Trace Abstraction-based Verification for Uninterpreted Programs

Weijiang Hong[1,2], Zhenbang Chen[1(✉)], Yide Du[1], and Ji Wang[1,2(✉)]

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
{hongweijiang17,zbchen,dyd1024,wj}@nudt.edu.cn

Abstract. The verification of uninterpreted programs is undecidable in general. This paper proposes to employ counterexample-guided abstraction refinement (CEGAR) framework for verifying uninterpreted programs. Different from the existing interpolant-based trace abstraction, we propose a congruence-based trace abstraction method for infeasible counterexample paths to refine the program's abstraction model, which is designed specifically for uninterpreted programs. Besides, we propose an optimization method that utilizes the decidable verification result for coherent uninterpreted programs to improve the CEGAR framework's efficiency. We have implemented our verification method and evaluated it on two kinds of benchmark programs. Compared with the state-of-the-art, our method is more effective and efficient, and achieves 3.6x speedups on average.

Keywords: Uninterpreted programs · CEGAR · Trace abstraction.

1 Introduction

Uninterpreted programs [15] belong to a class of programs in which there are uninterpreted functions [12]. An uninterpreted function f only has a function signature (*i.e.*, function name, and the types of input and output) but no other definitions. f only satisfies the common property, *i.e.*, given the same input, f produces the same output. Uninterpreted programs are motivated in many scenarios of program analysis and verification. For example, suppose we want to verify a partial program in which some functions are not defined. We can over-approximate the program by considering the undefined functions as uninterpreted functions. Even for well-defined programs, we can carry out a pre-analysis of a function by considering all the called functions in f as uninterpreted functions. Specially, the solving of the SMT formulas in the theory of equality and uninterpreted functions is decidable and has a PSPACE complexity [11].

* Weijiang Hong and Zhenbang Chen contributed equally to this work and are co-first authors. Zhenbang Chen and Ji Wang are the corresponding authors.

However, the verification problem of uninterpreted programs is generally undecidable [15], because of the loop structures.

There exists a sub-class of uninterpreted programs (called *coherent* uninterpreted programs) whose verification problem is decidable and PSPACE-complete [15]. As far as we know, there exist no verification methods designed specifically for the general uninterpreted programs. In this paper, we propose to leverage the idea of counterexample-guided abstraction refinement (CEGAR) [6] to verify safety properties of uninterpreted programs. Although the existing CEGAR-based verification methods, such as [8] and [1], are also applicable for uninterpreted programs, we argue that more efficient CEGAR-based verification for uninterpreted programs can be achieved by employing the abstraction and refinement methods designed specifically for uninterpreted programs.

Leveraging the decidable result of coherent program verification in [15], we propose in this paper a trace abstraction [8] based CEGAR framework to verify uninterpreted programs. Different from the traditional SMT-based trace feasibility checking [5] and interpolant-based trace abstraction [8], our framework provides a new congruence-based method for abstracting the infeasible counterexample traces. The congruence-based method directly captures the core invariant features (*i.e.*, equality and congruence closure [18]) of uninterpreted programs and improves the efficiency of trace abstraction-based refinement. Besides, based on the observation that some parts (even the ones containing complex loops) of an uninterpreted program are *coherent* [15] and can be efficiently verified, we propose an optimization for the CEGAR framework that verifies the program's coherent part first and then employs the CEGAR-based procedure to verify the remaining part, which can further improve the verification's efficiency.

The main contributions of this paper are as follows:

- We propose a CEGAR-based verification framework for uninterpreted programs. The framework verifies the program's coherent part first and then employs the CEGAR-based procedure to verify the non-coherent part.
- We propose a new congruence-based trace abstraction method which utilizes the verification method of coherent programs for checking the trace feasibility and constructing the trace abstraction.
- We have implemented our CEGAR framework for Boogie uninterpreted programs. The experimental results on several benchmarks indicate that: compared with the *state-of-the-art* work [7], *i.e.*, ULTIMATE, our method achieves in average 3.6x speedup for the verified programs.

Related work. Our work is closely related to the existing work for uninterpreted programs. In [15], the decidability result of coherent uninterpreted programs is discovered, which inspires our work. Following the work [15], Mathur *et al.* extend the decidable result to the programs with memory allocations [17]. In [20], La Torre *et al.* prove that the verification of coherent concurrent programs is decidable. Krogmeier *et al.* in [13] investigate the synthesis problem of uninterpreted programs and propose a decidable synthesis procedure for coherent uninterpreted programs. Besides, Mathur *et al.* in [16] study the verification problem of uninterpreted programs under different types of data models. Different from these approaches, we consider the verification problem of general uninterpreted programs. Another line is CEGAR-based program verification methods [1,3,4,8], which differ in the target programs, abstraction models, and refinement methods. Our work for uninterpreted programs is in the style of trace abstraction-based refinement [8]. Unlike the interpolant-based trace abstraction method [3,4,8–10], our abstraction is congruence-based and provides a new mechanism for the trace abstraction of uninterpreted programs.

Structure. The remainder of this paper is organized as follows. A brief summary of the backgrounds and an illustration of our method are presented in Section 2. The verification framework's details will be presented in Section 3. The evaluation of our method and its results are presented in Section 4. Finally, Section 5 concludes the paper.

2 Background and Illustration

This section briefly introduces uninterpreted programs. Then, we use a motivation example to illustrate our trace abstraction based verification method.

2.1 Uninterpreted Programs

Syntax Let Σ be a *finite* symbolic set, and $\mathbb{C} \subseteq \Sigma$ the constant set. The syntax of uninterpreted programs is defined in Figure 1, where $c \in \mathbb{C}, x, y, f \in \Sigma$, and $\bar{\mathbf{z}}$ denotes a tuple of symbols.

$$\begin{aligned} \langle \mathit{stmt} \rangle ::= {} & \mathbf{skip} \mid x := c \mid x := y \mid x := f(\bar{\mathbf{z}}) \\ & \mid \mathbf{assume}(\langle \mathit{cond} \rangle) \mid \mathbf{assert}(\langle \mathit{cond} \rangle) \\ & \mid \langle \mathit{stmt} \rangle \mathbin{⨾} \langle \mathit{stmt} \rangle \\ & \mid \mathbf{if}\ (\langle \mathit{cond} \rangle)\ \mathbf{then}\ \langle \mathit{stmt} \rangle\ \mathbf{else}\ \langle \mathit{stmt} \rangle \\ & \mid \mathbf{while}\ (\langle \mathit{cond} \rangle)\ \langle \mathit{stmt} \rangle \end{aligned}$$

$$\langle \mathit{cond} \rangle ::= x = y \mid \langle \mathit{cond} \rangle \wedge \langle \mathit{cond} \rangle \mid \neg \langle \mathit{cond} \rangle$$

Fig. 1. The syntax of uninterpreted programs.

