On Decidable Verification of Non-terminating Golog Programs

Jens Claßen, Martin Liebenberg and **Gerhard Lakemeyer**
Dept. of Computer Science
RWTH Aachen University
52056 Aachen
Germany
{classen,liebenberg,gerhard}@kbsg.rwth-aachen.de

Abstract

The high-level action programming language GOLOG has proven to be a useful means for the control of autonomous agents such as mobile robots. Usually, such agents perform open-ended tasks, and their control programs are hence non-terminating. Before deploying such a program to the robot, it is often desirable if not crucial to verify that it meets certain requirements, preferably by means of an automated method. For this purpose, Claßen and Lakemeyer recently introduced algorithms for the verification of temporal properties of non-terminating GOLOG programs, based on the first-order modal Situation Calculus variant $\mathcal{ES}$, and regression-based reasoning. However, while GOLOG's high expressiveness is a desirable feature, it also means that their verification procedures cannot be guaranteed to terminate in general. In this paper, we address this problem by showing that, for a relevant subset, the verification of non-terminating GOLOG programs is indeed decidable, which is achieved by means of three restrictions. First, we use the $\mathcal{ES}$ variant of a decidable two-variable fragment of the Situation Calculus that was introduced by Gu and Soutchanski. Second, we have to restrict the GOLOG program to contain ground action only. Finally, we consider special classes of successor state axioms, namely the context-free ones and those that only admit local effects.

1 Introduction

The GOLOG [De Giacomo *et al.*, 2000; Levesque *et al.*, 1997] family of high-level action programming languages and its underlying logic, the Situation Calculus [McCarthy and Hayes, 1969; Reiter, 2001], have proven to be useful means for the control of autonomous agents such as mobile robots [Burgard *et al.*, 1999]. Usually, the task of such an agent is open-ended, i.e. there is no predefined goal or terminal state that the agent tries to reach, but (at least ideally) the robot works indefinitely, and its corresponding control program is hence *non-terminating*.

As a simple example, consider a mobile robot whose task it is to remove dirty dishes from certain locations in an office on request. A program for this robot might look like this:

loop : **while** $(\exists x.OnRobot(x))$ **do**
$\pi x.unload(x)$
endWhile;
$\pi y.goToRoom(y)$;
while $(\exists x.DirtyDish(x,y))$ **do**
$\pi x.load(x,y)$
endWhile;
$goToKitchen$

We assume that the robot is initially in the kitchen, its home base. There is an infinite loop, where during each iteration the robot first unloads all dishes it carries, then selects a room in the office building, goes to this room, loads all dirty dishes in this room, and returns to the kitchen. Here, $DirtyDish(x,y)$ should be read as "dirty dish x is in room y" and $load(x,y)$ as "load dish x in room y." During the execution of the program, people can send requests indicating that there is a dirty dish in a certain room (not shown here).

Before actually deploying such a program on the robot and executing it in the real world, it is often desirable if not crucial to verify that it meets certain requirements such as safety, liveness and fairness properties, for example that "every request will eventually be served by the robot" or whether "it is possible that no request is ever served." Moreover, the verification is preferably done using an *automated* method, since manual, meta-theoretic proofs such as done in [De Giacomo *et al.*, 1997] tend to be tedious and prone to errors. For this purpose, Claßen and Lakemeyer [2008] recently proposed the logic $\mathcal{ESG}$, an extension of the modal Situation Calculus variant $\mathcal{ES}$ [Lakemeyer and Levesque, 2010] by constructs that allow to express temporal properties of GOLOG programs. They moreover provided algorithms for the verification of a subset of the logic that resembles the branching-time temporal logic CTL. Their methods rely on regression-based reasoning and a newly introduced graph representation of GOLOG programs to do a systematic exploration of a program's configuration space within a fixpoint approximation loop. While the procedures are proven to be sound, no general guarantee can be given for termination.

There are two reasons for this. On the one hand, to detect the convergence of the fixpoint loop, the algorithm has to check the equivalence of formulas that encode reachable

program configurations. Since these may be arbitrary first-order formulas, this already amounts to an undecidable problem. Furthermore, even if all equivalence checks can be performed in finite time (or if we assume a first-order oracle), the fixpoint computation may never converge.

A straight-forward approach to remedy this problem is to restrict the input language such that verification becomes decidable, as done for instance by Baader, Liu and ul Mehdi [2010]. Instead of using the full first-order expressiveness of the Situation Calculus or $\mathcal{ES}$, they resort to a dynamic extension [Baader *et al.*, 2005] of the decidable description logic $\mathcal{ALC}$ [Baader *et al.*, 2003] to represent pre- and postconditions of actions, where properties are expressed by a variant of LTL over $\mathcal{ALC}$ assertions [Baader *et al.*, 2008]. Second, they encode programs by finite Büchi automata instead of the fully-fledged GOLOG language. They could show that under these restrictions, verification reduces to a decidable reasoning task within the underlying description logic.

Although this is a step in the right direction, it requires harsh restrictions in terms of expressiveness. In particular, representing programs through Büchi automata loses one important feature of GOLOG, namely the possibility to include test conditions in the form of formulas. Moreover, representing action effects within $\mathcal{ALC}$ only allows for basic STRIPS-style addition and deletion of literals. While decidability can obviously not be achieved without any restrictions on the input languages, the high, first-order expressiveness of the Situation Calculus and GOLOG is typically considered a desirable feature and the reason why these languages were chosen in the first place, and one would rather give up as little as possible of it. Ideally, we could do the verification within the very same expressive formalism and with the same reasoning tools that are used for the actual control of the agent.

