Automatic Verification of Partial Correctness of Golog Programs

Naiqi Li **Yongmei Liu**
Department of Computer Science,
Sun Yat-sen University, Guangzhou 510006, China
linaiqi@mail2.sysu.edu.cn, ymliu@mail.sysu.edu.cn

Abstract

When Golog programs are used to control agents' behaviour in a high-level manner, their partial correctness naturally becomes an important concern. In this paper we propose a sound but incomplete method for automatic verification of partial correctness of Golog programs. We introduce the notion of extended regression, which reduces partial correctness of Golog programs to first-order entailment problems. During the process loop invariants are automatically discovered by heuristic methods. We propose progression of small models wrt Golog programs, which are used to filter out too strong heuristic candidates. In this way we combine the methods of static and dynamic analysis from the software engineering community. Furthermore, our method can also be adapted to verify state constraints. Experiments show that our method can not only handle sequential and nested loops uniformly in a reasonable among of time, but also be used to discover succinct and comprehensible loop invariants and state constraints.

1 Introduction

Building autonomous agents that follow human's high-level guiding instructions, and accordingly figure out a concrete plan to carry out, is always a challenge of artificial intelligence. One possible solution to achieve this goal is to use high-level programming language, such as Golog [Levesque *et al.*, 1997], which is a logic programming language based on the situation calculus [Reiter, 2001]. Provided with domain descriptions and possibly nondeterministic instructions, a Golog interpreter is able to generate a sequence of concrete actions, so programmers are released from the burden of specifying every single detail.

When Golog programs are used to control agents' behaviour in a high-level manner, their partial correctness naturally becomes an important concern. Roughly speaking, partial correctness states that if some properties hold at the beginning, then some other properties must hold after the program is executed. Liu [2002] proposed a Hoare-style proof system for partial correctness of Golog programs. However, she didn't address the automatic verification issue.

Partial correctness is also a central topic in the software engineering community. There have been two important approaches for verification: static analysis and dynamic analysis. Static analysis tries to verify programs by analyzing the source code without actually executing them. Usually verification by static analysis is guaranteed to be sound, but at the cost of inefficiency [Flanagan and Qadeer, 2002; Pasareanu and Visser, 2004]. On the contrary, dynamic analysis tries to discover faults by running the program with adequate test cases. Generally this approach leads to faster algorithms and implementations, but there is no guarantee that the program is correct even if all test cases behave as expected [Nguyen *et al.*, 2012].

When we human try to prove or disprove a mathematical assertion, we are using the static and dynamic analysis unconsciously, and integrate them in a sophistical way. We first attempt to prove it by following some routines and heuristics that were learned before; if a proof is not found, we may look for some examples to illustrate which step during the attempted proof goes wrong, and then fix it and continue the proof. If lucky we may also find a counterexample and thus disprove the assertion. One may go back and forth between reasoning and testing, until a proof or a disproof is achieved. Based on this observation, in this paper we propose a method that combines the static and dynamic analysis, which tries to imitate the reasoning process of human.

We introduce the notion of extended regression, which belongs to the static analysis paradigm. By extended regression one can regress a formula wrt a Golog program. It has a similar property as regression in the situation calculus: a formula holds after executing a program, if it's extended regression holds before the execution. Thus we reduce partial correctness of Golog programs to first-order entailment problems.

When extended regression is applied to a loop statement, we need to discover an appropriate loop invariant. Recently, we proposed a sound but incomplete method for automatic verification and discovery of state constraints in the situation calculus [Li *et al.*, 2013]. The basic idea presented in this paper is similar to the previous work: we start with a simple formula, and then strengthen it iteratively until it becomes a loop invariant. During this process, formulas used for strengthening are generated by heuristic methods. It turns out that the method for inferring loop invariants can also be adapted to verify state constraints.

The heuristic functions generate a set of heuristic candidates. When deciding which of the candidates should be selected for strengthening, we use small models to filter out the too strong ones, which is the dynamic analysis component in our method. Specifically, a set of initial small models are provided by the user, which can be viewed as several test inputs. We progress these initial small models wrt the program, associating each loop with a set of small models. These associated small models can be regarded as a set of possible program states. Later when heuristic functions generate a set of heuristic candidates, we select the one as strong as possible provided it is satisfied by the associated small models.

2 Preliminaries

2.1 The Situation Calculus and Golog

The situation calculus [Reiter, 2001] is a many-sorted first-order language for representing 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 $s \sqsubseteq s'$ meaning that situation s is a subhistory of situation s'; a binary predicate $Poss(a, s)$ meaning that action a is possible in situation s; a countable set of action functions, e.g., $move(x, y)$; and a countable set of relational fluents, i.e., predicates taking a situation term as their last argument, e.g., $ontable(x, s)$.

