Verifying Numerical Programs via Iterative Abstract Testing

Banghu Yin[1,2], Liqian Chen[1], Jiangchao Liu[1], Ji Wang[1,2] and Patrick Cousot[3]

[1]College of Computer, National University of Defense Technology, China
[2]State Key Laboratory of High Performance Computing, China
[3]New York University, USA

Abstract. When applying abstract interpretation to verification, it may suffer from the problem of getting too conservative over-approximations to verify a given target property, and being hardly able to generate counter-examples when the property does not hold. In this paper, we propose iterative abstract testing, to create a property-oriented verification approach based on abstract interpretation. Abstract testing employs forward abstract executions (i.e., forward analysis) together with property checking to mimic (regular) testing, and utilizes backward abstract executions (i.e., backward analysis) to derive necessary preconditions that may falsify the target property, and be useful for reducing the input space that needs further exploration. To verify a property, we conduct abstract testing in an iterative manner by utilizing dynamic partitioning to split the input space into sub-spaces such that each sub-space involves fewer program behaviors and may be easier to verify. Moreover, we leverage bounded exhaustive testing to verify bounded small sub-spaces, as a means to complement abstract testing based verification. The experimental results show that our approach has comparable strength with several state-of-the-art verification tools.

Keywords: Program verification · Abstract interpretation · Abstract Testing · Input space partitioning

1 Introduction

Abstract interpretation [18] has been successfully applied to static analysis, due to its soundness guarantee and scalability. It can automatically handle loops generally in a terminate and sound way, compared to other approaches such as bounded model checking and symbolic execution. And it allows the use of infinite abstract domains of program properties. In this paper, we focus on applying abstract interpretation to verify properties in numerical programs.

However, in the context of verification, there still exist several limitations over current abstract interpretation based approaches [25]. One limitation is that the generated invariants may be not precise enough to prove the target property, due to the too conservative over-approximation of the concrete semantics of the program. More clearly, the major sources of imprecision of abstract

interpretation come from the following aspects: (1) Most widely used abstract domains (such as intervals [18], octagons [41] and polyhedra [22]) have limitations in expressing disjunctive or non-linear properties, which are common in programs, for instance, at the joins of control-flows; (2) The widening operator in abstract interpretation which ensures the convergence of fixpoint iteration may bring severe precision loss, because widening often aggressively weakens unstable predicates in each iteration. Moreover, when one pass of the analysis does not provide precise enough invariants to prove the target property, it lacks a systematic approach (like counterexample-guided refinement) to refine the abstractions. Another limitation of most current abstract interpretation based approaches is that it can hardly generate counter-examples when a property does not hold.

In this paper, we propose an iterative approach to verify properties of numerical programs by exploiting iterative abstract testing based on abstract interpretation. We leverage the notion of "abstract testing" [19] to denote the process of abstract execution (i.e., forward analysis) of a program with an given abstract input (which represents a set of concrete inputs) and the checking of whether the abstract output satisfies the target property. The key idea of our approach is to perform abstract testing iteratively in a top-down manner, by refining the abstract input via partitioning, wherein the refinement process also makes use of the computed necessary precondition of violating the target assertion via the "inverse" abstract execution (i.e., backward analysis [21]). When the property has been checked to hold for all abstract sub-inputs in the partition, the iterative process stops and gives a proof. Another benefit of partitioning is that it enables to conduct verification via bounded exhaustive testing [11, 38] over an abstract sub-input of small size. The use of bounded exhaustive testing allows our approach to generate counter-examples when the target property does not hold. Overall, our approach not only can give a proof when the property holds, but also can supply a concrete counter-example when the property does not hold.

This paper makes the following contributions.

- We propose an iterative abstract testing based program verification algorithm with dynamic input partitioning. The partitioning enables our analysis to focus on smaller input spaces in each of which the program may involve fewer disjunctive or non-linear behaviors and thus may be easier to verify.
- We propose to use bounded exhaustive testing to complement abstract testing based verification. When the considered input space after partitioning is of small enough size, we could utilize bounded exhaustive testing to replace abstract testing. Bounded exhaustive testing can completely verify the target program even when it involves very complicated behaviors (which may be out of the verification capability of abstract interpretation), and can supply a concrete counterexample when the property does not hold.
- We have implemented the proposed approach in a tool called VATer and conducted experimental comparison between VATer and other available state-of-the-art verification tools. Our experiments on a set of verification benchmarks show that our approach is promising.

The rest of this paper is organized as follow: Section 2 gives an overview of our approach. Section 3 presents the main approach based on iterative abstract testing. Section 4 presents how to utilize bounded exhaustive testing. Section 5 provides the experimental results on the benchmarks. Section 6 discusses related work. Finally, the conclusions as well as future work are given in Section 7.

2 Overview

2.1 The framework

First, we give an overview of our verification framework, namely Verification based on Abstract Testing (VAT), as shown in Fig 1. Overall, given a numerical program P (in which each variable is of machine-bounded numerical type) and a property represented as an assertion ψ, VAT gives a proof when the assertion holds or generates concrete counter-examples when the assertion does not hold. In detail, VAT involves the iteration of the following phases.