In this paper, we show that this is indeed possible for a relevant subset of the formalism. In order to achieve decidability for first-order equivalence checks, we rely on results by Gu and Soutchanski [Gu and Soutchanski, 2010] who presented a modified version of the Situation Calculus built using a two-variable fragment of first-order logic and a variant of Reiter's regression operator such that the reasoning task of projection becomes decidable. Since (as we will see later) this is in itself not sufficient to guarantee the termination of the overall verification method, we moreover consider special classes of successor state axioms from the literature to be used in the agent's basic action theory, namely the *context-free* [Lin and Reiter, 1997] ones as well as those that only admit *local effects* [Liu and Levesque, 2005], and prove that under these prerequisites, a termination guarantee can be given for the verification methods if we restrict the GOLOG program to contain ground actions only. Note that our restrictions allow us to retain a great deal of (first-order) expressiveness, including test conditions in programs and conditional action effects.

The remainder of this paper is organized as follows. In the following section, we briefly recapitulate the logic $\mathcal{ESG}$. Section 3 then presents the verification procedures we consider. In Section 4, we present a decidable subset of $\mathcal{ES}$ that is similar to Gu and Soutchanski's two-variable Situation Calculus fragment. Sections 6 and 5 contain the main results of this paper, namely the decidability of the verification methods for the above mentioned classes of basic action theories. Section 7 reviews related work before we conclude in Section 8.

2 The Logic $\mathcal{ESG}$

2.1 Syntax

The language is a first-order modal dialect with equality and sorts of type *object* and *action*. It includes countably infinitely many standard names for each sort. Also included are both fluent and rigid predicate and function symbols. Fluents vary as the result of actions, but rigids do not. We assume that the fluents include unary predicates $Poss$ and Exo, whose argument is of type action and which will be used to specify when an action is executable or exogenous, respectively.

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

Program constructs are logical (built-in) symbols with a fixed meaning. The programs we consider are the ones admitted by the following grammar:

$$\delta ::= \quad t \mid \alpha? \mid \delta_1;\delta_2 \mid \delta_1|\delta_2 \mid \pi x.\delta \mid \delta_1\|\delta_2 \mid \delta^* \quad (1)$$

That is we allow primitive actions t (where t can be any action term), tests $\alpha?$ (where α is a static situation formula as defined below), sequence, nondeterministic branching, nondeterministic choice of argument, concurrency, and nondeterministic iteration. Moreover, conditionals and loops can be defined in terms of the above constructs:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{def}{=} [\phi?;\delta_1] \mid [\neg\phi?;\delta_2] \quad (2)$$

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \stackrel{def}{=} [\phi?;\delta]^*;\neg\phi? \quad (3)$$

The infinite loop, also abbreviated as δ^ω, is further given by:

$$\textbf{loop } \delta \textbf{ endLoop} \stackrel{def}{=} \textbf{while } \top \textbf{ do } \delta \textbf{ endWhile} \quad (4)$$

Formulas come in two different "flavours", as given by the following definitions:

Definition 1 (Situation Formulas)**.** The *situation formulas* are the least set such that

- if $t_1, \ldots, t_k$ are terms and P is a (fluent or rigid) k-ary predicate symbol, then $P(t_1, \ldots, t_k)$ is a situation formula;
- if t_1 and t_2 are terms, then $(t_1 = t_2)$ is a situation formula;
- if α and β are situation formulas, x is a variable, P is a (fluent or rigid) predicate symbol, δ is a program, and ϕ is a trace formula (defined below), then $\alpha \wedge \beta$, $\neg\alpha$, $\forall x.\alpha$, $\forall P.\alpha$, $\Box\alpha$, $[\delta]\alpha$ ("α holds after executing δ"), and $[\![\delta]\!]\phi$ ("temporal property ϕ holds for all executions of δ") are situation formulas.

Situation formulas, roughly, express properties wrt a given situation and possibly future situations, that is, the formulas may include references to future situations by means of $[\,\cdot\,]$, $\Box$, or $[\![\,\cdot\,]\!]$. Moreover, let $\langle\delta\rangle\alpha = \neg[\delta]\neg\alpha$ and $\langle\!\langle\delta\rangle\!\rangle\varphi =$

$\neg[\![\delta]\!]\neg\varphi$. A situation formula α is called *fluent* when it contains no $[\cdot]$, no $\Box$, and no $[\![\cdot]\!]$ operators, nor any of the special fluents $Poss$ and Exo. It is called *static* when it contains no $[\cdot]$, no $\Box$ and no $[\![\cdot]\!]$ operators. It is *bounded* when it contains no $\Box$ operators, no $[\![\cdot]\!]$ operators, and $[t]$ operators only in case the argument is an action term t.

Definition 2 (Trace Formulas)**.** The *trace formulas* are the least set such that

- if α is a situation formula, then it is also a trace formula;
- if ϕ and ψ are trace formulas and x is a variable, then $\phi \wedge \psi$, $\neg\phi$, $\forall x.\phi$, $\boldsymbol{X}\phi$ ("ϕ holds in the next situation"), and $\phi\ \boldsymbol{U}\ \psi$ ("ϕ holds until ψ holds") are also trace formulas.

Trace formulas, as the name suggests, are used to talk about *traces* of situations, i.e. finite or infinite sequences of actions. We will use them for representing the temporal properties of program execution traces. In addition to the usual abbreviations, we also have $\boldsymbol{F}\phi = (\top\ \boldsymbol{U}\ \phi)$ ("eventually ϕ") and $\boldsymbol{G}\phi = \neg\boldsymbol{F}\neg\phi$ ("always ϕ").

