Reasoning about State Constraints in the Situation Calculus

Naiqi Li[1] **Yi Fan**[1,2] **Yongmei Liu**[1]
[1]Department of Computer Science,
Sun Yat-sen University, Guangzhou 510006, China
[2]Institute for Integrated and Intelligent Systems,
Griffith University, Brisbane, Australia
ymliu@mail.sysu.edu.cn

Abstract

In dynamic systems, state constraints are formulas that hold in every reachable state. It has been shown that state constraints can be used to greatly reduce the planning search space. They are also useful in program verification. In this paper, we propose a sound but incomplete method for automatic verification and discovery of $\forall^*\exists^*$ state constraints for a class of action theories that include many planning benchmarks. Our method is formulated in the situation calculus, theoretically based on Skolemization and Herbrand Theorem, and implemented with SAT solvers. Basically, we verify a state constraint by strengthening it in a novel and smart way so that it becomes a state invariant. We experimented with the blocks world, logistics and satellite domains, and the results showed that, almost all known state constraints can be verified in a reasonable amount of time, and meanwhile succinct and intuitive related state constraints are discovered.

1 Introduction

When an agent is working in a dynamic world, she needs to recognize and verify some laws of the world. Two kinds of laws widely investigated are state invariants and state constraints [Lin, 2004]. Roughly, state invariants are formulas that if true in a state, will be true in all successor states, while state constraints are formulas that hold in any reachable state. The key difference between them is that the first is a first-order property while the second is a second-order property.

In planning, state constraints act as domain specific knowledge which reduces the search space greatly [Gerevini and Schubert, 1998]. Experiments show that they speed up planning drastically in some cases [Kautz and Selman, 1992; Gerevini and Schubert, 1998; Refanidis and Vlahavas, 2000]. State constraints can also be used to debug domain axiomatization. In C programs, we sometimes need to ensure that a pointer never moves out of an area, or a stack is never overflowed, so we are to verify whether they are state constraints.

The problems of discovering state constraints and invariants have been investigated in both the planning and reasoning about actions communities. The works from the planning community are more empirical. In the reasoning about action community, Lin [2004] developed a method for discovering state invariants in domains of unbounded size through exhaustive search in selected small domains. He did experiments in blocks world and logistics domains, and discovered a complete set of constraints in the second, that is, any illegal state must violate at least one of them. However, there is no soundness guarantee when his method is applied to non-$\forall^*$-formulas. Kelly and Pearce [2010] proposed a method for computing whether a formula ϕ holds in all successor states of a given state, but the procedure does not necessarily halt since it is based on first-order entailment and fixpoint computation, and no implementation was reported.

In this paper, we propose a sound but incomplete method for automatic verification and discovery of $\forall^*\exists^*$ state constraints for a class of action theories that include many planning benchmarks. Our method is formulated in the situation calculus, theoretically based on Skolemization and Herbrand Theorem, and implemented with SAT solvers.

Basically, we verify a state constraint by strengthening it with a sequence of formulas so that it becomes an invariant constraint, and the formulas used to strengthen it are called its joint constraints. So the problem becomes how to systematically obtain such joint constraints. We have developed a novel way to do so. Given a $\forall^*$ formula ϕ, assuming it holds in the initial state, we test if ϕ is a state invariant, using those previously verified constraints as the background information. If so we confirm that ϕ is a state constraint. Otherwise, we pick up a successor state s' of state s such that $\phi(s) \supset \phi(s')$ is not entailed by the existing set of state constraints. We regress $\phi(s')$ to get an equivalent formula $\psi(s)$. For the class of action theories we consider, $\psi(s)$ is also a $\forall^*$ formula. We convert $\psi(s)$ into CNF, and we call each conjunct of it a derivant of ϕ. The derivants of ϕ not logically implied by ϕ have the nice property that if ϕ can be strengthened to become an invariant constraint, then they are joint constraints of ϕ. Next we obtain a derivant C not implied by ϕ, use it to strengthen ϕ, and continue this procedure with $\phi \wedge C$. Besides, we also check if ϕ can be strengthened by some subset of C in order to obtain more intuitive and succinct joint constraints.

We did experiments in the blocks world, logistics and satellite domains, and the results showed that, almost all known constraints are verified in a reasonable time period, and meanwhile succinct and intuitive joint constraints are discovered.

2 Preliminaries

In this section, we introduce the background work of our paper, *i.e.*, Herbrand Theorem and Skolemization, the situation calculus, state constraints and state invariants.

2.1 Herbrand Theorem

Herbrand Theorem reduces the satisfiability of a set of $\forall^*$ first-order sentences to that of a possibly infinite set of propositional formulas.

Theorem 2.1. *Let $\mathcal{L}$ be a first-order language with at least one constant, and let Φ be a set of $\forall^*$ $\mathcal{L}$-sentences including the equality axioms for $\mathcal{L}$. Then Φ is unsatisfiable iff some finite set Φ_0 of $\mathcal{L}$-ground instances of sentences in Φ is propositionally unsatisfiable.*

Herbrand Theorem only applies to determining the satisfiability of $\forall^*$ sentences. In other cases, we need Skolemization.

