rangle,\ \langle v_0, \bot\rangle$$

Using all seven labels in $\textsc{Pre}[\mathcal{G}_\delta, X]$, we can determine $\textsc{And}(X, \textsc{Pre}[\mathcal{G}_\delta, X])$, which contains

$$\langle v_0, \psi \wedge \neg\mathit{Full}(queue)\rangle, \langle v_0, \psi \wedge \mathit{Empty}(queue)\rangle, \langle v_0, \psi \wedge \bot\rangle$$

and four other labels. This set becomes the new X. Iterating such steps, we will observe convergence after the fifth iteration, ending up with X consisting of

$\langle v_0, \psi \wedge \mathit{Empty}(queue)\rangle$
$\langle v_1, \psi \wedge \neg\mathit{HoldingCoffee} \wedge \mathit{Empty}(queue)\rangle$
$\langle v_2, \psi \wedge \mathit{HoldingCoffee} \wedge \mathit{Empty}(queue)\rangle$

Therefore, $\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle \boldsymbol{G}\psi] = \psi \wedge \mathit{Empty}(queue)$, which is clearly entailed by $\Sigma_0 \cup \Sigma_{\text{UNA}}$. The result is also intuitively clear: A trace without $sR(P)$ is possible just in case no $sR(p)$ has yet occurred and there are no initial requests in the queue that have to be served.

The example also nicely demonstrates that for the convergence of our method, the state space does not necessarily have to be finite in a strict sense; it suffices when it is finitely *representable*. Remember that the state of the $queue$ fluent (with an assumed length of 2) is given by a term of the form $list(n_1, n_2)$, where the n_i can be any of the countably infinitely many standard names of sort object. For the purpose of verification, it however does not matter *which* requests the queue actually contains, but only *how many*, with only finitely many possible quantities. With the expressive power of first-order quantification, this simple form of abstraction can easily be achieved. In the example, the three abstract states of $queue$ were

(Q1) $list(e, e)$
(Q2) $\exists x_1.\ queue = list(x_1, e)$
(Q3) $\exists x_1 \exists x_2.\ (x_1 \neq e) \wedge (x_2 \neq e) \wedge queue = list(x_1, x_2)$

where (Q1) was abbreviated as $\mathit{Empty}(queue)$ and (Q3) as $\mathit{Full}(queue)$. Had we propositionalized the theory using a fixed number of standard names as possible arguments in order to obtain finiteness, the state space would unnecessarily blow up exponentially. While it is easy to come up with an example where the algorithm does not terminate anymore, we believe that many practical applications share the properties of our example and that it therefore pays off to adopt the

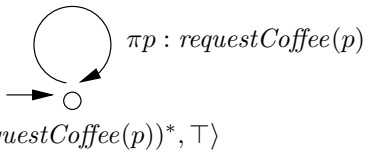

Figure 2: Characteristic graph of a simple iteration

higher expressivity of a first-order logic to avoid state space explosion. A worthwhile direction of future work would be the identification of classes of syntactically restricted basic action theories for which termination can be guaranteed, thus putting our claim on more solid grounds.

Verifying Terminating Programs

Finally, we can adapt the idea of the algorithm to define an extended form of Reiter's (2001) regression that is able to deal with arbitrary $[\delta]$ and $\langle\delta\rangle$ operators even in the presence of iteration, though it is of course again not guaranteed to terminate. Intuitively, when we want to decide whether for a fluent formula α, $\langle\delta\rangle\alpha$ is entailed, then we look for some trace that makes valid transitions *until* a final state is reached. For that purpose, we have a procedure $\textsc{Regr}[\delta, \alpha]$ that is similar to $\textsc{CheckEU}[\delta, \top, \alpha]$, but where the initialization of X also considers the nodes' finality conditions:

Procedure 4 $\textsc{Regr}[\delta, \alpha]$

$X := \textsc{Final}[\mathcal{G}_\delta, \alpha]$; $X_O := \textsc{Label}[\mathcal{G}_\delta, \top]$;
while $X \neq X_O$ **do**
  $X_O := X$;
  $X := X \cup \textsc{Pre}[\mathcal{G}_\delta, X]$;
end while
return $\textsc{InitLabel}[\mathcal{G}_\delta, X]$

The labels encoding the final configurations are:

$$\textsc{Final}[\mathcal{G}, \varphi] \stackrel{def}{=} \{\langle v, \psi \wedge \varphi\rangle \mid v = \langle \delta', \psi\rangle,\ \psi \neq \bot\}$$

The following theorem is an adaptation of the previous one:

