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:

```
loop: if $\neg Empty(queue)$
        then $\pi x$ { $selectRequest(x)$;
            $pickupCoffee$; $bringCoffee(x)$ }
        else $wait$
```

Here the robot maintains a finite queue of requests to be served in order of arrival. In each cycle of an infinite **loop**, if the *queue* is not empty, the robot chooses the next pending request x (action $selectRequest(x)$), gets a cup of coffee from the machine ($pickupCoffee$), and brings it to person x ($bringCoffee(x)$). Otherwise, it waits for a short period ($wait$). Requests are represented by exogenous actions $requestCoffee(x)$ that may occur at any time during execution, where x denotes one person (of unboundedly many).

Another example is removing dirty dishes:

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

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, $Dirty(x,y)$ reads as "dirty dish x is in room y" and $load(x,y)$ as "load dish x from room y", and there are exogenous $newdish(x,y)$ actions causing a new dirty dish x to appear in room y. Unlike the previous example, the choice operators π ("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 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 $\mathcal{ES}$ (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. $\mathit{plate} \neq \mathit{cup}$ and $\mathit{load}(x) \neq \mathit{unload}(y)$). Both predicate and function symbols can be *fluent*, meaning they may change due to the execution of actions (e.g. $\mathit{OnRobot}(x)$ and queue, also $\mathit{Poss}(a)$ for action executability). There are no situation terms; to refer to future situations, two modal operators can be used: $\Box\phi$ says that ϕ holds after any sequence of actions, and $[t]\phi$ means that ϕ holds after executing action t. A formula without $\Box$ and $[\cdot]$ 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) Σ is a set of axioms consisting of: (1) Σ_0, *the initial theory*, a finite set of fluent sentences describing the initial state of the world; (2) Σ_{pre}, a *precondition axiom*, of the form[1] $\Box \mathit{Poss}(a) \equiv \pi$, where π is a fluent formula with free variable a; (3) Σ_{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] $\Box[a]F(\vec{x}) \equiv \gamma_F$, where γ_F is a fluent formula with free variables a and $\vec{x}$ (similar for functions). ▲

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

$$\begin{aligned}
&\Box \mathit{Poss}(a) \equiv \\
&\quad (a = \mathit{wait}) \lor (a = \mathit{pickupCoffee} \land \neg \mathit{HoldingCoffee}) \lor \\
&\quad \exists x\, (a = \mathit{bringCoffee}(x) \land \mathit{HoldingCoffee}) \lor \\
&\quad \exists x\, (a = \mathit{requestCoffee}(x) \land x \neq e \land \mathit{LastFree}(\mathit{queue})) \lor \\
&\quad \exists x\, (a = \mathit{selectRequest}(x) \land x \neq e \land \mathit{IsFirst}(\mathit{queue}, x))
\end{aligned}$$

The SSAs for $\mathit{HoldingCoffee}$ and queue are:

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

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 $\mathit{Empty}(\mathit{queue})$, $\mathit{LastFree}(\mathit{queue})$, $\mathit{IsFirst}(x, \mathit{queue})$ as well as the operations $\mathit{Enqueue}(\mathit{queue}, x, y)$ and $\mathit{Dequeue}(\mathit{queue}, x, y)$ are then expressed through corresponding formulas, e.g.

$$\begin{aligned}
\mathit{Dequeue}(q_o, x, q_n) \;:=\; &\exists x_2 \ldots x_k.\ q_o = \langle x, x_2, \ldots, x_k \rangle \\
&\land q_n = \langle x_2, \ldots, x_k, e \rangle
\end{aligned}$$ ▲

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

$$\begin{aligned}
&\Box[a]\mathit{Dirty}(x, y) \equiv a = \mathit{newdish}(x, y) \lor \\
&\quad \mathit{Dirty}(x, y) \land a \neq \mathit{load}(x, y) \\
&\Box[a]\mathit{OnRobot}(x) \equiv \exists y.\ a = \mathit{load}(x, y) \lor \\
&\quad \mathit{OnRobot}(x) \land a \neq \mathit{unload}(x).
\end{aligned}$$ ▲

Next we define complex behaviours over primitive actions:

