Symbolic Verification of GOLOG Programs with First-Order BDDs

Jens Claßen
Knowledge-Based Systems Group
RWTH Aachen University, Germany
classen@kbsg.rwth-aachen.de

Abstract

GOLOG is an agent language that allows defining behaviours in terms of programs over actions defined in a Situation Calculus action theory. Often it is vital to verify such a program against a temporal specification before deployment. So far work on the verification of GOLOG has been mostly of theoretical nature. Here we report on our efforts on implementing a verification algorithm for GOLOG based on fixpoint computations, a graph representation of program executions, and a symbolic representation of the state space. We describe the techniques used in our implementation, in particular a first-order variant of OBDDs for compactly representing formulas. We evaluate the approach by experimental analysis.

1 Introduction

The agent language GOLOG (Levesque et al. 1997; De Giacomo, Lespérance, and Levesque 2000) allows to describe an agent’s behaviour in terms of a program over primitive actions defined in a Situation Calculus action theory (McCarthy and Hayes 1969; Reiter 2001). This very expressive, first-order formalism is particularly suited for scenarios where one has to cope with incomplete information and a possibly unbounded domain of objects. As an example, consider a robot whose task is to deliver coffee on request:

\[
\text{loop: while } (\exists x. \text{OnRobot}(x)) \text{ do } \\
\quad \pi x: \text{Dish unload}(x) \text{ endWhile; } \\
\quad \pi y: \text{Room } \{ \text{goto}(y); \\
\quad \quad \text{while } (\exists x. \text{Dirty}(x, y)) \text{ do } \\
\quad \quad \quad \pi x: \text{Dish load}(x, y) \text{ endWhile } \}; \\
\quad \text{goto(kitchen)}
\]

In each iteration of the infinite outer loop, the robot (initially located in the kitchen) first unloads all dishes it carries, selects a room in the building, moves there, collects all dirty dishes there, and returns to the kitchen. Here, \text{Dirty}(x, y) reads as “dirty dish } x \text{ is in room } y \text{” and } \text{load}(x, y) \text{ as “load dish } x \text{ from room } y \text{”, and there are exogenous newdish}(x, y) \text{ actions causing a new dirty dish } x \text{ to appear in room } y \text{. Unlike the previous example, the choice operators } \pi \text{ (“picks”) here only range over finite domains Dish and Room.}

Before deploying any such program onto a robot, one may want to verify it against some temporal specification, e.g. to ensure that every coffee request will eventually be served, or that no dirty dish remains in a room forever. There are essentially two approaches for the formal verification of such properties of GOLOG programs: First, use existing verification methods, particularly from the area of model checking. As the general verification problem for GOLOG is highly undecidable due to the language’s expressivity in terms of first-order quantification, range of action effects, and program constructs, this entails finding non-trivial restrictions that allow for a finite abstraction of the state space (Zarrieß and Claßen 2014; Zarrieß and Claßen 2016). A second approach is to devise GOLOG-specific verification algorithms that e.g. do similar fixpoint computations as (symbolic) model checking techniques, but that employ special representation and reasoning techniques for Situation Calculus theories and GOLOG programs (Claßen and Lakemeyer 2008; Claßen 2013). Due to the aforementioned undecidability, in general such a method is at most sound, but cannot be guaranteed to terminate, unless similar restrictions as for the abstraction approach are imposed (Claßen et al. 2014).

For the most part, work on GOLOG verification so far has been purely theoretical. In this paper, after briefly introducing formal preliminaries (Section 2), we report on our efforts on implementing such verification methods, in particular the GOLOG-specific fixpoint method. As it turns out, the largest obstacle is keeping the involved first-order formulas

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
at a manageable size, as they tend to blow up very quickly. Again taking inspiration from (symbolic) model checking, we propose to use a first-order variant of ordered binary decision diagrams (OBDDs) for their compact representation, along with other techniques (Section 3). We evaluate the approach’s practicality by experimental analysis (Section 4).