Theorem 6 *Let α be a fluent formula. If the computation of $\textsc{Regr}[\delta, \alpha]$ terminates, it is a fluent formula and*

$$\Sigma \cup \Sigma_{UNA} \models \langle\delta\rangle\alpha \text{ iff } \Sigma_0 \cup \Sigma_{UNA} \models \textsc{Regr}[\delta, \alpha].$$

Again, convergence is only given when the set of possible traces is finitely representable by some formula. Let us illustrate the algorithm briefly. The most interesting case is how it works in the case of an iteration, therefore consider the simple program $\delta = (\pi p.requestCoffee(p))^*$, which nondeterministically sends a finite number of coffee requests. The corresponding characteristic graph is depicted in Figure 2. Assume that we want to know whether there is some possible execution of this δ such that the queue will be full afterwards, that is whether $\Sigma \models \langle\delta\rangle Full(queue)$. The set X gets initialized to $\{\top \wedge Full(queue)\}$, which is equivalent to $\{(Q3)\}$. Since the regression of (Q3) can be simplified to (Q2), we obtain $\{(Q3), (Q2)\}$ in the first iteration of the algorithm and similarly $\{(Q3), (Q2), (Q1)\}$ in the second one. The regression of (Q1) is equivalent to $\bot$ and the algorithm converges; the resulting formula is the disjunction of (Q1)-(Q3), which is obviously entailed by Σ_0.

Related Work

$\mathcal{ESG}$ draws its inspiration from a number of other formalisms, most importantly dynamic logic (Harel, Kozen, & Tiuryn 2000), process logic (Harel, Kozen, & Parikh 1982; Harel, Kozen, & Tiuryn 2000), and temporal logics[5] (Emerson 1990). Despite the commonalities, there are some differences. In particular, although there are first-order extensions of these logics, their expressiveness is often restricted. Atomic actions, for instance, are usually assumed to be assignments of the form $x := t$, where x is a variable, and t is some term (Bohn *et al.* 1998). On the other hand, in the tradition of the situation calculus, $\mathcal{ESG}$ allows to define pre- and postconditions of actions by arbitrary formulas.

There is a wide spectrum of ongoing research in the area of model-checking. The fundamental techniques for temporal logics, from which, in the case of $\mathcal{CTL}$, our algorithm is clearly inspired, are discussed in (Clarke, Grumberg, & Peled 1999). Larger state spaces are tackled by resorting to symbolic model-checking, using an OBDD-based representation (Burch *et al.* 1990). Infinite state spaces are usually reduced to the finite case by means of predicate abstraction techniques (Chechik, Devereux, & Gurfinkel 2001; Bogunovi & Pek 2006). Many software model checkers are available, famous examples being NuSMV (Cimatti *et al.* 2002) and SPIN (Holzmann 2003).

Regarding the verification of Golog programs, besides the work of (De Giacomo, Ternovska, & Reiter 1997) already discussed in the introduction, a model checker for non-terminating ConGolog programs using encodings in the μ-calculus is reported in (Kalantari & Ternovska 2002). Liu (2002) presents a proof system in the style of Hoare logic for proving properties of terminating Golog programs. Kelly and Pearce (2007) present an algorithm for checking the persistence of formulas, which can be viewed as a special case of our method considering the fact that $\Box\alpha$ is equivalent to $\neg\langle\langle any^\omega\rangle\rangle\top\ \boldsymbol{U}\ \neg\alpha$. Verification of agent programs has naturally been studied for other action languages as well, e.g. in (de Boer *et al.* 2007), but is mostly limited to the propositional case.

Conclusion

Based on an existing modal fragment of the situation calculus we proposed a new logic which allows us to formulate properties of non-terminating Golog programs in a natural way. In particular, we are able to express conditions that need to hold during the execution of a program, where these conditions may use operators known from branching-time logic. We further presented a method to automatically verify such conditions which uses ideas from model checking and regression-based reasoning. We adapted it to verify postconditions of terminating programs. We remark that, since (Lakemeyer & Levesque 2004) also provide a regression method for formulas containing *Know* operators, both the logic and all results presented in this paper can be extended to the epistemic case with few modifications. In the

[5]Statements of temporal logics were first introduced into the situation calculus by (Gabaldon 2004), however only for finite sequences of actions.

future, we would like to explore verification methods for larger subclasses of the language.

Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft under grant La 747/14-1. We also thank the anonymous reviewers for their helpful comments.