Definition 2.2. Let ϕ be a first-order sentence in prenex normal form. We construct the functional form of ϕ by removing each existential quantifier $\exists y$ in the prenex and replacing y in the formula by $f(x_1, \ldots, x_n)$, where f is a new function symbol, and $x_1, \ldots, x_n$ are the universally quantified variables that precede $\exists y$ in the prefix. To form the functional forms of a set of sentences, it is necessary to make every Skolem function symbol introduced distinct from all Skolem function symbols in all other formulas.

Theorem 2.3. *A set Φ of sentences is satisfiable iff the set of functional forms of sentences in Φ is satisfiable.*

2.2 The Situation Calculus

We will not go over the language of the situation calculus here except to note 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 $s \sqsubset s'$ meaning that situation s is a proper subhistory of s'; a binary predicate $Poss(a, s)$ meaning that action a is possible in situation s; a finite number of action functions; a finite number of relational and functional fluents, *i.e.*, predicates and functions taking a situation term as their last argument.

Often, we need to restrict our attention to formulas that do not refer to any situations other than a particular one τ, and we call such formulas uniform in τ. We use $\phi(\tau)$ to denote that ϕ is uniform in τ. A situation s is executable if it is possible to perform the actions in s one by one:

$$Exec(s) \stackrel{def}{=} \forall a, s'.do(a, s') \sqsubseteq s \supset Poss(a, s').$$

In the situation calculus, a particular domain of application is specified by a basic action theory (BAT) of the form:

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

1. Σ is the set of the foundational axioms for situations, including a second-order induction axiom for situations:
$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s).$$
2. $\mathcal{D}_{ap}$ contains a single precondition axiom of the form $Poss(a, s) \equiv \Pi(a, s)$, where $\Pi(a, s)$ is uniform in s.
3. $\mathcal{D}_{ss}$ is a set of successor state axioms (SSAs), one for each 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.
4. $\mathcal{D}_{una}$ is the set of unique names axioms for actions.
5. $\mathcal{D}_{S_0}$, the initial KB, is a set of sentences uniform in S_0.

Actions in many domains have only local effects in the sense that if an action $A(\vec{c})$ changes the truth value of an atom $F(\vec{e}, s)$, then $\vec{e}$ is contained in $\vec{c}$. This contrasts actions that have non-local effects such as moving a briefcase, which will also move all the objects inside the briefcase without having mentioned them. We skip the the formal definition here.

We say that $\mathcal{D}_{ap}$ and $\mathcal{D}_{ss}$ are essentially quantifier-free if for each action function $A(\vec{y})$ and each fluent F, by using $\mathcal{D}_{una}$, both $\Pi(A(\vec{y}), s)$ and $\Phi_F(\vec{x}, A(\vec{y}), s)$ can be simplified to quantifier-free formulas.

Definition 2.4. A basic action theory is simple if every action is local-effect, $\mathcal{D}_{ap}$ and $\mathcal{D}_{ss}$ are essentially quantifier-free, and $\mathcal{D}_{S_0}$ is a consistent set of $\forall^*$ formulas.

Example 2.5. *Below are parts of a simple BAT:*

$Poss(stack(x, y), s) \equiv holding(x, s) \wedge clear(y, s) \wedge x \neq y$
$on(x, y, do(a, s)) \equiv a = stack(x, y) \vee$
$\quad on(x, y, s) \wedge a \neq unstack(x, y)$
$holding(x, do(a, s)) \equiv$
$\quad (\exists y)a = unstack(x, y) \vee a = pickup(x) \vee$
$\quad holding(x, s) \wedge (\neg\exists y)a = stack(x, y) \wedge a \neq putdown(x)$

We assume that $\mathcal{D}$ is simple throughout this paper.

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

Definition 2.6. We use $\hat{\mathcal{R}}_{\mathcal{D}}[\phi]$ 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 $Poss(\alpha, \sigma)$ with $\Pi(\alpha, \sigma)$, and further simplifying the result by using $\mathcal{D}_{una}$.

Theorem 2.7. $\mathcal{D} \models \phi \equiv \hat{\mathcal{R}}_{\mathcal{D}}[\phi]$.

2.3 State Constraints and State Invariants

We now formally define state constraints and state invariants. We use $\models$ to denote entailment in the situation calculus, and use $\models_{\text{FOL}}$ to denote classic first-order entailment. We follow the definition of state constraints in [Reiter, 2001]:

Definition 2.8. Given $\mathcal{D}$ and a formula $\phi(s)$, $\phi(s)$ is a state constraint for $\mathcal{D}$ if $\mathcal{D} \models \forall s.Exec(s) \supset \phi(s)$.

So state constraints are formulas which hold in all executable situations. Since $\mathcal{D}$ includes a second-order induction axiom for situations, verifying state constraints is a second-order reasoning task. We make

$$\mathcal{D}_{\mathcal{SC}} = \{\varphi(s) \mid \varphi(s) \text{ is a verified constraint}\}$$

an extra component of $\mathcal{D}$. We also abuse $\mathcal{D}_{\mathcal{SC}}$ as the conjunction of the elements in $\mathcal{D}_{\mathcal{SC}}$.