For the *atomic* statements, the statement **skip** does nothing and is the unit statement. $x := c$ is the constant assignment statement. $x := y$ is the assignment statement. $x := f(\bar{\mathbf{z}})$ assigns the term of the uninterpreted function f to variable x. The term accepts the tuple $\bar{\mathbf{z}}$. The assume statement $\mathbf{assume}(\langle \mathit{cond} \rangle)$ blocks the program states that does not satisfy $\langle \mathit{cond} \rangle$; otherwise, it is a **skip**. The assertion statement $\mathbf{assert}(\langle \mathit{cond} \rangle)$ requires that all the states reaching the statement should satisfy $\langle \mathit{cond} \rangle$. There are three compositive statements: sequential composition, if-then-else branch composition, and while loop composition. For conditions, there is only one atomic condition, *i.e.*, equality. It indicates that x is equal to y. Then, for the sake of brevity, we only include the conjunction and negation operations. Note that there only exist symbols in uninterpreted programs. Besides, we can also represent other kinds of relations as functions.

$$\frac{\mathcal{P} = \textbf{skip}}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S} \rangle} \qquad \frac{\mathcal{P} = x := c}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}[x \mapsto \mathcal{I}(c)] \rangle}$$

$$\frac{\mathcal{P} = x := y}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}[x \mapsto \mathcal{S}(y)] \rangle} \qquad \frac{\mathcal{P} = x := f(\bar{\mathbf{z}}) \quad (\mathcal{S}(\bar{\mathbf{z}}), v) \in \mathcal{I}(f)}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}[x \mapsto v] \rangle}$$

$$\frac{\mathcal{P} = \textbf{assume}\ (\mathcal{C}) \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \mathcal{C}}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S} \rangle} \qquad \frac{\mathcal{P} = \textbf{assume}\ (\mathcal{C}) \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \neg\mathcal{C}}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathsf{Blocked} \rangle}$$

$$\frac{\mathcal{P} = \textbf{assert}\ (\mathcal{C}) \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \mathcal{C}}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S} \rangle} \qquad \frac{\mathcal{P} = \textbf{assert}\ (\mathcal{C}) \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \neg\mathcal{C}}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathsf{Failed} \rangle}$$

$$\frac{\mathcal{P} = \mathcal{P}_1 \fatsemi \mathcal{P}_2 \quad \langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}_1} \langle \mathcal{M}, \mathcal{S}_1 \rangle \quad \langle \mathcal{M}, \mathcal{S}_1 \rangle \rightarrow_{\mathcal{P}_2} \langle \mathcal{M}, \mathcal{S}_2 \rangle}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}_2 \rangle}$$

$$\frac{\mathcal{P} = \textbf{if}\ (\mathcal{C})\ \textbf{then}\ \mathcal{P}_1\ \textbf{else}\ \mathcal{P}_2 \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \mathcal{C} \quad \langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}_1} \langle \mathcal{M}, \mathcal{S}_1 \rangle}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}_1 \rangle}$$

$$\frac{\mathcal{P} = \textbf{if}\ (\mathcal{C})\ \textbf{then}\ \mathcal{P}_1\ \textbf{else}\ \mathcal{P}_2 \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \neg\mathcal{C} \quad \langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}_2} \langle \mathcal{M}, \mathcal{S}_2 \rangle}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}_2 \rangle}$$

$$\frac{\mathcal{P} = \textbf{while}\ (\mathcal{C})\ \mathcal{P}_1 \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \neg\mathcal{C}}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S} \rangle}$$

$$\frac{\mathcal{P} = \textbf{while}\ (\mathcal{C})\ \mathcal{P}_1 \quad \langle \mathcal{M}, \mathcal{S} \rangle \models \mathcal{C} \quad \langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}_1} \langle \mathcal{M}, \mathcal{S}_1 \rangle \quad \langle \mathcal{M}, \mathcal{S}_1 \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}_2 \rangle}{\langle \mathcal{M}, \mathcal{S} \rangle \rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}_2 \rangle}$$

Fig. 2. Semantics rules for uninterpreted programs.

Semantics The semantics of an uninterpreted program is defined *w.r.t.* a data model $\mathcal{M} = (\mathcal{U}, \mathcal{I})$ for interpretation, where $\mathcal{U}$ is the universal element set for interpretation, and $\mathcal{I}$ interprets the symbols in Σ. Specifically, $\mathcal{I}$ interprets a symbol in $\mathbb{C}$ to an element in $\mathcal{U}$, and a function f as a relation of $\mathcal{U}$. Given a data model $\mathcal{M}$, the semantics of an uninterpreted program $\mathcal{P}$ is defined as a state transition graph. Each state $\mathcal{S} : \Sigma \to \mathcal{U}$ maps a symbol to an element in the universal set of the data model. The initial state is an empty map. Figure 2 gives the transition rules for the semantics, where $\mathsf{Blocked}$ and Fail are the special blocked and failed terminated states, respectively. $\mathcal{S}[x \mapsto e]$ is $\mathcal{S} \cup \{(x, e)\}$ if $x \notin dom(\mathcal{S})$; otherwise, it is defined as follows.

$$\mathcal{S}[x \mapsto e](y) = \begin{cases} e & y = x \\ \mathcal{S}(y) & otherwise \end{cases} \tag{1}$$

For the sake of brevity, we extend $\mathcal{S}$ to tuples and use $\mathcal{S}(\bar{\mathbf{z}})$ to denote the value tuple of $\bar{\mathbf{z}}$. For example, $\mathcal{S}((x, y))$ is $(\mathcal{S}(x), \mathcal{S}(y))$. Besides, we define state $\mathcal{S}$ satisfies a condition $\mathcal{C}$, *i.e.*, $\mathcal{S} \models \mathcal{C}$, as follows.

$$\begin{array}{ll} \mathcal{S} \models x = y & \textbf{iff}\ S(x) = S(y) \\ \mathcal{S} \models \mathcal{C}_1 \wedge \mathcal{C}_2 & \textbf{iff}\ \mathcal{S} \models \mathcal{C}_1 \wedge \mathcal{S} \models \mathcal{C}_2 \\ \mathcal{S} \models \neg\mathcal{C} & \textbf{iff}\ \mathcal{S} \not\models \mathcal{C} \end{array} \tag{2}$$

We use $\langle \mathcal{M}, \mathcal{S}_1 \rangle \rightsquigarrow_{\mathcal{P}} \langle \mathcal{M}, \mathcal{S}_2 \rangle$ to represent that state $\mathcal{S}_2$ can be reached from state $\mathcal{S}_1$ during the execution of $\mathcal{P}$ under the data model $\mathcal{M}$.

Verification Problem In this paper, we only consider the verification of uninterpreted programs *w.r.t.* reachability properties. There are assertions in the program. Each assertion can be specified by **assert**($\langle cond \rangle$), which requires that $\langle cond \rangle$ holds for the program in terms of any data models. We define $\mathcal{P}$ satisfies the assertions as follows.

$$\forall \mathcal{M} \bullet \langle \mathcal{M}, \emptyset \rangle \not\rightarrow_{\mathcal{P}} \langle \mathcal{M}, \mathsf{Fail} \rangle \tag{3}$$

It means that the Fail state is not reachable during the execution of $\mathcal{P}$ under *any data model.* In general, the verification problem of uninterpreted programs is undecidable [15]. Recently, a fragment of uninterpreted programs, called *coherent uninterpreted programs* [15], has been discovered, and its verification problem is decidable and has a PSPACE-complete time complexity.

Coherent Uninterpreted Programs The coherence is defined in a purely symbol-oriented manner. Each variable of the program is supposed to have an initial term. The statements in the program are interpreted as term rewritings. For example, if y's current term is t_y, $x := f(y)$ assigns term $f(t_y)$ to x. Then, along with the program's execution, the congruence closure relation [18] *w.r.t.* equality can also be inferred. The relation can be used to reason the equality or dis-equality of the program variables, which supports proving the program's assertions. Specifically, the coherent uninterpreted program requires that all of its traces satisfy the following two properties.

– **Memoizing:** whenever a term t is recomputed, there must be a variable x whose term value is equal to t *w.r.t.* the congruence closure relation.
– **Early assumes: assume**($x = y$) statement appears before the assignments of the variables whose term values are equal (also *w.r.t.* the congruence closure relation) to a super-term of x's term or y's term.