2.2 Semantics

Terms and formulas are interpreted with respect to *worlds*:

Definition 3 (Worlds)**.** Let $\mathcal{P}_O$ and $\mathcal{P}_A$ denote the set of primitive terms of sort object, and action, respectively, where a *primitive term* is of the form $f(n_1, \ldots, n_k)$, where all the n_i are standard names. Similarly, let $\mathcal{P}_F$ be the set of all primitive formulas $F(n_1, \ldots, n_k)$. Moreover, let $\mathcal{N}_O$ and $\mathcal{N}_A$ be the sets of all standard names of sort object and action, respectively, $\mathcal{N} = \mathcal{N}_O \cup \mathcal{N}_A$, and $\mathcal{Z} = {\mathcal{N}_A}^*$ the set of all finite sequences of action names. A world w then is a mapping

- $w : \mathcal{P}_O \times \mathcal{Z} \to \mathcal{N}_O$ and
- $w : \mathcal{P}_A \times \mathcal{Z} \to \mathcal{N}_A$ and
- $w : \mathcal{P}_F \times \mathcal{Z} \to \{0, 1\}$

satisfying the following constraints:

Rigidity: If R is a rigid function or predicate symbol, then for all $z, z' \in \mathcal{Z}$, $w[R(n_1, \ldots, n_k), z] = w[R(n_1, \ldots, n_k), z']$.

Unique names for actions: If $g(\vec{n})$ and $g'(\vec{n}')$ are two distinct primitive action terms, then for all $z \in \mathcal{Z}$, $w[g(\vec{n}), z] \neq w[g'(\vec{n}'), z]$.

Let $\mathcal{W}$ denote the set of all worlds.

A world thus maps primitive terms to co-referring standard names of the corresponding sort, and primitive formulas to truth values. The rigidity constraint ensures that rigid symbols do not take different values in different situations, as expected. We further incorporate the unique names assumption for actions into our logic's semantics, as opposed to the Situation Calculus where this is typically asserted axiomatically.

Definition 4 (Denotation of Terms)**.** Given a ground term t, a world w, and an action sequence $z \in \mathcal{Z}$, we define $|t|_w^z$ (read: "the co-referring standard name for t given w and z") by:

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

To interpret programs, we need the notion of *program configurations*. A configuration $\langle z, \delta\rangle$ consists of an action sequence z and a program δ, where intuitively z is the history of actions that have already been performed, while δ is the program that remains to be executed. Then we define the possible transitions and finality of programs as follows:

Definition 5 (Program Transition Semantics)**.** The transition relation $\xrightarrow{w}$ among configurations, given a world w, is the least set satisfying

1. $\langle z, t\rangle \xrightarrow{w} \langle z \cdot p, \top?\rangle$, if $p = |t|_w^z$;
2. $\langle z, \delta_1; \delta_2\rangle \xrightarrow{w} \langle z \cdot p, \gamma; \delta_2\rangle$, if $\langle z, \delta_1\rangle \xrightarrow{w} \langle z \cdot p, \gamma\rangle$;
3. $\langle z, \delta_1; \delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$, if $\langle z, \delta_1\rangle \in \mathcal{F}^w$ and $\langle z, \delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$;
4. $\langle z, \delta_1|\delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$, if $\langle z, \delta_1\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$ or $\langle z, \delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$;
5. $\langle z, \pi x.\delta\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$, if $\langle z, \delta_n^x\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$ for some $n \in \mathcal{N}_x$;
6. $\langle z, \delta^*\rangle \xrightarrow{w} \langle z \cdot p, \gamma; \delta^*\rangle$, if $\langle z, \delta\rangle \xrightarrow{w} \langle z \cdot p, \gamma\rangle$;
7. $\langle z, \delta_1\|\delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta'\|\delta_2\rangle$, if $\langle z, \delta_1\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$;
8. $\langle z, \delta_1\|\delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta_1\|\delta'\rangle$, if $\langle z, \delta_2\rangle \xrightarrow{w} \langle z \cdot p, \delta'\rangle$.

Above, $\mathcal{N}_x$ means the set of all standard names of the same sort as x, and δ_n^x refers to δ with x replaced by n.

The set of final configurations $\mathcal{F}^w$ of a world w is the smallest set such that

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

Temporal properties that we express by situation formulas refer to *traces*, as defined below.

Definition 6 (Traces)**.** A *trace* is a possibly infinite sequence of $action$ standard names. As a notational convention, we use τ to denote arbitrary traces, z for finite ones and π for infinite ones. Let $\Pi = {\mathcal{N}_A}^\omega$ be the set of all infinite traces, and $\mathcal{T} = \mathcal{Z} \cup \Pi$ the set of all traces. Furthermore, let $\pi^{(i)}$ stand for the finite sequence that consists of the first i elements of π, where $\pi^{(0)}$ is the empty sequence $\langle\rangle$.

We can now define the traces admitted by a given program:

Definition 7 (Traces of Programs)**.** Let $\xrightarrow{w}{}^*$ denote the reflexive and transitive closure of $\xrightarrow{w}$. Given a world w and a finite sequence of $action$ standard names z, the set of *traces* $\|\delta\|_w^z$ of a program δ is the set

$$\{z' \in \mathcal{Z} \mid \langle z, \delta\rangle \xrightarrow{w}{}^* \langle z \cdot z', \delta'\rangle,\ \langle z \cdot z', \delta'\rangle \in \mathcal{F}^w\} \cup$$
$$\{\pi \in \Pi \mid \langle z, \delta\rangle \xrightarrow{w} \langle z \cdot \pi^{(1)}, \delta_1\rangle \xrightarrow{w} \langle z \cdot \pi^{(2)}, \delta_2\rangle \xrightarrow{w} \cdots$$
$$\text{where for all } i \geq 0,\ \langle z \cdot \pi^{(i)}, \delta_i\rangle \notin \mathcal{F}^w\}$$

In words, the finite traces admitted by some δ given w and z are those that correspond to a finite number of transitions by means of which a final configuration is reachable. Its infinite traces are given by all infinite sequences of transitions that never visit any final configuration.