Our approach exploits verified state constraints to verify other constraints, so we have a slightly different definition of state invariants below:

Definition 2.9. Let $\mathcal{D}_{vc}$ be a set of formulas uniform in s, called the background information. A formula $\phi(s)$ is a state invariant for $\mathcal{D}$ wrt $\mathcal{D}_{vc}$ if

$$\mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \models_{\text{FOL}} \forall a, s.\mathcal{D}_{vc} \wedge \phi(s) \wedge Poss(a, s) \supset \phi(do(a, s)).$$

Given the background information $\mathcal{D}_{vc}$, state invariants are those that if true in a situation, will be true in all successor situations. Clearly, verifying state invariants is a first-order reasoning task. We will use $\mathcal{D}_{\mathcal{SC}}$ to collect those verified constraints, and use it as the background information. Obviously, if $\phi(s)$ is a state invariant wrt some subset of $\mathcal{D}_{\mathcal{SC}}$, then $\phi(s)$ is a state invariant wrt $\mathcal{D}_{\mathcal{SC}}$.

Definition 2.10. *We say $\phi(s)$ is a state invariant for $\mathcal{D}$, if $\phi(s)$ is a state invariant wrt a set of state constraints for $\mathcal{D}$.*

The following shows closure properties of constraints.

Proposition 2.11. *1. $\phi(s) \wedge \psi(s)$ is a constraint iff both $\phi(s)$ and $\psi(s)$ are constraints.*

2. If $\phi(s)$ is a constraint, then so is $\phi(s) \vee \psi(s)$.

The following result shows that we can verify a constraint by proving that it is an invariant.

Proposition 2.12. *If every element of $\mathcal{D}_{\mathcal{SC}}$ is an invariant, $\mathcal{D}_{S_0} \models_{\text{FOL}} \varphi(S_0)$, and $\varphi(s)$ is an invariant, then $\varphi(s)$ is a constraint.*

Our approach verifies a constraint by strengthening it to be an invariant, which is put into $\mathcal{D}_{\mathcal{SC}}$, so it can ensure that every element of $\mathcal{D}_{\mathcal{SC}}$ is an invariant.

3 Theoretical Foundations

In this section, we introduce the theoretical foundations of our work, *i.e.*, the concepts of joint constraints and derivants, and incomplete reasoning based on Hebrand Theorem.

3.1 Joint Constraints and Derivants

We now introduce the most important concepts of this paper.

Definition 3.1. An invariant constraint is a constraint which is also an invariant. If $\varphi(s) \wedge \eta(s)$ is an invariant constraint, we call $\varphi(s)$ a partial invariant constraint. If $\varphi(s) \wedge \eta(s)$ is a partial invariant constraint and $\eta(s)$ is not logically implied by $\varphi(s)$, we call $\eta(s)$ a joint constraint for $\varphi(s)$.

We collects a set of invariant constraints in $\mathcal{D}_{\mathcal{SC}}$. Given a $\forall^*$ partial invariant constraint, we will strengthen it with a sequence of joint constraints to approach an invariant constraint. Below we show how to obtain such joint constraints.

Theorem 3.2. *$\varphi(s)$ is a constraint for $\mathcal{D}$ iff $\mathcal{D}_{S_0} \models_{\text{FOL}} \varphi(S_0)$ and $\forall a.\hat{\mathcal{R}}_{\mathcal{D}}[Poss(a, s) \supset \varphi(do(a, s))]$ is a constraint.*

By Proposition 2.11 and Theorem 3.2, we have

Proposition 3.3. *If $\varphi(s)$ is a constraint for $\mathcal{D}$, then for any action function $A(\vec{x})$, the following is also a constraint:*

$$\forall \vec{x}.\hat{\mathcal{R}}_{\mathcal{D}}[Poss(A(\vec{x}), s) \supset \varphi(do(A(\vec{x}), s)].$$

Let $\varphi(s)$ be a formula $\forall \vec{y}\phi(s)$, where $\phi(s)$ is quantifier-free. As $\mathcal{D}$ is simple, $\hat{\mathcal{R}}_{\mathcal{D}}[Poss(A(\vec{x}), s) \supset \varphi(do(A(\vec{x}), s))]$ can be put into a formula $\forall \vec{y}\psi(s)$, where $\psi(s)$ is quantifer-free and can be put into CNF. So we have the definition below:

Definition 3.4. *Let $\varphi(s)$ be a $\forall^*$ formula. We transform $\hat{\mathcal{R}}_{\mathcal{D}}[Poss(A(\vec{x}), s) \supset \varphi(do(A(\vec{x}), s))]$ into a prenex CNF formula. We call each conjunct a derivant of $\varphi(s)$.*

By Proposition 2.11, we have the following proposition which shows that the derivants of constraints are constraints.

Proposition 3.5. *Let $\varphi(s)$ be a constraint, and $C(s)$ a derivant of $\varphi(s)$. Then $C(s)$ is a constraint.*

Below we show that the derivants of a partial invariant constraint φ that are not implied by φ are its joint constraints.

Theorem 3.6 (Main Theorem)**.** *If $\varphi(s) \wedge \eta(s)$ is an invariant constraint, and $C(s)$ is a derivant of $\varphi(s)$, then*

1. $\models_{\text{FOL}} \forall s.\mathcal{D}_{\mathcal{SC}} \wedge \varphi(s) \wedge \eta(s) \supset C(s)$.

2. $\varphi(s) \wedge C(s)$ is a partial invariant constraint.

