ECAI 2016
G.A. Kaminka et al. (Eds.)

doi:10.3233/978-1-61499-672-9-760

Automatic Verification of Golog Programs via Predicate Abstraction

Peiming Mo and **Naiqi Li** and **Yongmei Liu**[1]

Abstract. Golog is a logic programming language for high-level agent control. In a recent paper, we proposed a sound but incomplete method for automatic verification of partial correctness of Golog programs where we give a number of heuristic methods to strengthen given formulas in order to discover loop invariants. However, our method does not work on arithmetic domains. On the other hand, the method of predicate abstraction is widely used in the software engineering community for model checking and partial correctness verification of programs. Intuitively, the predicate abstraction task is to find a formula consisting of a given set of predicates to approximate a given first-order formula. In this paper, we propose a method for automatic verification of partial correctness of Golog programs which use predicate abstraction as a uniform method to strengthen given formulas. We implement a system based on the proposed method, conduct experiments on arithmetical domains and examples from the paper by Li and Liu. Also, we apply our method to the verification of winning strategies for combinatorial games.

1 Introduction

Among the various AI approaches, high-level programming tries to enable agents to understand and follow human's high-level instructions, and thus process intelligent behaviours. In the programming language Golog [12], the idea of high-level control is embodied in the statements of nondeterministic choices of actions and arguments, based on which the concrete actions to be performed are automatically generated by the interpreter. For robot programs which are expected to terminate, their partial correctness naturally becomes an important concern. Roughly speaking, a program is partially correct if desirable properties hold when it terminates on the condition that some properties hold before the execution of the program.

In a recent paper [13], we proposed a sound but incomplete method for automatic verification of partial correctness of Golog programs where we give a number of heuristic methods to strengthen given formulas in order to discover loop invariants. In our paper, the verification of Golog programs is achieved by the extended regression operator, which has a property similar to that of regression in the situation calculus: a formula holds after a program is executed, if its extended regression holds before the execution. With extended regression, we reduce partial correctness verification to a first-order theorem-proving task. When extended regression is applied to a loop statement, our method will repeatedly try to strengthen a candidate formula until it becomes a loop invariant. However, the heuristics that we developed to strengthen given formulas appear to be quite nonuniform, and our method does not work on arithmetic domains.

[1] Dept. of Computer Science, Sun Yat-sen University, China, e-mail: mopeim3@mail2.sysu.edu.cn & linaiqi@mail2.sysu.edu.cn & ymliu@mail.sysu.edu.cn.

On the other hand, the method of predicate abstraction is widely used in the software engineering community for model checking and partial correctness verification of programs. Intuitively, the predicate abstraction task is to find a formula consisting of a given set of predicates to approximate a given first-order formula. The main practical problem in model checking is the state explosion problem, and predicate abstraction has been successfully used by Ball et al. [2], Clarke et al. [3, 4], etc, in reducing the size of the state space via capturing only relevant features. The most difficult step for partial correctness verification is to discover a proper loop invariant for each loop statement. While the search space of invariants is generally infinite, the technique of predicate abstraction tries to approximate the intended invariant based on a finite set of predicates, and thus the search space becomes finite. Flanagan and Qadeer [9] propose a heuristic method to generate appropriate predicates for each program loop, and then use these predicates to infer the loop invariants. According to their experimental results, predicate abstraction can lead to a very effective verification system, which can infer the necessary invariants for all except 31 of the 396 routines in a 44,380 LOC program within one hour. Srivastava and Gulwani [17] combine templates and predicate verification to infer very expressive loop invariants.

There are also a few works about the verification of Golog programs. Liu [14] presents a Hoare-style proof system for partial correctness of Golog programs. In [5], Claßen and Lakemeyer propose a logic based on the situation calculus variant called $\mathcal{ES}$ to verify temporal properties of non-terminating Golog programs, and this line of research is continued by some recent publications [6, 19]. However, notably, all these works are mainly theoretic studies.

In this paper, we propose a method for automatic verification of partial correctness of Golog programs which uses predicate abstraction as a uniform method to strengthen given formulas. Specifically, the predicate abstraction method is used when a loop is being regressed. Obtaining the extended regression of a loop is equivalent to discovering a proper loop invariant, and we follow the idea proposed in [13]: start from an initial formula, and then repeatedly try to strengthen it until it becomes a loop invariant. While in that paper, the proper loop invariants are generated by handcrafted heuristics, in this paper we try to discover them by the method of predicate abstraction.

We also make use of small models and the technique of small model progression as in the previous paper. Intuitively, small models can be viewed as possible program states during the execution. Given a set of initial small models, which can be viewed as some initial test inputs, small model progression will run the given Golog program on the small models and thus associate a set of possible states with each loop statement. The associated small models can accelerate the process of predicate abstraction: if a candidate formula generated by the predicate abstraction is not satisfied by some of the small model-

s, it is not a proper loop invariant and can be discarded immediately. It is important to emphasize that although our method makes use of small models, it is still capable of verifying properties of programs over arbitrarily large finite domains and even infinite domains. The reason is that small models only serve for the purpose of filtering out formulas which cannot be loop invariants, and we need to resort to first-order theorem-proving to verify a formula is a loop invariant. This also makes our work fundamentally different from the method of model checking, where the systems being verified must be finite.

We implement a verification system based on the proposed method and conduct a set of experiments on it. Experimental results show that our method can prove all the Hoare-triples reported in [13]. The potential of our approach is further demonstrated by experiments on some other domains that involves functions and arithmetical loop invariants. We also apply our system to verify properties of strategies in two famous games $Pick\text{-}up\ Stone$ and $Chomp$, in which the verification of winning strategies is reduced to the verification of partial correctness of Golog programs.

The paper is organized as follows. In Section 2, we will introduce some background knowledge of our work. In Section 3, we will present the related concepts of predicate abstraction, and how to use them to infer loop invariants and thus verify the given Golog programs. In Section 4, we will discuss some technical details. Section 5 will present our experimental results in both arithmetical and non-arithmetical domains. In Section 6 we will show how to apply our method to verify game properties. Finally we will conclude this paper with a summary of contributions and a discussion of future work.

2 Preliminaries

In this section, we will first introduce the preliminaries knowledge. Then we will define Golog program and its partial correctness. In the third subsection we will discuss extended regression, which is introduced in our previous work and will play an important role in this paper. Finally we will introduce small model and its progression.