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. We call a uniform formula ϕ with all situation arguments eliminated a *situation-suppressed* formula, and use $\phi[s]$ to denote the uniform formula with all situation arguments restored with term s. A situation s is executable if it is possible to perform the actions in s one by one: $Exec(s) \doteq \forall a, s'.do(a, s') \sqsubseteq s \supset Poss(a, s')$.

In the situation calculus, a particular domain of application is specified by a basic action theory (BAT) of the form: $\mathcal{D} = \Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$, where

1. Σ is the set of the foundational axioms for situations.
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.

The formal semantics of Golog is specified by an abbreviation $Do(\delta, s, s')$, which is inductively defined as follows:

1. Primitive actions: For any action term α, $Do(\alpha, s, s') \doteq Poss(\alpha, s) \wedge s' = do(\alpha, s)$.
2. Test actions: For any situation-suppressed formula ϕ, $Do(\phi?, s, s') \doteq \phi[s] \wedge s = s'$.
3. Sequence: $Do(\delta_1; \delta_2, s, s') \doteq \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$.
4. Nondeterministic choice of two actions: $Do(\delta_1|\delta_2, s, s') \doteq Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.
5. Nondeterministic choice of action arguments: $Do((\pi x)\delta(x), s, s') \doteq (\exists x)Do(\delta(x), s, s')$.
6. Nondeterministic iteration: $Do(\delta^*, s, s') \doteq (\forall P).\{(\forall s_1)P(s_1, s_1) \wedge (\forall s_1, s_2, s_3)[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\} \supset P(s, s')$.

Conditionals and loops are defined as abbreviations:

if ϕ **then** δ_1 **else** δ_2 **fi** $\stackrel{def}{=} [\phi?; \delta_1]|[\neg\phi?; \delta_2]$,

while ϕ **do** δ **od** $\stackrel{def}{=} [\phi?; \delta]^*; \neg\phi?$.

In this paper we consider programs with **while** statements, but without nondeterministic iterations appearing alone.

2.2 State Invariants and State Constraints

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

State constraints are formulas which hold in all executable situations. We use $\mathcal{D}_{\mathcal{SC}}$ to denote a set of verified state constraints, and abuse $\mathcal{D}_{\mathcal{SC}}$ as the conjunction of its elements.

State invariants are formulas that if true in a situation, will be true in all successor situations. Verifying state invariants is a first-order reasoning task.

Definition 2. Let $\mathcal{D}_{\mathcal{SC}}$ be a set of verified state constraints. A formula $\phi(s)$ is a state invariant for $\mathcal{D}$ wrt $\mathcal{D}_{\mathcal{SC}}$ if

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

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

Proposition 1. If $\mathcal{D}_{S_0} \models_{\text{FOL}} \varphi(S_0)$, and $\varphi(s)$ is an invariant wrt $\mathcal{D}_{\mathcal{SC}}$, then $\varphi(s)$ is a constraint.

2.3 Regression and Model Progression

There are two important computational mechanisms for reasoning about actions and their effects in the situation calculus, namely regression and progression.

Here we define a one-step regression operator and state a simple form of the regression theorem [Reiter, 2001].

Definition 3. We use $\mathcal{R}_{\mathcal{D}}[\phi]$ 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 1. $\mathcal{D} \models \phi \equiv \mathcal{R}_{\mathcal{D}}[\phi]$.

Intuitively, the notion of progression is used to update the current world state after an action is executed. The progression of a theory is generally not first-order definable [Lin and

Reiter, 1997]. In this paper, progression refers to the progression of models. We use $prog^S[M, \alpha]$ to denote the new model generated by updating model M according to action α.

Under the close world assumption, a model can be represented by a set of ground atoms, which we call a small model.

3 Theoretical Foundations

3.1 Partial Correctness and Extended Regression

We begin with a formal definition of Hoare triple and partial correctness in the context of Golog programs.

Definition 4. A Hoare triple is of the form $\{P\}\delta\{Q\}$, where P and Q are situation-suppressed formulas, and δ is a Golog program. A Hoare triple $\{P\}\delta\{Q\}$ is said to be partially correct wrt $\mathcal{D}$ if $\mathcal{D} \models \forall s, s'.P[s] \wedge Do(\delta, s, s') \supset Q[s']$.

Next we present the notion of extended regression, which plays a significant role throughout this paper.

Definition 5. Given $\mathcal{D}$ and $\mathcal{D}_{\mathcal{SC}}$, the extended regression of formula $\phi(s)$ wrt program δ, denoted as $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta]$, is defined as follows:

- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \alpha] = \mathcal{R}_{\mathcal{D}}(Poss(\alpha, s) \supset \phi(do(\alpha, s)))$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \psi?] = \psi[s] \supset \phi(s)$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_1; \delta_2] = \hat{\mathcal{R}}_{\mathcal{D}}[\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_2], \delta_1]$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_1|\delta_2] = \hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_1] \wedge \hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_2]$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), (\pi x)\delta(x)] = (\forall x)\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta(x)]$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \textbf{while } \varphi \textbf{ do } \delta \textbf{ od}]$ is a formula (denoted as $\eta(s)$) satisfying the following two conditions:
  1. $\models_{\text{FOL}} \forall s.\eta(s) \wedge \varphi[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset \hat{\mathcal{R}}_{\mathcal{D}}[\eta(s), \delta]$.
  2. $\models_{\text{FOL}} \forall s.\eta(s) \wedge \mathcal{D}_{\mathcal{SC}} \supset \phi(s) \vee \varphi[s]$.