2 Preliminaries

We use a modal variant of the Situation Calculus called $ES$ (Lakemeyer and Levesque 2010). Due to limited space, we only give a brief overview (see references for details). The language is a first-order logic with equality and terms coming in two sorts, namely action and object, and where we make the unique names assumption for all ground terms (yielding e.g. plate  ≠ cup and load(x)  ≠ unload(y)). Both predicate and function symbols can be fluent, meaning they may change due to the execution of actions (e.g. $OnRobot(x)$ and queue, also $Poss(a)$ for action executability). There are no situation terms; to refer to future situations, two modal operators can be used: $[]\phi$ says that $\phi$ holds after any sequence of actions, and $[t]\phi$ means that $\phi$ holds after executing action $t$. A formula without $\Box$ and $[t]$ is called fluent formula. $\top$ denotes “true”, and $\bot$ means “false”. We can then represent a dynamic domain as follows:

Definition 1. A basic action theory (BAT) $\Sigma$ is a set of axioms consisting of: (1) $\Sigma_0$, the initial theory, a finite set of fluent sentences describing the initial state of the world; (2) $\Sigma_{pre}$, a precondition axiom, of the form $\Box Poss(a) \equiv \pi$, where $\pi$ is a fluent formula with free variable $a$; (3) $\Sigma_{post}$, a finite set of successor state axioms (SSAs), one for each fluent relevant to the application domain, encoding actions’ effects. The SSA for a fluent predicate $F$ has the form $^2\Box[a]F(\bar{x}) \equiv \gamma_F$, where $\gamma_F$ is a fluent formula with free variables $a$ and $\bar{x}$ (similar for functions).

Example 2. For the coffee robot, the initial theory is $\Sigma_0 = \{Empty(queue)\}$, and the precondition $\Sigma_{pre}$ is given by:

- $\Box Poss(a) \equiv (a = wait) \lor (a = pickupCoffee \land \neg HoldingCoffee) \lor$
- $\exists x (a = bringCoffee(x) \land HoldingCoffee) \lor$
- $\exists x (a = requestCoffee(x) \land x \neq e \land LastFree(queue)) \lor$
- $\exists x (a = selectRequest(x) \land x \neq e \land IsFirst(queue,x))$

The SSAs for HoldingCoffee and queue are:

- $\Box[a] HoldingCoffee \equiv a = pickupCoffee$
- $\lor HoldingCoffee \land \neg \exists x. a = bringCoffee(x)$

- $\Box[a] queue = y \equiv$
- $\exists x (a = requestCoffee(x) \land Enqueue(queue,x,y)) \lor$
- $\exists x (a = selectRequest(x) \land Dequeue(queue,x,y)) \lor$
- $queue = y \land$
- $\neg \exists x (a = requestCoffee(x) \lor a = selectRequest(x))$

A finite queue of size $k$ is here represented by a term $\langle x_1, \ldots, x_k \rangle$, where empty slots are denoted by the special constant $e$. The properties $Empty(queue)$, $LastFree(queue)$, $IsFirst(x,queue)$ as well as the operations $Enqueue(queue,x,y)$ and $Dequeue(queue,x,y)$ are then expressed through corresponding formulas, e.g.

$Dequeue(q_o,x,q_n) := \exists x_2,\ldots, x_k. q_o = \langle x, x_2, \ldots, x_k \rangle$

$\land q_n = \langle x_2, \ldots, x_k, e \rangle$

Example 3. For the dish robot, the initial theory is $\Sigma_0 = \{\neg \exists x. yDirty(x,y), \neg \exists x. OnRobot(x)\}$. For simplicity we assume every action is possible ($\Sigma_{pre} = \{\Box Poss(a) \equiv \top\}$). The SSAs are (omitting the robot’s location for simplicity):

- $\Box[a] Dirty(x,y) \equiv a = newdish(x,y) \lor$
- $Dirty(x,y) \land a \neq load(x,y)$
- $\Box[a] OnRobot(x) \equiv \exists y. a = load(x,y) \lor$
- $OnRobot(x) \land a \neq unload(x)$.

