On the Verification of Very Expressive Temporal Properties of Non-terminating Golog Programs

Jens Claßen and **Gerhard Lakemeyer**[1]

Abstract. The agent programming language GOLOG and the underlying Situation Calculus have become popular means for the modelling and control of autonomous agents such as mobile robots. Although such agents' tasks are typically open-ended, little attention has been paid so far to the analysis of non-terminating GOLOG control programs. Recently we therefore introduced a logic that allows to express properties of Golog programs using operators from temporal logics while retaining the full first-order expressiveness of the Situation Calculus. Combining ideas from classical symbolic model checking with first-order theorem proving we presented a verification method for a restricted subclass of temporal properties. In this paper, we extend this work by considering arbitrary temporal formulas. Our algorithm is inspired by classical $\mathcal{CTL}^*$ model checking, but introduces techniques to cope with arbitrary first-order quantification.

1 INTRODUCTION

The GOLOG [13, 6] family of agent programming languages and its underlying logic, the Situation Calculus [17, 16], have become popular means for the control of autonomous agents such as mobile robots. Typically, the task of such an agent is an open-ended one, i.e. there is no predefined goal or terminal state that the agent tries to reach eventually, but (at least ideally) the robot works indefinitely.

As a simple example, adapted from [8, 4], consider a mobile robot whose task it is to serve coffee to the people in an office environment. A program for this robot might look like this:

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

That is there is an infinite loop where at each cycle, if the robot's current queue is not empty, it selects the next request (which comes from person p) to be served next, gets a cup of coffee and brings it to p. Otherwise, the robot waits for one cycle. Requests from people may arrive at any time during the execution of the program.

Before actually deploying such a program on the robot and executing it in the real world, it is often desirable 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."

Somewhat surprisingly, the analysis and verification of non-terminating GOLOG programs has so far received little attention. One exception is the above mentioned work by De Giacomo, Ternovska and Reiter [8]. They formalize properties of programs using fixpoint operators expressed by formulas of second-order logic, and then do a manual, meta-theoretic proof to show that their program satisfies the given properties. As such proofs tend to be tedious and error-prone, it would be much more preferable to do an *automated* verification.

On the other hand, over the last decades, a huge variety of powerful verification methods and systems have been developed in the field of model checking [2, 1, 10]. Their application is particularly popular in the area of hardware verification and software engineering, and it seems more than natural to try to adapt and exploit the available results for the verification of GOLOG agents. However, for tractability and decidability, the methods and their underlying formalisms are typically restricted to propositional or very limited first-order expressiveness and, as the term "model checking" suggests, work on an explicit, finite model of the system. Combining or interfacing an existing model-checking tool with a GOLOG system thus unavoidably means a drastic loss in expression and modelling power, which is somewhat undesirable as its first-order expressiveness is usually the reason why the Situation Calculus was chosen in the first place. It would be much more desirable to be able to do the verification within the very same, expressive formalism that is used for the representation and actual control of the agent.

For these reasons, in [4] we introduced a new logic called $\mathcal{ESG}$ that allows to express and reason about properties of both terminating and non-terminating GOLOG programs. It is based on the modal Situation Calculus variant $\mathcal{ES}$ [11], but among other things extends it by a new modal construct $[\![\delta]\!]\phi$ which intuitively means that all possible execution traces of the program δ (i.e. sequences of primitive actions conforming to δ) satisfy ϕ, where ϕ may contain operators known from temporal logics. If δ is our example program, then the two properties stated above can be expressed as follows:

$$\forall p[\![\delta]\!]\boldsymbol{G}(Occ(requestCoffee(p)) \supset \boldsymbol{F}Occ(selectRequest(p))) \quad (1)$$

$$\langle\!\langle\delta\rangle\!\rangle\boldsymbol{G}\neg\exists p(Occ(selectRequest(p))) \quad (2)$$

(1) says that for all possible executions of δ ($[\![\delta]\!]$), it is always ($\boldsymbol{G}$) the case that whenever some $requestCoffee(p)$ action occurs, then there will eventually ($\boldsymbol{F}$) be some $selectRequest(p)$. The $\langle\!\langle\delta\rangle\!\rangle$ is the existential counterpart of $[\![\delta]\!]$, thus (2) says that there is some execution of δ such that a $selectRequest(p)$ never occurs.

[4] provides a sound method for verifying a limited class of program properties that resembles $\mathcal{CTL}$. This means that there it is required that the path quantifier $\langle\!\langle\delta\rangle\!\rangle$ is directly followed by a temporal operator like $\boldsymbol{G}$ or $\boldsymbol{F}$. Formula (2) above is an example of such a property, whereas (1) is not, as temporal operators appear nested and are combined with other subformulas through logical connectives.