Intuitively, in the definition of loop statement the first condition ensures the regression is a loop invariant, and the second condition guarantees this invariant is strong enough to entail the formula being regressed when the loop ends.

In the situation calculus, a formula holds after a sequence of actions are performed if and only if its regression can be entailed by the initial knowledge base. By structural induction we obtain a similar property for the extended regression operator, that is, a formula holds after the program is executed, if its extended regression holds before the execution.

Theorem 2. Let $\phi(s)$ be a formula, δ be a Golog program and $\mathcal{D}$ as before, we have
$\mathcal{D} \models \forall s.\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta] \supset \forall s'.[Do(\delta, s, s') \supset \phi(s')]$.

This furnishes us with a method for proving partial correctness by extended regression:

Corollary 1. A Hoare triple $\{P\}\delta\{Q\}$ is partially correct wrt $\mathcal{D}$ and $\mathcal{D}_{\mathcal{SC}}$, if $\models_{\text{FOL}} \forall s.P[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset \hat{\mathcal{R}}_{\mathcal{D}}[Q[s], \delta]$.

3.2 Heuristics for Finding Loop Invariants

The definition of extended regression is not specific about how to obtain a proper formula satisfying the two conditions of a loop statement. For the purpose of a concrete algorithm, we next discuss the weakest extended regression and how it inspires us in the heuristic approach.

Definition 6. The weakest extended regression of formula $\phi(s)$ wrt program δ, denoted as $\hat{\mathcal{W}}_{\mathcal{D}}[\phi(s), \delta]$, is the same as extended regression except for the case of loop statement.

- $F(s)^0 = \phi(s) \vee \varphi[s]$, $F(s)^{i+1} = \neg\varphi[s] \vee \hat{\mathcal{W}}_{\mathcal{D}}[F(s)^i, \delta]$. $\hat{\mathcal{W}}_{\mathcal{D}}[\phi(s), \textbf{while } \varphi \textbf{ do } \delta \textbf{ od}] = \bigwedge_{i=0}^{N} F(s)^i$, where N is a natural number s.t. $F(s)^N$ is equivalent to $F(s)^{N+1}$.

As the next proposition shows, $\hat{\mathcal{W}}_{\mathcal{D}}[\phi(s), \delta]$ is sufficient and necessary for $\phi(s)$ to be true after the execution of δ. The proof is another application of structural induction on δ.

Proposition 2. Let $\phi(s)$ be a formula, δ be a Golog program and $\mathcal{D}$ as before, we have
$\mathcal{D} \models \forall s.\hat{\mathcal{W}}_{\mathcal{D}}[\phi(s), \delta] \equiv \forall s'.[Do(\delta, s, s') \supset \phi(s')]$.

Weakest extended regression can be viewed as a process of strengthening. The process starts with $\phi(s) \vee \varphi[s]$, and in the i-th step we strengthen it with $F(s)^i$. If the process reaches a fixpoint, we obtain the weakest loop invariant. However, the fixpoint does not always exist, and even if it exists the strengthening process may be too slow to be practical. We are inspired to use a stronger formula to strengthen during each step, by which the convergence to fixpoint could be accelerated.

In what follows we identify three kinds of simple but useful heuristics. Each heuristics generates a set of heuristic candidates Δ, and the corresponding heuristic function returns one of them. It is guaranteed that every heuristic candidate in Δ is stronger than or equivalent to its original formula. Every formula can be written in the form of $\exists^*\xi_0(s) \vee \forall^*\xi_1(s)$, where $\xi_0(s)$ and $\xi_1(s)$ may have other quantifiers.

Sub-Disjunction Heuristics

For formula $\exists^*\xi_0(s) \vee \forall^*\xi_1(s)$, the heuristic candidate set is:
$\Delta_{SD}(\exists^*\xi_0(s) \vee \forall^*\xi_1(s)) = \{\exists^*\xi_0'(s) \vee \forall^*\xi_1'(s) |$
$\xi_i'(s)$ is a sub-disjunction of $\xi_i(s)$, $i \in \{0, 1\}\}$.