Next we define complex behaviours over primitive actions:

Definition 4. A program $\delta$ is built according to $\delta ::= t \mid \psi \mid \delta \mid \delta_1 \land \delta_2 \mid \exists \bar{x} \psi \mid \forall \bar{x} \psi \mid \Box \psi \mid \diamond \psi \mid \Box \delta \mid \diamond \delta$, where $t$ is an action term, $\psi$ a test for fluent formula $\psi$, $\delta$ means sequence, $\delta_1 \land \delta_2$ non-deterministic choice, $\exists \bar{x} \psi$ non-deterministic choice of argument, $\forall \bar{x} \psi$ non-deterministic iteration, and $\Box \delta$ interleaving. We then use $\psi_1 \mid \psi_2$ as abbreviation for $[\psi_1 \land \psi_2]$ and $\exists \bar{x} \psi$ as abbreviation for $[\exists \bar{x} \psi]$. Similarly, the finitary pi $\exists \bar{x} \psi$ for $\psi$ is $\exists \bar{x} \psi$, while $\forall \bar{x} \psi$ for $\psi$ is $\forall \bar{x} \psi$. Finally, $\Box$ for while $\top \land \psi$ and $\diamond$ for while $\top \land \psi$. Similarly, the finitary pi $\exists \bar{x} \psi$ for $\psi$ is $\exists \bar{x} \psi$, while $\forall \bar{x} \psi$ for $\psi$ is $\forall \bar{x} \psi$. Finally, $\Box$ for while $\top \land \psi$ and $\diamond$ for while $\top \land \psi$.

Example 5. Some temporal properties for the coffee robot:

- $EXEmpty(queue)$
- $\Box Poss(a)$
- $\delta_{e_{x_0}} = loop \exists x:Dish \pi y:Room newdish(x,y)$

Example 6. Some temporal properties for the coffee robot:

Prop1: $EXEmpty(queue)$

“Can the queue be empty after the first action?”

Prop2: $E(Empty(queue)) \land \Box \sigma$ HoldingCoffee

“Can the queue remain empty until grabbing coffee?”
Prop3: \( EG \neg \exists x \text{Occ}(\text{selectRequest}(x)) \)

"Is it possible that no request is ever served?"

(\( \text{Occ}(a) \) means \( a \) was the last action that occurred.) ▲

Example 7. Some temporal properties for the dish robot:

Prop1: \( EGD\text{Dirty}(\text{cup}_1, \text{room}_1) \)

"Is it possible that \( \text{cup}_1 \) remains dirty in \( \text{room}_1 \)?"

Prop2: \( AF\neg \text{Dirty}(\text{cup}_1, \text{room}_1) \)

"Will \( \text{cup}_1 \) in \( \text{room}_1 \) eventually be cleaned?"

Prop3: \( AF\text{Dirty}(\text{cup}_1, \text{room}_1) \)

"Will \( \text{cup}_1 \) eventually be dirty in \( \text{room}_1 \)?"

Prop4: \( E(\neg \exists y \text{Dirty}(\text{cup}_1, y) \cup \exists y \text{Dirty}(\text{cup}_1, y)) \)

"Is it possible that \( \text{cup}_1 \) becomes dirty somewhere?"

Prop5: \( AF\exists x, y \text{Dirty}(x, y) \)

"Will there eventually be a dirty dish somewhere?" ▲

3 Verification by Fixpoint Computation

The GOLOG-specific verification procedure (Clafien and Lakemeyer 2008; Clafien 2013) is inspired by classical symbolic model checking (McMillan 1993) in the sense that a systematic exploration of the state space is made by a fixpoint computation of preimages of state sets, however now involving first-order reasoning about actions. For this purpose, an \( \mathcal{ES} \) variant (Lakemeyer and Levesque 2010) of Reiter’s (2001) regression operator \( \mathcal{R} \) is employed, which replaces occurrences of \( \text{Poss}(t) \) and fluent atoms in the scope of a \( [t] \) by the right-hand side of the corresponding axiom in the BAT, for example (with simplifications):

\[
\begin{align*}
\mathcal{R}([\text{Poss}(\text{pickupCoffee})] & = \neg \text{HoldingCoffee} \\
\mathcal{R}([\text{load}(\text{cup})] x, y \text{Dirty}(x, y)) & = \exists x, y \text{Dirty}(x, y)
\end{align*}
\]

Furthermore, \( \mathcal{R} \) distributes over logical connectives.