Suppose that $\varphi(s)$ is a partial invariant constraint but not an invariant. Our approach will choose a derivant $C(s)$ of $\varphi(s)$ such that $\not\models_{\text{FOL}} \forall s.\mathcal{D}_{\mathcal{SC}} \wedge \varphi(s) \supset C(s)$, so $\varphi(s) \wedge C(s)$ is strictly stronger than $\varphi(s)$. Thus our approach will strictly strengthen $\varphi(s)$ towards an invariant constraint.

The notion below is used to describe our algorithms.

Definition 3.7. The length of a CNF formula F, written $|F|$, is defined to be the sum of the length of the clauses.

3.2 Incomplete Reasoning via Herbrand Theorem

In verifying state constraints, we need to check first-order entailment of $\forall^*\exists^*$ sentences, which can be reduced to checking the satisfiability of a set of sentences. By Skolemization, this can be reduced to checking the satisfiability of the functional forms of the sentences, which is a set of $\forall^*$ sentences.

We use an incomplete algorithm to check the satisfiability of a set of $\forall^*$ $\mathcal{L}$-sentences. We first test the satisfiability of a subset of $\mathcal{L}$-ground instances using some finite set $\mathcal{H}_0$ of terms in $\mathcal{L}$. If it is satisfiable, we test the satisfiability of a larger subset of $\mathcal{L}$-ground instances, using a finite set $\mathcal{H}_1$ of terms such that $\mathcal{H}_0 \subset \mathcal{H}_1$. We repeatedly do so with larger subset of terms until unsatisfiability is confirmed or some bound is reached. We now formalize the above idea.

Definition 3.8. Let $\mathcal{L}$ be a first-order language with at least one constant symbol. We define partial Herbarnd universe for $\mathcal{L}$ as follows: $\mathcal{H}_0^{\mathcal{L}}$ is the set of constants of $\mathcal{L}$. For $n \geq 0$, $\mathcal{H}_{n+1}^{\mathcal{L}}$ is the union of $\mathcal{H}_n^{\mathcal{L}}$ and the set of terms $f(\vec{t})$ such that f is a function of $\mathcal{L}$, and each t_i belongs to $\mathcal{H}_n^{\mathcal{L}}$.

Let Φ be a set of $\forall^*$ $\mathcal{L}$-sentences, and H a set of ground terms of $\mathcal{L}$. We use $\text{gnd}(\Phi)|H$ to denote the set of $\mathcal{L}$-ground instances of sentences in Φ using ground terms from H.

By Theorem 2.1 and Theorem 2.3, we obtain a theorem to establish entailment of $\forall^*\exists^*$ sentences as follows.

Theorem 3.9. *Let Ψ be a set of $\forall^*\exists^*$ sentences, and ψ a $\forall^*\exists^*$ sentence. Let Φ be the set of Skolem functional forms of $\Psi \cup \{\neg\psi\}$, and $\mathcal{L}$ the language of Φ. Then $\Psi \models_{\text{FOL}} \psi$ iff there exists some n such that gnd$(\Phi)|\mathcal{H}_n^{\mathcal{L}}$ is unsatisfiable.*

We use $\Psi \models_{\text{FOL}}^n \psi$ to denote that $\text{gnd}(\Phi)|\mathcal{H}_n^{\mathcal{L}}$ is unsatisfiable.

We remark that all $\forall^*\exists^*$ constraints in this paper are verified by $\text{gnd}(\Phi)|\mathcal{H}_1^{\mathcal{L}}$, so this method is empirically effective.

4 Algorithms

Below, we use $\psi = \phi(s) \uparrow s$ to mean that ψ is the formula obtained from $\phi(s)$ by replacing each fluent atom $F(\vec{t}, s)$ by $F(\vec{t})$, and we say ψ is situation-suppressed. We use $\psi[s]$ to denote the formula obtained from ψ by restoring the situation argument s to all predicates in ψ. Note that all input and output formulas of our algorithms are situation-suppressed.

4.1 Verifying State Constraints

Algo. 1 verifies if the input $\forall^*$ formula ϕ is a state constraint for the input BAT $\mathcal{D}$. If ϕ is verified to be a state constraint, the algorithm outputs an invariant constraint subsuming ϕ.

Algorithm 1: $veri\_SC(\phi, \mathcal{D}, \mathcal{M}, \beta)$

input : A $\forall^*$ formula ϕ, a BAT $\mathcal{D}$, a set $\mathcal{M}$ of small initial models, a bound β
output: an invariant constraint subsuming ϕ, or $False$, or $Non\text{-}deter$