Unification Heuristics

Every formula in $\Delta_{Uni}((\exists\vec{x})\xi_0(\vec{x}, s) \vee (\forall\vec{y})\xi_1(\vec{y}, s))$ is obtained by the following steps:

1. Rename $\vec{x}, \vec{y}$ to $\vec{u}, \vec{v}$ such that $\vec{u}$ and $\vec{v}$ share no variables.
2. Choose n distinct variables $\vec{c} \in \vec{u}$, and n (unnecessarily distinct) variables $\vec{d} \in \vec{v}$, let $\vec{u}' = \vec{u} \backslash \vec{c}$.
3. Obtain $\xi_0'(\vec{u}', \vec{d}, s)$ by replacing $\vec{c}$ with $\vec{d}$ in $\xi_0(\vec{u}, s)$.
4. Let $\forall\vec{v}\exists\vec{u}'.\xi_0'(\vec{u}', \vec{d}, s) \vee \xi_1(\vec{v}, s)$ be a formula in Δ_{Uni}.

Constant Heuristics

Let $\mathcal{C}$ be the set of all constants used in the Hoare triple. $\Delta_{Con}((\exists\vec{x})\xi_0(\vec{x}, s) \vee \forall^*\xi_1(s))$ is a set of formulas generated by replacing some variables in $\vec{x}$ with $\vec{c} \in \mathcal{C}$, or by changing some constants in $\xi_1(s)$ to universally quantified variables.

There is a dilemma when deciding which formula should our heuristic function return from the candidate set Δ. If the formula used for strengthening is too strong, we may end up with a too strong loop invariant. On the other hand, if the formula is too weak, the strengthening process may be too slow, just as the weakest extended regression. We solve this dilemma by using small models to filter out the too strong candidates. Among all the heuristic candidates we select the

one as strong as possible provided it is satisfied by the associated small models. This is the topic of the next subsection.

3.3 Progression of Small Models

We begin with a definition of the progression operator.

Recall that a small model M is represented by a set of ground atoms. Given a finite constant set D, we use $M[s]$ to denote the unique model such that only those atoms in M (with situation s restored) are satisfied.

Definition 7. We assume the ground terms are all constants from a finite set D. Given a small model M and a program δ, the progression of M wrt δ, denoted as $prog[M, \delta]$, results in a set of small models:

- $prog[M, \alpha] = 1.\ \emptyset$ if $M[s] \not\models Poss(\alpha, s)$.
  $2.\ \{prog^S[M, \alpha]\}$ if $M[s] \models Poss(\alpha, s)$.
- $prog[M, \psi?] = 1.\ \emptyset$ if $M[s] \not\models \psi[s]$.
  $2.\ \{M\}$ if $M[s] \models \psi[s]$.
- $prog[M, \delta_1; \delta_2] = prog[prog[M, \delta_1], \delta_2]$.
- $prog[M, \delta_1 | \delta_2] = prog[M, \delta_1] \cup prog[M, \delta_2]$.
- $prog[M, (\pi x)\delta(x)] = \bigcup\{prog[M, \delta(c)] | c \in D\}$.
- $prog[M, \delta^*] = \bigcup_{n \geq 0} prog[M, \delta^n]$, where δ^n is an abbreviation of jointing n copies of δ sequentially.

When $\mathcal{M}$ is a set of small models, we define
$prog[\mathcal{M}, \delta] = \bigcup\{\mathcal{M}' | M \in \mathcal{M}, prog[M, \delta] = \mathcal{M}'\}$.

When progressing wrt a loop the computation may never stop. In practice we preset a constant K, and let $prog[M, \delta^*] = \bigcup_{n=0}^{K} prog[M, \delta^n]$.

The first usage of the small models is to prove a Hoare triple is not partially correct, while the extended regression can only be used to prove the positive result.

Theorem 3. If a Hoare triple $\{P\}\delta\{Q\}$ is partially correct, and M is a small model that $M[s] \models P[s] \wedge \mathcal{D}_{\mathcal{SC}}$, then for all $M' \in prog[M, \delta]$ we have $M'[s] \models Q[s]$.

Another important usage of our small models is to inform us that the regression "goes wrong" during the process. In particular, if some loop invariant is not satisfied by its associated small models, the final regression result will be too strong to be successfully verified. One step further, when generating a loop invariant, any heuristic candidate not satisfied by its associated small models should be discarded.

Proposition 3. If $\models_{\text{FOL}} \forall s. P[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset \hat{\mathcal{R}}_{\mathcal{D}}[Q[s], \delta_1; \delta_2]$ and $M[s] \models P[s] \wedge \mathcal{D}_{\mathcal{SC}}$, then for all $M' \in prog[M, \delta_1]$ we have $M'[s] \models \hat{\mathcal{R}}_{\mathcal{D}}[Q[s], \delta_2]$.

4 Algorithms

In this section we introduce three algorithms in detail: the top-level main algorithm, the loop invariant inferring algorithm and the state constraint discovering algorithm. The implementations for extended regression $\hat{\mathcal{R}}_{\mathcal{D}}$ and small model progression $prog$ are basically the same as their definitions, and we omit the details. We assume that the underlying $\mathcal{D}$, $\mathcal{D}_{\mathcal{SC}}$ and a finite set of constants D used for grounding are given, without providing them explicitly.