Another ingredient for the algorithm are characteristic graphs, which encode reachable subprogram configurations. For any program \( \delta \), the graph \( G_\delta = (V, E, v_0) \) consists of a set of vertices \( V \), each of which corresponds to one reachable subprogram \( \delta' \), and where the initial node \( v_0 \) corresponds to the overall program \( \delta \). Edges \( E \) are labeled with tuples \( \pi \bar{x} : t/\psi \), intuitively denoting that a transition with action \( t \) can be taken after choosing instantiations for the variables \( \bar{x} \) under the condition that fluent formula \( \psi \) holds.

Figure 1 shows the graph for the coffee robot program, and Figure 2 the one for the dish robot domain with one room (with simplifications: action preconditions are omitted; conditions equal to \( \top \) are omitted; \( \pi \) is omitted when there are no variables to be instantiated; "\( = \)" indicates that there is one such edge instance for every element in the Dish domain).

The algorithm uses a set of labels \( \langle v, \psi \rangle \), one for each node \( v \in V \), where \( \psi \) is a fluent formula. Intuitively, a label \( \psi \) on a node \( v = \delta' \) represents all situations where \( \psi \) holds and \( \delta' \) remains to be executed. Below is the procedure for formulas of form \( E\phi \) (similar ones exist for \( EX \) and \( EU \)):

\[\text{Procedure 1 CheckEG}[\delta, \phi] \]

1. \( L' := \text{Label}(G_\delta, \phi); \quad L := \text{Label}(G_\delta, \phi); \)
2. \( \text{while } L \neq L' \text{ do} \)
3. \( L' := L; \quad L := L' \text{ AND Pre}[G_\delta, L'] \}
4. \( \text{return InitLabel}(G_\delta, L) \]

That is to say first the “old” labelling \( L' \) is initialized to label every node with \( \bot \) and the “current” labelling \( L \) marks every vertex with \( \phi \). While \( L \) and \( L' \) are not equivalent (\( \psi \equiv \psi' \) for every \( \langle v, \psi \rangle \in L, \langle v, \psi' \rangle \in L' \)), \( L \) is conjoined according to \( L_1 \text{ AND L}_2 := \{ (\langle v, \psi_1 \rangle, \langle v, \psi_2 \rangle) | \langle v, \psi_1 \rangle \in L_1, \langle v, \psi_2 \rangle \in L_2 \} \), with its pre-image

\[\text{Pre}[(\langle v, E, v_0 \rangle, L)] := \{ (\langle v, \text{Pre}[v, L] \rangle) | v \in V \}
\]

where \( \text{Pre}[v, L] \) stands for

\[\{ R[\phi \land [t] \psi] | v \xrightarrow{t/\phi} v' \in E, (v', \psi) \in L \}\]

Note the use of regression to eliminate the action term \( t \). Once converged, the procedure returns the label formula at the initial node \( v_0 \). The algorithm is sound as follows:

Theorem 8. If \( \text{CheckEG}[\delta, \phi] \) terminates, it returns a fluent formula \( \psi \) such that \( E\phi \) is valid in \( \delta \) w.r.t. \( \Sigma \) if \( \Sigma_0 \models \psi \).

The major challenge in implementing this method is that regression tends to “blow up” formulas exponentially (atoms are replaced by larger formulas), which quickly becomes unmanageable. We again drew inspiration from propositional model checking, where the introduction of ordered binary decision diagrams (OBDDs) (Bryant 1986) as a symbolic,
representing \( A \lor B \land \neg A \). OBDDs are often (depending on the variable order) more compact than other representations of propositional formulas, support efficient manipulation through Boolean operations, and are unique when fully reduced. Using appropriate data structures, the OBDDs of shared subformulas only need to be materialized once in memory (Brace, Rudell, and Bryant 1990).