1 $\Delta \leftarrow \{\phi[s]\}$
2 **while** $True$ **do**
3 remove a shortest formula $\varphi(s)$ from Δ
4 **if** $|\varphi(s)| \geq \beta$ **then return** $Non\text{-}deter$
5 **if** $\exists M \in \mathcal{M}. M \not\models \varphi(S_0)$ *or* $\mathcal{D}_{S_0} \not\models_{\text{FOL}} \varphi(S_0)$ **then**
6 **if** $\varphi(s)$ is integral **then return** $False$
7 **else** continue
8 IS_SI $\leftarrow True$
9 **foreach** *action function* $A(\vec{x})$ **do**
10 $re \leftarrow \hat{\mathcal{R}}_{\mathcal{D}}[Poss(A(\vec{x}), s) \supset \varphi(do(A(\vec{x}), s))]$
11 $C_1(s) \wedge \ldots \wedge C_n(s) \leftarrow \texttt{cnf}(re)$
12 **if** $\exists C_i(s). \not\models_{\text{FOL}} \forall \vec{x}, s. \varphi(s) \wedge \mathcal{D}_{\mathcal{SC}} \supset C_i(s)$ **then**
13 IS_SI $\leftarrow False$
14 **foreach** $\eta(s) \subseteq C_i(s)$ **do**
15 $\Delta \leftarrow \Delta \cup \{\forall \vec{x}, s. \varphi(s) \wedge \eta(s)\}$
16 break
17 **if** $IS\_SI = True$ **then return** $\varphi(s) \uparrow s$

Now we illustrate the two other inputs of the algorithm: a set $\mathcal{M}$ of small initial models and a bound β. Given $\varphi(s)$, in verifying whether $\varphi(s)$ is a constraint, we might strengthen it, so we will have to verify longer and longer formulas. We believe that most meaningful constraints can be proved after being strengthened a few times, so we use β to bound $|\varphi(s)|$ in Line 4. To speed up the evaluation of $\mathcal{D}_{S_0} \models_{\text{FOL}} \varphi(S_0)$, we use a small set $\mathcal{M}$ of small models of $\mathcal{D}_{S_0}$ in Line 5. If $\varphi(S_0)$ is falsified by a model, it cannot be entailed, and this improves efficiency greatly. We say a member of Δ is integral if it is the input formula or it is obtained in Line 15 with $\eta(s) = C_i(s)$.

Here's the work flow. In Line 3 the shortest formula is selected. In Line 11, the formula re is transformed into CNF, $C_1(s) \wedge \ldots \wedge C_n(s)$. Line 12 successfully proves that $\models_{\text{FOL}} \forall \vec{x}, s. \mathcal{D}_{\mathcal{SC}} \wedge \varphi(s) \wedge Poss(A(\vec{x}), s) \supset \varphi(do(A(\vec{x}), s)$, or it picks up a clause $C_i(s)$ not entailed by $\mathcal{D}_{\mathcal{SC}} \wedge \varphi(s)$ and use it to strengthen $\varphi(s)$. Line 14 considers all subsets of $C_i(s)$, and this is not very much a problem to efficiency, since in practice, $|C_i(s)|$ is relatively small. The motivation is that it is likely that both $C_i(s)$ and a subset $\eta(s)$ of it are joint constraints of $\varphi(s)$, and we prefer $\eta(s)$ to $C_i(s)$ for both understandability and efficiency reasons.

The following theorem shows that Algorithm 1 is sound.

Theorem 4.1. *Given an input ϕ, if Algorithm 1 returns φ, then $\varphi[s]$ is an invariant constraint subsuming $\phi[s]$; if it returns False, then $\phi[s]$ is not a constraint.*

Algo. 2 verifies if an input $\forall^*\exists^*$ formula is a constraint by checking if itself is an invariant. In the algorithm, we apply the $\models^n_{\text{FOL}}$ entailment relation as defined after Theorem 3.9, which we implement with a SAT solver. Below is the soundness theorem for Algo. 2.

Theorem 4.2. *Given an input ϕ, if Algorithm 2 returns True, $\phi[s]$ is a constraint; if it returns False, $\phi[s]$ is not.*

Algorithm 2: $ex\_query(\phi, \mathcal{D}, \mathcal{M})$

input : A $\forall^*\exists^*$ formula ϕ, $\mathcal{D}$, $\mathcal{M}$ as before
output: $True$, $False$, or $Non\text{-}deter$

1 **if** $\exists M \in \mathcal{M}. M \not\models \phi[S_0]$ *or* $\mathcal{D}_{S_0} \models^n_{\text{FOL}} \neg\phi[S_0]$ **then**
2 **return** $False$
3 **if** $\mathcal{D}_{S_0} \not\models^n_{\text{FOL}} \phi[S_0]$ **then return** $Non\text{-}deter$
4 **foreach** *action function* $A(\vec{x})$ **do**
5 $\xi(s) \leftarrow \hat{\mathcal{R}}_{\mathcal{D}}[Poss(A(\vec{x}), s) \supset \varphi(do(A(\vec{x}), s))]$
6 **if** $\not\models^n_{\text{FOL}} \forall \vec{x}, s. \mathcal{D}_{\mathcal{SC}} \wedge \varphi(s) \supset \xi(s)$ **then**
7 **return** $Non\text{-}deter$
8 **return** $True$

4.2 Enumerating State Constraints

Algo. 3 enumerates all $\forall^*$ formulas containing one or two literals, and collects those which are verified to be constraints, together with their joint constraints. Here the input Γ is the set of candidate formulas, and all other inputs have the same meanings as before. In Line 2, formulas in Γ falsified by a model of $\mathcal{M}$ or not entailed by $\mathcal{D}_{S_0}$ are eliminated. From Line 5 to Line 10 we traverse Γ and move those verified constraints to both Θ and $\mathcal{D}_{\mathcal{SC}}$. If a new constraint is found, then MORE_SC will be assigned True in Line 8, and we repeatedly traverse Γ until no more constraints are found.

