Automatic Verification of Liveness Properties in the Situation Calculus

Jian Li, Yongmei Liu*
Dept. of Computer Science, Sun Yat-sen University, Guangzhou 510006, China
lijian77@mail3.sysu.edu.cn, ymliu@mail.sysu.edu.cn

Abstract

In dynamic systems, liveness properties concern whether something good will eventually happen. Examples of liveness properties are termination of programs and goal achievability. In this paper, we consider the following theorem-proving problem: given an action theory and a goal, check whether the goal is achievable in every model of the action theory. We make the assumption that there are finitely many non-number objects. We propose to use mathematical induction to address this problem: we identify a natural number feature and prove by mathematical induction that for any values of the feature, the goal is achievable. Both the basis and induction steps are verified using first-order theorem provers. We propose a simple method to identify potential features which are the number of objects satisfying a certain formula by generating small models of the action theory and calling a classical planner to achieve the goal. We also propose to regress the goal via different actions and then verify whether the resulting goals are achievable. We implemented the proposed method and experimented with the blocks world domain and a number of other domains from the literature. Experimental results showed that most goals can be verified within a reasonable amount of time.

Introduction

Two kinds of important properties for dynamic systems are safety and liveness properties. Safety properties mean that something bad will never happen. Examples of safety properties are state constraints and partial correctness of programs. On the other hand, liveness properties state that something good will eventually happen. Examples of liveness properties are termination of programs and goal achievability. Model checking (Clarke et al. 1999) techniques and tools can be efficiently used to verify whether a single dynamic system has certain safety or liveness properties. However, when it comes to verify if a class of possibly infinite many dynamic systems with similar structures have certain properties, theorem-proving techniques have to be used. A class of dynamic systems with similar structures can be conveniently represented by basic action theories in the situation calculus (Reiter 2001).

There have been some works on automated verification of safety properties in the situation calculus. For example, Li, Fan, and Liu (2013) investigated automatic verification and discovery of state constraints. Li and Liu (2015) and Mo, Li, and Liu (2016) explored automatic verification of partial correctness of Golog programs via discovery of loop invariants.

There has been mainly one work on automated verification of liveness properties in the situation calculus. Lin (2008) proposed a method to verify if a goal is achievable in a set of initial states under an action theory. They introduced a notion of reduction between sets of states, and showed that if the set of the initial states can be reduced to one of its subsets, then the problem is equivalent to checking whether the goal is achievable in every initial state of the subset. However, their method depends on the restrictions that the preconditions and effects of the actions can be specified by quantifier-free formulas, and all the variables in the goal are existentially quantified.

There are also works on automated verification of more general temporal properties in the situation calculus. Claßen and Lakemeyer (2008) proposed a method to verify temporal properties of nonterminating Golog programs based on fixed-point computation and regression. Claßen (2018) reported an implementation of this method. However, in the experiments, each domain has a bounded number of objects. De Giacomo, Lespérance, and Pearce (2010) proposed a method to verify ATL-like properties of situation calculus game structures. Kmiec and Lespérance (2014) presented an evaluation-based implementation of this method for complete initial state theories. De Giacomo, Lespérance, and Patrizi (2016) investigated bounded action theories: such a theory entails that, in every situation, the number of object tuples in the extension of any fluent is bounded by a given constant. They showed that the verification of a first-order variant of μ-calculus is decidable for such theories.

In this paper, we consider the following theorem-proving problem: given an action theory and a goal, check whether the goal is achievable in every model of the action theory. We make the assumption that there are finitely many non-

*Corresponding author

number objects. The motivation of our work is to simulate humans to verify liveness. If humans are able to verify liveness for a class of possibly infinitely many dynamic systems, most likely, inductive proof is used. Thus we propose to use mathematical induction for liveness verification: we identify a natural number feature and prove by mathematical induction that for any values of the feature, the goal is achievable. Both the basis and induction steps are verified using first-order theorem provers.

We propose a simple method to identify potential features. First, we generate a pool of features of the form $\#x.\phi(x)$, meaning the number of objects x satisfying $\phi(x)$. Then we generate a number of small models of the initial KB using SMT solvers. Each small model together with the goal constitutes a classical planning problem. We get a solution for each planning problem using a classic planner such as the FF planner. A feature is useful for a planning problem if most actions in the solution make the values of the feature decrease. We consider a feature potential if it is useful for most planning problems. We also propose to regress the goal via different actions and then verify whether the resulting goals are achievable.

To illustrate our main idea, let us consider the example of verifying whether the goal $clear(A)$ is reachable in any initial state of blocks world, where A is a block constant. As humans, we can easily verify this by induction on the number of blocks above A, *i.e.*, $\#x.above(x, A)$. For this feature, the basis step involves verifying when there is no block above A, A is clear; and the induction step involves verifying when there is a block above A, an action can be executed to reduce the number of blocks above A. Both basis and induction steps can be easily verified by first-order theorem provers. We use the method described above to discover $\#x.above(x, A)$ as a potential feature.