Appendix: Characteristic Graphs

For a program δ, the *characteristic graph of* δ is given by $\mathcal{G}_\delta = \langle V, E, v_0\rangle$, where

- V is a finite set of vertices of the form $\langle \delta', \phi\rangle$, where δ' is a program and ϕ is a formula;
- E is a finite set of edges of the form $v_1 \overset{\pi\vec{x}:t/\psi}{\rightarrow} v_2$, where $v_1, v_2 \in V$ are nodes, $\vec{x}$ is a possibly empty list of variables, t is a term of sort action and ψ is a formula;
- $v_0 \in E$ is the initial node.

$\mathcal{G}_\delta$ is defined by induction on the structure of δ:

- $\delta = t$:
  $v_0 = \langle t, \bot\rangle$;
  $V = \{\langle t, \bot\rangle,\ \langle nil, \top\rangle\};\ E = \{\langle t, \bot\rangle \overset{t}{\rightarrow} \langle nil, \top\rangle\}$.
- $\delta = \varphi?$:
  $v_0 = \langle nil, \varphi\rangle;\ V = \{\langle nil, \varphi\rangle\};\ E = \emptyset$.
- $\delta = \delta_1; \delta_2$:
  Let $\mathcal{G}_{\delta_1} = \langle V_1, E_1, v_o^1\rangle$ and $\mathcal{G}_{\delta_2} = \langle V_2, E_2, v_o^2\rangle$, where $v_0^i = \langle \delta_i, \varphi_0^i\rangle$. Then
  $v_0 = \langle \delta_1; \delta_2, \varphi_0^1 \wedge \varphi_0^2\rangle$;
  $V = \{\langle \delta_1'; \delta_2, \varphi_1' \wedge \varphi_0^2\rangle \mid \langle \delta_1', \varphi_1'\rangle \in V_1\} \cup V_2$;
  $E = \{\langle \delta_1'; \delta_2, \varphi_1' \wedge \varphi_0^2\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1''; \delta_2, \varphi_1'' \wedge \varphi_0^2\rangle \mid \langle \delta_1', \varphi_1'\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1'', \varphi_1''\rangle \in E_1\} \cup \{\langle \delta_2', \varphi_2'\rangle \overset{\pi\vec{x}:t/\phi_2}{\rightarrow} \langle \delta_2'', \varphi_2''\rangle \mid \langle \delta_2', \varphi_2'\rangle \overset{\pi\vec{x}:t/\phi_2}{\rightarrow} \langle \delta_2'', \varphi_2''\rangle \in E_2\} \cup \{\langle \delta_1'; \delta_2, \varphi_1' \wedge \varphi_0^2\rangle \overset{\pi\vec{x}:t/\phi_2 \wedge \varphi_1'}{\rightarrow} \langle \delta_2', \varphi_2'\rangle \mid \langle \delta_2, \varphi_0^2\rangle \overset{\pi\vec{x}:t/\phi_2}{\rightarrow} \langle \delta_2', \varphi_2'\rangle \in E_2,\ \varphi_1' \neq \bot\}$.
  The idea here is to essentially leave $\mathcal{G}_{\delta_1}$ as it is, but where each δ_1' turns into $\delta_1'; \delta_2$ and each termination condition is augmented by φ_0^2, the condition under which δ_2 is final. $\mathcal{G}_{\delta_2}$ then remains unchanged and we have copies of edges originally going out of v_0^2 also at all $\delta_1'; \delta_2$ nodes, with the additional constraint that δ_1' is final, i.e. that φ_1' holds.
- $\delta = (\delta_1 | \delta_2)$:
  Let $\mathcal{G}_{\delta_1} = \langle V_1, E_1, v_o^1\rangle$ and $\mathcal{G}_{\delta_2} = \langle V_2, E_2, v_o^2\rangle$, where $v_0^i = \langle \delta_i, \varphi_0^i\rangle$. Then
  $v_0 = \langle (\delta_1 | \delta_2), \varphi_0^1 \vee \varphi_0^2\rangle$;
  $V = \{v_0\} \cup V_1 \cup V_2$;
  $E = \{v_0 \overset{\pi\vec{x}:t/\phi}{\rightarrow} v_i \mid v_0^i \overset{\pi\vec{x}:t/\phi}{\rightarrow} v_i \in E_i\} \cup E_1 \cup E_2$.
  Here we have a copy of each $\mathcal{G}_{\delta_i}$. From the new initial node, there are copies of the outgoing edges of both v_0^i. Thus, the first transition corresponds to the commitment to either δ_1 or δ_2, and all subsequent transitions have to be in the chosen subprogram.
- $\delta = \pi y.\delta_1$:
  Let $\mathcal{G}_{\delta_1} = \langle V_1, E_1, v_o^1\rangle$, where $v_0^1 = \langle \delta_1, \varphi_0^1\rangle$. Then