4.1 Main Algorithm

Algorithm 1 is the top-level algorithm of our system, which combines static and dynamic analysis. It tries to imitate the deduction process of a human reasoner: going back and forth between the processes of attempting a proof, and looking for a test case to illustrate why an attempted proof goes wrong.

Algorithm 1: $veri(P, \delta, Q, \mathcal{M})$

Input: Hoare-triple $\{P\}\delta\{Q\}$; $\mathcal{M}$ - initial small models that satisfy P and $\mathcal{D}_{\mathcal{SC}}$.

1 Let $asso$ be a function mapping each **while** construct in δ to a pair $\langle true, \emptyset \rangle$
2 $\langle \mathcal{M}', asso \rangle \leftarrow prog(\mathcal{M}, \delta, asso)$
3 **if** $\exists M' \in \mathcal{M}'$ s.t. $M'[s] \not\models Q[s]$ **then return** no
4 $counter \leftarrow 0$
5 **while** $counter < K_1$ **do**
6 $counter \leftarrow counter + 1$
7 $\langle reg(s), asso \rangle \leftarrow \hat{\mathcal{R}}_{\mathcal{D}}(Q[s], \delta, asso)$
8 **if** $\models_{\text{FOL}} \forall s. P[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset reg(s)$ **then return** yes
9 **while** $counter < K_1$ **do**
10 $counter \leftarrow counter + 1$
11 $M_0 \leftarrow genModel(\mathcal{D}_{\mathcal{SC}}, P)$
12 $\langle \mathcal{M}', asso \rangle \leftarrow prog(M_0, \delta, asso)$
13 **if** $\exists M' \in \mathcal{M}'$ s.t. $M'[s] \not\models Q[s]$ **then return** no
14 **if** $\exists$ loop δ_l s.t. $asso(\delta_l) = \langle Inv_t(s), \mathcal{M}_t \rangle$ and $\exists M_t \in \mathcal{M}_t$ s.t. $M_t[s] \not\models Inv_t(s)$ **then break**
15 **return** *non-deter*

At the beginning we initialize a function $asso$, mapping each **while** construct to a pair $\langle inv, mod \rangle$, where inv is the current candidate loop invariant and mod is the set of small models progressed to the beginning of the while construct. When we perform progression (Line 2, 12), the associated small models will be updated; similarly when we perform extended regression (Line 7), the associated candidate loop invariant will be updated.

In Line 2, we progress the set of initial small models wrt δ. Procedure $prog$ returns a pair $\langle \mathcal{M}', asso \rangle$, where $\mathcal{M}'$ is the progression result and $asso$ is the updated mapping function. If some model in $\mathcal{M}'$ does not satisfy Q, we find a counterexample and the system returns no. Variable $counter$ in Line 4 is used to guarantee the following loops terminate in finite steps. During each iteration of the outer loop, we first try to prove the positive result by extended regression in Line 7. When regressing a loop statement, the associated small models can be retrieved by $asso$ function and be used to filter out too strong heuristic candidates.

If the entailment in Line 8 can be proved by the theorem prover, the system returns yes. Otherwise we enter a loop starting at Line 9. In Line 11, we generate a small model M_0 that satisfies $P[s] \wedge \mathcal{D}_{\mathcal{SC}}$ and is different from those in $\mathcal{M}$ and any small model generated before. In procedure $genModel$, we ground $\mathcal{D}_{\mathcal{SC}}$ and P to a set of propositional formulas in CNF and call a SAT solver. Next, we progress M_0 wrt δ. If some resulted small model does not satisfy Q, we find a counterexample and the system returns no as

before. Then in Line 14, if there is a **while** construct δ_l such that one of its associated small models does not satisfy its associated loop invariant, we jump out of the loop and try the extended regression again. The inner loop guarantees that regression at Line 7 returns a new result each time.

Theorem 4. Algorithm $veri(P, \delta, Q, \mathcal{M})$ returns yes only if the Hoare triple $\{P\}\delta\{Q\}$ is partially correct, and returns no only if the Hoare triple is not partially correct.

4.2 Loop Invariant Inferring Algorithm

When regressing a loop statement, we call the $infer$ procedure listed as Algorithm 2.

Algorithm 2: $infer(\phi(s), \delta_l, asso)$

Input: $\phi(s)$ - formula being regressed; δ_l - loop statement being regressed; $asso$ as before