2.1 The Situation Calculus

The situation calculus [16] is a second-order language specifically designed for representing dynamic worlds. It includes 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.*, $move(x, y)$; a countable set of relational fluents, *i.e.*, predicates taking a situation term as their last argument, *e.g.*, $ontable(x, s)$; and a countable set of functional fluents, *i.e.*, functions taking a situation term as their last argument, *e.g.*, $height(x, s) = y$.

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 $situation\text{-}suppressed$ 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).
   For each relational fluent F:
   $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$,
   For each functional fluent f:
   $f(\vec{x}, do(a, s)) = y \equiv \Phi_f(\vec{x}, y, a, s)$,
   where $\Phi_F(\vec{x}, a, s)$ and $\Phi_f(\vec{x}, y, a, s)$ are 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.

In the situation calculus, state constraints are formulas that hold true in every executable situation. We follow the definition of state constraints in [16].

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

We use $\mathcal{D}_{\mathcal{SC}}$ to denote a set of verified state constraints, and abuse $\mathcal{D}_{\mathcal{SC}}$ as its conjunction.

Regression is an important computational mechanism for reasoning about actions and their effects, and here we present the one step regression operator and a simple form of the regression theorem stated in [13], and we add the functional regression in the definition.

Definition 2 We use $\mathcal{R}_{\mathcal{D}}[\phi]$ to denote the formula obtained from ϕ by the following steps:

1. Replace each functional fluent atom $f(\vec{t}, do(\alpha, \sigma))$ with $(\exists y).\Phi_f(\vec{t}, y, \alpha, \sigma) \wedge \phi[f(\vec{t}, do(\alpha, \sigma))/y]$, where $\phi[x/y]$ means that all x's in ϕ are replaced by y.
2. Replace each relational fluent atom $F(\vec{t}, do(\alpha, \sigma))$ with $\Phi_F(\vec{t}, \alpha, \sigma)$.
3. Replace each precondition atom $Poss(\alpha, \sigma)$ with $\Pi(\alpha, \sigma)$, and further simplify the result by using $\mathcal{D}_{una}$.

Theorem 1 If $\mathcal{R}_{\mathcal{D}}[\phi]$ is the formula regressed from ϕ, then $\mathcal{D} \models \phi \equiv \mathcal{R}_{\mathcal{D}}[\phi]$ holds.

2.2 Golog Programs and Partial Correctness

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** $\equiv [\phi?;\delta_1]|[\neg\phi?;\delta_2]$,
while ϕ **do** δ **od** $\equiv [\phi?;\delta]^*;\neg\phi?$.
Then, the partial correctness of a Hoare triple is defined as below:

Definition 3 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 $\mathcal{D} \models \forall s, s'.P[s] \wedge Do(\delta, s, s') \supset Q[s']$.

2.3 Extended Regression

Li and Liu [13] extended the regression of primitive actions to that of programs, which is called extended regression:

Definition 4 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:

- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \alpha] = \mathcal{R}_{\mathcal{D}}(Poss(\alpha, s) \supset \phi(do(\alpha, s)))$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \psi?] = \psi[s] \supset \phi(s)$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_1;\delta_2] = \hat{\mathcal{R}}_{\mathcal{D}}[\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_2], \delta_1]$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_1|\delta_2] = \hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_1] \wedge \hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta_2]$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), (\pi x)\delta(x)] = (\forall x)\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s), \delta(x)]$.
- $\hat{\mathcal{R}}_{\mathcal{D}}[\phi(s),$ **while** φ **do** δ **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 iff its regression can be entailed by the initial knowledge base. The extended regression has a similar property [13]:

Theorem 2 A Hoare-triple $\{P\}\delta\{Q\}$ is partially correct wrt $\mathcal{D}$, if $\mathcal{D}_{\mathcal{SC}}$ is a set of verified state constraints, and $\models_{\text{FOL}} \forall s.P[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset \hat{\mathcal{R}}_{\mathcal{D}}[Q[s], \delta]$.

2.4 Small Model Progression

As aforementioned, small models can be viewed as possible program states during the execution. In this paper, a small model is represented as a finite set of ground atoms that excludes arithmetical relations. We also make the closed world assumptions to our small models, which means that any non-arithmetical relation that does not appear is regarded as false.

Li and Liu [13] present the notion of small model progression, and use $prog^S[M, \alpha]$ to denote the new model generated by updating small model M according to primitive action α, $prog[M, \delta]$ to denote a set of small models resulting from the progression of M wrt program δ. In this paper, the initial small models are provided by hand for the reason of convenience.

Definition 5 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 $prog[M, \delta]$ in our methods is that they can inform us the Hoare triple $\{P\}\delta\{Q\}$ is not partial correct before starting the static analyses, which derives from the following theorem [13]:

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 $M'[s] \models Q[s]$, where $M[s]$ is a small model with situation s restored.

Besides, the small models can largely improve the efficiency of predicate abstraction because we use the small models to filter the wrong candidates before calling an SMT solver in the process of predicate abstraction. We can see these details in the next section.

3 Infer Invariants via Predicate Abstraction

In this section, we will first introduce the standard predicate abstraction, and how we adapt it to the bounded predicate abstraction to serve our purpose. And then we will present a concrete algorithm to compute the bounded predicate abstraction. Finally, we will see how it is used to discover loop invariants and thus verify the Hoare-triple.

The definition of standard predicate abstraction we use is the one presented in [9]:

Definition 6 (Standard Predicate Abstraction) Given a set of predicates $\mathcal{P} = \{p_1, ..., p_n\}$, for any formula Q, its abstraction $\alpha(Q)$ is defined as the strongest boolean combination on $\mathcal{P}$ such that $Q \supset \alpha(Q)$ is valid.

Now we present the predicate abstraction defined in our paper, and discuss how it is different from the standard one.

We say a formula ϕ is a $\forall^*\exists^*$ formula, if ϕ is of the form $\forall x_1...\forall x_n\exists y_1...\exists y_m\psi$, where $n, m \geq 0$ and ψ is a quantifier-free formula.