Theorem 4.3. *Algorithm 3 returns a set of state constraints.*

5 Experimental Results

We implemented the procedures outlined above in gcc (Version 4.7.2) and SWI-Prolog (Version 5.10.4). All experiments were run on a machine with Intel(R) Core(TM)2 Duo with 2.10 GHz CPU and 2.00GB RAM under Linux. Note that our experiments were not designed for comparison with other approaches, but to show that our approach indeed produces effective results in a reasonable amount of time.

We tested our method with the blocks world, logistics and satellite domains. In all our experiments, we assume that $\mathcal{D}_{S_0}$ is represented by a finite set of finite initial models, so we reduce $\mathcal{D}_{S_0} \models_{\text{FOL}} \varphi(S_0)$ to model checking. Below are the models used in blocks world (each model has exactly 3 blocks):

Algorithm 3: $enum\_SC(\Gamma, \mathcal{D}, \mathcal{M}, \beta)$

input : A set Γ of $\forall^*$ formulas, and $\mathcal{D}$, $\mathcal{M}$, β as before
output: A set Θ of state constraints

$\Theta \leftarrow \emptyset$
remove from Γ γ such that $\exists M \in \mathcal{M}.M \not\models \gamma[S_0]$ or $\mathcal{D}_{S_0} \not\models_{\text{FOL}} \gamma[S_0]$
repeat
  MORE_SC $\leftarrow False$
  foreach $\gamma \in \Gamma$ **do**
    $sc \leftarrow veri\_SC(\gamma, \mathcal{D}, \mathcal{M}, \beta)$
    if $sc \neq False$ *and* $sc \neq Non\text{-}deter$ **then**
      MORE_SC $\leftarrow True$
      add sc into both Θ and $\mathcal{D}_{\mathcal{SC}}$
      remove γ from Γ
until *not MORE_SC*;
return Θ

1. $\{clear(b_1), clear(b_2), clear(b_3), ontable(b_1), ontable(b_2), ontable(b_3), handempty\}$;
2. $\{clear(b_1), on(b_1, b_2), clear(b_3), ontable(b_2), ontable(b_3), handempty\}$;
3. $\{clear(b_1), holding(b_1), clear(b_2), ontable(b_2), clear(b_3), ontable(b_3)\}$.

For each domain, we first use Algo. 3 to enumerate constraints, and then with these constraints as the background information, we use Algo. 1 and Algo. 2 to process $\forall^*$ and $\forall^*\exists^*$ formulas, respectively. Experiments show that all invariants discovered by Lin [2004] can be verified by our method.

5.1 $\forall^*$ State Constraints

Blocks World

We used Algo. 3 and found the following constraints:
$\{\neg clear(x) \vee \neg on(y, x), \neg holding(x) \vee \neg on(y, x), \neg holding(x) \vee \neg on(x, y), \neg on(x, y) \vee \neg on(y, x), \neg on(x, y) \vee \neg ontable(x), \neg holding(x) \vee \neg handempty, \neg holding(x) \vee \neg ontable(x), \neg holding(x) \vee clear(x)\}$.
Some longer constraints are discovered during the process:
$\{on(x, z) \wedge on(y, z) \supset x = y, on(x, y) \wedge on(x, z) \supset y = z, holding(x) \wedge holding(y) \supset x = y\}$.
We also found some other constraints not discovered by Lin:
$\{\neg on(x, x), \neg on(x, y) \vee \neg on(y, x)\}$.

Logistics Domain

Lin regarded type information as domain specific knowledge, but we also found them. Some examples are:
$\neg city(x) \vee \neg location(x), \neg airplane(x) \vee \neg truck(x)$.
Besides type information, we also found the constraints:
$\{\neg in(x, y) \vee \neg city(x), \neg in(x, y) \vee \neg truck(x), \neg at(x, y) \vee \neg airplane(x), \neg at(x, y) \vee \neg at(y, x)\}$.
One constraint containing 3 literals is discovered:
$at(x, y) \wedge at(x, z) \supset y = z$.

Satellite Domain

This was first introduced by AIPS'00. It involves planning and scheduling of observation tasks performed by instruments on some satellites. Any satellite can carry a few instruments, denoted by the predicate on_board. If any instrument is turned on ($power\_on$), it will make its satellite's power no longer available ($power\_avail$) for other use. Different instruments support ($supports$) different modes. Before an instrument performs its observation task ($take\_image$), it must be able to calibrate on some object ($calibration\_target$), causing the state of ready ($calibrated$).

Below the first two constraints were discovered while the third one was verified. They states that (1) if an instrument on a satellite is powered on, then the electricity of that satellite will not be available for other use; (2) one instrument can be located on only one satellite; and (3) at most one instrument on a satellite can be powered on.
$\{on\_board(x_1, x_2) \wedge power\_on(x_1) \supset \neg power\_avail(x_2)$,
$on\_board(x_1, x_2) \wedge on\_board(x_1, x_3) \supset x_2 = x_3$,
$on\_board(x_1, x_3) \wedge power\_on(x_1) \wedge on\_board(x_2, x_3) \wedge power\_on(x_2) \supset x_1 = x_2\}$.