If an uninterpreted program $\mathcal{P}$ satisfies these two conditions, *i.e.*, $\mathcal{P}$ is coherent, the congruence closure relations that can be concluded from $\mathcal{P}$ are *complete*, which is the key to ensure the verification's completeness. Then, $\mathcal{P}$'s verification can be reduced to an emptiness checking problem of a finite state automata (FSA), which is the intersection of $\mathcal{P}$'s execution automata $\mathcal{A}_{\mathcal{P}}$ (sound approximation) and another FSA $\mathcal{A}_{\mathcal{U}}$ for checking feasibility. $\mathcal{A}_{\mathcal{U}}$ ensures the soundness and completeness of the feasibility checking for its coherent traces. Therefore, this verification method for coherent uninterpreted programs is sound and complete. More details can be found in [15].

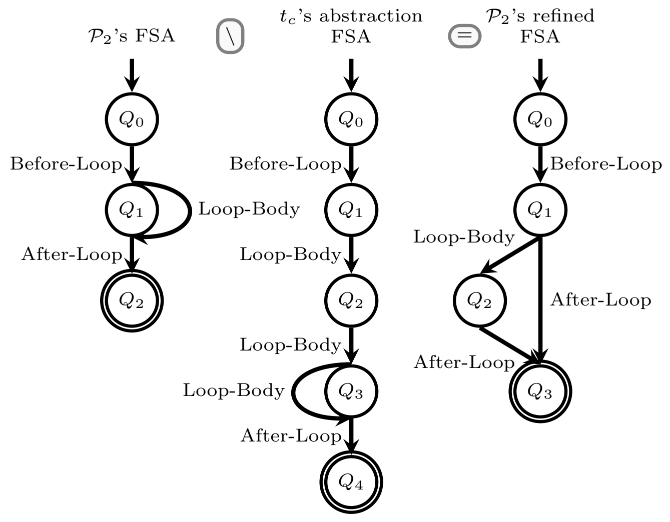
2.2 Trace Abstraction based CEGAR

Trace abstraction based CEGAR first abstracts a program $\mathcal{P}$ by an FSA $\mathcal{A}$, which is an over-approximation of $\mathcal{P}$. Then, a counter-example trace t is extracted from $\mathcal{A}$ if any counter-example traces exist. If t is feasible, a real counter-example is found; otherwise, t will be abstracted to an FSA $\mathcal{A}_c$, in which all the accepted traces are not feasible. Then, the abstraction $\mathcal{A}$ is refined to $\mathcal{A} \cap \neg\mathcal{A}_c$. If $\mathcal{A}$ contains no counter-example traces, $\mathcal{P}$ is verified to satisfy the assertions; otherwise, the trace abstraction based refinement continues.

Figure 3(a) shows an example program $\mathcal{P}$ to demonstrate our framework. For this program, the assertion in the last line is truly a fact. However, the program is not a coherent uninterpreted program. The **assume** statement in the **else** branch does not satisfy the requirement of *early assumes*, because $f(t)$ may be already computed and dropped in before while loop. Thus, we cannot use the verification method in [15] to verify this program. Besides, if we use ULTIMATE [7], *i.e.*, a state-of-the-art verification tool implementing trace abstraction based CEGAR [8], to verify the program, the refinement process does not terminate.

```
x := t;
y := k;
if (t != s){
  assume(t = k);
  while (y != z){
    x := f(x);
    y := f(y);
    z := g(z);
  }
}else{
  while (y != z){
    x := f(x);
    y := f(y);
    z := g(z);
  }
  assume(t = k);
}
assert(x = y);
```

(a) Program



(b) Refinement

Fig. 3. A motivation example, where Before-Loop, Loop-Body and After-Loop represent $x := t \mathbin{⨾} y := k \mathbin{⨾} \textbf{assume}(t = s)$, $\textbf{assume}(y \neq z) \mathbin{⨾} x := f(x) \mathbin{⨾} y := f(y) \mathbin{⨾} z := g(z)$, and $\textbf{assume}(y = z) \mathbin{⨾} \textbf{assume}(t = k) \mathbin{⨾} \textbf{assume}(x \neq y)$, respectively.

Notice that $\mathcal{P}$ can be separated into two sub-programs $\mathcal{P}_1$ and $\mathcal{P}_2$, which corresponds to the **true** and **false** branch cases, respectively. $\mathcal{P}_1$ is a coherent program, but $\mathcal{P}_2$ is not, because $\mathcal{P}_2$ violates the requirement of *early assumes*. We use $\mathcal{P}_2$ to demonstrate our CEGAR procedure. $\mathcal{P}_2$'s FSA is the first one in Figure 3(b), where each state represents the one after the transition of a sequence of statements (for the sake of brevity). Note that the statement $\textbf{assert}(x = y)$ will be replaced by $\textbf{assume}(x \neq y)$, and the verification problem *w.r.t.* $\textbf{assert}(x = y)$ will be converted into a reachability problem *w.r.t.* $\textbf{assume}(x \neq y)$. Then, suppose we get a counterexample trace t in which the loop body is executed *three* iterations. t is not a coherent trace, but any finite

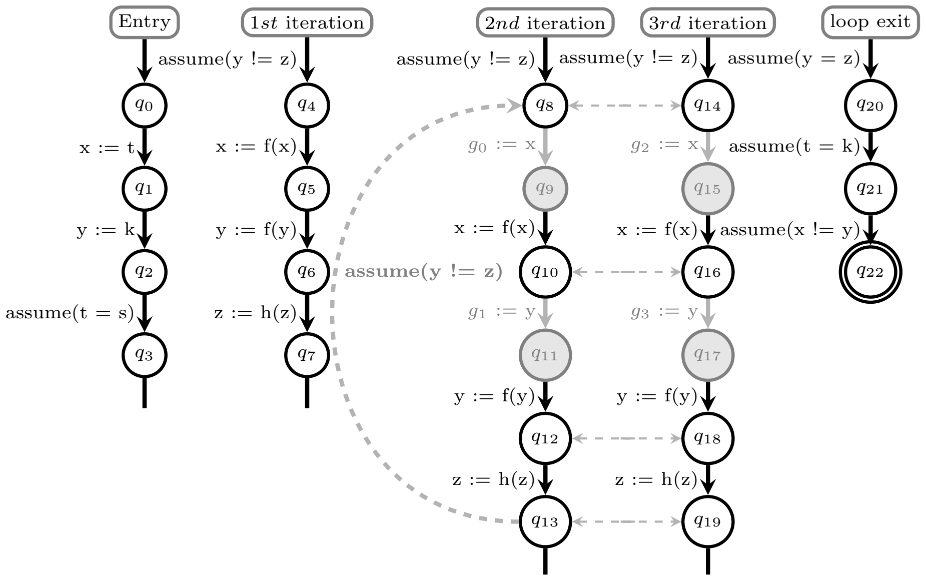


Fig. 4. The demonstration of abstracting infeasible trace.

trace is k-coherent [15]. So we translate t to a coherent trace t_c by adding k ghost variables. Figure 4 shows the trace t_c, which is divided into several segments. The gray states (q_9, q_{11}, q_{15} and q_{17}) and transitions are related to ghost variables. Then, along with t_c, we can compute the congruence closure relation based state transitions. Each state consists of three parts: an equality relation E of variables, a dis-equality relation D, and a set F of function relations. For example, after executing the first two statements, q_2's E is as follows.

$$\{(x,x),(x,t),(t,x),(y,y),(y,k),(k,y),(z,z),(t,t),(k,k),(s,s)\}$$

q_2's D is $\emptyset$ and q_2's F is $\{R_f, R_g\}$, where both R_f and R_g are the relation $\{([v]_E, \mathsf{Undef}) \mid v \in \{x,y,z,s\}\}$, in which $[v]_E$ represents v's equivalent set *w.r.t.* q_2's E, and Undef means that the input's value is undefined.