Definition 7 (Predicate Abstraction) Given a set of predicate $\mathcal{P}$, a formula ϕ, and a set of small models $\mathcal{M}$, we say a formula $\forall^*\exists^*C$ is the predicate abstraction of ϕ and $\mathcal{M}$ under $\mathcal{P}$, if C is a clause over $\mathcal{P}$ s.t. $\mathcal{M} \models \forall^*\exists^*C$ and $\mathcal{D}_{\mathcal{SC}} \models \forall^*\exists^*C \supset \phi$, and there is no shorter clause C' over $\mathcal{P}$ s.t. $\forall^*\exists^*C'$ satisfies these conditions.

The predicate abstraction we developed differs from the standard one in the following ways: 1. in the standard predicate abstraction all free variables in $\alpha(Q)$ are implicitly regarded as universally quantified, but in our definition both universal and existential quantifiers are possible; 2. the predicate abstraction in our approach results in a quantified clause, while the standard predicate abstraction results in

a formula; 3. the conditions in our definition are different from that in the standard definition.

The concept of predicate abstraction is already enough to solve many non-arithmetical verification problems. But during our research we observe that many loop invariants in arithmetical domains are of a particular form, which inspires us a more general definition.

Definition 8 (Bounded Predicate Abstraction) Let $Fun0$ and $Const$ denote the sets of 0-ary functional fluents and constants respectively, and $\mathcal{P}$, ϕ and $\mathcal{M}$ as before. A bounded predicate abstraction of ϕ and $\mathcal{M}$ under $\mathcal{P}$ is a formula of the form $\varphi = \forall^*\exists^* bound(\vec{x}) \supset C(\vec{x})$ s.t. $\mathcal{M} \models \varphi$ and $\mathcal{D}_{\mathcal{SC}} \models \varphi \supset \phi$, where $C(\vec{x})$ is a clause over $\mathcal{P}$, and $bound(\vec{x})$ is of the form $\bigwedge_i lb_i \leq x_i \leq hb_i$ where $lb_i, hb_i \in \{\vec{x}\} \cup Fun0 \cup Const$. There is no shorter $C'(\vec{x})$ s.t. it has a bound that makes $\forall^*\exists^* bound'(\vec{x}) \supset C'(\vec{x})$ satisfy these conditions.

In the algorithms below, we assume that $\mathcal{D}$ and $\mathcal{D}_{\mathcal{SC}}$ are given without providing them in the arguments explicitly.

Algorithm 1 is used to compute the bounded predicate abstraction. In our implementation, we only consider $\forall^*$ and $\forall^*\exists$ formulas because of efficiency. The idea is quite simple: we enumerate clauses from short ones to long ones, and then enumerate the different combinations of the quantifiers, until a formula that satisfies the conditions is found. In line 5, the algorithm calls an SMT solver to check whether the first-order entailment does hold (an SMT solver can decide whether a first-order formula is satisfiable with respect to some background theory such as linear arithmetic). If the SMT solver does not terminate in a given time, we treat the result as false. Similar cases are treated in the same way during algorithms 2 and 3.

Algorithm 1: $boundPA(\phi, \mathcal{M}, \mathcal{P})$

Input: ϕ - formula to be strengthened; $\mathcal{M}$ - the set of small models; $\mathcal{P}$ the set of predicate candidates
Output: The bounded predicate abstraction φ'

1 **repeat** *Enumerate a clause $C(\vec{x})$ over $\mathcal{P}$ (enumerate from short ones to long ones)*
2 **repeat** *Enumerate a bound $bound(\vec{x})$*
3 Let $\varphi = bound(\vec{x}) \supset C(\vec{x})$
4 **repeat** *Enumerate φ' as $\forall^*\varphi$ or $\forall^*\exists\varphi$*
5 **if** $\mathcal{M} \models \varphi'$ *and* $\mathcal{D}_{\mathcal{SC}} \models \varphi' \supset \phi$ **then**
6 **return** φ'
7 **return** false

Next we will see how to apply the bounded predicate abstraction to strengthen a formula into loop invariant. The procedure $infer$ listed as Algorithm 2 is similar to the corresponding algorithm in [13]. It is used to infer a loop invariant when a loop statement is being regressed.

In Algorithm 2, we use the association function $asso(\delta_l)$ to map each loop statement δ_l to a set of small models $\mathcal{M}$. Intuitively $\mathcal{M}$ is the set of small models that are progressed to the beginning of the loop. For every loop, the loop invariant should be satisfied by all the associated small models. In Line 2, we will first discover a set of linear and inequality invariants Δ_{inv} with a high efficient method, the detail of which will be discussed in the next section. The idea here is to discover the simple linear and inequality invariants first, and then use them to discover the difficult ones later. In Line 3, we start with the candidate loop invariant as $\phi(s) \vee \varphi[s]$, and we try to repeatedly

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

Input: $\phi(s)$ - formula being regressed; δ_l - loop statement being regressed; $asso$ - maps each **while** construct to a set of small models
Output: A loop invariant for δ_l.

1 Let $\delta_l =$ **while** φ **do** δ **od**
2 $\Delta_{inv} \leftarrow getLinearInvs(\delta_l, asso(\delta_l))$
3 $\eta(s) \leftarrow \phi(s) \vee \varphi[s]$;$counter \leftarrow 0$
4 **while** $counter < K$ **do**
5 $counter \leftarrow counter + 1$
6 $reg(s) \leftarrow \hat{\mathcal{R}}_{\mathcal{D}}(\eta(s), \delta)$
7 **if** $\models_{\text{FOL}} \eta(s) \wedge \varphi[s] \wedge \mathcal{D}_{\mathcal{SC}} \wedge \Delta_{inv} \supset reg(s)$ **then**
8 **return** $\eta(s) \wedge \Delta_{invs}$
9 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 $\mathcal{P} \leftarrow genPredicate(A_i(s), \delta_l)$
11 $\varphi(s) \leftarrow boundPA(A_i(s), asso(\delta_l), \mathcal{P})$
12 **if** $\varphi(s) = \emptyset$ **then**
13 **return** $unknown$
14 $\eta(s) \leftarrow \eta(s) \wedge \varphi(s)$
15 **return** $unknown$

strengthen it until it becomes an invariant in the following loop. In the following loop that starts from Line 4, the variable $counter$ and a preset constant K are used to make sure that the procedure always terminates (with the premise that the underlying theorem prover always terminates). In each iteration, if the regressed result can be entailed at Line 7, we return $\eta(s) \wedge \Delta_{invs}$ as the loop invariant. If the entailment cannot be proved, in Line 9 we select one of the unentailed conjunct $A_i(s)$, and then use the information of $A_i(s)$ and the loop δ_l to generate a predicate set $\mathcal{P}$. Line 11 is our abstraction algorithm. If the abstraction result is empty, it means the predicate set $\mathcal{P}$ we generated is not precise or the Hoare-triple is wrong, then we return unknown in Line 13 under that situation, else we apply the predicate abstraction on it to obtain a strengthening at Line 14. The bounded predicate abstraction requires a set of small models and a set of predicate candidates. The set of small models is retrieved from the $asso$ function, and the predicate candidates are automatically generated in Line 10. We will discuss how to generate predicates in further details in the next section.