The latter is representative for a broader class of properties that resembles $\mathcal{CTL}^*$, i.e. where the usage of temporal operators within a path quantifier is unrestricted and we may make free use of logical

[1] RWTH Aachen University, Germany, ⟨classen|gerhard⟩@cs.rwth-aachen.de

connectives, including first-order quantification. Many of the typical properties one may want to verify for a non-terminating agent are only expressible in this manner, most importantly liveness and fairness conditions such as the above "every request will eventually be served", or "whenever the battery is low, it will get recharged in time", or "the floor gets cleaned infinitely often" etc. In this paper, we therefore propose a verification algorithm for this more general class of properties. As for the less general method described in [4], there is of course no free lunch here: dealing with arbitrary first-order action theories and properties, and thus resorting to first-order theorem proving, comes at the price of losing decidability. The algorithm we present is sound, but not guaranteed to terminate.

The paper is organized as follows. Section 2 recapitulates the $\mathcal{ESG}$ language definition, and Section 3 does so for the $\mathcal{ESG}$ equivalent of Reiter's basic action theories and regression operator. Our main contribution is in Section 4 where we present our verification algorithm. We discuss related work in Section 5, after which we conclude.

2 THE LOGIC $\mathcal{ESG}$

2.1 Syntax

The language is a first-order[2] modal dialect with equality and two sorts *object* and *action*. For each sort, there are countably many standard names $n_1, n_2, \ldots$ which are syntactically treated like constants. Predicate and function symbols of any arity can be either rigid or fluent. The latter vary as the result of actions, while the former do not. The fluents include the unary predicates *Poss*, *Occ*, and *Exo*.

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 the usual abbreviations.

Terms are formed in the usual way. 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. The former express properties of situations, the latter are used to 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 and t_2 are terms, then $(t_1 = t_2)$ is a situation formula;
3. 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;
4. If α and β are situation formulas, and v is a first-order variable, then the following are also situation formulas: $(\alpha \wedge \beta)$, $\neg\alpha$, $\forall v.\alpha$.

For the *trace formulas*:

1. Every situation formula is a trace formula;
2. If ϕ and ψ are trace formulas and v is a first-order variable, then $\phi \wedge \psi$, $\neg\phi$, $\forall v.\phi$, $\boldsymbol{X}\phi$ 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* and *Exo* a *fluent formula*. A *bounded formula* does not contain $[\![\delta]\!]$ and contains only $[t]$ constructs whose t is an atomic action.

Finally, the set of *programs* is given by below grammar:

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

Here, t is any (not necessarily ground) term of sort action and α a bounded formula. In the presented order, the constructs mean a primitive action, a test, sequence of programs, nondeterministic choice between programs, nondeterministic choice of argument, concurrent (interleaved) execution of programs, and nondeterministic iteration. Further control structures are defined by:

$$\textbf{if } \alpha \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{def}{=} \alpha?;\delta_1 \mid \neg\alpha?;\delta_2 \quad (3)$$

$$\textbf{while } \alpha \textbf{ do } \delta \textbf{ endWhile} \stackrel{def}{=} (\alpha?;\delta)^*;\neg\alpha? \quad (4)$$

$$\textbf{loop } \delta \stackrel{def}{=} \textbf{while } \top \textbf{ do } \delta \textbf{ endWhile} \quad (5)$$

We also abbreviate **loop** δ as δ^ω. We use $\top$ to denote truth, defined as $\forall x.(x = x)$, and $\bot$ for falsity, i.e. $\neg\top$. The usual $\mathcal{CTL}^*$ path quantifiers can be introduced by defining $\boldsymbol{E}\phi$ as $\langle\!\langle any^\omega\rangle\!\rangle\phi$ and $\boldsymbol{A}\phi$ as $[\![any^\omega]\!]\phi$, where any is shorthand for $\pi a.a$. Further we abbreviate $(\top\ \boldsymbol{U}\ \phi)$ as $\boldsymbol{F}\phi$ ("eventually") and $\neg\boldsymbol{F}\neg\phi$ as $\boldsymbol{G}\phi$ ("always"). The $\Box\alpha$ construct from [11] is understood as shorthand for $\boldsymbol{AG}\alpha$.

2.2 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 $w \in \mathcal{W}$ 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.