Situation and Trace Formulas

We are now equipped to define the truth of formulas:

Definition 8 (Truth of Situation and Trace Formulas)**.** Given a world $w \in \mathcal{W}$ and a situation formula α, we define $w \models \alpha$ as $w, \langle\rangle \models \alpha$, where for any $z \in \mathcal{Z}$:

1. $w, z \models F(t_1, \ldots, t_k)$ iff $w[F(n_1, \ldots, n_k), z] = 1$, where $n_i = |t_i|^z_w$;
2. $w, z \models (t_1 = t_2)$ iff n_1 and n_2 are identical, where $n_i = |t_i|^z_w$;
3. $w, z \models \alpha \wedge \beta$ iff $w, z \models \alpha$ and $w, z \models \beta$;
4. $w, z \models \neg\alpha$ iff $w, z \not\models \alpha$;
5. $w, z \models \forall x.\alpha$ iff $w, z \models \alpha^x_n$ for all $n \in \mathcal{N}_x$;
6. $w, z \models \Box\alpha$ iff $w, z \cdot z' \models \alpha$ for all $z' \in \mathcal{Z}$;
7. $w, z \models [\delta]\alpha$ iff for all finite $z' \in \|\delta\|^z_w$, $w, z \cdot z' \models \alpha$;
8. $w, z \models [\![\delta]\!]\phi$ iff for all $\tau \in \|\delta\|^z_w$, $w, z, \tau \models \phi$.

The truth of trace formulas ϕ is defined as follows for $w \in \mathcal{W}$, $z \in \mathcal{Z}$, and traces $\tau \in \mathcal{T}$:

1. $w, z, \tau \models \alpha$ iff $w, z \models \alpha$, if α is a situation formula;
2. $w, z, \tau \models \phi \wedge \psi$ iff $w, z, \tau \models \phi$ and $w, z, \tau \models \psi$;
3. $w, z, \tau \models \neg\phi$ iff $w, z, \tau \not\models \phi$;
4. $w, z, \tau \models \forall x.\phi$ iff $w, z, \tau \models \phi^x_n$ for all $n \in \mathcal{N}_x$;
5. $w, z, \tau \models \boldsymbol{X}\phi$ iff $\tau = p \cdot \tau'$ and $w, z \cdot p, \tau' \models \phi$;
6. $w, z, \tau \models \phi \; \boldsymbol{U} \; \psi$ iff there is z' such that $\tau = z' \cdot \tau'$ and $w, z \cdot z', \tau' \models \psi$ and for all $z'' \neq z'$ with $z' = z'' \cdot z'''$, $w, z \cdot z'', z''' \cdot \tau' \models \phi$.

2.3 Basic Action Theories and Regression

Definition 9. A *basic action theory* (BAT) $\Sigma = \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}} \cup \Sigma_{\text{exo}}$ describes the dynamics of a specific application domain, where

1. Σ_0, *the initial database*, is a finite set of fluent sentences describing the initial state of the world.
2. Σ_{pre} is a *precondition axiom* of the form $\Box Poss(a) \equiv \pi$, with π being a fluent formula, whose only free variable is a, describing precisely the conditions under which a is a possible action.
3. Σ_{post} is a finite set of *successor state axioms* (SSAs), one for each fluent relevant to the application domain, incorporating Reiter's [Reiter, 2001] solution to the frame problem, and encoding the effects the actions have on the different fluents. The SSA for a fluent predicate has the form $\Box[a]F(\vec{x}) \equiv \gamma^+_F \vee F(\vec{x}) \wedge \neg\gamma^-_F$, whereas the one for a functional fluent is of the form $\Box[a]f(\vec{x}) = y \equiv \gamma^+_f \vee (f(\vec{x}) = y) \wedge \neg\exists y'\gamma^{+\,y}_{f\;y'}$, where γ^+_F and γ^-_F are fluent formulas with free variables $\vec{x}$, and γ^+_f one with free variables among $\vec{x}$ and y.
4. Σ_{exo} is the *exogenous actions axiom*, having the form $\Box Exo(a) \equiv \chi$, where χ is again a fluent formula with the free variable a. It is used to express the necessary and sufficient conditions under which an action is exogenous, i.e. not controlled by the agent, but by "nature".

Our algorithm relies on the equivalent of Reiter's *regression* operator $\mathcal{R}[\alpha]$. Roughly, the idea is that, whenever we encounter a subformula of the form $[t]F(\vec{x})$ within α, where t is an action term, we may substitute it by the right-hand side of the successor state axiom of the fluent 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, $Poss(t)$ and $Exo(t)$ are replaced by the right-hand sides of the corresponding axiom. By iteratively applying such substitutions, we eventually get a fluent formula that describes exactly the conditions on the initial situation under which the original, non-static formula holds:

Theorem 10. *Let Σ be a BAT and α a bounded sentence. Then $\mathcal{R}[\alpha]$, the regression of α, is a fluent sentence and $\Sigma \models \alpha$ iff $\Sigma_0 \models \mathcal{R}[\alpha]$.*

3 Verification in $\mathcal{ESG}$

We encode the space of reachable program configurations by a *characteristic graph* $\mathcal{G}_\delta = \langle v_0, V, E\rangle$ for a given program δ. The nodes V in such a graph are of the form $\langle\delta', \phi\rangle$, denoting the remaining program of a current run and the condition under which execution may terminate there. v_0 is the initial node. 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. Due to lack of space, we omit the formal definition of characteristic graphs and refer the interested reader to [Claßen and Lakemeyer, 2008]. Figure 1 shows the graph corresponding to $\delta_{robot}\|\delta_{exo}$, where δ_{robot} denotes the control program presented in the introduction and δ_{exo} is the encoding of exogenous actions. Here it consists simply of the $requestDDR(x, y)$, which should be read as "requesting the removal of dirty dish x from room y." The nodes are $v_0 = \langle\delta_{robot}\|\delta_{exo}, \bot\rangle$ and $v_1 = \langle(\delta_1; \delta_{robot})\|\delta_{exo}, \bot\rangle$, where δ_1 is the program

$$(\pi x.DirtyDish(x, y)?; load(x, y))^*;$$
$$\neg\exists x.DirtyDish(x, y)?; goToKitchen.$$