If there exist two variables that have a relation in both one state's E and D, the state is *inconsistent*. If a trace can result in an inconsistent state, the trace is *infeasible*. In total, there are 23 states along the coherent trace t_c. State q_{22} is inconsistent, because x and y are equal but also in the dis-equality relation of q_{22} (implied by the last statement **assume**$(x \neq y)$). Hence, t_c is infeasible.

So we abstract t_c to refine $\mathcal{P}_2$'s FSA. The abstraction for t_c is as follows. We first remove the transitions and the states related to ghost variables and remove the ghost variables' definitions from each state. For t_c, we remove q_9, q_{11}, q_{15} and q_{17} and their incoming transitions (*i.e.*, the gray states and transitions in Figure 4). Then, we scan the remaining states and transitions and try to match the equal states. Two states are equal if they have the same E and D. Figure 4 shows the pairs of equivalent states by $\leftarrow\text{-}\text{-}\text{-}\text{-}\rightarrow$. Because each state in the second iteration has a corresponding equivalent state in the third iteration, we can add a transition (dotted line) from q_{13} to q_8. This abstraction ensures that each trace in the abstraction is infeasible, because the core reason of inconsistency is **assume**$(t = k)$ and **assume**$(x \neq y)$. Same as t_c, any trace in the abstraction can always be transferred to a coherent trace, whose last state is inconsistent.

The abstraction's FSA is the second FSA in Figure 3(b), in which the loop body is executed two or more iterations. We refine P_2's FSA (*i.e.*, the first one in Figure 3(b)) by removing the traces in t_c's abstraction. The result is the third FSA in Figure 3(b), in which only two traces exist. These two traces are also infeasible. Therefore, P_2 is proved to satisfy the assertion by three rounds of refinement.

Optimization We can use the aforementioned CEGAR-based procedure to verify the whole program in Figure 3(a), which takes 8 rounds of refinement to complete the verification. Instead of doing that, by observing that $\mathcal{P}_1$ is a coherent program, we verify $\mathcal{P}_1$ by employing the verification method in [15] and then employ the CEGAR-based procedure to verify $\mathcal{P}_2$, which significantly reduces the refinement rounds. In contrast, the optimized verification only needs 3 rounds of refinement. Such optimization effectively improves the verification's efficiency.

3 Verification Framework

Figure 5 shows our verification framework. There are two stages in the framework. In the first stage, the framework partitions $\mathcal{P}$ into different parts. For the coherent parts, the framework verifies them by the verification method in [15]. Now, our framework only partitions each if-then-else branch statement into two parts. The loop statement is atomic and will not be partitioned.

The second stage verifies the remaining parts via CEGAR-based procedure. The framework constructs the FSA abstracting $\mathcal{A}_u$ for each sub-program $\mathcal{P}_u$. If there exists an accepted trace t, *i.e.*, a statement sequence driving from an initial state to an accepted state of $\mathcal{A}_u$, the framework employs a congruence-based method based on [15] to check t's feasibility (Section 3.1). If t is feasible, a real counterexample is found, and the framework terminates. Otherwise, the framework constructs an FSA $\mathcal{A}_t$ that abstracts the infeasible trace t (Section 3.2). In principle, all the traces in $\mathcal{A}_t$ are infeasible and equivalent with t *w.r.t.* t's core reason for infeasibility. Then, $\mathcal{A}_u$ is refined by removing the traces in $\mathcal{A}_t$, *i.e.*, $\mathcal{A}_u = \mathcal{A}_u \cap \neg\mathcal{A}_t$. If there is no accepted trace in $\mathcal{A}_u$, $\mathcal{P}_u$ is verified to satisfy the property. Otherwise, the iteration will continue from $\mathcal{A}_u$ until the verification succeeds (*i.e.*, find a real counterexample or prove all $\mathcal{P}_u$s' satisfaction to the property) or timeout.

3.1 Congruence-based Feasibility Checking

We use congruence-based term state transitions to check a coherent trace's feasibility [15]. Each term state represents the current equality, dis-equality and function relations. Each term state is formally defined as follows.

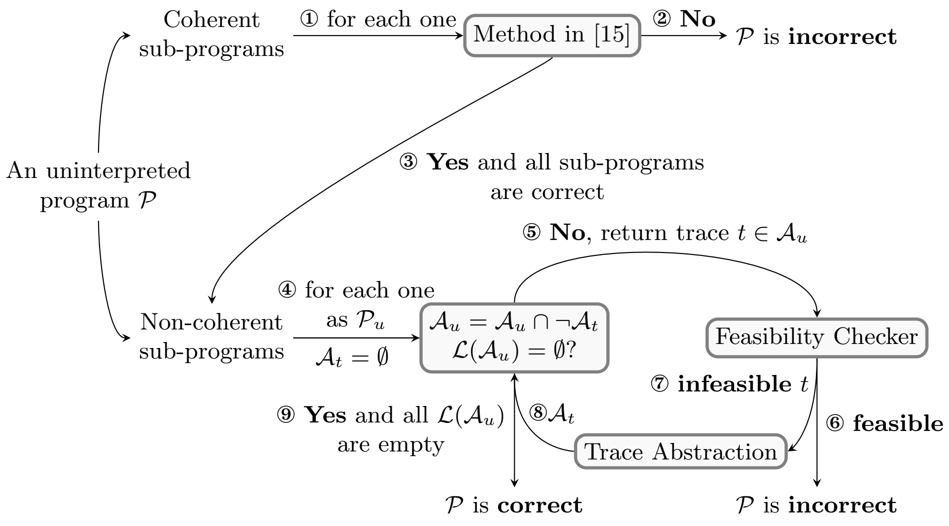

Coherent sub-programs
① for each one
Method in [15]
② No
$\mathcal{P}$ is incorrect
An uninterpreted program $\mathcal{P}$
③ Yes and all sub-programs are correct
⑤ No, return trace $t \in \mathcal{A}_u$
④ for each one as $\mathcal{P}_u$
$\mathcal{A}_t = \emptyset$
Non-coherent sub-programs
$\mathcal{A}_u = \mathcal{A}_u \cap \neg\mathcal{A}_t$
$\mathcal{L}(\mathcal{A}_u) = \emptyset$?
Feasibility Checker
⑦ infeasible t
⑥ feasible
⑨ Yes and all $\mathcal{L}(\mathcal{A}_u)$ are empty
⑧$\mathcal{A}_t$
Trace Abstraction
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect

Fig. 5. Verification framework.