Finally a Hoare-triple is verified by the following main algorithm, which can be regarded as a simplified version of the main algorithm in [13]. Intuitively, the main algorithm will first try to use small models and progression to find a counterexample. If no counterexample is found, it will apply the extended regression and reduce the verification problem to first-order entailment problem. If all such attempts fail, the algorithm simply returns $non\text{-}deter$. In this algorithm, the procedure $prog$ is almost the same as the definition of small model progression, except that it also updates the association function during the process. The procedure returns no if a counterexample is found at Line 3, and returns yes if the regressed result can be entailed by the precondition. For other cases, the system returns $non\text{-}deter$.

Theorem 4 $veri(P, \delta, Q, \mathcal{M})$ returns yes only if $\{P\}\delta\{Q\}$ is partial correct; returns no only if $\{P\}\delta\{Q\}$ is not partially correct.

Proof: Firstly, if $veri(P, \delta, Q, \mathcal{M})$ returns yes, it means that $\models_{\text{FOL}} \forall s.P[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset reg(s)$. Then according to Theorems 2, $\{P\}\delta\{Q\}$

Algorithm 3: $veri(P, \delta, Q, \mathcal{M})$

Input: Hoare-triple $\{P\}\delta\{Q\}$; $\mathcal{M}$ - initial small models that satisfy P and $\mathcal{D}_{\mathcal{SC}}$.
Output: Returns yes if Hoare-triple is proved to be correct; returns no if Hoare-triple is proved to be wrong; otherwise returns $non\text{-}deter$

1 Let $asso$ maps each **while** construct in δ to $\emptyset$
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 $reg(s) \leftarrow \hat{\mathcal{R}}_{\mathcal{D}}(Q[s], \delta, asso)$
5 **if** $\models_{\text{FOL}} \forall s.P[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset reg(s)$ **then return** yes
6 **return** *non-deter*

is partial correct. Secondly, according to Theorems 3, if $\{P\}\delta\{Q\}$ is partial correct, then $\forall M' \in \mathcal{M}'$ we should have $M'[s] \models Q[s]$ (we guarantee that all initial small models satisfy P and $\mathcal{D}_{\mathcal{SC}}$). So if $\exists M' \in \mathcal{M}'$ s.t. $M'[s] \not\models Q[s]$, then $\{P\}\delta\{Q\}$ is not partial correct.

4 Technical Details

In this section, we will discuss some technical issues of our algorithm in further details. Firstly we will show how to generate predicate candidates. Secondly we will discuss the bound generation and disturbance problem. Finally we will present a method to efficiently discover linear and inequalities loop invariants.

4.1 Generate Predicate Candidates

The success of Algorithm 1 largely depends on the given predicate candidates. The following shows how we generate this set.

Let p be a predicate symbol with k arguments. We say a ground atom $p(t_1, ..., t_k)$ is a predicate candidate, where $t_1, ..., t_k$ are terms. Our system will automatically generate a set of predicate candidates and use them in the predicate abstraction.

We use $genPredicate(\phi, \delta_l)$ to denote the set of predicate candidates generated by the formula ϕ and the loop statement δ_l. Intuitively, formula ϕ is the formula being abstracted and δ_l is the loop being regressed. Suppose the Hoare-triple being verified is $\{P\}\delta\{Q\}$, we regard formulas P and Q as globally accessible and will use them in the generation process. The set of predicate candidates is computed in the following steps:

1. Initialize $\mathcal{P}$ as an empty set.
2. $\mathcal{P} \leftarrow \mathcal{P} \cup \{p | p$ is a predicate appears in P, Q or $\phi\}$.
3. $\mathcal{P} \leftarrow \mathcal{P} \cup \{p | \psi?$ is a test action in δ_l and p is a predicate in $\psi\}$.
4. Suppose that the predicate candidates generated in the previous steps are $\mathcal{P} = \{p_1(\vec{t_1}), ..., p_n(\vec{t_n})\}$. We collect all the terms that appear as arguments of some predicates, which is $\mathcal{T} \leftarrow \{t | \exists p_i(\vec{t_i}) \in \mathcal{P}.t \in \vec{t_i}\}$.
5. The final set of predicate candidates is $\mathcal{P}' \leftarrow \{p_i(\vec{t_i}') | \vec{t_i}' \in \mathcal{T}\}$.

Example 1 Suppose that the predicates generated in the first three steps are $\{in(x, f(y)), x < f(y)\}$. The set of terms is $\{x, f(y)\}$, so we will generate $\{in(x, x), in(x, f(y)), in(f(y), x), f(y) < x, x < f(y), in(f(y), f(y))\}$ as the set of new predicates. Theoretically $x < x$ and $f(y) < f(y)$ should also be generated, but with the knowledge of mathematics it is easy to verified that they are equivalent to $false$, so we will remove them during the process.

4.2 Generate Bounds

When generating bounds in the process of bounded predicate abstraction, there are two problems worth noting. The first problem is how to construct the set $Const$ as the constants being enumerated. The second problem is that in most verifications, the bounds will be 'disturbed' so we also need to consider the disturbance of bounds.

Recall that in bounded predicate abstraction, a bound is of the form $\bigwedge_i lb_i \leq x_i \leq hb_i$ where $lb_i, hb_i \in \{\vec{x}\} \cup Fun0 \cup Const$. In this formula, $\{\vec{x}\}$ and the set of 0-ary functional fluents $Fun0$ can be obtained from the clause $C(\vec{x})$ and the domain description respectively. However, it is impossible to construct $Const$ as all the constants in the domain, since the set of constants may be huge or even infinite. In order to restrict the number of bounds being enumerated, we heuristically assign different variables with different sets of $Const$, and make use of the information in the small models. The approach can be better demonstrated by the example below.