The verification algorithms work on *labels* of the characteristic graph, where a label is given by $\langle v, \psi\rangle$ with $v \in V$ and ψ being a fluent formula. Intuitively, it represents all program configurations corresponding to v as well as all worlds w and action name sequences z satisfying ψ. A *labelling* is then given by a set of labels, one for each node of the graph. We need the following operations on labellings, as formalized below: Initial labelling with a formula, conjunction and disjunction of labellings, extraction of the label formula from

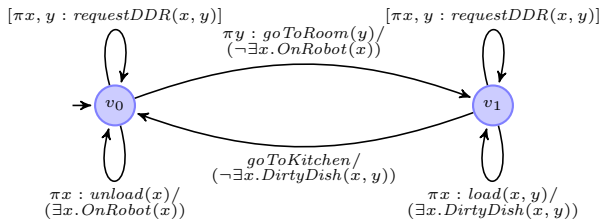

Figure 1: Characteristic graph for the robot example

the initial node, and the pre-image of a labelling:

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

$$L_1 \textsc{ And } L_2 \stackrel{def}{=} \{\langle v, \psi_1 \wedge \psi_2\rangle \mid \langle v, \psi_1\rangle \in L_1, \langle v, \psi_2\rangle \in L_2\}$$

$$L_1 \textsc{ Or } L_2 \stackrel{def}{=} \{\langle v, \psi_1 \vee \psi_2\rangle \mid \langle v, \psi_1\rangle \in L_1, \langle v, \psi_2\rangle \in L_2\}$$

$$\textsc{InitLabel}[\langle V, E, v_0\rangle, L] \stackrel{def}{=} \psi \text{ such that } \langle v_0, \psi\rangle \in L$$

$$\textsc{Pre}[\langle V, E, v_0\rangle, L] \stackrel{def}{=} \{\langle v, \textsc{Pre}[v, L]\rangle \mid v \in V\}$$

where

$$\textsc{Pre}[v, L] \stackrel{def}{=} \bigvee \{\mathcal{R}[\exists \vec{x}.\phi \wedge [t]\psi] \mid v \xrightarrow{\pi\vec{x}:t/\phi} v' \in E,\ \langle v', \psi\rangle \in L\}.$$

Roughly, the pre-image of a label gives us a description of the predecessor configuration of that label. (Note the use of regression to eliminate the action term t.)

The verification algorithm works on a CTL-like fragment of $\mathcal{ESG}$:

$$\varphi ::= (t_1 = t_2) \mid F(\vec{t}) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\varphi \mid \langle\!\langle\delta\rangle\!\rangle \boldsymbol{G}\varphi \mid \langle\!\langle\delta\rangle\!\rangle \varphi\, \boldsymbol{U}\, \varphi \tag{5}$$

where we assume that δ is a non-terminating program of the form ${\delta_1}^\omega \| \cdots \| {\delta_k}^\omega$. The algorithm then applies the following transformation.

Definition 11. Let Σ be a BAT and φ a formula according to (5). Then $\mathcal{C}[\varphi]$, the *verification transformation* of φ wrt Σ, is inductively defined by

1. $\mathcal{C}[(t_1 = t_2)] = (t_1 = t_2)$;
2. $\mathcal{C}[F(\vec{t})] = F(\vec{t})$;
3. $\mathcal{C}[\varphi_1 \wedge \varphi_2] = \mathcal{C}[\varphi_1] \wedge \mathcal{C}[\varphi_2]$;
4. $\mathcal{C}[\neg\varphi] = \neg\mathcal{C}[\varphi]$;
5. $\mathcal{C}[\exists x.\varphi] = \exists x.\mathcal{C}[\varphi]$;
6. $\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle \boldsymbol{G}\, \varphi] = \textsc{CheckEG}[\delta, \mathcal{C}[\varphi]]$;
7. $\mathcal{C}[\langle\!\langle\delta\rangle\!\rangle \varphi\, \boldsymbol{U}\, \psi] = \textsc{CheckEU}[\delta, \mathcal{C}[\varphi], \mathcal{C}[\psi]]$.

The procedure for the case of the "always" operator $\boldsymbol{G}$ is as follows:

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

$L' := \textsc{Label}[\mathcal{G}_\delta, \bot]$;
$L := \textsc{Label}[\mathcal{G}_\delta, \varphi]$;
while $L \not\equiv L'$ **do**
  $L' := L$;
  $L := L' \textsc{ And } \textsc{Pre}[\mathcal{G}_\delta, L']$;
end while
return $\textsc{InitLabel}[\mathcal{G}_\delta, L]$

The while loop is exited once $L \equiv L'$ holds, defined as follows:

$$L_1 \equiv L_2 \text{ iff for all } v \text{ with } \langle v, \psi_1\rangle \in L_1 \text{ and } \langle v, \psi_2\rangle \in L_2, \models \psi_1 \equiv \psi_2.$$

A similar procedure is used for the "until" operator $\boldsymbol{U}$:

Procedure 2 $\textsc{CheckEU}[\delta, \varphi_1, \varphi_2]$

$L' := \textsc{Label}[\mathcal{G}_\delta, \bot]$;
$L := \textsc{Label}[\mathcal{G}_\delta, \top]$;
while $L \not\equiv L'$ **do**
  $L' := L$;
  $L := L' \textsc{ And } \textsc{Pre}[\mathcal{G}_\delta, L']$;
end while
$L' := \textsc{Label}[\mathcal{G}_\delta, \top]$;
$L := \textsc{Label}[\mathcal{G}_\delta, \varphi_2] \textsc{ And } L$;
while $L \not\equiv L'$ **do**
  $L' := L$;
  $L := L' \textsc{ Or } (\textsc{Label}[\mathcal{G}_\delta, \varphi_1] \textsc{ And } \textsc{Pre}[\mathcal{G}_\delta, L'])$;
end while
return $\textsc{InitLabel}[\mathcal{G}, L]$

The algorithm is sound in the following sense:

Theorem 12 ([Claßen and Lakemeyer, 2008])**.** *Let Σ be a BAT, δ a program and φ a fluent sentence. Then if the computation of $\mathcal{C}[\varphi]$ terminates, it is a fluent sentence and $\Sigma \models \varphi$ iff $\Sigma_0 \models \mathcal{C}[\varphi]$.*

4 Decidability

The algorithms presented in the previous section cannot be guaranteed to terminate for two reasons. On the one hand, equivalence checks over first-order formulas as applied in the conditions of the while loops are in general undecidable. On the other hand, even if all equivalence checks terminate, the fixpoint approximation loops may never converge.