We implemented a verification system based on the proposed method and conducted a set of experiments on it. Experimental results showed that most goals can be verified within a reasonable amount of time.

Preliminaries

In this section, we introduce the situation calculus and classical planning.

The situation calculus (Reiter 2001) is a many-sorted first-order language suitable for describing dynamic worlds. There are three disjoint sorts: $action$ for actions, $situation$ for situations, and $object$ for everything else. A situation calculus language has the following components: a constant S_0 denoting the initial situation; a binary function $do(a, s)$ denoting the successor situation to s resulting from performing action a; a binary predicate $Poss(a, s)$ meaning that action a is possible in situation s; a binary predicate $s \sqsubset s'$ meaning that situation s is a subhistory of situation s'; action functions, *e.g.*, $move(x, y)$; a finite number of relational fluents, *i.e.*, predicates taking a situation term as their last argument, *e.g.*, $ontable(x, s)$; and a finite number of situation-independent predicates and functions. We ignore functional fluents.

To formalize our assumption that there are finitely many non-number objects, we introduce a sort nat for natural numbers, and a function symbol μ of sort $object \rightarrow nat$. The intended interpretation of μ is a coding of objects into natural numbers.

If a formula refers to a particular situation τ, we call it uniform in τ. We use $\phi(s)$ to mean that formula ϕ is uniform in s. When a formula ϕ is uniform in s, we often write its situation-suppressed version, *i.e.*, the formula resulting from ϕ by removing the situation argument from every fluent. For example, we write $above(x, A) \vee above(x, B)$ to mean the formula $above(x, A, s) \vee above(x, B, s)$.

We use $s \leq s'$ to represent that $s \sqsubseteq s'$ and it is possible to perform the actions from s to s' one by one:

$$s \leq s' \doteq s \sqsubseteq s' \wedge \forall a, s^*.s \sqsubseteq do(a, s^*) \sqsubseteq s' \rightarrow Poss(a, s^*).$$

We use $exec(s)$ to denote $S_0 \leq s$, meaning s is an executable situation. For $k \geq 0$, we define $s \leq_k s'$, meaning $s \leq s'$ by at most k actions, as follows:

- $s \leq_0 s' \doteq s' = s$;
- $s \leq_{k+1} s' \doteq s \leq_k s' \vee$
  $\exists s'', a.s \leq_k s'' \wedge Poss(a, s'') \wedge s' = do(a, s'')$.

In this paper, we use $s \leq_k s'$ for small k's.

A particular domain of application where there are finitely many non-number objects is specified by a finite model basic action theory (FMBAT) of the following form:

$\mathcal{D} = \Sigma \cup \mathcal{P} \cup \mathcal{C} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{sc}$, where

1. Σ is the set of the foundational axioms for situations, including $do(a_1, s_1) = do(a_2, s_2) \rightarrow a_1 = a_2 \wedge s_1 = s_2$.
2. $\mathcal{P}$ is the second-order axiomatization of Peano arithmetic.
3. $\mathcal{C}$ is the set of the following axioms:
   - $\forall x, y.\mu(x) = \mu(y) \rightarrow x = y$;
   - $\exists n \forall x.\mu(x) \leq n$.

   The above axioms state that the codings of different objects are different and there is a largest coding.
4. $\mathcal{D}_{ap}$ is a set of action precondition axioms, one for each action function A, of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$, where $\Pi_A(\vec{x}, s)$ is uniform in s.
5. $\mathcal{D}_{ss}$ is a set of successor state axioms (SSAs), one for each relational fluent F, of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is uniform in s.
6. $\mathcal{D}_{una}$ is the set of unique names axioms for actions.
7. $\mathcal{D}_{S_0}$, the initial KB, is a set of sentences uniform in S_0.
8. $\mathcal{D}_{sc}$ is a set of sentences of the form $\forall s.exec(s) \rightarrow \psi(s)$, where ψ is called a state constraint.

We use the following notation:

- $\mathcal{D}^- = \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup$
  $\{do(a_1, s_1) = do(a_2, s_2) \rightarrow a_1 = a_2 \wedge s_1 = s_2\}$;
- $\mathcal{D}_{sc}^- = \{\psi \mid \forall s.exec(s) \rightarrow \psi(s)$ is in $\mathcal{D}_{\text{sc}}\}$.

Since the foundational axioms include a second-order induction axiom for situations, reasoning in the situation calculus is, in general, a second-order reasoning task. We use $\models$ to denote entailment in the situation calculus, and use

$\models_{fo}$ to denote classic first-order entailment. In this paper, we use first-order theorem provers to check for first-order entailments.