Definition 4. A *program* δ is built according to

$$\delta ::= t \mid \psi? \mid \delta;\delta \mid \delta|\delta \mid \pi x \delta \mid \delta^* \mid \delta \| \delta,$$

where t is an action term, $\psi?$ a test for fluent formula ψ, $\delta;\delta$ means sequence, $\delta|\delta$ non-deterministic choice, $\pi x \delta$ non-deterministic choice of argument, δ^* non-deterministic iteration, and $\delta\|\delta$ interleaving. We then use **if** ϕ **then** δ_1 **else** δ_2 as abbreviation for $[\phi?;\delta_1] \,|\, [\neg\phi?;\delta_2]$, **while** ϕ **do** δ for $[\phi?;\delta]^*;\neg\phi?$, and **loop:**$\delta$ for **while** $\top$ **do** δ. Similarly, the finitary pick $\pi x{:}\{c_1, \ldots, c_k\}$ stands for $\delta^x_{c_1} \mid \cdots \mid \delta^x_{c_k}$, where δ^x_c means δ with variable x instantiated by c. ▲

Exogenous actions can be incorporated by having a loop that non-deterministically picks and executes one such action

$$\delta_{exo} = \textbf{loop}\ \pi x{:}\mathit{Dish}\ \pi y{:}\mathit{Room}\ \mathit{newdish}(x, y)$$

run concurrently with the actual control program δ, i.e. in the verification one analyzes the behaviour of $\delta \,\|\, \delta_{exo}$.

We use a transition semantics for programs as defined in (Claßen and Lakemeyer 2008; Claßen 2013), where as opposed to classical CONGOLOG (De Giacomo, Lespérance, and Levesque 2000), transitions are by physical actions, and tests are merely *conditions* under which a transition may be taken. Again, we leave out the formal details here.

Definition 5. Our temporal language is similar to CTL, but where in place of propositions we allow for arbitrary fluent sentences ψ, and disallow nesting of path quantifiers:

$$\begin{aligned}
&\varphi ::= \psi \mid \neg\varphi \mid \varphi \land \varphi \\
&\Psi ::= \varphi \mid \neg\Psi \mid \Psi \land \Psi \mid \boldsymbol{EX}\varphi \mid \boldsymbol{EG}\varphi \mid \boldsymbol{E}(\varphi\, \boldsymbol{U}\, \varphi)
\end{aligned}$$

We write $\boldsymbol{A}\Phi$ (Φ holds on *all* paths) for $\neg\boldsymbol{E}\neg\Phi$, $\boldsymbol{F}\Phi$ (*eventually* Φ) for $\top\, \boldsymbol{U}\, \Phi$ and $\boldsymbol{G}\Phi$ (*globally* Φ) for $\neg\boldsymbol{F}\neg\Phi$. ▲

Example 6. Some temporal properties for the coffee robot:

Prop1: $\boldsymbol{EX}\mathit{Empty}(\mathit{queue})$
"Can the queue be empty after the first action?"

Prop2: $\boldsymbol{E}(\mathit{Empty}(\mathit{queue})\ \boldsymbol{U}\ \mathit{HoldingCoffee})$
"Can the queue remain empty until grabbing coffee?"

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

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

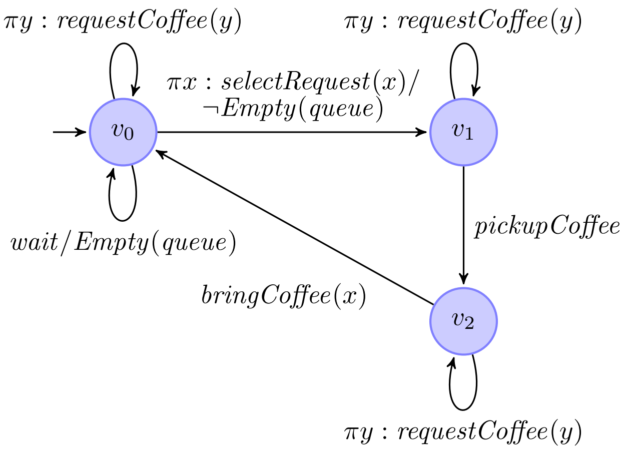

$\pi y : \mathit{requestCoffee}(y)$
$\pi y : \mathit{requestCoffee}(y)$
$\pi x : \mathit{selectRequest}(x)/$
$\neg \mathit{Empty}(\mathit{queue})$
v_0
v_1
$\mathit{wait}/\mathit{Empty}(\mathit{queue})$
$\mathit{pickupCoffee}$
$\mathit{bringCoffee}(x)$
v_2
$\pi y : \mathit{requestCoffee}(y)$