As for the first source of non-termination, we can exploit results by Gu and Soutchanski [2010] who present a two-variable fragment of the Situation Calculus for which the projection problem (solved by means of regression) is decidable. Here we capture this fragment as a subset of the situation formulas of $\mathcal{ESG}$. We refer to this sublanguage as $\mathcal{ES}^2$.

Definition 13. $\mathcal{ES}^2$ is the subset of situation formulas according to Definition 1 that do not contain any $[\![\cdot]\!]$ operators and where $[t]$ operators are restricted to action terms t. In addition the following constraints are satisfied:

- there are no object terms other than the variables x and y or rigid constant symbols;
- all action function symbols have at most two arguments;
- fluents have at most two arguments.

In $\mathcal{ES}^2$ a *regressable* formula has to be bounded and its action terms have to be ground. Furthermore, the regression operator $\mathcal{R}$ is modified such that by means of appropriate substitutions, no new variable is introduced in the process of regression. For details, the interested reader is referred to Gu and Soutchanski's article [2010]. We then have that projection is decidable in $\mathcal{ES}^2$:

Theorem 14. *Let α be a regressable sentence of $\mathcal{ES}^2$ without standard names and Σ a BAT in $\mathcal{ES}^2$. Then $\Sigma \models \alpha$ is decidable.*

Proof. (Sketch) The proof idea for this theorem is to map $\mathcal{ES}^2$ to the decidable fragment $\mathcal{L}_{SC}^{DL}$ introduced by Gu and Soutchanski. We use a similar reduction as Lakemeyer and Levesque [2010] who embed $\mathcal{ES}$ in the original Situation Calculus. Thus, because $\mathcal{L}_{SC}^{DL}$ is decidable, $\mathcal{ES}^2$ is decidable too. □

Resorting to a decidable base logic is unfortunately not sufficient to also eliminate the second source of non-termination of the verification algorithms. To see why, consider a simple BAT with the single fluent F whose successor state axiom is $\Box[a]F(x) \equiv \exists y.F(y) \wedge S(x,y)$ (where S is rigid). Let δ be the program **loop** : t for some ground action t and $\langle\!\langle\delta\rangle\!\rangle \mathbf{G}F(c)$ the sentence to verify, for some constant c. The characteristic graph of δ has only one node v_0 and one edge from v_0 to v_0 with the label t. Applying Procedure 1, we get the following label sets L in subsequent iterations:

$$
\begin{aligned}
L_0 &= \{\langle v_0, F(c)\rangle\},\\
L_1 &= \{\langle v_0, F(c) \wedge \exists y.F(y) \wedge S(c,y)\rangle\},\\
L_2 &= \{\langle v_0, F(c) \wedge [\exists y.F(y) \wedge S(c,y)] \wedge\\
&\qquad \exists y.\exists x.F(x) \wedge S(y,x) \wedge S(c,y)\rangle\},\\
L_3 &= \{\langle v_0, F(c) \wedge [\exists y.F(y) \wedge S(c,y)] \wedge\\
&\qquad \exists y.\exists x.F(x) \wedge S(y,x) \wedge S(c,y)\\
&\qquad \exists y.\exists x.[\exists y.F(y) \wedge S(x,y)] \wedge S(y,x) \wedge S(c,y)\rangle\},\\
&\ldots
\end{aligned}
$$

Obviously, none of the formulas in this sequence is equivalent to its predecessor, and hence the algorithm never converges. Note also that we remain within $\mathcal{ES}^2$ due to re-using the two variable symbols x and y.

5 Decidability with context-free BATs

The first possibility is to restrict oneself to BATs with context-free SSAs:

Definition 15 (Context-free Successor State Axioms [Lin and Reiter, 1997])**.** A successor state axiom is *context-free* if its effect conditions, $\gamma_F^+(\vec{x},a)$ and $\gamma_F^-(\vec{x},a)$, contain no fluents (but maybe rigids). A BAT is context-free if each successor state axiom is context-free.

In order to ensure our prerequisite that formulas to be regressed only contain ground terms, we prohibit the usage of the non-deterministic pick operator π. Note that this is not such a harsh restriction as this still allows to use a "pseudo-pick" that quantifies over a finite domain of constants:

$$\pi x : \{c_1, \ldots, c_k\}.\delta \stackrel{def}{=} \delta^x_{c_1} | \cdots | \delta^x_{c_k}.$$

We then have the following theorem:

Theorem 16. *If Σ is a context-free BAT and δ a program without pick operators, Procedures 1 and 2 will terminate.*

Proof. (Sketch) The central property for the proof of this theorem is the following:

$$\mathcal{R}\big[[t_i]\mathcal{R}\big[[t_n]\ldots[t_i]\ldots[t_1]\varphi\big]\big] \equiv \mathcal{R}\big[[t_n]\ldots[t_i]\ldots[t_1]\varphi\big].$$