Example 1. In blocks world, an agent is only allowed to perform two kinds of actions: $mt(x)$ (move x to the table, provided x is clear and not on table) and $move(x, y)$ (move block x to block y, provided x and y are clear). There are four fluents: $clear(x)$, $ontable(x)$, $on(x, y)$ and $above(x, y)$. We have the following axioms where free variables are implicitly universally quantified:

Action precondition axioms:

- $Poss(mt(x), s) \equiv clear(x, s) \wedge \neg ontable(x, s)$
- $Poss(move(x, y), s) \equiv clear(x, s) \wedge clear(y, s) \wedge x \neq y$

Successor state axioms:

- $on(x, y, do(a, s)) \equiv a = move(x, y) \vee on(x, y, s) \wedge \neg(\exists z) a = move(x, z) \wedge a \neq mt(x)$
- $ontable(x, do(a, s)) \equiv a = mt(x) \vee ontable(x, s) \wedge \neg(\exists y) a = move(x, y)$
- $above(x, y, do(a, s)) \equiv above(x, y, s) \wedge \neg(\exists z) a = move(x, z) \wedge a \neq mt(x) \vee \exists z. above(z, y, s) \wedge a = move(x, z) \vee a = move(x, y)$
- $clear(x, do(a, s)) \equiv clear(x, s) \wedge \forall y. a \neq move(y, x) \vee \exists y. on(y, x, s) \wedge (a = mt(y) \vee \exists z. a = move(y, z))$

State constraints (Cook and Liu 2003):

- $on(x, y) \equiv above(x, y) \wedge \neg(\exists z)(above(x, z) \wedge above(z, y))$;
- $clear(x) \equiv \neg(\exists y) on(y, x)$;
- $ontable(x) \equiv \neg(\exists y) on(x, y)$;
- $\neg above(x, x)$;
- $above(x, y) \wedge above(y, z) \rightarrow above(x, z)$;
- $above(x, y) \wedge above(x, z) \rightarrow y = z \vee above(y, z) \vee above(z, y)$;
- $above(y, x) \wedge above(z, x) \rightarrow y = z \vee above(y, z) \vee above(z, y)$;
- $ontable(x) \vee (\exists y) above(x, y) \wedge ontable(y)$;
- $clear(x) \vee (\exists y) above(y, x) \wedge clear(y)$;
- $above(x, y) \rightarrow (\exists z) on(x, z) \vee (\exists w) on(w, y)$

An important computational mechanism for reasoning about actions is regression. Here we define a one-step regression operator, and state a simple form of the regression theorem (Reiter 2001).

Definition 1. We use $\mathcal{R}(\phi)$ to denote the formula obtained from ϕ by replacing each fluent atom $F(\vec{t}, do(\alpha, \sigma))$ with $\Phi_F(\vec{t}, \alpha, \sigma)$ and each precondition atom $Poss(A(\vec{t}), \sigma)$ with $\Pi_A(\vec{t}, \sigma)$, and further simplifying the result by using $\mathcal{D}_{una}$.

Proposition 1. $\mathcal{D} \models \phi \equiv \mathcal{R}(\phi)$.

A classical planning problem is a tuple $P = \langle \mathcal{O}, F, A, I, G \rangle$ where $\mathcal{O}$ is a set of objects, F is a set of fluents, A is a set of actions, I is an initial state and G is a goal condition. Each action $a \in A$ is a pair $(pre(a), cond(a))$ where $pre(a)$ is the precondition, and $cond(a)$ is a set of conditional effects. Each conditional effect is a pair (C, E) where C is the condition and E is the effect. Each of the goal, preconditions, conditions and effects is a set of literals. A literal is an atom p or its negation $\neg p$, which are complements to each other.

A state s can be treated as the set of literals holding in s. Action a is applicable in state s if $pre(a) \subseteq s$, and the resulting set of triggered effects, written $\mathit{eff}(s, a)$, is the union of E such that $(C, E) \in cond(a)$ and $C \subseteq s$. The result of applying a in s is a new state $\theta(s, a) = s / \neg \mathit{eff}(s, a) \cup \mathit{eff}(s, a)$, *i.e.*, the state obtained from s as follows: for each literal $l \in \mathit{eff}(s, a)$, first delete from s the complement of l and then add l.

A solution for a planning problem P is an action sequence $\pi = \langle a_1, ..., a_n \rangle$ that induces a state sequence $\langle s_0, s_1, ..., s_n \rangle$ such that $s_0 = I$, $G \subseteq s_n$, and for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and $s_i = \theta(s_{i-1}, a_i)$.

Main Framework

In this section, we introduce the main framework of our verification method.

We first formally define our verification problem.

Definition 2. Given an FMBAT $\mathcal{D}$, a goal $\varphi(s)$, *i.e.*, a formula φ uniform in s, we say φ is achievable if $\mathcal{D} \models \exists s. exec(s) \wedge \varphi(s)$.

We propose to use mathematical induction to address this problem: we identify a quantitative feature and prove by mathematical induction that for any values of the feature, the goal is achievable. Both the basis and induction steps are verified using first-order theorem provers. There are two kinds of quantitative features: when the domain involves natural numbers, we identify candidate features of the form $\phi(n)$, where $\phi(n)$ is a formula with a free variable n of sort natural number; when the domain does not involve natural numbers, we identify candidate features of the form $\#x.\phi(x)$, denoting the number of objects x satisfying the formula $\phi(x)$. We call the first kind number features, and the second kind size features. We will introduce our method for generating potential features in the next section.