  $v_0 = \langle \pi y.\delta_1, \exists y.\varphi_0^1\rangle$;
  $V = \{v_0\} \cup V_1$;
  $E = \{v_0 \overset{\pi y, \pi\vec{x}:t/\phi_1}{\rightarrow} v_1' \mid v_0^1 \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} v_1' \in E_1\} \cup E_1$.
  In this case we extend a copy of $\mathcal{G}_{\delta_1}$ with a new initial node whose outgoing edges contain the additional choice for y and whose termination condition also requires the existence of an appropriate y.
- $\delta = (\delta_1 \| \delta_2)$:
  Let $\mathcal{G}_{\delta_1} = \langle V_1, E_1, v_o^1\rangle$ and $\mathcal{G}_{\delta_2} = \langle V_2, E_2, v_o^2\rangle$, where $v_0^i = \langle \delta_i, \varphi_0^i\rangle$. Then
  $v_0 = \langle (\delta_1 \| \delta_2), \varphi_0^1 \wedge \varphi_0^2\rangle$;
  $V = \{\langle (\delta_1' \| \delta_2'), \varphi_1' \wedge \varphi_2'\rangle \mid \langle \delta_1', \varphi_1'\rangle \in V_1, \langle \delta_2', \varphi_2'\rangle \in V_2\}$;
  $E =$
  $\{\langle (\delta_1' \| \delta_2'), \varphi_1' \wedge \varphi_2'\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle (\delta_1'' \| \delta_2'), \varphi_1'' \wedge \varphi_2'\rangle \mid \langle \delta_1', \varphi_1'\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1'', \varphi_1''\rangle \in E_1,\ \langle \delta_2', \varphi_2'\rangle \in V_2\} \cup$
  $\{\langle (\delta_1' \| \delta_2'), \varphi_1' \wedge \varphi_2'\rangle \overset{\pi\vec{x}:t/\phi_2}{\rightarrow} \langle (\delta_1' \| \delta_2''), \varphi_1' \wedge \varphi_2''\rangle \mid \langle \delta_2', \varphi_2'\rangle \overset{\pi\vec{x}:t/\phi_2}{\rightarrow} \langle \delta_2'', \varphi_2''\rangle \in E_2,\ \langle \delta_1', \varphi_1'\rangle \in V_1\}$.
  The set of vertices here is something like the Cartesian product of the nodes of the $\mathcal{G}_{\delta_i}$. Edges are such that either a transition in δ_1 or one in δ_2 is taken, and the other program remains unchanged.
- $\delta = (\delta_1)^*$:
  Let $\mathcal{G}_{\delta_1} = \langle V_1, E_1, v_o^1\rangle$. Then
  $v_0 = \langle (\delta_1)^*, \top\rangle$;
  $V = \{v_0\} \cup \{\langle \delta_1'; (\delta_1)^*, \varphi_1'\rangle \mid \langle \delta_1', \varphi_1'\rangle \in V_1\}$;
  $E = \{v_0 \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1'; (\delta_1)^*, \varphi_1'\rangle \mid v_0^1 \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1', \varphi_1'\rangle \in E_1\} \cup \{\langle \delta_1'; (\delta_1)^*, \varphi_1'\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1''; (\delta_1)^*, \varphi_1''\rangle \mid \langle \delta_1', \varphi_1'\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1'', \varphi_1''\rangle \in E_1\} \cup \{\langle \delta_1'; (\delta_1)^*, \varphi_1'\rangle \overset{\pi\vec{x}:t/\phi_1 \wedge \varphi_1''}{\rightarrow} v_0 \mid \langle \delta_1', \varphi_1'\rangle \overset{\pi\vec{x}:t/\phi_1}{\rightarrow} \langle \delta_1'', \varphi_1''\rangle \in E_2,\ \varphi_1'' \neq \bot\}$.
  Here, we introduce a new initial node, which has $\top$ as the termination condition, and which has copies of the leaving edges of the initial node of $\mathcal{G}_{\delta_1}$. The new v_0 furthermore has ingoing edges for each transition that leads to a node at which program execution may terminate.

To obtain simpler graphs, it is safe to drop any nodes and edges that are unreachable (in the graph theoretic sense) from v_0. In addition, we can identify both $(nil; \delta)$ and $(\delta; nil)$ with δ, which further simplifies the resulting graph.