That is, regressing φ through the same ground action multiple times produces an equivalent result as only regressing once through that action. Because the program (and thus the characteristic graph) has only finitely many actions all of which are ground, there are ony finitely many such sequences of actions to consider. We then exploit the fact that the bodies of all loops in the procedures are monotone, i.e. they either always produce a subsumer of the previous label formula, or a subsumed one. Hence, eventually the label set converges. □

6 Decidability with local-effect BATs

The other option to ensure termination is to restrict ourselves to BATs whose SSAs are local-effect:

Definition 17 (Local-effect Successor State Axioms [Liu and Levesque, 2005])**.** A successor state axiom is *local-effect* if both $\gamma_F^+(\vec{x},a)$ and $\gamma_F^-(\vec{x},a)$ are disjunctions of formulas of the form $\exists\vec{z}[a = A(\vec{y}) \wedge \phi(\vec{y})]$, where A is an action function, $\vec{y}$ contains $\vec{x}$, $\vec{z}$ is the remaining variables of $\vec{y}$. ϕ is called a *context formula* and contains no quantifiers. A BAT is local-effect if each successor state axiom is local-effect.

Then we have:

Theorem 18. *If Σ is a local-effect BAT and δ a program without pick operators, Procedures 1 and 2 will terminate.*

Proof. (Sketch) The proof of this theorem relies on the fact that we have only finitely many action terms in the graph (all of which are ground) and only finitely many fluents in the action theory. Furthermore, instantiating a successor state axiom by a ground action during regression yields a quantifier-free formula. Since there are only finitely many such instantiations and only finitely many edge condition formulas in the graph, we get finitely many equivalence classes of possible label formulas. Using the monotonicity argument again, termination is guaranteed. □

Example 19. Recapitulating the example from the beginning, we show here a verification run for a local-effect BAT. Fortunately, the example is already in the two-variable fragment. We only need to change the program slightly by replacing the pick operators by the pseudo-picks. Then we have the

following successor state axioms:

$$\Box[a]\,DirtyDish(x,y) \equiv a = requestDDR(x,y) \vee DirtyDish(x,y) \wedge \neg[a = load(x,y)]$$
$$\Box[a]\,OnRobot(x) \equiv \exists y.\, a = load(x,y) \vee OnRobot(x) \wedge \neg[a = unload(x)].$$

We omit other fluents like the location of the robot for simplicity. Additionally, we have the following precondition axiom:

$$\Box Poss(a) \equiv [\exists x,y.\, a = requestDDR(x,y)] \vee [\exists x,y.\, a = load(x,y)] \vee [\exists x.\, a = unload(x)].$$

The only exogenous actions axiom is

$$\Box Exo(a) \equiv \exists x,y.\, a = requestDDR(x,y).$$

Finally, the following is the GOLOG program δ'_{robot} with pseudo-picks, where d_i is a constant for a dish and r_i for a room:

loop : **while** $(\exists x.OnRobot(x))$ **do**
$\pi x : \{d_1,d_2,d_3\}.unload(x)$
endWhile;
$\pi y : \{r_1,r_2\}.goToRoom(y)$;
while $(\exists x.DirtyDish(x,y))$ **do**
$\pi x : \{d_1,d_2,d_3\}.load(x,y)$
endWhile;
$goToKitchen$

We now want to verify the following formula φ for δ'_{robot}: $\neg\exists x,y.\langle\!\langle \delta'_{robot}\rangle\!\rangle \boldsymbol{G} DirtyDish(x,y)$. This means there cannot be any infinite run of δ'_{robot} where some dish in some room remains dirty forever. The algorithm starts with $\mathcal{C}[\varphi]$ resulting in $\neg\exists x,y.\textsc{CheckEG}[\delta'_{robot}, DirtyDish(x,y)]$. Then Procedure 1 starts with the following label set:

$$L_0 = \{\langle v_0, DirtyDish(x,y)\rangle, \langle v_1, DirtyDish(x,y)\rangle\}$$

$\textsc{Pre}[v_0, L_0]$
$\equiv \mathcal{R}[(\neg\exists x.OnRobot(x)) \wedge [goToRoom(r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\neg\exists x.OnRobot(x)) \wedge [goToRoom(r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_1,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_2,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_3,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_1,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_2,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_3,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.OnRobot(x)) \wedge [unload(d_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.OnRobot(x)) \wedge [unload(d_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.OnRobot(x)) \wedge [unload(d_3)]DirtyDish(x,y)]$
$\equiv (\neg\exists x.OnRobot(x)) \wedge DirtyDish(x,y) \vee$
$x = d_1 \wedge y = r_1 \vee DirtyDish(x,y) \vee$
$x = d_2 \wedge y = r_1 \vee DirtyDish(x,y) \vee$
$x = d_3 \wedge y = r_1 \vee DirtyDish(x,y) \vee$
$x = d_1 \wedge y = r_2 \vee DirtyDish(x,y) \vee$
$x = d_2 \wedge y = r_2 \vee DirtyDish(x,y) \vee$
$x = d_3 \wedge y = r_2 \vee DirtyDish(x,y) \vee$
$(\exists x.OnRobot(x)) \wedge DirtyDish(x,y)$
$\equiv x = d_1 \wedge y = r_1 \vee x = d_2 \wedge y = r_1 \vee$
$x = d_3 \wedge y = r_1 \vee x = d_1 \wedge y = r_2 \vee$
$x = d_2 \wedge y = r_2 \vee x = d_3 \wedge y = r_2 \vee$
$DirtyDish(x,y)$