We complement inductive verification with direct verification and verification by regression. When the goal is inconsistent with the state constraints, we immediately know that the goal is unachievable. When $\mathcal{D}^- \models_{fo} \exists s. S_0 \leq_k s \wedge \varphi(s)$, we immediately know that the goal is achievable. We can also verify that the goal φ is achievable by verifying the regression of it wrt a ground action α, *i.e.*, $\mathcal{R}(Poss(\alpha, s) \wedge \varphi(do(\alpha, s)))$, is achievable.

Algorithm 1 attempts to verify by direct verification and inductive verification. If φ is inconsistent with the state constraints, then return no. If $\mathcal{D}^- \models_{fo} \exists s. S_0 \leq_k s \wedge \varphi(s)$, then return yes. Otherwise, generate a set of features F by calling the procedure $genFeatures$. Then for each generated feature f, if φ can be proved by induction on f, then return yes. Finally, return unknown.

Proposition 2. *Given an FMBAT $\mathcal{D}$, and a goal φ, if $\mathcal{D}_{sc}^- \models_{fo} \neg\varphi$, then φ is not achievable.*

Example 2. In blocks world, let the goal φ be $on(A, B) \wedge on(B, A)$. Since $\mathcal{D}_{sc}^- \models_{fo} \neg\varphi$, φ is not achievable.

Algorithm 1: $diverify(\mathcal{D}, \varphi)$

Input: FMBAT $\mathcal{D}$, goal φ
Output: $yes/no/unknown$

if $\mathcal{D}_{sc}^{-} \models_{fo} \neg\varphi$ **then return** no
if $\mathcal{D}^{-} \models_{fo} \exists s.S_0 \leq_k s \wedge \varphi(s)$ **then return** yes
$F \leftarrow genFeatures(\mathcal{D}, \varphi)$
foreach $f \in F$ **do**
  if $indcution(\mathcal{D}, \varphi, f) = \top$ **then return** yes
return $unknown$

Example 3. In blocks world, suppose the initial KB contains $clear(A, S_0) \wedge clear(B, S_0)$. Let $\varphi = on(A, B) \vee on(B, A)$. Then φ is achievable via direct verification since we have: $\mathcal{D}^{-} \models_{fo} \exists s.S_0 \leq_1 s \wedge \varphi(s)$.

Algorithm 2 does a breadth-first search to see if there exists a sequence of actions $\delta = \alpha_1; \ldots \alpha_n$ such that the regression of φ wrt δ can be verified by calling $diverify$. First, if φ is inconsistent with the state constraints, then return no. Now put the goal φ in an empty queue. While the queue Q is not empty, do the following: remove the first element ψ from Q; call *diverify* on ψ; if the result is yes then return yes; if the result is unknown, then generate a set A of ground actions, where the constants are those appearing in the basic action theory or the goal, and for each α of them, put the regressed goal $\mathcal{R}(Poss(\alpha, s) \wedge \psi(do(\alpha, s)))$ into the queue. Note that we cannot use depth-first search, because this might lead us to an infinite branch.

Algorithm 2: $verify(\mathcal{D}, \varphi)$

Input: FMBAT $\mathcal{D}$, goal φ
Output: $yes/no/unknown$

if $\mathcal{D}_{sc}^{-} \models_{fo} \neg\varphi$ **then return** no
$Q \leftarrow$ empty queue
put φ in Q
while Q *is not empty and not timing-out* **do**
  remove the first element ψ from Q
  $r \leftarrow diverify(\mathcal{D}, \psi)$
  if $r = yes$ **then return** yes
  if $r = unknown$ **then**
    $A \leftarrow genActions(\mathcal{D}, \psi)$
    foreach $\alpha \in A$ **do**
      put $\mathcal{R}(Poss(\alpha, s) \wedge \psi(do(\alpha, s)))$ in Q
return $unknown$

Proposition 3. *Given an FMBAT $\mathcal{D}$, a goal φ, and a ground action α, if $\mathcal{R}(Poss(\alpha, s) \wedge \varphi(do(\alpha, s)))$ is achievable, then φ is achievable.*

Example 4. In blocks world, let the goal φ be $on(A, B)$, and let α be $move(A, B)$. Let ψ be $\mathcal{R}(Poss(\alpha, s) \wedge \varphi(do(\alpha, s)))$, *i.e.*, $clear(A) \wedge clear(B)$. Then ψ can be proved achievable by induction on $\#x.above(x, A) \vee above(x, B)$.

Algorithm 3 does inductive verification on the given feature f. When f is a number feature $\phi(n)$, the first step proves that $\exists n.\phi(n, S_0)$ holds, the basis step involves proving when $\phi(0)$ holds, the goal is achievable in at most k steps, and the inductive step involves proving when $\phi(n)$ holds and $n > 0$, for some $n' < n$, $\phi(n')$ is achievable in at most k steps. Now suppose f is a size feature $\#x.\phi(x)$. Due to our assumption that there are finitely many non-number objects, we do not need a step like that of verifying $\exists n.\phi(n, S_0)$. The basis step involves verifying when $\exists x\phi(x)$ does not hold, the goal is achievable. The inductive step involves verifying for some action function A, when $\exists x\phi(x)$ holds, there exists an action argument $\vec{y}$ such that $A(\vec{y})$ is possible and the execution of $A(\vec{y})$ makes $\#x.\phi(x)$ decrease.