1 Let $\delta_l =$ **while** φ **do** δ **od**
2 $\eta(s) \leftarrow \phi(s) \vee \varphi[s]$; $counter \leftarrow 0$
3 **while** $counter < K_2$ **do**
4 $\quad counter \leftarrow counter + 1$
5 $\quad \langle reg(s), asso\rangle \leftarrow \hat{\mathcal{R}}_{\mathcal{D}}(\eta(s), \delta, asso)$
6 $\quad$ **if** $\models_{\text{FOL}} \eta(s) \wedge \varphi[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset reg(s)$ **then**
7 $\quad\quad asso(\delta_l).inv = \eta(s)$
8 $\quad\quad$ **return** $\langle \eta(s), asso\rangle$
9 $\quad$ Let $reg(s) \equiv A_1(s) \wedge ... \wedge A_n(s)$, choose a $A_i(s)$ s.t. $\not\models_{\text{FOL}} \eta(s) \wedge \varphi[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset A_i(s)$
10 $\quad$ Let $A_i(s) \equiv \exists^* \xi_0(s) \vee \forall^* \xi_1(s)$
11 $\quad$ Transform $\xi_1(s)$ into conjunction $C_1(s) \wedge ... \wedge C_t(s)$
12 $\quad$ Choose $C_i(s)$ s.t. $\not\models_{\text{FOL}} \eta(s) \wedge \varphi[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset \exists^* \xi_0(s) \vee \forall^* C_i(s)$
13 $\quad can(s) \leftarrow heur(\exists^* \xi_0(s) \vee \forall^* C_i(s), asso(\delta_l).mod)$
14 $\quad \eta(s) \leftarrow \eta(s) \wedge can(s)$
15 $asso(\delta_l).inv = false$
16 **return** $\langle false, asso\rangle$

The candidate invariant $\eta(s)$ is initialized as $\phi(s) \vee \varphi[s]$. In each step of the following iteration, we regress $\eta(s)$ wrt the loop body. If the entailment in Line 6 can be proved, then $\eta(s)$ is an appropriate invariant for δ_l. We update the $asso$ function accordingly, and return $\langle \eta(s), asso\rangle$ as the result. If the entailment cannot be proven, we transform the regression result to a conjunction, choosing a conjunct $A_i(s)$ which causes the failure of the entailment. If $A_i(s)$ is not entailed, there must be some $C_i(s)$ such that $\exists^* \xi_0(s) \vee \forall^* C_i(s)$ cannot be entailed. In function $heur$ we repeatedly apply the three heuristics in a greedy manner. Among all the possible candidates we select the one as strong as possible provided that it is satisfied by the associated small models $asso(\delta_l).mod$. Finally we strengthen $\eta(s)$ with the returned formula.

Proposition 4. Procedure $infer(\phi(s), \delta_l, asso)$ returns $\langle \eta(s), asso'\rangle$, where $\eta(s)$ is a formula that satisfies the definition of $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_l]$.

4.3 State Constraints Discovering Algorithm

We can strengthen a formula $\phi(s)$ to a state invariant by regressing it wrt a constraint checking program $\delta_{\mathcal{SC}}$.

Definition 8. Suppose that the action functions in the application domain are $a_1(\vec{X_1}), ..., a_n(\vec{X_n})$. The constraint checking program $\delta_{\mathcal{SC}}$ is defined to be
$\delta_{\mathcal{SC}} \doteq$ **while** $true$ **do** $(\pi\vec{X_1})a_1(\vec{X_1}) \,|...|\, (\pi\vec{X_n})a_n(\vec{X_n})$ **od**.

Proposition 5. Loop invariant of $\delta_{\mathcal{SC}}$ is also a state invariant.

When strengthening $\phi(s)$ to a loop invariant by regressing it wrt $\delta_{\mathcal{SC}}$, the only change of Algorithm 2 is that the initial formula to be strengthened is $\phi(s)$ instead of $\phi(s) \vee true$, which is trivial. By Propositions 1 and 5, if $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_{\mathcal{SC}}]$ is entailed by the initial KB then every conjunct of it, including $\phi(s)$, is a state constraint. In our implementation, the initial KB is represented by a set of small models. So checking the entailment could be done efficiently.

The constraint discovering algorithm follows the guess-and-check paradigm as [Li *et al.*, 2013] does, and we adopt the candidate enumerating method from [Rintanen, 2000]. Our candidate formulas are of the form $(x_1 \neq x_1' \wedge ... \wedge x_n \neq x_n') \supset (L_1 \vee ... \vee L_m)$, where L_i are literals while x_i and x_i' are variables appearing as arguments of the literals. In practice we restrict $n \leq 1$ and $m \leq 2$. The candidate set Σ is initialized as $\Sigma = \{P_1(\vec{x_1}), \neg P_1(\vec{x_1}), P_2(\vec{x_2}), ...\}$, where $P_1(\vec{x_1}), P_2(\vec{x_2}), \ldots$ are all the predicates in the domain. If we could verify it is a state constraint, we move it along with the formulas used for strenghtening to $\mathcal{D}_{\mathcal{SC}}$. Otherwise, we replace it with some weaker candidate constraints. There are three weakening operations for a given candidate constraint: adding a disjunct to the consequent, adding a conjunct $x \neq y$ to the antecedent and identifying two variables.

5 Experimental Results

We implemented the algorithms in SWI-Prolog. All experiments were conducted on a personal laptop equipped with 2.60 GHz CPU and 4.00GB RAM under Linux. We used E Prover [Schulz, 2013] as the underlying theorem prover, and MiniSat [Eén and Sörensson, 2003] as the SAT solver.