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 $|t|_w^z$ by:

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

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

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

$\mathcal{N}_v$ refers to the set of standard names of the same sort as v. α_n^v means α with every free occurrence of v replaced by n. $\|\delta\|^w(z)$, which is defined below, maps, given w 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 8 only requires that α holds after all finite sequences for $[\delta]\alpha$ to be true, which implies that any formula

[2] The second-order features from [4] are unnecessary for the purpose of this paper and therefore omitted.

is vacuously true "after" the execution of a non-terminating program δ^ω. On the other hand, in rule 7, the trace formula ϕ must hold for any trace τ, be it finite or not. The truth of trace formulas is given by:

1. $w, z, \tau \models \alpha$ iff $w, z \models \alpha$, where 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. $w, z, \tau \models \forall v.\phi$ iff $w, z, \tau \models \phi^v_n$ for all $n \in \mathcal{N}_v$;
5. $w, z, \tau \models \boldsymbol{X}\phi$ iff $\tau = n \cdot \tau'$ and $w, z \cdot n, \tau' \models \phi$;
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 $z' = z'' \cdot z'''$, $w, z \cdot z'', z''' \cdot \tau' \models \phi$.

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.

2.2.1 The Meaning of Programs

Due to limited space, we will only repeat the formal definition of the program semantics (with slight notational adaptations). For more detailed explanations, the interested reader is referred to [4].

A *configuration* $\langle w, z, \delta\rangle$ consists of a world w, a sequence of (already performed) actions z, and a program δ (remaining to be executed). The set of *final* configurations $\mathcal{F}$ is the least set satisfying:

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

The transition relation $\rightarrow$ among configurations is given by:

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

where *nil* is shorthand for $\top?$. Note that the above is basically the transition semantics of CONGOLOG [6], but with the slight modification that tests are not viewed as *transitions* like physical actions, but as *conditions* under which a transition may be taken or the program may terminate. Thus, it is not necessary to define synchronized variants of the **if** and **while** constructs as in [6], but we can use the definitions (3) and (4) given in Section 2.1.

Let $Pre(\pi)$ mean the set of all prefixes of a trace π and $\xrightarrow{*}$ be the reflexive transitive closure of $\rightarrow$. The set $\|\delta\|^w(z)$ of execution traces of the program δ, given w, z, then is

$$\{z' \mid \langle w, z, \delta\rangle \xrightarrow{*} \langle w, z \cdot z', \delta'\rangle \text{ and } \langle w, z \cdot z', \delta'\rangle \in \mathcal{F}\} \cup$$
$$\{\pi \mid \forall z' {\in} Pre(\pi),\ \langle w, z, \delta\rangle \xrightarrow{*} \langle w, z \cdot z', \delta'\rangle \text{ and } \langle w, z \cdot z', \delta'\rangle \notin \mathcal{F}\}.$$

3 BASIC ACTION THEORIES AND REGRESSION

A *basic action theory* (BAT) $\Sigma = \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}} \cup \Sigma_{\text{exo}} \cup \Sigma_{\text{UNA}}$ 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. In the coffee robot example, we might have $\Sigma_0 = \{\neg \mathit{HoldingCoffee},\ \mathit{Empty}(\mathit{queue})\}$.
2. Σ_{pre} is a *precondition axiom* of the form $\Box \mathit{Poss}(a) \equiv \pi$, with π being a fluent formula, whose only free variable is a, describing precisely the conditions under which a is a possible action:

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

3. Σ_{post} is a finite set of *successor state axioms* (SSAs), one for each fluent relevant to the application domain, incorporating Reiter's [17] 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 $\Box[a]F(\vec{x}) \equiv \gamma_F$, whereas the one for a functional fluent is of the form $\Box[a]f(\vec{x}) = y \equiv \gamma_f$, where γ_F is a fluent formula with free variables $\vec{x}$, and γ_f one with free variables among $\vec{x}$ and y. In the coffee robot domain, we have a relational fluent $\mathit{HoldingCoffee}$ and a functional fluent queue:

$$\begin{aligned}\Box[a]\mathit{HoldingCoffee} \equiv\ & a = \mathit{pickupCoffee}\ \vee \\ & \mathit{HoldingCoffee} \wedge \neg\exists p.\ a = \mathit{bringCoffee}(p),\end{aligned}$$

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

4. Σ_{exo} is the *exogenous actions axiom*, having the form $\Box \mathit{Exo}(a) \equiv \chi$, 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". In our example, we have only one such action, thus the axiom is $\Box \mathit{Exo}(a) \equiv \exists p.a = \mathit{requestCoffee}(p)$. If $\delta_{exo} = (\pi a.\mathit{Exo}(a)?; a)^\omega$ and δ_{ctrl} is the actual control program of the agent, we use $(\delta_{ctrl} \| \delta_{exo})$ in our further analysis.
5. Σ_{UNA} are unique names for actions axioms.