$\textsc{Pre}[v_1, L_0]$
$\equiv \mathcal{R}[(\neg\exists x.DirtyDish(x,r_1)) \wedge [goToKitchen]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\neg\exists x.DirtyDish(x,r_2)) \wedge [goToKitchen]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_1,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_2,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_3,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_1,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_2,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[[requestDDR(d_3,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.DirtyDish(x,r_1)) \wedge [load(d_1,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.DirtyDish(x,r_1)) \wedge [load(d_2,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.DirtyDish(x,r_1)) \wedge [load(d_3,r_1)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.DirtyDish(x,r_2)) \wedge [load(d_1,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.DirtyDish(x,r_2)) \wedge [load(d_2,r_2)]DirtyDish(x,y)] \vee$
$\mathcal{R}[(\exists x.DirtyDish(x,r_2)) \wedge [load(d_3,r_2)]DirtyDish(x,y)]$
$\equiv x = d_1 \wedge y = r_1 \vee x = d_2 \wedge y = r_1 \vee x = d_3 \wedge y = r_1 \vee$
$x = d_1 \wedge y = r_2 \vee x = d_2 \wedge y = r_2 \vee x = d_3 \wedge y = r_2 \vee$
$DirtyDish(x,y)$

$L_1 = L_0$ AND $\textsc{Pre}[\mathcal{G}_\delta, L_0]$
$= \{\langle v_0, DirtyDish(x,y)\rangle, \langle v_1, DirtyDish(x,y)\rangle\}$

Now, $L_0 \equiv L_1$, i.e. the algorithm terminates and returns $\neg\exists x,y.DirtyDish(x,y)$. Thus, there is no run with some dish forever remaining dirty in some room iff there is no dirty dish initially. Intuitively, this is correct because $\boldsymbol{G}\phi$ means that ϕ persists to hold during the *entire* run, including the initial situation. Therefore, only if a dish is dirty initially it may happen that it never gets cleaned, namely when the robot never visits the corresponding room. Note that excluding this from happening would still allow the case where a dirty dish occurs at a later time of the run (due to some *requestDDR* action) and never gets cleaned from that moment on.

7 Related Work

The verification of non-terminating GOLOG programs was first discussed by De Giacomo, Ternovska and Reiter [1997], but only in the form of manual, meta-theoretic proofs, where properties were expressed using μ-calculus fixpoint formulas instead of temporal modalities. The $\mathcal{ESG}$ language and the

automated verification methods used in this paper were introduced by Claßen and Lakemeyer [2008] and later extended to a larger subset [2010]. However, they proved their algorithms only to be sound, but could not give a general termination guarantee. De Giacomo, Lespérance and Pearce [2010] applied the idea of verifying GOLOG programs through iterative fixpoint approximations using characteristic graphs in the context of games and multi-agent systems, where properties are expressed in Alternating-Time Temporal Logic. De Giacomo, Lespérance and Patrizi [2012] define the class of bounded action theories, for which they show that the verification of a certain class of first-order μ-calculus temporal properties is decidable.

8 Conclusion

In this paper, we showed that the problem of verifying non-terminating GOLOG programs is indeed decidable for a relevant subset of the formalism, which was achieved by means of three restrictions. First, we used the $\mathcal{ES}$ variant of a decidable two-variable fragment of the Situation Calculus as introduced by Gu and Soutchanski. Second, we have to restrict the GOLOG program to contain ground action only. Finally, we considered special classes of successor state axioms, namely the context-free ones and those that only admit local effects. Interesting lines of future work would be to come up with a solution for re-introducing the original pick operator and to obtain complexity results for our approach.

References

[Baader *et al.*, 2003] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[Baader *et al.*, 2005] Franz Baader, Carsten Lutz, Maja Miličić, Ulrike Sattler, and Frank Wolter. Integrating description logics and action formalisms: First results. In *Proc. AAAI 2005*, pages 572–577. AAAI Press, 2005.

[Baader *et al.*, 2008] Franz Baader, Silvio Ghilardi, and Carsten Lutz. LTL over description logic axioms. In *Proc. KR 2008*, pages 684–694. AAAI Press, 2008.

[Baader *et al.*, 2010] Franz Baader, Hongkai Liu, and Anees ul Mehdi. Verifying properties of infinite sequences of description logic actions. In *Proc. ECAI 2010*, pages 53–58. IOS Press, 2010.

[Burgard *et al.*, 1999] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1–2):3–55, 1999.

[Claßen and Lakemeyer, 2008] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Proc. KR 2008*, pages 589–599. AAAI Press, 2008.

[Claßen and Lakemeyer, 2010] Jens Claßen and Gerhard Lakemeyer. On the verification of very expressive temporal properties of non-terminating Golog programs. In *Proc. ECAI 2010*, pages 887–892. IOS Press, 2010.

[De Giacomo *et al.*, 1997] Giuseppe De Giacomo, Evgenia Ternovska, and Raymond Reiter. Non-terminating processes in the situation calculus. In *Working Notes of "Robots, Softbots, Immobots: Theories of Action, Planning and Control", AAAI'97 Workshop*, 1997.

[De Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[De Giacomo *et al.*, 2010] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *Proc. KR 2010*, pages 445–455. AAAI Press, 2010.

[De Giacomo *et al.*, 2012] Giuseppe De Giacomo, Yves Lespérance, and Fabio Patrizi. Bounded situation calculus action theories and decidable verification. In *Proc. KR 2012*. AAAI Press, 2012.

[Gu and Soutchanski, 2010] Yilan Gu and Mikhail Soutchanski. A description logic based situation calculus. *Annals of Mathematics and Artificial Intelligence*, 58(1–2):3–83, 2010.

[Lakemeyer and Levesque, 2010] Gerhard Lakemeyer and Hector J. Levesque. A semantic characterization of a useful fragment of the situation calculus with knowledge. *Artificial Intelligence*, 175(1):142–164, 2010.

[Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.

[Lin and Reiter, 1997] Fangzhen Lin and Raymond Reiter. How to progress a database. *Artificial Intelligence*, 92(1–2):131–167, 1997.

[Liu and Levesque, 2005] Yongmei Liu and Hector J. Levesque. Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions. In *Proc. IJCAI 2005*, pages 522–527. Professional Book Center, 2005.

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. American Elsevier, New York, 1969.

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