Our system successfully verified 22 programs in 9 domains. Among them the $logistics$ domain is from the AIPS'00 planning competition, and the other 8 domains are adapted from those used in [Hu, 2012]. For each domain, we discovered state constraints in advance, and used them as background knowledge in the later verifications.

5.1 Program Verification Examples

We use two examples to demonstrate the power of our system.

$Green$ Domain Example

Figure 1 presents an example in the $green$ domain, which resembles the *blocks world* domain. We introduce two new predicates $green$, $collected$ and a new action $collect$. The goal is to collect all green blocks.

The program is composed of two sequential loops. First we unstack all towers onto the table, during which we collect all green blocks held by the robot. The second loop examines all the blocks remaining on the table and collect the green ones. The loop invariants for the two loops are listed as $Inv1$ and $Inv2$. To save space and to improve readability, we omit the initial formula being strengthened and manually simplify the final result. Our algorithm verified this program in 46.0s.

$\{handempty \wedge \forall x.green(x) \supset ontable(x) \vee \exists y.on(x,y)\}$
while $(\exists x,y)on(x,y)$ **do**
$(\pi u,v)(unstack(u,v);$
if $green(u)$ **then** $collect(u)$ **else** $putdown(u)$ **fi**) **od** ;
while $(\exists x)(ontable(x) \wedge green(x))$ **do**
$(\pi u)($ **if** $green(u)$ **then** $pickup(u); collect(u)$ **fi**) **od**
$\{\forall x.green(x) \supset collected(x)\}$

$Inv1$: ... $\wedge \forall x.green(x) \supset$
$ontable(x) \vee collected(x) \vee \exists y.on(x,y)$
$Inv2$: ... $\wedge \forall x.green(x) \supset ontable(x) \vee collected(x)$

Figure 1: Example of the $Green$ Domain

$\{holding(0) \wedge rob\_at(locA) \wedge \forall[o,obj].at(o,locB)\}$
while $\exists[x,obj].at(x,locB)$ **do**
$move(locA,locB);$
while $\exists[y,obj].at(y,locB) \wedge \neg holding(2)$ **do**
$(\pi[y',obj])rob\_load(y',locB)$ **od**;
$move(locB,locA);$
while $\exists[y,obj].gripped(y)$ **do**
$(\pi[y',obj])rob\_unload(y',locA)$ **od od**
$\{\forall[o,obj].at(o,locA)\}$

$Inv1$: ... $\wedge \forall[o,obj].at(o,locA) \vee at(o,locB)$
$Inv2,3$: ... $\wedge \forall[o,obj].at(o,locA) \vee at(o,locB) \vee gripped(o)$

Figure 2: Example of the $Gripper+$ Domain

$Gripper+$ Domain Example

Figure 2 lists the verification of a program in the $gripper+$ domain. There is a robot with two hands, so it is able to hold two objects at the same time. We use $holding(x)$ to denote that the robot is currently holding x objects, where x could be 0, 1 or 2. With the robot starting at $locA$, it tries to move all objects from $locB$ to $locA$.

The program works as follows: As long as there are objects remaining at $locB$, the robot moves there, and loads objects until all its hands are occupied. Then it moves back to $locA$, unloads all objects that it gripes. Formula $Inv1$ is the loop invariant of the outer loop. The loop invariants of the two inner loops, denoted as $Inv2$ and $Inv3$, are identical ones. Our system succeeded in verifying the Hoare triple in 32.0s.

We hope to use the two examples above to demonstrate the following points: (1) our method is able to verify some interesting programs, and during the process succinct and comprehensible loop invariants are discovered; (2) the method can handle programs with sequential loops and nested loops in a uniform way; (3) the verification process usually terminates in a reasonable amount of time for such size of programs.

5.2 State Constraint Discovering Examples

When discovering state constraints we assume that type information is given. For instance, in predicate $at(x,y)$ of the $logistic$ domain, x is of type obj and y is of type loc. In this case we include $\forall x,y.at(x,y) \supset obj(x) \wedge loc(y)$, $\forall x.\neg obj(x) \vee \neg loc(x)$ in $\mathcal{D}_{\mathcal{SC}}$ beforehand.

Constraints discovered in the $green$ domain cover the 12 ones of the $blocks\ world$ domain presented in [Li *et al.*, 2013]. Since $green$ domain introduces two new predicates $green$ and $collected$, we discovered 5 more constraints:
$\{collected(x) \supset \neg clear(x), collected(x) \supset \neg on(x,y),$
$collected(x) \supset \neg holding(x), collected(x) \supset \neg on(y,x),$
$collected(x) \supset \neg ontable(x)\}$

Our constraint discovering algorithm is much faster than that in [Li *et al.*, 2013]. We used 103.0s to discover all the 17 constraints in the $green$ domain, while it took our previous system about 15min to discover the 12 ones in *blocks world*.