Appendix: Proofs

Theorem 1 Let α be a sentence of $\mathcal{ES}$ without epistemic operators. Then $\models_{\mathcal{ES}} \alpha$ iff $\models_{\mathcal{ESG}} \alpha$.

Proof: We show that for each α of classical $\mathcal{ES}$,

$$w,z,u \models_{\mathcal{ESG}} \alpha \text{ iff } w,z,u \models_{\mathcal{ES}} \alpha.$$

The claim is obvious for the cases $H(\vec{t})$, $V(\vec{t})$, $(t_1 = t_2)$, $\alpha \wedge \beta$, $\neg\alpha$, $\forall v.\alpha$, and $\forall V.\alpha$ since their definition did not change. The cases $[t]\alpha$ and $\Box\alpha$ remain.

- For an atomic action term t, the only possible transition step is $(t,z) \underset{w,u}{\rightarrow} (nil, z\cdot n)$, where $n = |t|^w_z$. Further $(nil, z\cdot n) \in \mathcal{F}^{w,u}$, therefore $\|t\|^w_u(z) = \{n\}$. Then we have: $w,z,u \models_{\mathcal{ESG}} [t]\alpha$ iff (by definition) for all $z' \in \|t\|^w_u(z)$, $w, z\cdot z', u \models_{\mathcal{ESG}} \alpha$ iff $w, z\cdot n, u \models_{\mathcal{ESG}} \alpha$, where $n = |t|^w_z$ iff (by induction) $w, z\cdot n, u \models_{\mathcal{ES}} \alpha$, where $n = |t|^w_z$ iff (by definition) $w,z,u \models_{\mathcal{ES}} [t]\alpha$.
- First note that $(\pi a.a, z') \underset{w,u}{\rightarrow} (nil, z'\cdot n)$ for any sequence z' and any action name n, therefore $((\pi a.a)^\omega, z) \underset{w,u}{\rightarrow} (nil; (\pi a.a)^\omega, z\cdot n)$ for any n and $(nil; (\pi a.a)^\omega, z') \underset{w,u}{\rightarrow} (nil; (\pi a.a)^\omega, z'\cdot n)$ for any z' and n. Further obviously $((\pi a.a)^\omega, z) \notin \mathcal{F}^{w,u}$ and $(nil; (\pi a.a)^\omega, z') \notin \mathcal{F}^{w,u}$, hence $\|(\pi a.a)^\omega\|^w_u(z) = \Pi$ (the set of all infinite paths). Then we have $w,z,u \models_{\mathcal{ESG}} \Box\alpha$ iff (by definition) $w,z,u \models_{\mathcal{ESG}} [\![(\pi a.a)^\omega]\!]\neg(\top \boldsymbol{U} \neg\alpha)$ iff (by definition) for all $\tau \in \|(\pi a.a)^\omega\|^w_u(z)$, $w,z,\tau,u \models_{\mathcal{ESG}} \neg(\top \boldsymbol{U} \neg\alpha)$ iff (by the above) for all infinite π, $w,z,\pi,u \models_{\mathcal{ESG}} \neg(\top \boldsymbol{U} \neg\alpha)$ iff (by definition) for all infinite π, $w,z,\pi,u \not\models_{\mathcal{ESG}} (\top \boldsymbol{U} \neg\alpha)$ iff (by definition) for all infinite π and all z' with $\pi = z'\cdot\pi'$, $w, z\cdot z', \pi', u \not\models_{\mathcal{ESG}} \neg\alpha$ iff (by definition) for all infinite π' and all z', $w, z\cdot z', \pi', u \models_{\mathcal{ESG}} \alpha$ iff (since $\alpha \in \mathcal{ES}$ is a situation formula, i.e. independent from π') for all z', $w, z\cdot z', u \models_{\mathcal{ESG}} \alpha$ iff (by induction) for all z', $w, z\cdot z', u \models_{\mathcal{ES}} \alpha$ iff (by definition) $w,z,u \models_{\mathcal{ES}} \Box\alpha$. ∎

The next theorem requires the following lemma:

Lemma 7 *For any w, z, u, δ and δ':*

1. $z' \in \|\delta;\delta'\|^w_u(z)$ *iff* $z' = z''\cdot z'''$
   where $z'' \in \|\delta\|^w_u(z)$ *and* $z''' \in \|\delta'\|^w_u(z\cdot z'')$.
2. $z' \in \|\delta^*\|^w_u(z)$ *iff* $z' = z_0\cdot z_1\cdots z_k$, $k \geq 0$, $z_0 = \langle\,\rangle$,
   where $z_{i+1} \in \|\delta\|^w_u(z\cdot z_0\cdots z_i)$ *for* $1 \leq i < k$.