Algorithm 3: $induction(\mathcal{D}, \varphi, f)$

Input: FMBAT $\mathcal{D}$, goal φ, feature f
Output: $\top/\bot$
if f is a number feature $\phi(n)$ and the following hold:
1. $\mathcal{D}_{S_0} \models_{fo} \exists n.\phi(n, S_0)$
2. $\mathcal{D}^{-} \models_{fo} \forall s.\mathcal{D}_{sc}^{-}(s) \wedge \phi(0, s) \rightarrow \exists s'.s \leq_k s' \wedge \varphi(s')$
3. $\mathcal{D}^{-} \models_{fo} \forall n, s.\mathcal{D}_{sc}^{-}(s) \wedge \phi(n, s) \wedge n > 0 \rightarrow \exists n', s'.s \leq_k s' \wedge \phi(n', s') \wedge n' < n$

then return $\top$ **else return** $\bot$
if f is a size feature $\#x.\phi(x)$ and the following hold:
1. $\mathcal{D}^{-} \models_{fo} \forall s.\mathcal{D}_{sc}^{-}(s) \wedge \neg\exists x\phi(x, s) \rightarrow \exists s'.s \leq_k s' \wedge \varphi(s')$
2. **forsome** action function A we have $\mathcal{D}^{-} \models_{fo} \forall s.\mathcal{D}_{sc}^{-}(s) \wedge \exists x\phi(x, s) \rightarrow \exists\vec{y}.Poss(A(\vec{y}), s) \wedge \exists z(\phi(z, s) \wedge \neg\phi(z, do(A(\vec{y}), s))) \wedge \forall z(\neg\phi(z, s) \rightarrow \neg\phi(z, do(A(\vec{y}), s)))$

then return $\top$ **else return** $\bot$

Proposition 4. *Given an FMBAT $\mathcal{D}$, a goal φ, and a feature f, if Algorithm 3 returns $\top$, then φ is achievable.*

Proof. We only prove the case that f is a number feature $\phi(n)$. The proof for the other case is similar. We prove that $\mathcal{D} \models \exists s.S_0 \leq s \wedge \varphi(s)$. Let M be a model of $\mathcal{D}$, and let σ be an executable situation of M s.t. $M, \sigma \models \phi(n, s)$ for some n. We prove $M, \sigma \models \exists s'.s \leq s' \wedge \varphi(s')$ by induction on n. Hence, as a special case, since $M \models \phi(n, S_0)$ for some n (due to the first entailment in Alg. 3), we have $M \models \exists s.S_0 \leq s \wedge \varphi(s)$. Basis: $M, \sigma \models \phi(0, s)$. By the second entailment in Alg. 3, $M, \sigma \models \exists s'.s \leq s' \wedge \varphi(s')$. Inductive step: $M, \sigma \models \phi(n, s)$ where $n > 0$. By the third entailment in Alg. 3, $M, \sigma \models \exists s'.s \leq s' \wedge \phi(n', s')$ for some $n' < n$. Hence there is an executable situation σ' reachable from σ such that $M, \sigma' \models \phi(n', s)$. By induction, $M, \sigma' \models \exists s'.s \leq s' \wedge \varphi(s')$. Hence $M, \sigma \models \exists s'.s \leq s' \wedge \varphi(s')$. □

Example 5. The **Oil lamp** domain is introduced by (Kmiec and Lespérance 2014). There is an infinite row of oil lamps, one for each integer. Each oil lamp x has an igniter which

can be flipped if lamp $x+1$ is on. Once we do action $flip(x)$, lamp x will be turned on. We use $on(x, s)$ to represent that lamp x is on in situation s. We have the following axioms:

- $Poss(flip(x), s) \equiv on(x + 1, s) \wedge \neg on(x, s)$;
- $on(x, do(a, s)) \equiv on(x, s) \vee \neg on(x, s) \wedge a = flip(x)$;
- $\exists n.on(n, S_0)$.

Let φ be $on(0)$. Then φ can be proved achievable via induction on $on(n)$, because the following hold:

1. $\mathcal{D}_{S_0} \models_{fo} \exists n.on(n, S_0)$;
2. $\mathcal{D}^- \models_{fo} \forall s.on(0, s) \rightarrow on(0, s)$;
3. $\mathcal{D}^- \models_{fo} \forall n, s.on(n + 1, s) \rightarrow$
   $on(n, s) \vee Poss(flip(n, s)) \wedge on(n, do(flip(n), s))$.