Figure 1: Characteristic Graph for the Coffee-Serving Robot

Prop3: $\boldsymbol{EG}\neg\exists x\, \mathit{Occ}(\mathit{selectRequest}(x))$
"Is it possible that no request is ever served?"
($\mathit{Occ}(a)$ means a was the last action that occurred.) ▲

Example 7. Some temporal properties for the dish robot:

Prop1: $\boldsymbol{EG}\mathit{Dirty}(\mathit{cup}_1, \mathit{room}_1)$
"Is it possible that cup_1 remains dirty in room_1?"

Prop2: $\boldsymbol{AF}\neg\mathit{Dirty}(\mathit{cup}_1, \mathit{room}_1)$
"Will cup_1 in room_1 eventually be cleaned?"

Prop3: $\boldsymbol{AF}\mathit{Dirty}(\mathit{cup}_1, \mathit{room}_1)$
"Will cup_1 eventually be dirty in room_1?"

Prop4: $\boldsymbol{E}(\neg\exists y \mathit{Dirty}(\mathit{cup}_1, y)\ \boldsymbol{U}\ \exists y \mathit{Dirty}(\mathit{cup}_1, y))$
"Is it possible that cup_1 becomes dirty somewhere?"

Prop5: $\boldsymbol{AF}\exists x, y \mathit{Dirty}(x, y)$
"Will there eventually be a dirty dish somewhere?" ▲

3 Verification by Fixpoint Computation

The GOLOG-specific verification procedure (Claßen and Lakemeyer 2008; Claßen 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 $\mathit{Poss}(t)$ 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):

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

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

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

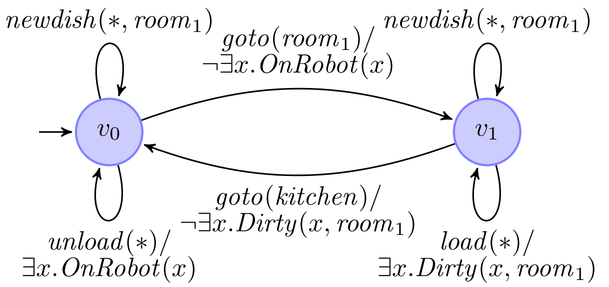

$\mathit{newdish}(*, \mathit{room}_1)$
$\mathit{goto}(\mathit{room}_1)/$
$\neg\exists x. \mathit{OnRobot}(x)$
$\mathit{newdish}(*, \mathit{room}_1)$
v_0
v_1
$\mathit{goto}(\mathit{kitchen})/$
$\neg\exists x. \mathit{Dirty}(x, \mathit{room}_1)$
$\mathit{unload}(*)/$
$\exists x. \mathit{OnRobot}(x)$
$\mathit{load}(*)/$
$\exists x. \mathit{Dirty}(x, \mathit{room}_1)$

Figure 2: Characteristic Graph for the Dish-Cleaning Robot

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

Procedure 1 CHECKEG$[\delta, \phi]$

```
$L' :=$ LABEL$[\mathcal{G}_\delta, \bot]$;    $L :=$ LABEL$[\mathcal{G}_\delta, \phi]$;
while $L \not\equiv L'$ do
    { $L' := L$;   $L := L'$ AND PRE$[\mathcal{G}_\delta, L']$ };
return INITLABEL$[\mathcal{G}_\delta, L]$
```

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 ϕ. 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

$$L_1 \text{ AND } L_2 := \{\langle v, \psi_1 \wedge \psi_2\rangle \mid \langle v, \psi_1\rangle \in L_1, \langle v, \psi_2\rangle \in L_2\}$$

with its pre-image

$$\text{PRE}[\langle V, E, v_0\rangle, L] := \{\langle v, \text{PRE}[v, L]\rangle \mid v \in V\}$$