5.2 Discovering Joint Constraints

Algo. 1 iteratively strengthens the input formula ϕ, and when it verifies ϕ to be a constraint, it also discovers ϕ's joint constraints. In contrast Lin enumerates candidate formulas of the given syntactic forms and check if they are indeed invariants.

Example 5.1. *1. We were to test whether $\phi_1 = holding(x) \supset \neg ontable(x)$ is a state constraint.*

2. ϕ_1 is not proved to be a state invariant, so the system strengthened it with $\phi_2 = \neg on(x_1, x_2) \vee \neg ontable(x_1)$.

3. Again, $\phi_1 \wedge \phi_2$ is not a state invariant so the system strengthened it with $\phi_3 = \neg on(x_1, x_2) \vee \neg holding(x_1)$.

4. After getting $\phi_4 = \neg on(x_1, x_2) \vee \neg on(x_1, x_3) \vee x_2 = x_3$, $\phi_1 \wedge \ldots \wedge \phi_4$ was proved to be a state invariant, and thus, $\phi_1, \ldots, \phi_4$ were all proved to be state constraints.

You see, all discovered constraints are succinct and intuitive.

5.3 $\forall^*\exists^*$ State Constraints

Lin [2004] discovered the following constraints in the blocks world and logistics domains:
$\{holding(x) \vee clear(x) \vee (\exists y) on(y, x)$,
$ontable(x) \vee holding(x) \vee (\exists y) on(x, y)$,
$handempty \vee (\exists x) holding(x), \}$ and
$\{\forall(x, package).\exists(y, vehicle) in(x, y) \vee \exists(y, place) at(x, y)$,
$\forall(x, palce).\exists(y, city) inCity(x, y)$,
$\forall(x, vehicle).\exists(y, place) at(x, y)\}$.
Using $\models^1_{\text{FOL}}$, our approach verified them and the following:
$\{\forall x \exists y.on\_board(x, y), \forall x \exists y.supports(x, y)$,
$\forall x \exists y.calibrated(x) \supset calibration\_target(x, y)$,
$\forall x \exists y.power\_avail(x) \vee on\_board(y, x) \wedge power\_on(y)\}$.

5.4 Performance

Table 1 shows the time costs in verifying constraints in the three domains (B.W. abbreviates blocks world). The column Enum Binary shows the time of enumerating all unary and binary constraints using Algo. 3, in terms of minutes. The columns $\forall^*$ formulas and $\forall^*\exists^*$ formulas both contain two sub-columns: Min and Max, where Min is the minimal time cost of verifying a constraint among those we experimented

with, and Max shows the maximal. When enumerating constraints, the values of β are as shown in Table 2. However, when verifying a single constraint, we use a sufficiently large bound. In both Table 1 and Table 2, when we verify a $\forall^*$ formula, we let $\mathcal{D}_{\mathcal{SC}} = \emptyset$, *i.e.*, we do not consider background information. When we verify a $\forall^*\exists^*$-formula, we let $\mathcal{D}_{\mathcal{SC}}$ be the set of constraints returned by Algo. 3, *i.e.*, we verify them after enumerating constraints using Algo. 3.

Domain	Enum Binary	$\forall^*$ Formulas		$\forall^*\exists^*$ Formulas	
		Min	Max	Min	Max
B.W.	15min	1.8s	1402.8s	0.5s	0.5s
Logistics	17min	19.2s	68.8s	6.3s	7.0s
Satellite	10min	2.5s	14.8s	5.7s	7.8s

Table 1. Time Costs in Verifying State Constraints

Table 1 shows, that the difference between Max and Min for verifying a $\forall^*$ constraint using Algo. 1, is much greater than that for verifying a $\forall^*\exists^*$ constraint using Algo. 2. This is because Algo. 1 iteratively strengthens a formula $\phi(s)$ until an invariant is obtained, while Algo. 2 returns Non-deter once $\phi(s)$ is not proved to be an invariant. In verifying $\forall^*$ constraints, B.W. has the maximal time difference while satellite domain has the minimal, because B.W. is the most structured domain, and some constraints entangle dramatically. One would notice that the maximal time cost of verifying one constraint in B.W. even surpasses the total time of enumerateing constraints. That is because Algo. 3 exploits previously verified constraints to speed up further verifications, so it can avoid many costly operations of discovering joint constraints.

Table 2 shows how much time Algo. 1 has spent before returning False or Non-deter. Here Min and Max have the same meanings as above. In logistics and satellite domains, no formulas are non-determined, that is, all formulas are verified or rejected, so we fill '-' in the respective grids. This shows that Algo. 1 happens to be *complete* in these domains.

Domain	β	$\lvert\mathcal{M}\rvert$	False		Non-deter	
			Min	Max	Min	Max
B.W.	9	3	0.5s	1.8s	6s	123s
Logistics	5	4	0.1s	0.9s	-	-
Satellite	4	3	0.1s	1.0s	-	-

Table 2. Time Costs in Failed Verifications

In B.W., it takes Algo. 1 quite long ($\geq$6s) before Non-deter is returned, and the worst case needs 123s. When Algo. 1 returns False, it takes very short time in all three domains, because most rejected formulas are falsified by one of the initial models and are rejected immediately.