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:

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

$$\begin{aligned}&\mathit{Dequeue}(q_o, p, q_n) \overset{def}{=} (p \neq e)\ \wedge \exists x_2 \ldots \exists x_k. \\ &\qquad q_o = \mathit{list}(p, x_2, \ldots, x_k)\ \wedge\ q_n = \mathit{list}(x_2, \ldots, x_k, e)\end{aligned}$$

Our algorithm relies on the $\mathcal{ES}$ equivalent of Reiter's *regression* operator. The idea behind it is that whenever we encounter a subformula of the form $[t]F(\vec{x})$, where t is a primitive action, 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, $[t]\,Occ(t')$ is replaced by $(t = t')$ and $Poss(t)$ and $Exo(t)$ by the right-hand sides of the corresponding axiom. Iteratively applying such substitutions, we get a fluent formula that describes exactly the conditions on the initial situation under which the original, non-static formula holds:

Theorem 1 *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 $\Sigma_0 \models \mathcal{R}[\alpha]$.*

4 PROGRAM VERIFICATION

Verification of non-terminating programs means checking whether some BAT Σ satisfies a property expressed by a formula of $\mathcal{ESG}_{\mathcal{CTL}^*}$:

$$\alpha ::= (t_1 = t_2) \mid F(\vec{t}) \mid \neg\alpha \mid \alpha \wedge \alpha \mid \exists x.\alpha \mid \langle\langle\delta\rangle\rangle\varphi$$

Of course the $\langle\langle\delta\rangle\rangle\varphi$ subformulas, where we assume that the δ is a non-terminating program of the form $\delta_1^\omega \| \cdots \| \delta_k^\omega$, are the most interesting part. The idea is to replace them by fluent formulas that are equivalent wrt Σ and then check the result against Σ_0, which can be done using standard first-order theorem proving. [4] provided verification methods only for the case where the φ above was restricted to be a temporal subformula $(\phi\ \boldsymbol{U}\ \psi)$ or $\boldsymbol{X}\phi$ without nested temporal operators. Here we allow φ to be any formula of $\mathcal{ESG}_{\mathcal{LTL}}$:

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

4.1 Characteristic Graphs

We encode the space of reachable program configurations by a *characteristic graph* $\mathcal{G}_\delta$ 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 [4]. Figure 1 shows the graph corresponding to $\delta_{coffee}\|\delta_{exo}$, where δ_{coffee} denotes the control program presented in the introduction and δ_{exo} is the encoding of exogenous actions. The nodes are $v_0 = \langle\delta_{coffee} \parallel \delta_{exo}, \bot\rangle$, $v_1 = \langle(pickupCoffee; bringCoffee(p); \delta_{coffee}) \parallel \delta_{exo}, \bot\rangle$, and $v_2 = \langle(bringCoffee(p); \delta_{coffee}) \parallel \delta_{exo}, \bot\rangle$.

4.2 The Algorithm

The method we propose is inspired by the classical propositional $\mathcal{LTL}$ model checking algorithm as presented in [20]. There, the idea roughly is to combine the finite transition system (a model of the behaviour of the system) with a finite nondeterministic Büchi automaton which encodes the set of infinite traces that satisfy the input formula. The result is again a finite graph structure, and whether or not the property in question holds can then be determined on

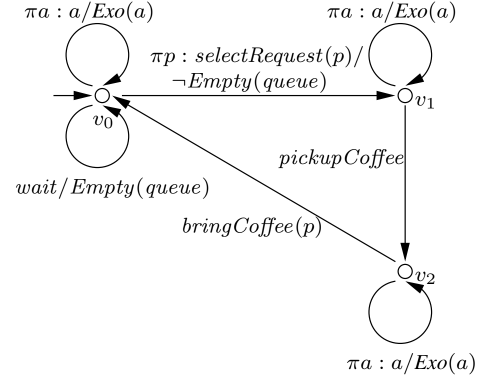

Figure 1. Characteristic Graph for the program $\delta_{coffee}\|\delta_{exo}$

a purely graph-theoretical basis. In our case, the representation of the behaviour of the system consists of two parts. On the one hand, the characteristic graph of the program δ represents the robot's behaviour, whereas the dynamics of the physical environment is encoded in its basic action theory Σ. Since both Σ and δ as well as the input formula φ may contain first-order quantification, the state space is in general not finite, which is why the set of traces satisfying φ cannot be simply captured by some finite graph structure. We rather have to use an implicit representation in terms of first-order formulas and thus resort to first-order theorem proving.