We discovered 3 constraints in the $gripper+$ domain:
$\{\forall[o,obj][l,loc].gripped(o) \supset \neg at(o,l),$
$\forall[l_1,loc][l_2,loc].rob\_at(l_1) \wedge rob\_at(l_2) \supset l_1 = l_2,$
$\forall[x,num][y,num].holding(x) \wedge holding(y) \supset x = y\}$

5.3 Summary

Domain	Constraint		Verification			
	#A	T	#A	#N	T.avg	T.max
CornerA	2	3.0	1	0	1.0	1.0
Delivery	6	116.0	4	2	1.5	3.0
Green	17	103.0	3	2	24.7	46.0
Gripper	3	26.0	3	1	3.0	6.0
Gripper+	3	26.0	3	2	17.7	32.0
Logistics	5	61.0	2	2	6.0	7.0
Recycle	7	139.0	2	1	15.5	18.0
Transport	2	1137.0	2	1	0.5	1.0
Trash	3	12.0	2	0	0.0	0.0

Table 1: Experimental Results on 9 Domains

We summarize the experimental results as Table 1. Under the Constraint column, #A is the number of all constraints we discovered, and T shows the total time cost. Under the Verification column, #A is the number of all Hoare triples we tested, and all of them were successfully verified. #N is the number of non-trivial verifications. We say a verification is trivial, if during the extended regression, every $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s),$ **while** φ **do** δ **od**$]$ returns $\phi(s) \vee \varphi[s]$ as its loop invariant. T.avg and T.max are the average and maximal time costs of all verifications. Time costs are measured in seconds.

However, there are also a few programs in other domains that our system failed to verify. One reason is that the domains involve arithmetic, which calls for special heuristic functions; another reason is that our method suffers from the scalability problem, mostly due to the inefficiency of the underlying theorem prover.

6 Conclusion

In this paper we have proposed a sound but incomplete method for verifying partial correctness of Golog programs. The basic idea is to use extended regression to reduce the verification of a Hoare triple to a first-order entailment problem. When regressing a loop statement, we discover a proper loop invariant by strengthening a formula with heuristic methods and use small models to filter out too strong candidates.

We summarize our main contributions as follows. Firstly we have developed a practical method with solid theoretical foundations for verifying partial correctness of Golog programs. Secondly, we have proposed heuristic methods for discovering loop invariants, and identified three kinds of

simple but useful heuristics. Thirdly, our algorithm combines static and dynamic analysis from the software engineering community. Fourthly, the method can also be adapted to discover state constraints. Lastly, we have implemented our algorithms, and experimental results show that our method can not only handle sequential and nested loops uniformly in a reasonable among of time, but also discover succinct and comprehensible loop invariants and state constraints.

There are at least three topics for future research. Firstly we would like to introduce heuristics for programs that involve arithmetic. The second challenge is to extend the method such that it could verify total correctness. Finally, we are interested in combining our approach with other successful methods, for example the predicate abstraction method in [Flanagan and Qadeer, 2002].

Acknowledgements

We thank Hector Levesque and Gerhard Lakemeyer for many helpful discussions about this paper. We are grateful to Yuxiao Hu for his valuable help with this work. Thanks also to the anonymous reviewers for useful comments. Yongmei Liu is also affiliated to the Guangdong Key Laboratory of Information Security Technology, Sun Yat-sen University, China. This work was partially supported by the Natural Science Foundation of China under Grant No. 61073053.

References

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proc. of SAT-03*, pages 502–518, 2003.

[Flanagan and Qadeer, 2002] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Conference Record of POPL-02*, pages 191–202, 2002.

[Hu, 2012] Yuxiao Hu. *Generation and Verification of Plans with Loops*. PhD thesis, Department of Computer Science, University of Toronto, 2012.

[Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–83, 1997.

[Li *et al.*, 2013] Naiqi Li, Yi Fan, and Yongmei Liu. Reasoning about state constraints in the situation calculus. In *Proc. IJCAI-13*, 2013.

[Lin and Reiter, 1997] Fangzhen Lin and Raymond Reiter. How to progress a database. *Artificial Intelligence*, 92(1-2):131–167, 1997.

[Liu, 2002] Yongmei Liu. A hoare-style proof system for robot programs. In *Proc. AAAI-02*, pages 74–79, 2002.

[Nguyen *et al.*, 2012] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proc. of ICSE-12*, pages 683–693, 2012.

[Pasareanu and Visser, 2004] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In *Proc. of 11th SPIN Workshop*, pages 164–181, 2004.

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

[Rintanen, 2000] Jussi Rintanen. An iterative algorithm for synthesizing invariants. In *Proc. of AAAI-00*, pages 806–811, 2000.

[Schulz, 2013] Stephan Schulz. System description: E 1.8. In *Proc. of 19th LPAR*, pages 735–743, 2013.