Definition 1. *Given an uninterpreted program $\mathcal{P}$ and its variable set V, a term state S is defined as a triple (E, D, F), where*

- *$E \subseteq V \times V$ is the equality relation, and we use $[V]_E \subseteq 2^V$ to represent the set of V's equivalent classes w.r.t. E.*
- *$D \subseteq [V]_E \times [V]_E$ is the dis-equality relation.*
- *F is the set of function definitions, and each function $f : [V]_E \times \ldots \times [V]_E \to [V]_E$ defines a relation between equivalent variable classes w.r.t. E.*

We use $[x]_E$ to represent x's equivalent variable class *w.r.t.* E. A state $S = (E, D, F)$ is *inconsistent* when the following condition holds.

$$\exists x, y \in V \bullet (x, y) \in E \wedge ([x]_E, [y]_E) \in D \tag{4}$$

Two states S_1 and S_2 are equivalent (denoted by $S_1 \equiv S_2$) if they have the same E and D. The beginning term state S_{ini} is $(\mathsf{ID}_V, \emptyset, F_{\mathsf{Undef}})$, where ID_V is $\{(x, x) \mid x \in V\}$, *i.e.*, any variable should be equal to itself, and each function in F_{Undef} gives Undef for any inputs. Then, given a coherent trace $t = \langle stmt_1, ..., stmt_n\rangle$ of the program $\mathcal{P}$, we can derive t's state transitions starting from S_{ini} by the rules in Figure 6, wherein the following definitions are used. We use $E[x :\equiv y]$ below to denote changing x's equivalent elements to those that are equivalent with y, wherein $E\downarrow_V$ represents E's projection on the variable set V.

$$E[x :\equiv y] ::= E\downarrow_{V\setminus\{x\}} \cup \{(x, y'), (y', x) \mid (y, y') \in E\} \cup \{(x, x)\} \tag{5}$$

$D[x, E']$ below represents redefining D *w.r.t.* a new equality relation E' and removing x related dis-equality relations.

$$D[x, E'] ::= \{([x_1]_{E'}, [x_2]_{E'}) \mid \{x_1, x_2\} \subseteq V \setminus \{x\} \wedge ([x_1]_E, [x_2]_E) \in D\} \tag{6}$$

$$\frac{\mathit{Stmt} = \mathbf{skip}}{(E, D, F) \rightarrow_{\mathit{Stmt}} (E, D, F)}$$

$$\frac{\mathit{Stmt} = x := y \wedge E' = E[x :\equiv y]}{(E, D, F) \rightarrow_{\mathit{Stmt}} (E', D[x, E'], F[x, E'])}$$

$$\frac{\mathit{Stmt} = x := f(z) \wedge F[\![f]\!]([z]_E) = [y]_E \wedge E' = E[x :\equiv y]}{(E, D, F) \rightarrow_{\mathit{Stmt}} (E', D[x, E'], F[x, E'])}$$

$$\frac{\mathit{Stmt} = x := f(z) \wedge F[\![f]\!]([z]_E) = \mathsf{Undef} \wedge E' = E \downarrow_{V \setminus \{x\}} \cup \{(x, x)\}}{(E, D, F) \rightarrow_{\mathit{Stmt}} (E', D[x, E'], F[x, E', f(z)])}$$

$$\frac{\begin{array}{c}\mathit{Stmt} = \mathbf{assume}(x = y) \wedge E' = \mathsf{LCC}(E, F, x, y) \wedge \\ D' = \{([x_1]_{E'}, [x_2]_{E'}) \mid ([x_1]_E, [x_2]_E) \in D\}\end{array}}{(E, D, F) \rightarrow_{\mathit{Stmt}} (E', D', F[E'])}$$

$$\frac{\mathit{Stmt} = \mathbf{assume}(x \neq y)}{(E, D, F) \rightarrow_{\mathit{Stmt}} (E, D \cup \{([x]_E, [y]_E), ([y]_E, [x]_E)\}, F)}$$

Fig. 6. Transition rules for term states, where $\mathsf{LCC}(E, F, x, y)$ represents the congruence closure of E after adding the equality between x and y to E. Note that, $x := c$ can be replaced by $x := x_c$ where x_c is a never-used variable and its initial term is c. $\mathbf{assert}(\langle \mathit{cond} \rangle)$ has been replaced by $\mathbf{assume}(\neg \langle \mathit{cond} \rangle)$ during verification.

We use $F[\![f]\!]$ to represent the function f in F and $F[E']$ defined as follows to represent redefining the functions in F *w.r.t.* the equality relation E'.

$$F[E'][\![f]\!]([x_1]_{E'}, \ldots, [x_n]_{E'}) ::= \begin{cases} [u]_{E'} & [u]_E = F[\![f]\!]([x_1]_E, \ldots, [x_n]_E) \\ \mathsf{Undef} & \mathsf{otherwise} \end{cases} \tag{7}$$

Then, based on $F[E']$, we define $F[x, E']$ to represent redefining the functions in F *w.r.t.* E' when x is assigned with another variable or constant. The function relations defined on x are modified to Undef.

$$F[x, E'][\![f]\!]([x_1]_{E'}, \ldots, [x_n]_{E'}) ::= \begin{cases} F[E'][\![f]\!]([x_1]_{E'}, \ldots, [x_n]_{E'}) & x \notin \{u, x_1, \ldots, x_n\} \\ \mathsf{Undef} & \mathsf{otherwise} \end{cases} \tag{8}$$

Besides, if x is assigned with an uninterpreted function expression $f(z)$ and z's value defined by f is Undef, the functions in F are redefined as follows.

$$F[x, E', f(z)][\![g]\!]([y]_{E'}) ::= \begin{cases} F[x, E'][\![g]\!]([y]_{E'}) & g \neq f \\ [x]_{E'} & g = f \wedge x \notin \{y\} \wedge y \in [z]_E \\ [u]_{E'} & g = f \wedge x \notin \{u, y\} \wedge \\ & [u]_E = F[\![f]\!]([y]_E) \\ \mathsf{Undef} & \mathsf{otherwise} \end{cases} \tag{9}$$

For the functions other than f, the definitions are the same as those in $F[x, E']$; otherwise, the values of z's equivalent variables (not including x) are $[x]_{E'}$, and f's relations defined on x are modified to Undef.

We use $S_1 \rightsquigarrow_t S_2$ to represent that S_2 can be reached after executing the trace t starting from S_1. Then, a trace t is *infeasible* (denoted as $\mathsf{infeasible}(t)$)

if an *inconsistent* state S_{inc} can be reached from the beginning state S_{ini}, *i.e.*, where $t_1 \preceq t$ represents that t_1 is a prefix of t.

$$\exists t_1 \bullet t_1 \preceq t \wedge S_{ini} \rightsquigarrow_{t_1} S_{inc} \tag{10}$$

Example Suppose that the current term state (E, D, F) is as follows, where $V = \{x, y, z\}$ and there is only one uninterpreted function f.

$$E = \{(x,x),(y,y),(z,z),(y,z),(z,y)\} \quad D = \{([x]_E,[y]_E)\}$$

$$F[\![f]\!] = \{f([x]_E) = [y]_E, f([y]_E) = \mathsf{Undef}, f([z]_E) = \mathsf{Undef}\}$$

Then, after executing $x := f(z)$, we can obtain the next term state (E', D', F') (denoted as S'), *i.e.*, $(E, D, F) \rightarrow_{x:=f(z)} (E', D', F')$, which is as follows according to the forth rule in Figure 6.

$$E' = E\downarrow_{V\setminus\{x\}} \cup\{(x,x)\} = \{(y,y),(z,z),(y,z),(z,y),(x,x)\}$$

$$D' = D[x,E'] = \{([x_1]_{E'},[x_2]_{E'}) \mid \{x_1,x_2\} \subseteq \{y,z\} \wedge ([x_1]_E,[x_2]_E) \in D\} = \{\}$$

$$F' = F[x,E',f(z)]$$

$F[x, E', f(z)]$ defines f as follows.

$$\begin{array}{ll} F[x,E',f(z)][\![f]\!]([x]_{E'}) = \mathsf{Undef} & \text{because } x \in \{x\} \\ F[x,E',f(z)][\![f]\!]([y]_{E'}) = [x]_{E'} & \text{because } x \notin \{y\} \wedge y \in [z]_E \\ F[x,E',f(z)][\![f]\!]([z]_{E'}) = [x]_{E'} & \text{because } x \notin \{z\} \wedge z \in [z]_E \end{array}$$

3.2 Trace Abstraction

Based on an infeasible coherent trace t_c, we can abstract t_c to an FSA, which represents the equivalent infeasible traces of t_c. Algorithm 1 shows the details of trace abstraction. The inputs are an infeasible trace t_c and the program $\mathcal{P}$, and the output is the abstraction FSA $\mathcal{A}_t$. The algorithm removes the useless states and transitions and generalizes the loop bodies with the same state transitions.