Formally, let $\varphi \in \mathcal{ESG}_{\mathcal{LTL}}$. We need to consider all temporal subformulas of φ, i.e. all its subformulas of the form $(\phi_i\ \boldsymbol{U}\ \psi_i)$ (with free variables $\vec{x}_i$) and of the form $\boldsymbol{X}\phi_j$ (with free variables $\vec{x}_j$). As an example consider property (1), which says that on all executions of our program δ, whenever there is some $requestCoffee(p)$ action, eventually there will be a $selectRequest(p)$. If r stands for $requestCoffee$ and s for $selectRequest$, the formula is shorthand for

$$\forall p.\neg\langle\langle\delta\rangle\rangle(\top\ \boldsymbol{U}\ (Occ(r(p)) \wedge \neg(\top\ \boldsymbol{U}\ Occ(s(p)))))). \tag{6}$$

The part behind the $\langle\langle\delta\rangle\rangle$ quantifier is an $\mathcal{ESG}_{\mathcal{LTL}}$ formula and contains the following two temporal subformulas with free variable p:

$$(\top\ \boldsymbol{U}\ Occ(s(p))) \tag{7}$$

$$(\top\ \boldsymbol{U}\ (Occ(r(p)) \wedge \neg(\top\ \boldsymbol{U}\ Occ(s(p))))) \tag{8}$$

The idea is now to break down the truth of temporal formulas at infinite execution traces into three different parts:

1. **Local consistency:**
   $LocCons[\varphi]$ is the set of local consistency constraints for all of φ's temporal subformulas of the form $(\phi_i\ \boldsymbol{U}\ \psi_i)$:
   $$\forall\vec{x}_i.\ \ \psi_i \supset (\phi_i\ \boldsymbol{U}\ \psi_i) \tag{9}$$
   $$\forall\vec{x}_i.\ \ (\phi_i\ \boldsymbol{U}\ \psi_i) \wedge \neg\psi_i \supset \phi_i \tag{10}$$
2. **Single-step consistency:**
   $Trans[\varphi]$ denotes the set of the transition constraints for all temporal subformulas of φ, which express that from one situation to the next, the well-known expansion law for the until operator must hold, and ensure that the semantics of the next operator is obeyed:
   $$\forall\vec{x}_i.\ \ (\phi_i\ \boldsymbol{U}\ \psi_i) \equiv \psi_i \vee (\phi_i \wedge [a](\phi_i\ \boldsymbol{U}\ \psi_i)) \tag{11}$$
   $$\forall\vec{x}_j.\ \ \boldsymbol{X}\phi_j \equiv [a]\phi_j \tag{12}$$
3. **Eventual compliance:**
   For each $(\phi_i\ \boldsymbol{U}\ \psi_i)$ subformula, we need that infinitely often
   $$Accept_i \overset{def}{=} (\phi_i\ \boldsymbol{U}\ \psi_i) \supset \psi_i. \tag{13}$$

In order to be able to reason with these properties solely based on first-order theorem proving, we introduce a new predicate symbol $U_i(\vec{x}_i)$ for each $(\phi_i \boldsymbol{U} \psi_i)$ subformula (whose free variables are $\vec{x}_i$) and similarly a new symbol $X_j(\vec{x}_j)$ for each $\boldsymbol{X}\phi_j$ subformula (whose free variables are $\vec{x}_j$). In our example we thus get $U_1(p)$ for (7) and $U_2(p)$ for (8). We use $\phi\!\downarrow$ to denote the result of replacing temporal subformulas by their corresponding predicates.

To ensure eventual compliance, we use a property that is similar to the acceptance criterion of Büchi automata, but extends it to the first-order case. For that purpose, we introduce a new fluent A_i for each $(\phi_i \boldsymbol{U} \psi_i)$ subformula, having the successor state axiom

$$\square[a]A_i(\vec{x}_i) \equiv (Accept_i\!\downarrow \vee (A_i(\vec{x}_i) \wedge \neg AccAll[\varphi])) \qquad (14)$$

where

$$AccAll[\varphi] \stackrel{def}{=} \bigwedge_i \forall \vec{x}_i A_i(\vec{x}_i). \qquad (15)$$

The idea is that the A_i "collect" all instances of $Accept_i$, and $AccAll[\varphi]$ becomes true once all A_i hold for all $\vec{x}_i$, after which they are reset to false. The algorithm then basically tries to prove the existence of an execution trace on which $AccAll[\varphi]$ is satisfied infinitely often, i.e. where always eventually $AccAll[\varphi]$ holds.