Example 2 Suppose the clause $C(\vec{x}) = vis(x, y)$, and the associated small models of the loop that being regressed is $\{vis(1,1), vis(1,2), vis(1,3), vis(2,1), vis(2,2), vis(2,3)\}$. Variable x is the first argument of predicate vis, and from the small model we know the range of the first argument of predicate vis is $[1, 2]$. So for variable x, we will construct its $Const$ as $\{1, 2\}$. Similarly, for variable y its $Const$ is $\{1, 3\}$. As a result, the bounded predicate abstraction method will finally generate formulas like $\forall x, y.1 \leq x \leq 2 \wedge 1 \leq y \leq 3 \supset vis(x, y)$.

Another problem is the disturbance of the bounds. In many verifications, the actually desired bound may be just a little different from that generated by the previous method. For example, we may generate $0 \leq x \leq xpos$, while the desired bound is $1 \leq x \leq xpos + 1$. In order to enable our system discover such bounds, we will also enumerate bounds with disturbance $\pm k$, where k is a small constant, but we replace k with 1 under the consideration of efficiency and experience. To put it more formally, if $lb_i \leq x \leq hb_i$ is enumerated by the previous method, we will also enumerate other 4 bounds $lb_i \pm 1 \leq x \leq hb_i \pm 1$ as its disturbance.

4.3 Discover Linear Relations and Inequalities

To further improve the performance, we treat the loop invariants that are linear relations and inequalities specially. In the software engineering community, there are many available works on efficiently discovering linear and inequality invariants, such as [8] and [18].

Since the linear and inequality invariants can be discovered with special method much faster than by using predicate abstraction, our algorithm try to discover them in advance. This is demonstrated as function $getLinearInvs$ at Line 2 of Algorithm 2. Our implementation of $getLinearInvs$ follows the guess-and-check paradigm.

Firstly, the system will guess a set of candidate invariants based on dynamic checking and linear regression as follows. Suppose the set of small models is $\mathcal{M} = \{M_1, ..., M_k\}$. From each small model M_i, we obtain a set of function-value pairs of all 0-ary functional fluents, such as $\{f_0 = v_{0i}, f_1 = v_{1i}, ..., f_n = v_{ni}\}$. The inequality and linear invariant candidates are generated by the following methods:

- Inequality candidates: We consider inequality invariant candidates of the forms $f_u < f_v$ and $f_u < C$, where f_u, f_v are 0-ary functional fluents and C is a constant in the Hoare-triple. An inequality formula will be included as an invariant candidate if it is satisfied in every model.

- Linear candidates: In our implementation, we consider all linear relations of the form $A \cdot f_u + B \cdot f_v + C \cdot f_w + D = 0$. This is equivalent to solving the following system of linear equations:

$$A \cdot v_{u1} + B \cdot v_{v1} + C \cdot v_{w1} + D = 0$$
$$\cdots\cdots$$
$$A \cdot v_{uk} + B \cdot v_{vk} + C \cdot v_{wk} + D = 0$$

  Suppose $\langle a, b, c, d\rangle$ is the solution of the system, we will include the linear relation $a \cdot f_u + b \cdot f_v + c \cdot f_w + d = 0$ as an invariant candidate.

After the guessing step, we obtain a set of candidate invariants Δ, and then each formula in Δ will be checked statically. The checking step is also an application of the extended regression. Suppose the loop is **while** φ **do** δ, for every candidate invariant $\eta \in \Delta$ we will compute its regression $reg(s) = \hat{\mathcal{R}}_{\mathcal{D}}[\eta[s], \delta]$ and then check that whether the entailment $\models_{\text{FOL}} \eta[s] \wedge \varphi[s] \wedge \mathcal{D}_{\mathcal{SC}} \supset reg(s)$ can be proved. If we can prove the entailment, we know $\eta[s]$ is indeed a loop invariant. Otherwise, we will simply give up the candidate.

To ensure the efficiency, we will not try to strengthen the candidate invariant as Algorithm 2 does when the system fails to prove the entailment. However, experiments show that even such simple strategy is enough to discover and validate lots of useful linear and inequality invariants.

5 Experimental Results

We have implemented our algorithms to a system by using SWI-Prolog, Java and Z3. Java is used to discover linear relations because of its efficiency and powerful API, and Z3 [7] is the SMT solver which is used to prove the first-order entailments. They use the I/O interface to work together.

Our system has run on six arithmetical domains (with five succeed and one failed) and all the non-arithmetic domains used in [13]. Among all the arithmetical domains, $1D$, $PrizeA1$, $Arith$ and $Sort$ are adapted from those used in [10]. $Find$ is a modified version from [9], and $Addition$ is designed by ourselves. All experiments were conducted on a machine with 3.30 GHz CPU and 4.00GB RAM under Linux. In all our experiments the initial small models are manually provided, and only 1 or 2 simple initial small models are sufficient for each domain.

5.1 Arithmetical Domains

The arithmetic experiments are derived from the following domains:
1D: The program is to visit all the elements in an array from right to left.
PrizeA1: This program is to visit all the $N \cdot N$ cells from row 1 to row N in the outer loop, and from left to right during every row in the inner loop.
Arith: This program is to increase a variable $numy$ (initialized as 0) to $2N$. It contains a single loop, and $numy$ will increase by two during each iteration.
Find: This program sets the i-th element of a boolean array inb to be true if the i-th element of an integer array ina is nonzero, and to false otherwise. And we treat $ina(X)$ as a 1-ary functional fluent and $inb(X)$ as a 1-ary relational fluent.
Addition: This program is to increase a variable sum (initialized by 0) to $A \cdot B$ through a nested loop. The outer loop repeats A times and the inner loop B times, the variable sum is increased by one during every inner loop's iteration.
Sort: The program sorts an array by using a single loop.

In the experiments of the arithmetical domains, we provide the set of state constraints as $\mathcal{D}_{\mathcal{SC}}$ in advance. For all the experiments, we only provide constraints of type information. For example in the $1D$ domain, the argument x of $vis(x)$ is expected to be an integer, so we will include the constraint $\forall x.vis(x) \supset int(x)$. In the following, we will use $\forall x \in int.\phi(x)$ to denote $\forall x.int(x) \supset \phi(x)$.