First, the algorithm constructs an initial automata to accept the infeasible coherent trace t_c (Lines 2−5). Especially, the statements after the inconsistent state S_m make self loops on S_m (Line 4). Then, it recognizes those *ghost* variables P_v in $\mathcal{P}$ and t_c, including $\mathcal{P}$'s observing variables [14] and the variables added to t_c for making t_c coherent (Line 7). The algorithm removes each state's relations that are related to P_v (Line 8). After that, the states brought by ghost variables are removed (Line 9), like the states (q_9, q_{11}, q_{15} and q_{17}) in Figure 4. The transitions of the remaining states that are connected by those deleted states are established (Line 11). So far, the algorithm removes all the useless states.

Next, the algorithm matches the states that are the entry state of the same loop body (denoted by $\mathsf{LES}(S_i, S_j)$) and have the same state transitions in the loop body, which is defined as follows.

$$\begin{aligned} \mathsf{match}(S_i,S_j) ::= {} & \mathsf{LES}(S_i,S_j)\wedge \\ & \exists k \bullet \forall 0 \le m \le k \bullet S_{i+m} \equiv S_{j+m} \wedge \\ & \quad \forall 1 \le n \le k \bullet \mathit{stmt}_{i+n} = \mathit{stmt}_{j+n} \end{aligned} \tag{11}$$

Algorithm 1: Generalize(t_c, $\mathcal{P}$)

```
Input: An infeasible coherent trace $t_c = \langle stmt_1, ..., stmt_n \rangle$ and $t_c$'s transitions are
    $S_0 \rightarrow_{stmt_1} S_1 ... \rightarrow_{stmt_m} S_m$, where $S_0$ is $S_{ini}$, $S_m$ is inconsistent, and $m \leq n$.
Output: The generalization automata $\mathcal{A}_t = (\Sigma, S, T, S_I, S_F)$
 1: // Automata initialization
 2: $\Sigma \leftarrow \{stmt_1, ..., stmt_m, ..., stmt_n\}$
 3: $S \leftarrow \{S_0, S_1, ..., S_m\}$
 4: $T \leftarrow \{(S_i, stmt_{i+1}, S_{i+1}) \mid 0 \leq i \leq m-1\} \cup \{(S_m, stmt_{j+1}, S_m) \mid m \leq j \leq n-1\}$
 5: $S_I, S_F \leftarrow S_0, S_m$
 6: // Ghost elimination
 7: $P_v \leftarrow \mathsf{ghostVariables}(\mathcal{P}, t_c)$
 8: $S \leftarrow \{S_i \downarrow_{V \setminus P_v} \mid 0 \leq i \leq m\}$      //$S_i \downarrow_V$ represents $S_i$'s projection on $V$
 9: $S \leftarrow S \setminus \{S_i \mid S_i \in S \wedge \exists S_j \in S \bullet S_j \equiv S_i \wedge j < i\}$
10: // Connect these states interrupted by the states brought by ghost variables
11: $T \leftarrow T \cup \{(S_i, stmt_j, S_j) \mid S_i \in S \wedge j = \underset{k>i \wedge S_k \in S}{\arg\min}\ k\}$
12: // Loop construction
13: for $\mathcal{E} \in [S]_{\mathsf{match}}$ do
14:     $i, j \leftarrow \underset{S_k \in \mathcal{E}}{\arg\min}\, k,\ \underset{k>i \wedge S_k \in \mathcal{E}}{\arg\min}\ k$ //Get the first and second entry states in $\mathcal{E}$
15:     $\mathcal{C} \leftarrow \mathsf{getCondition}(stmt_{i+1})$          // Get the $S_i$'s loop condition $\mathcal{C}$
16:     $e \leftarrow \underset{k>i \wedge S_{k-1} \rightarrow_{\mathbf{assume}(\neg\mathcal{C})} S_k}{\arg\min}\ k$     // Get the state $S_e$ after exiting the loop
17:     $T \leftarrow T \setminus \{(S_f, stmt, S_t) \mid S_t = S_j\}$      // Remove the transitions to $S_j$
18:     $T \leftarrow T \setminus \{(S_f, stmt, S_t) \mid S_t = S_e\}$      // Remove the transitions to $S_e$
19:     $T \leftarrow T \cup \{(S_{j-1}, \mathbf{assume}(\mathcal{C}), S_i)\}$      // Generalize by adding a loop
20:     $T \leftarrow T \cup \{(S_{j-1}, \mathbf{assume}(\neg\mathcal{C}), S_e)\}$     // Add the edge for exiting the loop
21: end for
22: $\mathcal{A}_t \leftarrow (\Sigma, S, T, S_I, S_F)$ // $\mathcal{A}_t$'s unreachable states and transitions are removed
23: return $\mathcal{A}_t$
```

For example, $\mathsf{match}(S_8, S_{14})$ holds in Figure 4. If $\mathsf{match}(S_i, S_j)$ and $\mathsf{match}(S_j, S_k)$ hold, then $\{S_i, S_j, S_k\}$ constitutes an equivalent state class with respect to match.

After that, for each equivalent state class in $[S]_{\mathsf{match}}$, the algorithm keeps just one loop body (*i.e.*, the first one) and generalizes the trace by introducing a loop in the FSA (Lines 14−20). The algorithm get the first and second entry states (S_i, S_j) in each equivalent state class and the corresponding state S_e after exiting the loop (Lines 14−16). Based on these information, the algorithm constructs a loop. For example, in Figure 4, (S_i, S_j, S_e) can be (S_8, S_{14}, S_{20}). This loop construction advanced by cutting and adding the corresponding edges and states (Lines 17−20). We do this for each equivalent state class to complete the generalization of the loop bodies.

Finally, the algorithm removes the unreachable states and transitions in the FSA (Line 22) (omitted for the sake of space). The following theorem ensures the correctness of the trace abstraction algorithm.

Theorem 1 (Soundness). *Given an infeasible coherent trace t_c of the uninterpreted program $\mathcal{P}$, each accepted trace of* Generalize($t_c, \mathcal{P}$) *is also infeasible.*

Proof. The key point of proof is that each generalization step in Algorithm 1 does not introduce any feasible accepted traces. We prove by contradiction.