Similar to the one presented in [4], our algorithm works on a set L of labels on the characteristic graph of δ. A label is of the form $\langle v, \psi\rangle$, where $v = \langle \delta', \cdot\rangle$ is some node in the graph and ψ is a fluent formula, and intuitively represents the set of all configurations $\langle w, z, \delta'\rangle$ where $w, z \models \psi$. The algorithm is depicted below:

Algorithm 1 CHECKLTL$[\delta, \varphi]$

$L_O := \emptyset$; $L := \textsc{Label}[\mathcal{G}_\delta, LocCons[\varphi]\!\downarrow \wedge AccAll[\varphi]]$;
while $L \not\equiv L_O$ **do**
  $L_O := L$; $L := L \cup \textsc{Pre}[\mathcal{G}_\delta, L]$;
end while
$L_O := \emptyset$;
while $L \not\equiv L_O$ **do**
  $L_O := L$; $L := \textsc{And}(L, \textsc{Pre}[\mathcal{G}_\delta, L])$;
end while
return $\textsc{Elim}[\langle \vec{X}, \vec{U}\rangle, \varphi\!\downarrow \wedge \textsc{InitLabel}[\mathcal{G}_\delta, L]]$

It starts with initializing the set of labels to the ones representing the configurations where local consistency and $AccAll[\varphi]$ hold:

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

The first while loop then does a least fixpoint computation in order to determine the set of configurations from which such an "acceptance" state is *eventually* reachable, after which the second while loop computes a greatest fixpoint of labels representing those configurations for which this is in turn *always* the case. In the latter case, labels need to be combined conjunctively in the following sense:

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

The convergence criterion "$L \not\equiv L_O$" is violated if for each $\langle v, \psi\rangle \in L$ there is some $\langle v, \psi'\rangle \in L_O$ with $\Sigma_{\text{UNA}} \models \psi \equiv \psi'$ and vice versa.

In each single cycle of the loops, we need to project the set of labels back to the one representing all possible predecessor configurations which is done using the preimage operator:

$$\textsc{Pre}[\langle V, E, v_0\rangle, L] = \{\langle v', \psi'\rangle \mid v' \stackrel{\pi\vec{x}:t/\phi}{\rightarrow} v \in E,\ \langle v, \psi\rangle \in L\}$$

where ψ' is determined as follows:

$$\psi' = \textsc{Elim}[\langle \vec{\mathcal{U}}, \vec{\mathcal{X}}\rangle, \mathcal{R}[\exists \vec{x}.\phi \wedge [t]\psi \wedge Trans[\varphi]^a_t\!\downarrow] \wedge LocCons[\varphi]\!\downarrow]$$

Intuitively, the formula reads as follows. To end up in v with label ψ through edge $v' \stackrel{\pi\vec{x}:t/\phi}{\rightarrow} v$, we need to pick instantiations for the $\vec{x}$, the transition condition ϕ has to hold and ψ must be true after doing action t. Further the single-step and local consistency constraints need to be respected. In order to obtain a new label that is itself a first-order formula, we use regression to eliminate all references to the successor situation, treating A_i as a normal fluent with successor state axiom (14). To regress the auxiliary predicates U_i and X_j, we introduce yet another predicate $\mathcal{U}_i$ for each U_i and $\mathcal{X}_j$ for each X_j (with the same arities), replace each occurrence of U_i (X_j) in the successor situation by $\mathcal{U}_i$ ($\mathcal{X}_j$), and otherwise leave them unchanged:

$$\begin{array}{llll} \mathcal{R}[U_i(\vec{t}_i)] \stackrel{def}{=} U_i(\vec{t}_i), & \mathcal{R}[[t]\,U_i(\vec{t}_i)] \stackrel{def}{=} \mathcal{U}_i(\vec{t}_i) \\ \mathcal{R}[X_j(\vec{t}_j)] \stackrel{def}{=} X_j(\vec{t}_j), & \mathcal{R}[[t]X_j(\vec{t}_j)] \stackrel{def}{=} \mathcal{X}_j(\vec{t}_j) \end{array}$$

Let us apply a few steps of the algorithm to our example. First note that both instances of the local consistency constraint (10) are vacuously true as each ϕ_i is $\top$. The set of initial labels in the example thus is $\{\langle v, \beta \wedge \forall p A_1(p) \wedge \forall p A_2(p)\rangle \mid v \in V\}$, where β denotes

$$\forall p(Occ(s(p)) \supset U_1(p)) \wedge \forall p(Occ(r(p)) \wedge \neg U_1(p) \supset U_2(p)).$$