We will take $1D$ and $Sort$ as examples to demonstrate our system, while the results are summarized in Table 1. We only mention the situation arguments explicitly in the precondition axioms and successor state axioms, and omit them in the Hoare-triple and the loop invariant formulas.

Figure 1 presents the Hoare-triple and verification results of the $1D$ domain. It only contains one loop. The constant len denotes the length of the array. The action $move\_left$ has two effects: firstly it decreases $xpos(s)$ by one, where $xpos(s)$ is a 0-ary functional fluent used as a pointer, and then it makes the predicate $vis(xpos(s), s)$ to be true. The resulted loop invariant is the conjunction of $Invs$ and $Linear$, where $Invs$ is discovered by the predicate abstraction algorithm, while $Linear$ is a set and the conjunction of its elements denotes the useful linear relations between terms. This domain is verified by our system in 1.0s.

Hoare-triple:
$\{xpos = len\}$
while $0 < xpos$ **do** $move\_left$ **od**
$\{\forall x \in int.0 < x < len + 1 \supset vis(x)\}$

Action Precondition Axiom:
$Poss(move\_left, s) \equiv xpos(s) > 0$
Successor State Axiom:
$xpos(do(a, s)) = y \equiv xpos(s) = y \vee$
$\qquad xpos(s) = y + 1 \wedge a = move\_left$
$vis(x, do(a, s)) \equiv vis(x, s) \vee$
$\qquad \neg vis(x, s) \wedge a = move\_left \wedge xpos(s) = x + 1$
$Invs$: $\forall x \in int.xpos < x < len + 1 \supset vis(x)$
$Linear$: $\{xpos < len + 1\}$

Figure 1. Example of the $1D$ Domain

In Figure 2, the program sorts an array with length len in a single loop. During each iteration, $xpos(s)$ moves right if $in(xpos(s), s) < in(xpos(s) + 1, s)$, otherwise, it swaps the value of $in(xpos(s), s)$ and $in(xpos(s) + 1, s)$ and then resets $xpos(s)$ to be 1. After 224.0s, an invariant (the conjunction of $Invs$ and $Linear$) is found and the triple is verified.

Now we report our experimental results on some other arithmetic domains. We can see in Table 1 that every domain has two rows in its Invariant column. They are $Invs$ and $Linear$, and their meanings are as discussed in $1D$ domain. Recall that to generate the invariants, our system firstly tries to discover linear and inequality relations with more efficient method. We can see the results of $Arith$ and $Addition$ in Table 1 that this process is efficient. If there is no any useful linear relations, the system then will go on to call the predicate abstraction method which can discover more general invariants, and it will still be a fast process if the predicate set is small and the length of the resulted clause is short, such as the case in $1D$. But it will cost lots of time if the predicate set is large and the resulted clause is complex, such as the cases in $Find$ and $Sort$.

The failure of $PrizeA1$ is mainly because its verification requires a conjunction of two clauses ($\varphi_1 \wedge \varphi_2$) to strengthen the given formula by doing the abstraction only once, but Algorithm 1 can merely output φ_i which can not be found here because neither

Hoare-triple:
$\{xpos = 1\}$
while $xpos < len$ **do**
$\pi(x_1, x_2 \in int.\langle xpos = x_1 \wedge x_1 + 1 = x_2\rangle?)$
$\langle in(x_1) < in(x_2)\rangle?; move\_right|$
$\langle -(in(x_1) < in(x_2))\rangle?; swap(x_1, x_2); re\_set$
od
$\{\forall a1, a2 \in int.a1 < a2 \supset in(a1) < in(a2)\}$

Action Precondition Axiom:
$Poss(move\_right, s) \equiv xpos(s) < len$
$Poss(re\_set, s) \equiv 0 < xpos(s) < len$
$Poss(swap(x, y), s) \equiv true$
Successor State Axiom:
$xpos(do(a, s)) = y \equiv xpos(s) = y \vee y = 1 \wedge a = re\_set \vee$
$xpos(s) = y - 1 \wedge a = move\_right$
$in(x, do(a, s)) = y \equiv$
$in(x, s) = y \vee \exists z.in(z, s) = y \wedge a = swap(x, z)$
$Invs$: $\forall a1, a2 \in int.a1 < a2 < xpos + 1 \supset in(a1) \leq in(a2)$
$Linear$: $\{0 < a1, 0 < xpos\}$

Figure 2. Example of the $Sort$ Domain

Domain	Invariant		Time
1D	$\forall x \in int.xpos < x \supset vis(x)$		1.0
	$\{xpos < len + 1\}$		
PrizeA1	-		-
Arith	$numy = 10 \vee i < N$		1.0
	$\{0 \leq i, i < N + 1,$ $-10 \cdot i + 5 \cdot numy = 0\}$		
Find	$\forall i \in int.i < xpos \supset$ $\neg inb(i) \vee ina(i) = 0$		444.0
	$\{0 \leq xpos, xpos < length + 1\}$		
Addition	inner	$sum = A \cdot B \vee$ $i + 1 < A \vee j < B$	2.0
		$\{0 \leq i, i < A, 0 \leq j,$ $j < B + 1, 0 \leq sum,$ $i \leq sum, j \leq sum,$ $20 \cdot i + 4 \cdot j - 4 \cdot sum = 0\}$	
	outer	$sum = A \cdot B \vee i < A$	
		$\{-25 \cdot i + 5 \cdot sum = 0\}$	
Sort	$\forall a1, a2 \in int.a1 < a2 < xpos + 1 \supset$ $in(a1) \leq in(a2)$		230.0
	$\{0 < a1, 0 < xpos\}$		

Table 1. Performance of the Arithmetic Domains

$\varphi_1 \supset \phi$ nor $\varphi_2 \supset \phi$ holds. If we adjust Algorithm 1 to search the conjunctions of clauses, its search space will explode.

5.2 Comparisons on the Non-Arithmetic Domains

In the experiments of non-arithmetic domains, we also manually provide the state constraints in advance. The constraints we use here are exactly those reported in our previous work [13], where all the constraints are automatically discovered and verified.

We can see in Table 2 that our method can successfully cover all the domains used in [13]. #A is the number of all Hoare-triples we tested, and T.avg and T.max are the average and maximal time costs of all verifications. Time costs are measured in seconds. All verifications in domains $CornerA$, $Transport$ and $Trash$ are trivial, i.e, 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, so T.maxs and T.avgs are reported to be 0. Because our method can generate more general predicates during the verification, it can discover more general invariants theoretically. But as we can see in Table 2 that the generality is at the cost of efficiency due to the large search space.