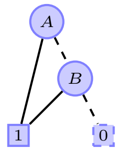
where PRE$[v, L]$ stands for

$$\bigvee \{\mathcal{R}[\phi \wedge [t]\psi] \mid v \xrightarrow{t/\phi} v' \in E,\ \langle v', \psi\rangle \in L\}.$$

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* CHECKEG$[\delta, \phi]$ *terminates, it returns a fluent formula ψ such that $\boldsymbol{EG}\phi$ is valid in δ wrt. Σ iff $\Sigma_0 \models \psi$.*

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, implicit representation of the state space caused a huge leap in the size of problems one could handle (Burch et al. 1992). An example for an OBDD over variable order $A < B$ is



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:

$$\begin{aligned}
\exists x.\, \phi(x,\cdot) \lor \psi(x,\cdot) &\leadsto (\exists x \phi(x,\cdot)) \lor (\exists x \psi(x,\cdot)) \\
\forall x.\, \phi(x,\cdot) \land \psi(x,\cdot) &\leadsto (\forall x \phi(x,\cdot)) \land (\forall x \psi(x,\cdot)) \\
\exists x.\, \phi(x,\cdot) \land \psi(\cdot) &\leadsto (\exists x \phi(x,\cdot)) \land \psi(\cdot) \\
\forall x.\, \phi(x,\cdot) \lor \psi(\cdot) &\leadsto (\forall x \phi(x,\cdot)) \lor \psi(\cdot)
\end{aligned}$$

Furthermore, quantifiers can be eliminated using:

$$\begin{aligned}
\exists x.\, x = y \land \phi(x,\cdot) &\leadsto \phi(y,\cdot) \\
\forall x.\, x \neq y \lor \phi(x,\cdot) &\leadsto \phi(y,\cdot)
\end{aligned}$$

For example, the formula

$$\exists x.\, (P(x) \lor \forall y.\, P(x) \land Q(x) \land \neg P(y))$$

is rewritten to

$$\boxed{\exists x P(x)} \lor \left(\boxed{\exists x (P(x) \land Q(x))} \land \boxed{\forall y \neg P(y)} \right).$$

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 $\exists x P(x)$ and B for $\exists x P(x) \land Q(x)$, 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 $\exists x P(x)$, we can further simplify it to A.

Q	Graph [ms]	Prop1 [ms]	Prop2 [ms]	Prop3 [ms]
1	65	61	358	270
2	65	162	563	1706
3	65	400	704	45250
5	65	797	678	–
10	66	3949	1238	–

Table 1: Evaluation Results for the Coffee Robot Domain

R	D	Graph [ms]	Prop1 [ms]	Prop2 [ms]	Prop3 [ms]	Prop4 [ms]	Prop5 [ms]
1	1	23	89	39	92	158	361
1	2	26	98	39	84	171	635
2	1	43	134	50	115	229	635
2	2	89	144	50	119	269	1111
3	1	108	180	58	140	283	911
1	3	40	100	40	85	181	909
2	3	151	145	51	122	261	1451
3	2	243	194	62	158	346	1531
3	3	429	186	62	160	353	2054
5	5	4492	300	92	222	526	5282
10	10	162798	589	214	459	1106	–

Table 2: Evaluation Results for the Dish Robot Domain

4 Experiments

We implemented a GOLOG interpreter in SWI Prolog. In contrast to many prototypes (Reiter 2001) that use Prolog's negation-as-failure mechanism and incorporate the closed-word assumption, ours operates on full FOL formulas (represented as Prolog terms). For the FOL reasoning tasks of deciding label set equivalence and entailment by the initial theory, we embedded the E theorem prover (Schulz 2013).

Tables 1 and 2 present evaluation results for the coffee and dish robot domains, respectively. All experiments were conducted on an Intel® Core$^{\text{TM}}$ i5-7500U @2.70GHz with 16GB of RAM. Rows correspond to instances with varying queue sizes (Q) in the coffee domain as well as numbers of rooms (R) and dishes (D) in the dish domain. Columns report the time in milliseconds needed to compute the characteristic graph and subsequently verify the properties from Examples 2 and 3, with a timeout threshold of 300s.