Now let us determine ψ' for the edge $v_0 \stackrel{wait/Empty(queue)}{\rightarrow} v_0$ and the initial label of v_0. The transition condition is $\alpha_1 = Empty(queue)$, which remains unchanged by regression. Regressing $Occ(s(p))$ through $wait$ yields $(wait = s(p))$, which reduces to $\perp$ with unique names for actions, and similar for $r(p)$. The regression of β therefore is equivalent to $\top$. The regression of $\forall p A_1(p) \wedge \forall p A_2(p)$ further is

$$\begin{array}{ll} \alpha_2 = & \forall p.\, U_1(p) \equiv [Occ(s(p)) \vee \mathcal{U}_1(p)] \wedge \\ & \forall p.\, U_2(p) \equiv [(Occ(r(p)) \wedge \neg U_1(p)) \vee \mathcal{U}_2(p)], \end{array}$$

and $Trans[\varphi]^a_{wait}\!\downarrow$ regresses to the conjunction of

$$\begin{array}{l} \forall p.[U_1(p) \supset Occ(s(p))] \vee A_1(p) \wedge \neg AccAll[\varphi] \\ \forall p.[U_2(p) \supset (Occ(r(p)) \wedge \neg U_1(p))] \vee A_2(p) \wedge \neg AccAll[\varphi] \end{array}$$

which we will call α_3. Then $\psi' = \textsc{Elim}[\langle \mathcal{U}_1, \mathcal{U}_2\rangle, \alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge \beta]$.

Finally we eliminate all occurrences of the $\mathcal{U}_i$ and $\mathcal{X}_j$ by the $\textsc{Elim}[\cdot]$ operator. $\textsc{Elim}[\vec{P}, \alpha]$ means that we replace α by an equivalent formula not mentioning the predicates $\vec{P}$. This amounts to second-order quantifier elimination, i.e. to determine a purely first-order formula equivalent to a given $\exists P_1, \ldots, \exists P_k \beta$, where β does not contain further second-order quantifiers. The predicate elimination is necessary because of the nondeterministic nature of the single-step consistency law (11): whereas the successor state axioms for normal fluents determine unique values for the respective fluents if their values in the current situation are known, the value of $\phi_i \boldsymbol{U} \psi_i$ (and thus U_i) for the successor situation is not uniquely determined through (11). To "forget" [14] what is known about the successor situation and thus obtain a first-order formula only talking about the current situation, we have to eliminate all occurrences of $\mathcal{U}_i$ and $\mathcal{X}_j$.

To implement $\textsc{Elim}[\cdot]$ we require in the general case a sound (but necessarily incomplete) second-order quantifier elimination technique such as SCAN [5]. Note that in many practical cases (though not in the example above), a simple syntactic manipulation is sufficient, namely when the predicates to be eliminated have arity zero, which happens when we do not quantify into temporal subformulas from the outside. In this special case, to eliminate P, let $\alpha^P_\top$ denote α where each occurrence of P is substituted by $\top$, and similarly $\alpha^P_\perp$ be α with P replaced by $\perp$. Then $\textsc{Elim}[P, \alpha] = \alpha^P_\top \vee \alpha^P_\perp$. Multiple predicates can further be eliminated recursively.

If finally the second loop of the algorithm terminates, we extract the formulas in labels at the initial node:

$$\text{INITLABEL}[\langle V, E, v_0\rangle, L] = \bigvee\{\psi \mid \langle v_0, \psi\rangle \in L\}$$

The result the algorithm returns is sound in the following sense:

Theorem 2 *Given $\alpha \in \mathcal{ESG}_{\mathcal{CTL}^*}$, let β be the result of replacing each $\langle\!\langle\delta\rangle\!\rangle\varphi$ subformula by the output of* CHECKLTL$[\delta, \varphi]$ *in it. If this terminates, β is a fluent formula and $\Sigma \models \alpha$ iff $\Sigma_0 \cup \Sigma_{UNA} \models \beta$.*

In our example CHECKLTL$[\delta, \varphi]$ eventually converges and produces an output equivalent to $\bot$. The original, negated formula (6) thus holds, and that independent of any particular Σ_0, as it is a property that is already inherent in the program δ itself.

5 RELATED WORK

As stated earlier, verification of non-terminating GOLOG programs has so far received surprisingly little attention in the Situation Calculus community. Exceptions include the mentioned meta-theoretic proofs of [8], doing classical propositional model checking with BAT representations [9], and [4], the basis for this paper. De Giacomo, Lespérance and Pearce [7] further present related techniques for verifying ATL [19] properties of games and multi-agent scenarios.

From a broader perspective, there is much work related to ours in some respects. There are a number of model checking techniques for "first-order" $\mathcal{LTL}$. Typically they restrict the usage of first-order expressiveness to ensure decidability. In [21] for instance, first-order terms are allowed, but quantification is not. [15] presents a verification method for programs represented by Büchi automata, where properties are expressed using a combination of $\mathcal{LTL}$ with a decidable description logic. An epistemic, modal variant of the Situation Calculus with branching-time temporal operators is presented in [12]. [19] discusses ATL model checking of multi-agent systems.