When comparing our new experimental results with the previous ones, both machines we use are equipped with 4.00GB RAM and operate under Linux. But our new machine is equipped with an Intel i5, 3.30 GHz CPU, while the old one uses i7, 2.60 GHz CPU.

Domain	Old Method			New Method		
	#A	T.max	T.avg	#A	T.max	T.avg
CornerA	1	1.0	1.0	1	0	0
Delivery	4	3.0	1.5	4	514.0	150.2
Green	3	46.0	24.7	2	1388.0	1062.0
Gripper	3	6.0	3.0	3	2138.0	815.7
Logistics	2	7.0	6.0	2	23.0	11.5
Recycle	2	18.0	15.5	2	964.0	483.5
Transport	2	1.0	0.5	2	0	0
Trash	2	0	0	2	0	0

Table 2. Performance of the Non-Arithmetic Domains

6 Applications in Strategy Verification

In this section we use our method to verify the winning property and executability of strategies in combinatorial games.

Giving a strategy, our first job is to encode it into a Golog program δ that can be recognized by our system.

Definition 9 A strategy $\mathcal{S}$ of a player A is a finite set (with size m) that denotes the moves of A under some given conditions, i.e, every element of $\mathcal{S}$ is a pair $\langle \phi_i, \sigma_i \rangle$ which means that A will choose σ_i as her next move under the condition of ϕ_i, where ϕ_i is a formula that is true in the given game state, and σ_i is an action according to ϕ_i.

Then a winning strategy (WS) of A is a strategy such that no matter how the opponent plays, A is guaranteed to win. It can be encoded into a Golog program δ_{WS} as below:

- $\delta_{WS} \doteq [\phi_1?; \sigma_1]|...|[\phi_m?; \sigma_m]$

For example, in the $Pick\text{-}up\ Stone$ game, there are two players A and B taking turns to pick up i stones on the table ($i \in \{1, 2, 3\}$). We use the variable n to denote the number of stones left on the table ($n > 0$ initially). The one who picks the last stone loses the game. We say that A has a winning strategy if n satisfies the property ($n\%4 \neq 1$) every time in her turn. Then the WS of A should be $\{\langle n\%4 = 0, pick(3)\rangle, \langle n\%4 = 2, pick(1)\rangle, \langle n\%4 = 3, pick(2)\rangle\}$.

6.1 Partial Correctness of WS

To verify the partial correctness of WS, we should construct a Hoare-triple $\{P\}\delta_P\{Q\}$, where $\{P\}$ is the pre-condition that always denotes the initial state of the game, $\{Q\}$ is the post-condition that we call win-condition, and δ_P is as below:

- $\delta_{P_1} \doteq$ **while** φ **do** $(\pi\overrightarrow{X})a(\overrightarrow{X}); \delta_{WS}$ **od**, or
- $\delta_{P_2} \doteq$ **while** φ **do** $\delta_{WS}; (\pi\overrightarrow{X})a(\overrightarrow{X})$ **od**

where φ is the termination condition of the game, $a(\overrightarrow{X})$ is the action in the game, and the choice of δ_{P_1} or δ_{P_2} depends on which player moves first. Specifically, if the player of the strategy being verified moves first we will use δ_{P_2}, and otherwise we will use δ_{P_1}.

Take the previous $Pick\text{-}up\ Stone$ game as an example, the $\{P\}\delta_P\{Q\}$ of A can be encoded as:

$$\{n\%4 \neq 1 \wedge turn(A)\}\delta_{P_2}\{turn(A)\}$$

Domain	Partial Correctness		Executability	
	Inv	T	Inv	T
$Pick\text{-}up\ Stone$	$(n > 0 \vee turn(A)) \wedge$ $(n\%4 = 0 \vee turn(B)) \wedge$ $(n\%4 \neq 0 \vee n = 0)$	13.0	$\forall x \in int.$ $(n \geq x \wedge (x = 1 \vee x = 2 \vee x = 3)) \supset$ $(n - x)\%4 = 0 \vee (n - x)\%4 = 2 \vee (n - x)\%4 = 3$	3.0
$Chomp\ 2 \times N$	$(length(row[1]) > 0 \vee turn(A)) \wedge$ $(length(row[1]) = 0 \vee turn(B))$	6.0	#T	1.0
$Chomp\ N \times N$	$(length(row[1]) = 0 \vee turn(A)) \wedge$ $(length(row[1]) = 0 \vee turn(B))$	8.0	#T	1.0

Table 3. Experimental Results of WSs

where $\delta_{P_2} \doteq$ **while** $n > 0$ **do** $\delta_{WS}; (\pi x) pick(x)$; **od**, and $\delta_{WS} \doteq [n\%4 = 0?; pick(3)]|[n\%4 = 2?; pick(1)]|[n\%4 = 3?; pick(2)]$. Here the win-condition is $\{turn(A)\}$ because after B picking the last stone it should be A's turn.

Nevertheless, how does it make sure that every time in her turn, A (the winner) can take a move according to her WS, i.e, every time in A's turn, the game state s can entail the disjunction of situations in her WS ($s \supset \bigvee \phi_i$)? And is it enough to say that WS is really a winning strategy of A by only proving the partial correctness of $\{P\}\delta_P\{Q\}$? The answer is no. Assume that we change the WS of A to a wrong WS' by replacing its first element $\langle n\%4 = 0, pick(3)\rangle$ by $\langle n\%4 = 0, pick(2)\rangle$, there will be a counterexample if B performs $pick(1)$ under the pre-condition of $\{n = 4 \wedge turn(A)\}$, which leaves only one stone to A' last turn. Since A has no strategy to deal with such situation in WS', the executability of δ_P will be invalid.

Therefore, in order to ensure the correctness of a WS, it is the executability that should also be taken into consideration.