Multiple approaches have been proposed for lifting OBDDs to the first-order case. While most require the formula to be in prenex form or quantifier-free (Groote and Tveretina 2003; Wang, Joshi, and Khardon 2007), a variant of the first-order algebraic decision diagrams introduced by Sanner and Boutilier (2009) seems best suited for our purposes. The idea is to "expose the propositional structure of a first-order formula" by pushing quantifiers inside as deep as possible by repeatedly applying the following rewrite rules:

\[
\begin{align*}
\exists x. \phi(x, \cdot) \lor \psi(x, \cdot) & \leadsto (\exists x\phi(x, \cdot)) \lor (\exists x\psi(x, \cdot)) \\
\forall x. \phi(x, \cdot) \land \psi(x, \cdot) & \leadsto (\forall x\phi(x, \cdot)) \land (\forall x\psi(x, \cdot)) \\
\exists x. \phi(x, \cdot) \land \psi(\cdot) & \leadsto (\exists x\phi(x, \cdot)) \land \psi(\cdot) \\
\forall x. \phi(x, \cdot) \lor \psi(\cdot) & \leadsto (\forall x\phi(x, \cdot)) \lor \psi(\cdot)
\end{align*}
\]

Furthermore, quantifiers can be eliminated using:

\[
\begin{align*}
\exists x. x = y \land \phi(x, \cdot) & \leadsto \phi(y, \cdot) \\
\forall x. x \neq y \lor \phi(x, \cdot) & \leadsto \phi(y, \cdot)
\end{align*}
\]

For example, the formula

\[
\exists x. (P(x) \lor y) \land Q(x) \land \neg P(y)
\]

is rewritten to

\[
(\exists x(P(x) \land Q(x)) \lor \forall y \neg P(y)).
\]

The boxes illustrate the propositional structure of the formula. In its OBDD representation, each box is treated as a propositional atom. If \( A \) stands for \( \exists xP(x) \) and \( B \) for \( \exists xP(x) \land Q(x) \), we obtain the OBDD shown above.

We integrated this solution into our implementation. While Sanner and Boutilier suggest to apply the rules "working from the innermost to the outermost quantifiers", we found that the opposite worked better for our setting. Moreover, we apply the method recursively to subformulas within quantifiers, and simplify using unique names wherever possible. Finally, before reconstructing the FOL formula from an OBDD, we put its propositionalized version into clausal form, determine its prime implicates, use the theorem prover to identify subsumption relations within and between clauses, and simplify accordingly: If e.g. the reduced OBDD yields \( A \lor B \), since \( \exists x(P(x) \land Q(x)) \) entails \( \exists xP(x) \), we can further simplify it to \( A \).
evant to the query property. Moreover, different decidable fragments require different abstractions, whereas the fix-point approach works irrespective of which action theory is used; there, decidable subclasses merely yield termination guarantees (Claßen et al. 2014).

5 Discussion
The verification of temporal properties of Situation Calculus theories and GLOG programs has received increasing attention in recent years. One other, major line of research explores bounded action theories (De Giacomo, Lespérance, and Patrizi 2016; De Giacomo et al. 2016), which have an infinite object domain, but where the number of object tuples in each fluent’s extension will never exceed a certain bound. Similar to (Zarrieß and Claßen 2016; Zarrieß and Claßen 2016), this allows for a finite abstraction of the state space.

However, for the most part, work so far has been of a purely theoretical nature, and the issue of implementing and practically evaluating these methods has been neglected. A notable exception is due to Kmiec and Lespérance (2014) who present a verification system for ATL properties of Situation Calculus games structures. However, their system checks state properties by evaluation rather than entailment (i.e., assumes complete information) and does not detect convergence automatically. Li and Liu (2015) furthermore present a fully automated system and employ first-order theorem proving, but address the (different) task of proving partial correctness of terminating GLOG programs.

While the evaluation scenarios presented here may seem like small toy examples, note that they are somewhat representative in size and complexity of tasks performable by our group’s household robot Caesar (Ferrie et al. 2013; Hofmann et al. 2016; Gierse et al. 2016) Nonetheless, there is clearly much work left to do to increase the practicality of our system. In particular, a more principled approach than the current ad-hoc combination of techniques we use to compactly represent first-order formulas would be in order.

Acknowledgments. This work was supported by the German Research Foundation (DFG), research unit FOR 1513 on Hybrid Reasoning for Intelligent Systems, project A1.

References