- *Automata initialization:* For Lines 2∼5 in Algorithm 1, the traces accepted by automata are in the form of $\langle stmt_1, ..., stmt_m, ...\rangle$. If there exists a feasible trace $t = \langle stmt_1, ..., stmt_m, ..., stmt_l\rangle$, then it requires that all the states reached by the trace should be consistent, which is not true since t will enter the inconsistent state S_m after executing $\langle stmt_1, ..., stmt_m\rangle$. Therefore, *Automata initialization* does not introduce accepted traces.
- *Ghost elimination:* For Lines 7∼11 in Algorithm 1, the only difference between the accepted trace t before this step and the accepted trace t' after this step is that t' does not contain the statements related to these ghost variables. If there exists a t' that is feasible, the corresponding t must be feasible since the ghost variables only observe the program states and do not change the trace's feasibility. However, t obtained by *Automata initialization* should be an infeasible trace, which results in a conflict. Therefore, *Ghost elimination* does not introduce accepted traces.
- *Loop construction*: Similar to ghost elimination, the difference between the accepted trace t before loop construction and the accepted trace t' after this step is that t' may have different copies of loop bodies. According to the match's definition, the executions of these loop bodies do not change the state that exits the loop, which implies that t and t' have the same feasibility. If t' is feasible, the corresponding t must be feasible. However, t obtained by *Ghost elimination* should be an infeasible trace, which results in a conflict. Therefore, *Loop construction* does not introduce any accepted traces either.

In total, Generalize$(t_c, \mathcal{P})$ does not introduce any feasible accepted traces.

4 Evaluation

We have implemented our method as a prototype[3] in Python. Our prototype supports the uninterpreted programs in Boogie language [2]. We evaluate our method's effectiveness and efficiency by applying the prototype on verifying *general* uninterpreted programs. Since there is no standard benchmark of uninterpreted programs, we designed a program generator to generate Boogie uninterpreted programs automatically. The generator composes the component programs randomly by branch operator. The types of the component programs are as follows: the ones satisfying *memoizing* or not, the ones satisfying *early assumes* or not, the ones containing *if-else* or not, and the ones containing *while* or not. The generator covers the representative cases of uninterpreted programs.

We compare our prototype with the state-of-the-art tool ULTIMATE [7] on the benchmark programs in terms of effectiveness and efficiency. We use 10 minutes as the time threshold for each verification task. All the experiments were carried out on a machine with eight cores and 8G memory, and the operating system is Ubuntu 18.04.

[3] Our implementation and benchmark are available at the GitHub repository `https://github.com/Verifier4UP/Trace-Refinement-based-Verification`

Table 1. The experimental results, where TO stands for timeout and the grey cell means that the corresponding tool performs better. The first column lists the benchmark programs and the second column shows the Lines of Code (LoC) of each program. The third and fourth columns show the results of ULTIMATE. The columns between fifth and ninth show the results of our method. The columns named **Result** show the verification results. The columns named **Time** show the time of verification. The column **Partition** shows the time for partition in our method. The column **C**(#) shows the time of verifying coherent sub-programs and the number of coherent sub-programs. The column **NC**(#) shows the time of verifying non-coherent sub-programs and the number of non-coherent sub-programs.

Program	**LoC**	**ULTIMATE**		**Our Method**				
		Result	**Time**	**Result**	**Time**	**Partition**	**C**(#)	**NC**(#)
benchmark0	41	incorrect	1.816	incorrect	1.424	0.175	0.04(2)	1.209(10)
benchmark1	43	correct	1.958	correct	1.162	0.137	0.086(6)	0.938(12)
benchmark2	54	TO	TO	correct	33.647	0.612	0.245(12)	32.79(15)
benchmark3	49	TO	TO	correct	10.101	0.248	0.115(8)	9.738(10)
benchmark4	41	incorrect	1.606	incorrect	0.212	0.178	0.034(3)	0(9)
benchmark5	46	TO	TO	correct	3.010	0.138	0.076(6)	2.796(12)
benchmark6	42	TO	TO	correct	2.011	0.176	0.038(4)	1.797(8)
benchmark7	52	TO	TO	correct	15.395	0.387	0.327(6)	14.682(12)
benchmark8	44	incorrect	1.593	incorrect	0.091	0.077	0.014(3)	0(9)
benchmark9	40	incorrect	1.582	incorrect	0.066	0.045	0.021(4)	0(4)
benchmark10	46	incorrect	1.796	incorrect	0.200	0.186	0.014(5)	0(13)
benchmark11	54	TO	TO	correct	9.522	0.605	0.608(15)	8.308(12)
benchmark12	37	incorrect	1.589	incorrect	0.089	0.076	0.013(3)	0(5)
benchmark13	39	incorrect	1.559	incorrect	0.115	0.101	0.014(2)	0(6)
benchmark14	50	correct	46.887	correct	254.083	214.727	39.356(8)	0(0)
benchmark15	41	incorrect	1.587	incorrect	0.107	0.094	0.013(6)	0(6)
benchmark16	40	correct	95.587	correct	0.842	0.19	0.219(7)	0.433(1)
benchmark17	47	incorrect	1.562	incorrect	9.650	0.105	0.034(3)	9.51(9)
benchmark18	46	incorrect	1.585	incorrect	73.801	60.686	13.054(4)	0.061(4)
benchmark19	49	correct	1.909	correct	186.273	172.073	14.2(12)	0(0)
benchmark20	37	incorrect	1.614	incorrect	1.139	0.07	0.054(4)	1.014(4)
benchmark21	39	incorrect	1.645	incorrect	1.404	0.05	0.043(3)	1.311(5)
benchmark22	48	incorrect	1.613	incorrect	0.165	0.143	0.022(6)	0(2)
benchmark23	54	TO	TO	correct	3.191	0.325	0.212(12)	2.655(15)
benchmark24	41	incorrect	1.592	incorrect	0.211	0.199	0.011(4)	0(8)
benchmark25	44	incorrect	1.56	incorrect	9.361	0.167	0(0)	9.193(12)
benchmark26	47	TO	TO	correct	1.723	0.202	0.179(10)	1.342(8)
benchmark27	48	incorrect	1.627	incorrect	0.231	0.198	0.033(5)	0(13)
benchmark28	45	incorrect	1.674	incorrect	0.298	0.206	0.092(7)	0(5)
benchmark29	45	TO	TO	correct	1.843	0.073	0.065(6)	1.705(6)
benchmark30	57	correct	1.748	correct	31.438	26.878	2.906(10)	1.654(8)
benchmark31	39	incorrect	1.589	incorrect	0.071	0.045	0.026(2)	0(6)
benchmark32	41	incorrect	1.588	incorrect	3.836	0.218	0.036(2)	3.583(10)
benchmark33	46	incorrect	1.556	incorrect	0.152	0.139	0.013(3)	0(9)
benchmark34	55	TO	TO	correct	9.955	0.456	0.091(6)	9.408(21)
benchmark35	51	incorrect	1.606	incorrect	60.027	41.231	18.776(9)	0.021(3)
benchmark36	43	incorrect	1.619	incorrect	0.121	0.107	0.014(6)	0(6)
benchmark37	41	incorrect	1.598	incorrect	1.466	0.118	0.103(6)	1.244(6)
benchmark38	49	TO	TO	correct	2.138	0.203	0.11(6)	1.826(12)
benchmark39	47	TO	TO	correct	3.380	0.127	0.086(6)	3.167(12)
benchmark40	42	TO	TO	correct	1.662	0.129	0.067(5)	1.465(7)
benchmark41	41	incorrect	1.586	incorrect	0.281	0.242	0.039(8)	0(4)
benchmark42	22	correct	1.645	correct	0.023	0.007	0.005(2)	0.011(1)
benchmark43	46	incorrect	1.626	incorrect	33.946	32.568	1.379(8)	0(0)
benchmark44	40	correct	1.914	correct	2.064	0.087	0.066(2)	1.911(6)
benchmark45	53	TO	TO	correct	18.148	0.379	0.083(5)	17.686(13)
benchmark46	28	TO	TO	correct	0.185	0.029	0.031(2)	0.124(1)
benchmark47	46	correct	1.806	correct	1.736	0.149	0.146(9)	1.441(9)
benchmark48	53	correct	1.899	correct	27.209	23.601	3.608(12)	0(0)
benchmark49	44	TO	TO	correct	10.421	0.176	0.156(4)	10.089(4)