6.2 Executability of WS

Since we have specified WS, we can use our system to verify the executability of WS too. But now, the Hoare-triple may be a little different. Because it is the executability that we want to verify, the post-condition should be the disjunction of situations in WS, and the while loop in δ_P should be replaced by a non-deterministic loop. With the same pre-condition, now the Hoare-triple is $\{P\}\delta_E\{\bigvee \phi_i\}$, where δ_E is:

- $\delta_{E_1} \doteq [(\pi \overrightarrow{X}) a(\overrightarrow{X}); \delta_{WS}]^*; (\pi \overrightarrow{X}) a(\overrightarrow{X})$
- $\delta_{E_2} \doteq [\delta_{WS}; (\pi \overrightarrow{X}) a(\overrightarrow{X})]^*$

Similarly, which of the two should be used depends on which player moves first. If player being analysed moves first we will use δ_{E_2}, and otherwise we will use δ_{E_1}.

When referring to the WS of A in the $Pick\text{-}up\ Stone$ game, we can encode its Haore-triple $\{P\}\delta_E\{Q\}$ as:

$$\{n\%4 \neq 1 \wedge turn(A)\}\delta_{E_2}\{n\%4 = 0\ \vee n\%4 = 2 \vee n\%4 = 3\}$$

where $\delta_{E_2} \doteq [\delta_{WS}; (\pi x) pick(x)]^*$ and δ_{WS} as before.

6.3 Summary of Experiments

Besides $Pick\text{-}up\ Stone$, we have tried another game $Chomp$ [15], which has two versions as below:

Chomp 2×N: There is a $2 \times N$ grid, and each point contains a cookie. The positions of them can be represented as a two dimensional array $row[2][N]$ that starts from 1, and the one in $row[1][1]$is poisonous. There are two players A and B taking turns to pick one of the remaining cookies, and once she picks $row[a][b]$, she must eat all the cookies in $row[i][j]$, where $a \leq i \leq 2$ and $b \leq j \leq N$. The one who has to eat the poisonous cookie loses the game. We say that the first player A has a winning strategy by keeping the number of the first row of cookies one greater than the number of the second row every time after her move.

Chomp N×N: This game is similar to $Chomp\ 2 \times N$, but this time there are N rows of cookies initially. Player A has a winning strategy too, that is to eat $row[2][2]$ in her first step and then keep the length of the first row equivalent to the length of the first column every time after her move.

Table 3 shows that the WSs of these three games are correct. And interestingly, when proving their partial correctness, the discovered invariants are much simpler than we thought (for example in $Pick\text{-}up\ Stone$, the intuitive invariant is $n = 0 \supset turn(A) \wedge n > 0 \supset turn(B) \wedge n\%4 = 1$), but the results are valid according to the proof system introduced in [1]. #T means that a verification is trivial, i.e, 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.

7 Conclusion

In this paper, we propose a uniform method to verify the partial correctness of Golog programs that may involve functions and arithmetic. We summarize our main contributions as follows: Firstly, we combine the extended regression with the predicate abstraction method, which results in a uniform verification method that is capable of handling programs with functions and arithmetic. Secondly, to make our approach effective and feasible, we develop some techniques like generating predicates, generating bounds and efficiently discovering linear and inequality invariants. Thirdly, we have implemented a verification system, conducted a set of experiments and compared it with our previous work, showing the capability and potential of our approach. Lastly, we have also applied the method to the verification of winning strategies in two famous games, $Pick\text{-}up\ Stone$ and $Chomp$.

As for the future work, we would like to further improve the performance of our system and make the approach more practical. One possible direction is to introduce human-machine interaction to the process of predicate abstraction. For example, a user may analyse the output of the system and add new predicates into the candidate set, or provide some invariant templates to accelerate the search. Another direction we have in mind is to combine our method with algorithmic learning, such as applying the CDNF algorithm as in the work of [11], where the intended loop invariant is learned by interacting with a mechanical teacher. We believe that by posing informative queries during the interaction, the algorithmic learning method can guide and thus speed up the search process.

Acknowledgments

We thank the anonymous reviewers for helpful comments. This work received support from the Natural Science Foundation of China under Grant No. 61572535.

REFERENCES

[1] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog, *Verification of Sequential and Concurrent Programs*, Texts in Computer Science, Springer, 2009.

[2] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani, 'Automatic predicate abstraction of C programs', in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pp. 203–213, (2001).

[3] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, 'Counterexample-guided abstraction refinement', in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, (2000).

[4] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, 'Counterexample-guided abstraction refinement for symbolic model checking', *J. ACM*, **50**(5), 752–794, (2003).

[5] Jens Claßen and Gerhard Lakemeyer, 'A logic for non-terminating golog programs', in *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pp. 589–599, (2008).

[6] Jens Claßen, Martin Liebenberg, Gerhard Lakemeyer, and Benjamin Zarrieß, 'Exploring the boundaries of decidable verification of non-terminating golog programs', in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence.*, pp. 1012–1019, (2014).

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner, 'Z3: an efficient SMT solver', in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pp. 337–340, (2008).

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao, 'The daikon system for dynamic detection of likely invariants', *Sci. Comput. Program.*, **69**(1-3), 35–45, (2007).

[9] Cormac Flanagan and Shaz Qadeer, 'Predicate abstraction for software verification', in *Conference Record of POPL-02*, pp. 191–202, (2002).

[10] Yuxiao Hu, *Generation and Verification of Plans with Loops*, Ph.D. dissertation, Department of Computer Science, University of Toronto, 2012.

[11] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi, 'Termination analysis with algorithmic learning', in *Computer Aided Verification - 24th International Conference*, pp. 88–104, (2012).

[12] 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).

[13] Naiqi Li and Yongmei Liu, 'Automatic verification of partial correctness of golog programs', in *Proc. of IJCAI-15*, pp. 3113–3119, (2015).

[14] Yongmei Liu, 'A hoare-style proof system for robot programs', in *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pp. 74–79, (2002).

[15] Richard J Nowakowski, *Games of no chance*, volume 29, Cambridge University Press, 1998.

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

[17] Saurabh Srivastava and Sumit Gulwani, 'Program verification using templates over predicate abstraction', in *PLDI*, pp. 223–234, (2009).

[18] Westley Weimer ThanhVu Nguyen, Deepak Kapur and Stephanie Forrest, 'Using dynamic analysis to discover polynomial and array invariants', in *ICSE*, pp. 683–693, (2012).

[19] Benjamin Zarrieß and Jens Claßen, 'Verifying CTL* properties of GOLOG programs over local-effect actions', in *ECAI 2014 - 21st European Conference on Artificial Intelligence*, pp. 939–944, (2014).