Example 6. In blocks world, let the goal φ be $clear(A)$, and let feature $f = \#x.above(x, A)$. Then φ can be proved achievable via induction on f, because the following hold:

1. $\mathcal{D}^- \models_{fo} \forall s.\mathcal{D}^-_{sc}(s) \wedge \neg(\exists x) above(x, A, s) \rightarrow clear(A, s)$;
2. $\mathcal{D}^- \models_{fo} \forall s.\mathcal{D}^-_{sc}(s) \wedge (\exists x) above(x, A, s) \rightarrow$
   $\exists y.Poss(mt(y), s) \wedge$
   $above(y, A, s) \wedge \neg above(y, A, do(mt(y), s)) \wedge$
   $\forall z.\neg above(z, A, s) \rightarrow \neg above(z, A, do(mt(y), s))$

We end with the soundness theorem of our verification method:

Theorem 1. *Given an FMBAT $\mathcal{D}$ and a goal φ, if Algorithm 2 returns yes, then φ is achievable; if it returns no, then φ is not achievable.*

Proof. If Algorithm 2 returns no, by Proposition 2, φ is unreachable. If Algorithm 2 returns yes, it must be the case that $diverify(\mathcal{D}, \psi)$ returns yes for some ψ in the queue. Thus ψ is obtained from φ by regressing it over a sequence of actions. By Proposition 4, ψ is achievable. By repeatedly applying Proposition 3, φ is achievable. □

Generation of Quantitative Features

In this section, we introduce how we generate a set of potential features, given a basic action theory and a goal.

We use induction in the verification of liveness. The key to using induction is to identify a numerical feature which will decrease after performing actions. So we came up with the idea that potential features can be learned from the solutions of small instances by observing what features are decreased as actions are taken. It may not be the case that every action performed makes this feature decrease. So we identify those features such that most actions make them decrease in most planning instances.

First, we describe how we generate a pool of features. A literal is an atom or the negation of an atom. A clause is the disjunction of literals, and a term is the conjunction of literals. The size of a clause or a term is the number of literals in it. For a domain involving numbers, the pool of features is the set of terms $\phi(n)$ whose size is bounded. For a domain not involving numbers, the pool of features is the set of $\#x.\phi(x)$ where $\phi(x)$ is a clause with a bounded size.

Example 7. In Oil lamp, if we set the bound of feature size as 2, the following are examples of generated features: $on(n)$, $on(n) \wedge on(n + 1)$.

Example 8. The **Corner** domain is from (Bonet, Palacios, and Geffner 2009): An agent can move in four directions on a $N \times N$ grid. Initially, she is at position (N, N). The goal is to move to position $(1, 1)$. If we set the bound of feature size to 1, the following are examples of generated features: $at(1, n)$, $\neg at(n, 1)$, $at(n, N)$, $\neg at(N, n)$, $at(n, n)$.

Example 9. In blocks world, let the goal φ be $clear(A) \wedge clear(B)$. Set the bound of feature size to 2. Then the following are examples of generated features:
$on(x, A) \vee on(x, B)$, $\neg on(x, A) \vee above(A, x)$,
$above(x, A) \vee above(x, B)$, $above(A, x) \vee above(B, x)$.

We propose to learn potential size features from the solutions of several small planning instances. We input the union of $\mathcal{D}_{S_0}$, $\{\neg\varphi\}$, and state constraints into an SMT solver, where φ is the goal formula. If the set is satisfiable, the solver will output a model. The small model, used as the initial state, together with φ and the actions, constitutes a classic planning problem. We can generate multiple small models by ensuring that they have different numbers of objects. A number feature $\phi(n)$ is a term; the selection of number features is relatively simple. Our method does not generate small models for domains involving numbers.

Algorithm 4 generates a set of potential features for a given FMBAT $\mathcal{D}$ and a goal φ. If $\mathcal{D}$ involves numbers, simply return a pool of number features. Otherwise, generate a pool F of size features. Then, by using an SMT solver, we generate a number of small models of the initial KB together with the state constraints. Each small model together with the goal constitutes a planning problem. For each planning problem P, we do the following: first, solve it with a classic planner and get a solution $\langle a_1, \ldots, a_n \rangle$. Then we compute the sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ by applying the effects of actions. For a state s_i, for a size feature $f = \#x.\phi(x)$, we can compute the value of f at s_i, *i.e.*, $\#x.\phi(x, s_i)$. Thus, we can compute the number m of actions a_i which make f decrease, *i.e.*, $\#x.\phi(x, s_{i+1}) < \#x.\phi(x, s_i)$. If the ratio m/n is over a certain threshold λ_1, then we say f is a useful feature for P. If a feature is useful for more than λ_2 percentage of the planning problems, we consider it a potential feature. Finally, we return the set of potential features.

Example 10. In blocks world, let $\varphi = clear(A) \wedge clear(B)$. Let $f_1 = \#x.above(x, A) \vee above(x, B)$, and $f_2 = \#x.above(x, A) \vee on(x, B)$. We generate two initial models described in Figure 1. By calling a classic planner, we get the following solutions:

- For I_1, $mt(D), mt(C), mt(B), mt(E)$. The values of f_1 are decreased 4 times, and f_2 4 times too.
- For I_2, $mt(C), move(D, C), move(E, D), move(F, E), move(G, F)$. The values of f_1 are decreased 5 times, and f_2 3 times.