Below we list some non-determined formulas in blocks world when we use Algo. 3 to enumerate constraints:
$\{clear(x) \vee ontable(x), clear(x) \vee handempty,$
$clear(x) \vee \neg on(x,y), clear(x) \vee \neg holding(x),$
$ontable(x) \vee \neg on(y,x), \neg on(x,y) \vee \neg holding(y),$
$handempty \vee \neg on(x,y), \neg on(x,y) \vee \neg on(y,x)\}$.
In fact, none of them is a constraint except the last one. We then ran Algo. 1 on $\neg on(x,y) \vee \neg on(y,x)$ with $\beta = 14$, and it was verified to be a constraint. In fact it costs us the most time – 1402.8s as Table 1 shows. This indicates that its verification depends on the discovery of quite a few joint constraints.

6 Related Work

Various other communities, *e.g.*, model-checking, program verification and planning communities, are also interested in state constraints. We discuss two typical works below.

Bradley and Manna [2007; 2011] from the model-checking community developed a sound and complete algorithm to check safety properties (state constraints) on large circuits, *i.e.*, either the specification formula is proved, or a counterexample trace is returned. In contrast, we are verifying constraints on a first-order dynamic system. Generally this is a second-order reasoning problem where no sound and complete algorithms exist, so our strategy is to develop an incomplete method which reduces this problem to a series of first-order reasoning problems and solves them with a SAT solver. In addition, it is trivial to reduce our framework, methods and notions to theirs, when we restrict our domains to be finite.

Conincidently both their and our works strengthen a state constraint to be an invariant constraint (inductive invariant). They use the negation of one of the states that lead to violation of the specification formula, while we use an unentailed derivant of the regressed formula. Both works exclude some unreachable predecessor states so that successor states do not violate the specification. Both works prefer shorter clauses. While we use small models to help eliminate useless clauses, they use a more sophisticated technique based on lattice fixpoint theory to compute the minimal useful clause.

Rintanen [2000] from the planning community applied a procedure to a set Σ_i of candidate formulas to generate a refined set Σ_{i+1} of formulas iteratively, until $\Sigma_i = \Sigma_{i+1}$, so as to obtain a set of invariants (invariant constraints). His method is very efficient when every candidate formula is a unit or binary clause, and it is guaranteed that his algorithm terminates in polynomial time.

Both he and Lin realized that many constraints fail to be invariants, so they both attempted to combine formulas of certain predefined syntactic forms, and verified whether the conjunctions as a whole are invariants. In contrast, our approach takes a single formula as the input, and smartly strengthen it with unentailed derivants towards an invariant if possible.

7 Conclusions

In this paper, we have developed a sound but incomplete method for automatic verification and discovery of $\forall^*\exists^*$ state constraints. Our method is to prove a state constraint by strengthening it to be an invariant with joint constraints. We experimented with the blocks world, logistics and satellite domains, and the results showed that, our method is efficient and able to verify all constraints earlier discovered in [Lin, 2004].

Now we summarize our contributions. Firstly we have developed a practically effective method with solid theoretical foundations for verifying if a given $\forall^*\exists^*$ formula is a state constraint. Secondly, if a state constraint is not proved to be a state invariant, our method will systematically strengthen it with derivants and hence discover joint constraints. Thirdly, so far as we know, we are the first to propose a method with soundness guarantee for verifying $\forall^*\exists^*$ constraints. Lastly, experiments show that practically meaningful and useful constraints can be efficiently verified with our approach, and suc-

cinct and intuitive joint constraints can be discovered. Here, the second and the third points distinguish our work from [Lin, 2004]. Another difference is: our method is based on Herbrand Theorem and Skolemization and implemented with a SAT solver, while Lin's work is based on exhaustive search in selected small domains.

However, our work has two limitations. Firstly our method can only verify partial invariant constraints. Secondly when verifying $\forall^*\exists^*$ constraints, our method is of poor scalability.

There are at least two topics for future research. Firstly, our current approach to strengthening formulas can only be applied to $\forall^*$ constraints, so we will extend our method to handle $\forall^*\exists^*$ constraints. Secondly, we would like to know under what conditions, a state constraint can be strengthened to be a state invariant, so we know the conditions under which our algorithms are complete.

Acknowledgments

We thank the anonymous reviewers for helpful comments. This work was supported by the Natural Science Foundation of China under Grant No. 61073053.

References

[Bradley and Manna, 2007] A. R. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In *FMCAD*, pages 173–180, 2007.

[Bradley, 2011] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[Gerevini and Schubert, 1998] A. Gerevini and L. K. Schubert. Inferring state constraints for domain-independent planning. In *AAAI/IAAI*, pages 905–912, 1998.

[Kautz and Selman, 1992] H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[Kelly and Pearce, 2010] R. F. Kelly and A. R. Pearce. Property persistence in the situation calculus. *Artif. Intell.*, 174(12-13):865–888, 2010.

[Lin, 2004] F. Lin. Discovering state invariants. In *KR*, pages 536–544, 2004.

[Refanidis and Vlahavas, 2000] I. Refanidis and I. P. Vlahavas. Exploiting state constraints in heuristic state-space planning. In *AIPS*, pages 363–370, 2000.

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

[Rintanen, 2000] J. Rintanen. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, pages 806–811, 2000.