Proof: These can be proved by an induction on the length of z' and on k, respectively. ∎

Theorem 3 Let δ be a program without $\|$ and α be a formula. Let δ^p be δ with every atomic action t replaced by $Poss(t)?;t$. Then $\models Do(\delta,\alpha) \equiv \langle\delta^p\rangle\alpha$.

Proof: We prove $w,z,u \models Do(\delta,\alpha)$ iff $w,z,u \models \langle\delta^p\rangle\alpha$ by induction on the structure of δ. Because of limited space, we only consider the most interesting case δ^*. Obviously $(\delta^*)^p = (\delta^p)^*$.

"⇒": Let $w,z,u \models Do(\delta^*,\alpha)$. Therefore for all u' with $u' \sim_P u$, if $w,z,u' \models \Box(\alpha \supset P)$ and $w,z,u' \models \Box(Do(\delta,P) \supset P)$, then $u'[P,z] = 1$. Now let u_0 be a variable map with $u_0[P,z'] = 1$ iff there is $z'' \in \|(\delta^*)^p\|^w_u(z')$ with $w, z'\cdot z'', u \models \alpha$ and which is otherwise like u. Then clearly $u_0 \sim_P u$. Moreover, we will show that

(i) $w,z,u_0 \models \Box(\alpha \supset P)$
(ii) $w,z,u_0 \models \Box(Do(\delta,P) \supset P)$

From this it follows that $u_0[P,z] = 1$, which by assumption implies that there is $z'' \in \|(\delta^*)^p\|^w_u(z)$ with $w, z\cdot z', u \models \alpha$, therefore $w,z,u \models \langle(\delta^*)^p\rangle\alpha$.

To prove (i), let $w, z\cdot z', u_0 \models \alpha$. Assuming that α does not contain free occurrences of P, also $w, z\cdot z', u \models \alpha$. Since $\langle\,\rangle \in \|(\delta^*)^p\|^w_u(z\cdot z')$ for any z', we have $u_0[P, z\cdot z'] = 1$.

For (ii), let $w, z\cdot z', u_0 \models Do(\delta,P)$. By induction, $w, z\cdot z', u_0 \models \langle\delta^p\rangle P$. This means there is $z'' \in \|\delta^p\|^w_{u_0}(z\cdot z')$ and $u_0[P, z\cdot z'\cdot z''] = 1$. Since P does not occur freely in δ, also $z'' \in \|\delta^p\|^w_u(z\cdot z')$. By definition of u_0, there is now some $z''' \in \|(\delta^*)^p\|^w_u(z\cdot z'\cdot z'')$ with $w, z\cdot z'\cdot z''\cdot z''', u \models \alpha$. By using Lemma 7 item 2 twice, we obtain $z''\cdot z''' \in \|(\delta^p)^*\|^w_u(z\cdot z')$, therefore $u_0[P, z\cdot z'] = 1$.

"⇐": Let $w,z,u \models \langle(\delta^*)^p\rangle\alpha$ and

(i) $w,z,u' \models \Box(\alpha \supset P)$ and
(ii) $w,z,u' \models \Box(Do(\delta,P) \supset P)$

for some u' with $u' \sim_P u$. We have to show that $u'[P,z] = 1$. By induction, (ii) is equivalent to

(iii) $w,z,u' \models \Box(\langle\delta^p\rangle P \supset P)$.

By assumption, $z' \in \|(\delta^*)^p\|^w_u(z)$ and $w, z\cdot z', u \models \alpha$. By Lemma 7 item 2, $z' = z_0\cdot z_1\cdots z_k$, where $z_0 = \langle\,\rangle$ and $z_{i+1} \in \|\delta^p\|^w_u(z\cdot z_0\cdots z_i)$ for $1 \leq i < k$. Using (i) we obtain $u'[P, z\cdot z'] = 1$. Applying (iii) repeatedly, we get $u'[P, z\cdot z_0\cdots z_i] = 1$ for all $0 \leq i \leq k$. In particular, $u'[P,z] = 1$. ∎

The following lemma, needed for the next theorem, gives a formal characterization of how characteristic graphs encode the transition relation and finality of program configurations:

Lemma 8 *Let $\mathcal{G}_\delta = \langle V, E, v_0\rangle$. Then*

1. $v_0 = \langle\delta,\phi\rangle$ *for some ϕ.*
2. $\langle\delta,z\rangle \underset{w,u}{\overset{*}{\rightarrow}} \langle\delta',z\cdot z'\rangle$ *iff:*
   $z' = n_0\cdots n_{k-1}$ *($k \geq 0$) and there are $v_0,\ldots,v_k \in V$ and standard names $\vec{n}_0,\ldots,\vec{n}_{k-1}$ such that*
   - $v_i = \langle\delta_i,\phi_i\rangle$ *for* $1 \leq i \leq k$;
   - $v_i \overset{\pi\vec{x}_i:t_i/\psi_i}{\rightarrow} v_{i+1}$ *for* $1 \leq i < k$;
   - $n_i = |(t_i)^{\vec{x}_{i-1}\cdots\vec{x}_o}_{\vec{n}_{i-1}\cdots\vec{n}_0}|^{z\cdot n_0\cdots n_{i-1}}_w$ *for* $1 \leq i \leq k$;
   - $w, z\cdot n_0\cdots n_{i-1}, u \models (\psi_i)^{\vec{x}_{i-1}\cdots\vec{x}_o}_{\vec{n}_{i-1}\cdots\vec{n}_0}$ *for* $1 \leq i \leq k$;
   - $\delta' = (\delta_k)^{\vec{x}_{k-1}\cdots\vec{x}_o}_{\vec{n}_{k-1}\cdots\vec{n}_0}$.
3. $\langle\delta,z\rangle \underset{w,u}{\overset{*}{\rightarrow}} \langle\delta',z\cdot z'\rangle$ *and* $\langle\delta',z\cdot z'\rangle \in \mathcal{F}^{w,u}$ *iff the conditions listed in the previous item hold and additionally*
   - $w, z\cdot z', u \models (\phi_k)^{\vec{x}_{k-1}\cdots\vec{x}_o}_{\vec{n}_{k-1}\cdots\vec{n}_0}$.

Proof: The first item can be shown by an induction on the structure of δ. Item 2 and 3 follow by an outer induction on δ and an inner one on the length of the computation. ∎

Theorem 5 Let $\varphi \in \mathcal{ESG}_{\mathcal{CTL}}$. If the computation of $\mathcal{C}[\varphi]$ wrt Σ terminates, it is a fluent formula and

$$\Sigma \cup \Sigma_{\text{UNA}} \models \varphi \quad \text{iff} \quad \Sigma_0 \cup \Sigma_{\text{UNA}} \models \mathcal{C}[\varphi].$$

Proof: (Sketch) Let $w \models \Sigma$. We prove that $w, z \models \varphi$ iff $w, z \models \mathcal{C}[\varphi]$ by induction on the structure of φ. It can also be proved that $\mathcal{C}[\varphi]$ is a fluent formula, thus $w, \langle\,\rangle \models \mathcal{C}[\varphi]$ iff $w', \langle\,\rangle \models \mathcal{C}[\varphi]$ for any w' agreeing with w on the fluents' values in the initial situation, as defined by Σ_0.

The cases without $\langle\!\langle\delta\rangle\!\rangle$ quantifiers are easy. The intuition for the correctness of the other cases is the following.

$w, z \models \langle\!\langle\delta\rangle\!\rangle \boldsymbol{X}\varphi$ iff there is some first step n in some execution trace of δ such that $w, z \cdot n \models \varphi$; the latter by induction being the same as $w, z \cdot n \models \mathcal{C}[\varphi]$. By Lemma 8, this holds iff $\mathcal{G}_\delta$ contains some edge, leaving v_0, and labelled $\pi\vec{x} : t/\psi$, where n is the denotation of $t^{\vec{x}}_{\vec{n}}$ in w, $w, z \models \psi^{\vec{x}}_{\vec{n}}$ for some $\vec{n}$ and $w, z \cdot n \models \mathcal{C}[\varphi]$. Using the correctness of regression from (Lakemeyer & Levesque 2004), Lemma 4, this is the same as when $w, z \models \mathcal{R}[\vec{x}.\psi \wedge [t]\mathcal{C}[\varphi]]$, which is one of the disjuncts in $\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle \boldsymbol{X}\varphi]$.