If we set both λ_1 and λ_2 to 0.8, f_1 is chosen as a potential feature, but f_2 is not.

Algorithm 4: $genFeatures(\mathcal{D}, \varphi)$

Input: FMBAT $\mathcal{D}$, goal φ
Output: a set of potential features

if $\mathcal{D}$ involves numbers **then**
  return a pool of number features
Generate a pool F of size features
$\mathcal{P} \leftarrow genProblems(\mathcal{D}, \varphi)$
foreach $P \in \mathcal{P}$ **do**
  Generate a solution $\pi = \langle a_1, \ldots, a_n \rangle$ for P
    using a classic planner
  Compute the sequence of states $s_0, s_1 \ldots, s_n$ for π
  foreach $f \in F$ **do**
    Count the number m of actions a_i that reduce f,
      i.e., $\#x.\phi(x, s_{i+1}) < \#x.\phi(x, s_i)$
    f is useful for P **if** $m/n \geq \lambda_1$
return *the set of features which is useful*
  for $\geq \lambda_2$ *percentage of problems*

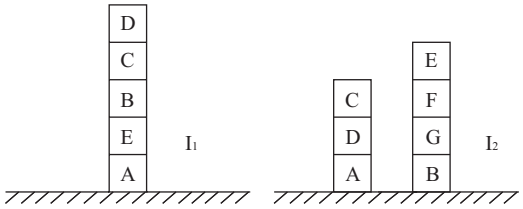

Figure 1: Two initial blocks world models

Experimentation

We implemented our verification method in python3 using the FF planner (Hoffmann and Nebel 2001) and the SMT solver Z3 (de Moura and Bjørner 2008). All experiments were run on a Linux machine with 2.40GHz CPU and 16GB RAM. We set up the implementation parameters as follows: The time-out bound for Z3 is 5 seconds; when generating potential features, the bound of feature size is 2, the number of planning problems is 3, $\lambda_1 = 0.8$ and $\lambda_2 = 1.0$; the k in $\leq_k$ is set to 2.

Table 1 shows our experimental results for blocks world. The first column $goal$ represents the goal to verify, the second column $feature$ gives the feature which is used to do inductive verification, and the third column $time$ shows the total time for verification (with unit second). The third goal $ontable(A) \wedge \forall x(x \neq A \rightarrow above(x, A))$ means that there is a single tower with A at the bottom. The fifth goal $\forall x.x = A \vee above(A, x)$ means that there is a single tower with A at the top. The first three goals can be verified directly by induction. The fourth goal $on(A, B)$ is verified by regressing it to $clear(A) \wedge clear(B)$, which can be verified directly by induction. For the fifth goal, our system is not able to generate a potential feature and hence not able to accomplish the verification. In fact, there are no suitable features using our feature language.

We also experimented with the following domains involving numbers. For each domain, we include the axioms.

$goal$	$feature$	$time$(s)
$clear(A)$	$\#x.above(x, A)$	56.78
$\forall x.ontable(x)$	$\#x.\neg ontable(x)$	70.63
$ontable(A) \wedge$ $\forall x(x \neq A \rightarrow$ $above(x, A))$	$\#x.\neg above(x, A)$	52.42
$on(A, B)$	$\#x.above(x, A) \vee$ $above(x, B)$	168.66
$\forall x.x = A \vee$ $above(A, x)$	-	-

Table 1: Experimental results for blocks world

Oil lamp See Example 5.

Oil lamp* The same as **Oil lamp** except that light x can be flipped if both light $x + 1$ and light $x + 2$ are on.

- $Poss(flip(x), s) \equiv on(x{+}1, s) \wedge on(x{+}2, s) \wedge \neg on(x, s)$

Chop tree (Sardiña et al. 2004): There is a tree that can be chopped down, but the agent does not know how many times that she needs to chop at it in order to bring it down. We introduce a $size(n)$ fluent to mean that n is the number of chops that is needed to bring down the tree. We want to verify that the tree will eventually be cut down. We use the encoding from (Lin 2008): $down(s)$ means the tree is down in situation s, action $chop(m, n)$ decreases the size of the tree from m to n, provided $n = m - 1$.

- $Poss(chop(m, n), s) \equiv$
  $\neg down(s) \wedge size(m, s) \wedge n = m - 1$
- $down(do(a, s)) \equiv a = chop(1, 0)$
- $size(n, do(a, s)) \equiv \exists m.a = chop(m, n)$
- $\exists n.size(n, S_0)$

Pick up Stone: There are n stones in the initial state. An agent can pick up one stone if there are an odd number of stones, and she can pick up two stones if there are an even number of stones. We want to verify that eventually no stone can be picked up. We use symbols: $nstone(n, s)$ means the number of stones in situation s is n, action $take(m)$ means picking up m stones.