4.1 Effectiveness & Efficiency

Table 1 shows the experimental results. In the total 50 programs (half correct and half incorrect), our tool completes the verification tasks of 50 programs (100%), while ULTIMATE completes the verification of 34 programs (68%), which indicates the effectiveness of our method. For the 34 programs verified by both our method and ULTIMATE, our method performed better on 23 programs (67.6%). On average, our method achieves 3.6x speedups for the verified programs, indicating that our method is efficient. The speedup calculation is as follows, where $T(\texttt{Ours})$ and $T(\texttt{ULTIMATE})$ are the verification time of our prototype and ULTIMATE, respectively.

$$speedup = \begin{cases} \frac{T(\texttt{ULTIMATE})}{T(\texttt{Ours})}, & T(\texttt{ULTIMATE}) > T(\texttt{Ours}) \\ 0, & T(\texttt{ULTIMATE}) = T(\texttt{Ours}) \\ -\frac{T(\texttt{Ours})}{T(\texttt{ULTIMATE})}, & T(\texttt{ULTIMATE}) < T(\texttt{Ours}) \end{cases} \tag{12}$$

For the programs on which ULTIMATE performs better than us (*e.g.*, benchmark14), our method usually spends much time on the coherence checking in the program partition, whose complexity is the same as verifying coherent programs.

4.2 The Results of Different Trace Abstraction Methods

The CEGAR module in our framework plays an important role in tackling with non-coherent programs. To further inspect our trace abstraction method's effectiveness, we compare the verification performance under three different configurations: partition with congruence-based CEGAR module, partition with interpolant-based CEGAR module (ULTIMATE), and ULTIMATE. We only conduct this experiment on the 25 correct programs in the above benchmark. The reason is that for the incorrect programs, the running under different configurations might terminate early by finding different true counterexamples.

As shown in Figure 7, our method, that is, partition with congruence-based CEGAR module, performs better on 17 programs while the other two perform better on 2 and 6 programs, respectively. This result indicates that for small programs (the sub-programs after the partition), the congruence-based trace abstraction is more efficient than the interpolant-based trace abstraction. Besides, the program partition and the congruence-based trace abstraction are both necessary.

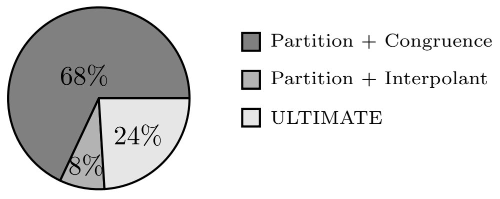


Fig. 7. The influence of different configurations. The percentage indicates the ratio of programs in which the corresponding configurations performs better.

4.3 The Results on SV-COMP Benchmark

To further evaluate the performance of the congruence-based verification and the trace abstraction in our method, we selected 46 benchmark C programs from the *loops* category of SV-COMP[4] [19] and manually transformed them into Boogie programs. In these C programs, there are only equality and dis-equality constraints, and the expressions can be modeled as uninterpreted functions. Besides, these programs can no longer be partitioned into sub-programs, which means that the whole program is either coherent or non-coherent. We compare our prototype and ULTIMATE on these benchmark programs.

Figure 8 shows the results, where the x-axis displays the indexes of the programs, and the y-axis shows the speedup value (calculated by Equation 12) of the verification time. In the total 46 programs, our method performs better on 44 programs (95.7%) than ULTIMATE. On average, our method achieves 1.90x speedups. Besides, for the 29 coherent programs, our method achieves 2.13x speedups on average; for the remaining 17 non-coherent programs, our method achieves 1.52x speedups on average. These results indicate that the congruence-based verification method and the congruence-based trace abstraction method are effective and efficient on the benchmark.

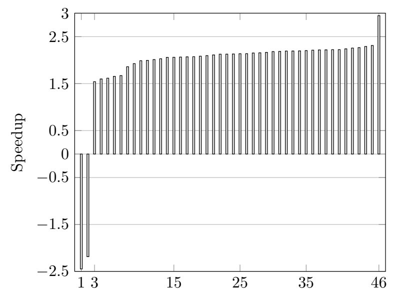


Fig. 8. The evaluation results on SV-COMP benchmark.

5 Conclusion and Future work

This paper applies a CEGAR framework for verifying uninterpreted programs. With the help of program partition, we partition an uninterpreted program into coherent and non-coherent sub-programs. Then, for the coherent sub-programs, we verify them by the method in [15]. For the remaining sub-programs, we propose a congruence-based trace abstraction method to carry out the CEGAR loop. We have implemented our method, and the experimental results indicate that our method is more effective and efficient than the state-of-the-art.

The future work lies in several directions: (1) extend our verification framework to other uninterpreted programs, such as the ones in [17] that have dynamic memory allocations; (2) enhance the trace abstraction method further, *e.g.*, by automated synthesized loop invariants; (3) more extensive evaluation on more uninterpreted programs.

Acknowledgments. This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and the NSFC Programs (No. 62172429, 61632015, 62032024, and 61690203).

[4] SV-COMP is one of the most popular benchmarks for software verification.

References

1. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. pp. 203–213. ACM (2001)
2. Boogie Language: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/
3. Cassez, F., Jensen, P.G., Larsen, K.G.: Verification and parameter synthesis for real-time programs using refinement of trace abstraction. Fundam. Informaticae **178**(1-2), 31–57 (2021)
4. Cassez, F., Ziegler, F.: Verification of concurrent programs using trace abstraction refinement. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 233–248. Springer (2015)
5. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A.F., Parker, D. (eds.) Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7385, pp. 248–254. Springer (2012)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000)
7. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol - (competition contribution). In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 641–643. Springer (2013)
8. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009)
9. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 471–482. ACM (2010)
10. Hoder, K., Kovács, L., Voronkov, A.: Playing in the grey area of proofs. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 259–272. ACM (2012)
11. Kozen, D.: Automata and computability. Undergraduate texts in computer science, Springer (1997)
12. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer (2016)

13. Krogmeier, P., Mathur, U., Murali, A., Madhusudan, P., Viswanathan, M.: Decidable synthesis of programs with uninterpreted functions. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 634–657. Springer (2020)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
15. Mathur, U., Madhusudan, P., Viswanathan, M.: Decidable verification of uninterpreted programs. Proc. ACM Program. Lang. **3**(POPL), 46:1–46:29 (2019)
16. Mathur, U., Madhusudan, P., Viswanathan, M.: What's decidable about program verification modulo axioms? In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 158–177. Springer (2020)
17. Mathur, U., Murali, A., Krogmeier, P., Madhusudan, P., Viswanathan, M.: Deciding memory safety for single-pass heap-manipulating programs. Proc. ACM Program. Lang. **4**(POPL), 35:1–35:29 (2020)
18. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
19. SV-benchmarks: https://github.com/sosy-lab/sv-benchmarks
20. Torre, S.L., Madhusudan, P.: Reachability in concurrent uninterpreted programs. In: Chattopadhyay, A., Gastin, P. (eds.) 39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2019, December 11-13, 2019, Bombay, India. LIPIcs, vol. 150, pp. 46:1–46:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)