$w, z \models \langle\!\langle\delta\rangle\!\rangle \boldsymbol{G}\varphi$ iff $w, z \cdot z' \models \varphi$ (thus also $w, z \cdot z' \models \mathcal{C}[\varphi]$ by induction) for all prefixes z' of some infinite execution trace π of δ. By Lemma 8, z' corresponds to some path $t_1, \ldots, t_k$ in $\mathcal{G}_\delta$, starting in v_0. We can express that $\mathcal{C}[\varphi]$ holds along this path by a fluent formula using regression:

$$w, z \models \mathcal{C}[\varphi] \wedge \mathcal{R}[t_1, \mathcal{C}[\varphi] \wedge \mathcal{R}[t_2, \mathcal{C}[\varphi] \wedge \cdots \mathcal{R}[t_k, \mathcal{C}[\varphi]] \cdots]]$$

It is these formulas that CHECKEG computes iteratively in the set of labels, and when the procedure terminates, it means that the finitely many resulting disjuncts in $\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle \boldsymbol{G}\varphi]$ suffice to capture all of them.

$w, z \models \langle\!\langle\delta\rangle\!\rangle \phi\, \boldsymbol{U}\, \psi$ iff for some execution trace τ of δ and some prefix z' of τ, $w, z \cdot z' \models \psi$ and for all prefixes z'' of z', $w, z \cdot z'' \models \phi$. The property can be expressed for z' of length up to k, again using Lemma 8 and regression, as a formula of the form

$$\begin{aligned} w, z \models & \mathcal{C}[\psi] \vee \mathcal{C}[\phi] \wedge \mathcal{R}[t_1, \mathcal{C}[\psi] \vee \\ & \mathcal{C}[\phi] \wedge \mathcal{R}[t_2, \mathcal{C}[\psi] \vee \cdots \mathcal{C}[\phi] \wedge \mathcal{R}[t_k, \mathcal{C}[\psi]] \cdots]] \end{aligned}$$

which is basically what CHECKEU computes for increasing k until convergence.∎

References

Bogunovi, N., and Pek, E. 2006. Model checking procedures for infinite state systems. In *Proc. of ECBS-06*, 419–425.

Bohn, J.; Damm, W.; Grumberg, O.; Hungar, H.; and Laster, K. 1998. First-order-CTL model checking. In *Proc. of the 18th Conf. on Foundations of Software Technology and Theoretical Computer Science*, 283–294.

Burch, J.; Clarke, E.; McMillan, K.; Dill, D.; and Hwang, L. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1–33.

Chechik, M.; Devereux, B.; and Gurfinkel, A. 2001. Model-checking infinite state-space systems with fine-grained abstractions using SPIN. *Lecture Notes in Computer Science* 2057:16–36.

Cimatti, A.; Clarke, E.; Giunchiglia, E.; Giunchiglia, F.; Pistore, M.; Roveri, M.; Sebastiani, R.; and Tacchella, A. 2002. NuSMV 2: An OpenSource tool for symbolic model checking. In *Proc. of CAV 2002*, 359–364.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model Checking*. MIT Press.

de Boer, F. S.; Hindriks, K. V.; van der Hoek, W.; and Meyer, J.-J. C. 2007. A verification framework for agent programming with declarative goals. *J. Applied Logic* 5(2):277–302.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1–2):109–169.

De Giacomo, G.; Ternovska, E.; and Reiter, R. 1997. Non-terminating processes in the situation calculus. In *Working Notes of "Robots, Softbots, Immobots: Theories of Action, Planning and Control", AAAI'97 Workshop*.

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B): formal models and semantics*. MIT Press. 995–1072.

Gabaldon, A. 2004. Precondition control and the progression algorithm. In *Proc. of KR-04*, 634–643.

Harel, D.; Kozen, D.; and Parikh, R. 1982. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences* 25(2):144–170.

Harel, D.; Kozen, D.; and Tiuryn, J. 2000. *Dynamic Logic*. MIT Press.

Holzmann, G. J. 2003. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional.

Kalantari, L., and Ternovska, E. 2002. A model checker for verifying ConGolog programs. In *AAAI-02*, 953–954.

Kelly, R. F., and Pearce, A. R. 2007. Property persistence in the situation calculus. In *Proc. of IJCAI-07*, 1948–1953.

Lakemeyer, G., and Levesque, H. 2004. Situations, si! situation terms, no! In *Proc. of KR-2004*, 516–526.

Lakemeyer, G., and Levesque, H. J. 2005. Semantics for a useful fragment of the situation calculus. In *Proc. of IJCAI-05*, 490–496.

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

Liu, Y. 2002. A hoare-style proof system for robot programs. In *Proc. of AAAI-02*, 74–79.

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. New York: American Elsevier. 463–502.

Pelov, N., and Ternovska, E. 2005. Reducing inductive definitions to propositional satisfiability. In *Proc. of ICLP-05*, 221–234.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.