6 CONCLUSION

We presented[3] an algorithm for verifying a very expressive class of properties for GOLOG programs, where temporal operators can appear nested and combined through logical connectives, including arbitrary first-order quantification. Just as the less general method in [4], the algorithm is sound, but not guaranteed to terminate. There, the two reasons for this are that in each cycle of the algorithm's loop, equivalences of general first-order formulas have to be tested, an already undecidable problem. For another, even if all equivalence checks terminate, the loop may never reach a fixpoint, but may keep generating new labels. The step of moving from $\mathcal{CTL}$- to $\mathcal{CTL}^*$-like properties in this paper introduces yet another potential source of non-termination, namely when the predicate elimination operator ELIM (Section 4.2) is applied to predicates of arity greater than zero.

While this may sound bad at first, in particular compared to the computational properties of state-of-the-art model checking techniques, it nonetheless helps to understand the theoretical ideal case of GOLOG verification that our algorithm somewhat represents. We also argue that in many practical cases, as for the example used in this paper, the worst case does not arise, but the method converges despite (cautious) usage of first-order quantification. It is then a worthwhile line of future research to identify subclasses of BATs and programs for which termination actually can be ensured. Probably the most important next step in assessing the practicality of our algorithms will however be through experimental studies. We currently work on a PROLOG-based implementation, and preliminary experiments are quite promising. It turned out though that our methods' tractability crucially depends on the representation for first-order formulas. While previously we experimented with clausal form representations and theorem provers, ongoing work indicates that a first-order variant of binary decision diagrams [18] is much more beneficial, both in terms of compactness of representation and reasoning efficiency.

[3] A more detailed discussion including all proofs will be available in [3].

ACKNOWLEDGEMENTS

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

REFERENCES

[1] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, 'NuSMV2: An OpenSource tool for symbolic model checking', in *CAV*, pp. 359–364, (2002).

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

[3] J. Claßen, *Planning and Verification in the Agent Language Golog*, Ph.D. dissertation, RWTH Aachen University, 2010. In preparation.

[4] J. Claßen and G. Lakemeyer, 'A logic for non-terminating Golog programs', in *KR*, pp. 589–599, (2008).

[5] D.M. Gabbay and H.J. Ohlbach, 'Quantifier elimination in second-order predicate logic', in *KR*, pp. 425–435, (1992).

[6] G. De Giacomo, Y. Lespérance, and H.J. Levesque, 'ConGolog, a concurrent programming language based on the situation calculus', *Artif. Intell.*, **121**(1-2), 109–169, (2000).

[7] G. De Giacomo, Y. Lespérance, and A.R. Pearce, 'Situation calculus-based programs for representing and reasoning about game structures', in *KR*, (2010).

[8] G. De Giacomo, E. Ternovska, and R. 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).

[9] Y. Gu and I. Kiringa, 'Model checking meets theorem proving: a situation calculus based approach', in *Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR-06) at KR2006*, Lake District of the UK, (June 2006).

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

[11] G. Lakemeyer and H.J. Levesque, 'Situations, si! Situation terms, no!', in *KR*, pp. 516–526, (2004).

[12] Gerhard Lakemeyer, 'The situation calculus: A case for modal logic', *Journal of Logic, Language and Information*, (2010). In press, DOI 10.1007/s10849-009-9117-6.

[13] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R.B. Scherl, 'Golog: A logic programming language for dynamic domains', *J. Log. Program.*, **31**(1-3), 59–83, (1997).

[14] F. Lin and R. Reiter, 'Forget it!', in *In Proceedings of the AAAI Fall Symposium on Relevance*, pp. 154–159, (1994).

[15] H. Liu, *Computing Updates in Description Logics*, Ph.D. dissertation, Dresden University of Technology, 2010.

[16] J. McCarthy and P. Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', in *Machine Intelligence 4*, 463–502, American Elsevier, New York, (1969).

[17] R. Reiter, *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, MIT Press, September 2001.

[18] S. Sanner and C. Boutilier, 'Practical solution techniques for first-order MDPs', *Artificial Intelligence*, **173**(5-6), 748–788, (2009).

[19] W. van der Hoek, A. Lomuscio, and M. Wooldridge, 'On the complexity of practical ATL model checking', in *AAMAS*, pp. 201–208, (2006).

[20] M.Y. Vardi and P. Wolper, 'An automata-theoretic approach to automatic program verification (preliminary report)', in *LICS*, pp. 332–344. IEEE Computer Society, (1986).

[21] F. Wang, S. Tahar, and O.A. Mohamed, 'First-order LTL model checking using MDGs', in *ATVA*, ed., F. Wang, volume 3299 of *Lecture Notes in Computer Science*, pp. 441–455. Springer, (2004).