These results constitute a significant improvement in the sense that our original, naive implementation (without OBDDs or other techniques) could not solve even the smallest instances, as the blow-up due to regression was too severe. Note that the method however did not fail from exceeding memory limitations, but rather, once formulas reached a certain size, the embedded theorem prover would not terminate (within reasonable time, say 24 hours) when called to decide their equivalence. It is outside our expertise to identify the exact reason for this (we regard the prover as black box), but our observation is that apart from sheer size, quantifier depth has a strong impact. The same phenomenon occurred with other theorem provers we tried (Vampire, iProver). Only once we employ our techniques to rewrite formulas into a simpler, equivalent form, verification becomes feasible.

Finally, we also experimented with verification by finite abstraction of the state space (and then using a classical model checker). However, we found that this method is far from being competitive since constructing a complete, bisimilar abstraction is very expensive: For "local-effect" theories, where actions may only affect objects they explicitly mention as argument (e.g. the dish domain), its size can be up to double-exponential (Zarrieß and Claßen 2014). The fixpoint method on the other hand only explores states relevant to the query property. Moreover, different decidable fragments require different abstractions, whereas the fixpoint 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 GOLOG 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.[3] Similar to (Zarrieß and Claßen 2014; 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 GOLOG 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 (Ferrein 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

Brace, K. S.; Rudell, R. L.; and Bryant, R. E. 1990. Efficient implementation of a BDD package. In *DAC 1990*, 40–45. IEEE Computer Society Press.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2):142–170.

Claßen, J., and Lakemeyer, G. 2008. A logic for non-terminating Golog programs. In *KR 2008*, 589–599. AAAI Press.

Claßen, J.; Liebenberg, M.; Lakemeyer, G.; and Zarrieß, B. 2014. Exploring the boundaries of decidable verification of non-terminating Golog programs. In *AAAI 2014*, 1012–1019. AAAI Press.

Claßen, J. 2013. *Planning and Verification in the Agent Language Golog*. Ph.D. Dissertation, Department of Computer Science, RWTH Aachen University.

De Giacomo, G.; Lespérance, Y.; Patrizi, F.; and Sardiña, S. 2016. Verifying ConGolog programs on bounded situation calculus theories. In *AAAI 2016*, 950–956. AAAI Press.

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

De Giacomo, G.; Lespérance, Y.; and Patrizi, F. 2016. Bounded situation calculus action theories. *Artificial Intelligence* 237:172–203.

Ferrein, A.; Niemueller, T.; Schiffer, S.; and Lakemeyer, G. 2013. Lessons learnt from developing the embodied AI platform CAESAR for domestic service robotics. In *Papers from the AAAI 2013 Spring Symposium on Designing Intelligent Robots: Reintegrating AI II*, Technical Report SS-13-04. AAAI Press.

Gierse, G.; Niemueller, T.; Claßen, J.; and Lakemeyer, G. 2016. Interruptible task execution with resumption in Golog. In *ECAI 2016*, 1265–1273. IOS Press.

Groote, J. F., and Tveretina, O. 2003. Binary decision diagrams for first order predicate logic. *Journal of Logic and Algebraic Programming* 57(1–2):1–22.

Hofmann, T.; Niemueller, T.; Claßen, J.; and Lakemeyer, G. 2016. Continual planning in Golog. In *AAAI 2016*, 3346–3353. AAAI Press.

Kmiec, S., and Lespérance, Y. 2014. Infinite states verification in game-theoretic logics: Case studies and implementation. In *EMAS 2014, Revised Selected Papers*, volume 1244 of *LNAI*, 271–290. Springer.

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

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1–3):59–83.

Li, N., and Liu, Y. 2015. Automatic verification of partial correctness of Golog programs. In *IJCAI 2015*, 3113–3119. AAAI Press.

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

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

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

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

Schulz, S. 2013. System description: E 1.8. In *LPAR 2013*, volume 8312 of *LNCS*, 735–743. Springer.

Wang, C.; Joshi, S.; and Khardon, R. 2007. First order decision diagrams for relational MDPs. In *IJCAI 2007*, 1095–1100. AAAI Press.

Zarrieß, B., and Claßen, J. 2014. Verifying CTL* properties of Golog programs over local-effect actions. In *ECAI 2014*, 939–944. IOS Press.

Zarrieß, B., and Claßen, J. 2016. Decidable verification of Golog programs over non-local effect actions. In *AAAI 2016*, 1109–1115. AAAI Press.

[3] Our coffee domain is an instance of a bounded theory.