- $Poss(take(m), s) \equiv \exists n.nstone(n, s) \wedge n > 0 \wedge$
  $(n\%2 = 1 \wedge m = 1 \vee n\%2 = 0 \wedge m = 2)$
- $nstone(n, do(a, s)) \equiv$
  $\exists m.a = take(m) \wedge nstone(m + n, s)$
- $\exists n.nstone(n, S_0)$

Corner: See Example 8. We use $at(m, n, s)$ to mean the agent is at position (m, n) in situation s. There are four actions: up, $down$, $left$, and $right$.

- $Poss(left, s) \equiv \exists m, n.at(m, n, s) \wedge m > 0, \ldots$
- $at(m, n, do(a, s)) \equiv$
  $a = up \wedge at(m, n{-}1, s) \vee a = down \wedge at(m, n{+}1, s) \vee$
  $a = left \wedge at(m{+}1, n, s) \vee a = right \wedge at(m{-}1, n, s)$
- $at(N, N, S_0)$

$Domain$	$goal$	$feature$	$time$(s)
oil lamp	$on(0)$	$on(n)$	4.23
oil lamp*	$on(0)$	$on(n) \wedge on(n+1)$	5.70
chop tree	$down$	$size(n)$	3.99
pick stone	$nstone(0)$	$nstone(n)$	7.88
corner	$at(1,1)$	$at(n,n)$	3.21

Table 2: Experimental results for other domains

Table 2 shows the experimental results for the above domains. All domains are successfully verified. For the Pick up Stone domain, the proof implies a generalized plan to achieve the goal: when the initial number of stones is even, the agent repeatedly picks up two stones; when the initial number of stones is odd, the agent first picks up one stone, and then repeatedly picks up two stones. Intuitively, the domain Corner can be verified by a double induction. However, our system verifies it by a single induction, and the feature used for induction is $at(n, n)$. So the proof implies a generalized plan to achieve the goal: for n from N to 2, move from position (n, n) to $(n-1, n-1)$ using two actions.

Conclusions

In this paper, we have developed a sound but incomplete method for automated theorem proving of liveness properties. The main idea of our method is to prove goal achievability by induction on quantitative features where both basis and induction steps can be proved by first-order theorem provers. We propose a simple method to identify potential features which are the number of objects satisfying a certain formula by generating small models of the action theory and calling a classic planner to achieve the goal. We also propose to regress the goal via different actions and then verify whether the resulting goals are achievable. Compared with Lin's method to prove goal achievability, a clear advantage of our method is that we do not restrict the form of the basic action theory or the goal.

There are two limitations of our current system. Firstly, it only supports single induction. In the future, we will remove this limitation to allow nested induction. Secondly, our system only generates simple features in the form of quantifier-free clauses or terms. In the future, we will explore more expressive features represented by quantified formulas such as $\exists x, y.at(x, y) \wedge n = x + y$. Moreover, in our work, the proof of goal achievability implies a generalized plan to achieve the goal, as we show for the Pick up Stone and Corner examples. Generalized planning aims at finding a single solution which works for possibly infinitely many similar planning problems (Levesque 2005; Srivastava, Immerman, and Zilberstein 2008). So far a challenge for generalized planning is to guarantee the correctness of solutions. In the future, we will explore the application of our work in generalized planning with correctness guarantee.

Acknowledgments

We acknowledge support from the Natural Science Foundation of China under Grant No. 61572535.

References

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. ICAPS-09*.

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

Claßen, J., and Lakemeyer, G. 2008. A logic for non-terminating golog programs. In *Proc. KR-08*, 589–599.

Claßen, J. 2018. Symbolic verification of golog programs with first-order bdds. In *Proc. KR-18*, 524–529.

Cook, S. A., and Liu, Y. 2003. A complete axiomatization for blocks world. *J. Log. Comput.* 13(4):581–594.

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

De Giacomo, G.; Lespérance, Y.; and Pearce, A. R. 2010. Situation calculus based programs for representing and reasoning about game structures. In *Proc. KR-10*, 445–455.

de Moura, L. M., and Bjørner, N. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, 337–340.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.* 14:253–302.

Kmiec, S., and Lespérance, Y. 2014. Infinite states verification in game-theoretic logics: Case studies and implementation. In *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014*, 271–290.

Levesque, H. J. 2005. Planning with loops. In *Proc. IJCAI-05*, 509–515.

Li, N., and Liu, Y. 2015. Automatic verification of partial correctness of golog programs. In *Proc. IJCAI-15*, 3113–3119.

Li, N.; Fan, Y.; and Liu, Y. 2013. Reasoning about state constraints in the situation calculus. In *Proc. IJCAI-13*.

Lin, F. 2008. Proving goal achievability. In *Proc. KR-08*, 621–628.

Mo, P.; Li, N.; and Liu, Y. 2016. Automatic verification of golog programs via predicate abstraction. In *Proc. ECAI-2016*.

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

Sardiña, S.; De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2004. On the semantics of deliberation in indigolog - from theory to implementation. *Ann. Math. Artif. Intell.* 41(2-4):259–299.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proc. AAAI-08*, 991–997.