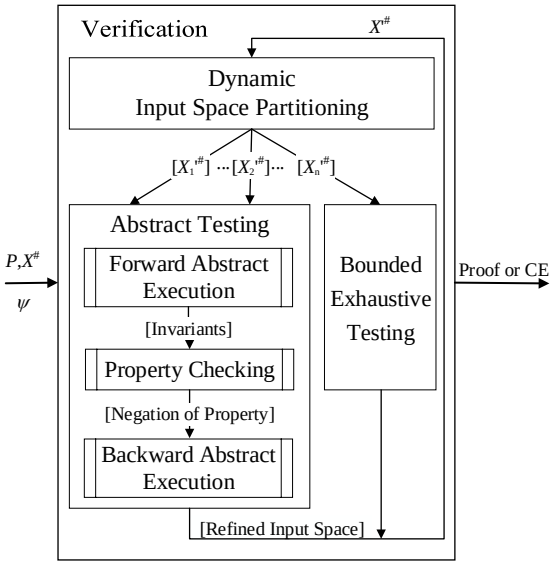

Fig. 1. The main framework of our approach.

Forward abstract execution. It acts as the abstract execution engine of abstract testing, which takes (one pass) forward abstract interpretation to generate program invariants under the given abstract input $X^{\sharp}$. Forward abstract execution "executes" a program in the sense of an abstract semantics instead of a concrete one, thus has the ability to consider several (possibly unbounded) concrete executions at a time.

Property checking. It mimics the oracle of abstract testing. It checks the logic relation between the invariants generated by forward abstract execution and the

assertion ψ. In concrete testing, the verifying result can be only "True" or "False" for each test case. However, in abstract testing, three possible verifying results may be returned: "True","False" or "Unknown". If the result is "Unknown", we need further exploration on $X^\sharp$.

Backward abstract execution. It performs a backward analysis based on abstract interpretation for $X^\sharp$ that needs further exploration, starting from the assertion location and assuming that the negation of the target assertion holds. It essentially computes the necessary precondition for the failure of the target assertion, which results in a refined abstract input $X'^\sharp \subseteq X^\sharp$ at the program entry. If $X'^\sharp$ is the empty set, it means that the assertion holds true for the abstract input $X^\sharp$. Otherwise, we need continue to explore $X'^\sharp$.

Input space partitioning. To further explore the abstract input case $X'^\sharp$, VAT partitions $X'^\sharp$ into a set of sub-inputs $\{X_1'^\sharp, X_2'^\sharp, ..., X_n'^\sharp\}$. Then VAT checks further for each sub-input $X_i'^\sharp$ separately. For a sub-input $X_i'^\sharp$, if its size is small enough, VAT uses bounded exhaustive testing, otherwise it uses abstract testing (on top of forward and backward abstract execution and property checking). This phase mimics the abstract test case generation of abstract testing.

Bounded exhaustive testing. When the number of concrete inputs in the considered abstract input $X_i'^\sharp$ is small, VAT uses bounded exhaustive testing to check the assertion for all possible concrete inputs. The rationale behind bounded exhaustive testing is that the failure of assertions can be mainly revealed within small bounds, and exhaustively testing within the bounds ensures that no "corner case" is missed [33].

When this whole verification process terminates, VAT will find a concrete counter-example, or provide a complete proof, or a resource limit is reached.

2.2 An illustrating example

Now we illustrate our approach by verifying the assertion ψ (i.e., y *!= 1225*) in the example P shown in Fig. 2(a). P implements the mathematical function shown in Fig. 2(b). From the mathematical function, one could know that when the input is n *= 49* and *flag = 1*, the program will result in y *= 1225* at Line 10 (while all other inputs will satisfy the assertion ψ). Thus the assertion ψ actually does not hold in program P.

Verifying ψ (whether it holds or not) in P automatically and completely is challenging for the following reasons. First, there is no restriction given on the input variables n and *flag*. Thus, without considering any mathematical background behind the program, $2^{32}*2^{32}$ cases of possible values of the input variables need to be considered if they are 32-bit integers. Directly using exhaustive testing to verify ψ in P would cause too much overhead. Second, the loop condition at Line 3 (i.e., $x < n$) depends on the symbolic input variable n. Symbolically executing all feasible program paths (through unrolling) is not possible, owing to the potentially infinite number of paths. Thus symbolic execution or bounded model checking can hardly verify ψ in P automatically and completely. Third, P involves disjunctive and non-linear behaviors, which is out of the expressiveness

```
1  void fun(int n, int flag){
2    int x = 0,y = 0;
3    while(x < n){
4      x = x + 1;
5      if(flag == 1)
6        y = y + x;
7      else
8        y = y − x * x;
9    }
10   assert(y != 1225);
11 }
```

(a)

$$y = \begin{cases} 0 & n \leq 0 \\ \frac{1}{2}n(n+1) & 1 \leq n, flag = 1 \\ -\frac{1}{6}n(n+1)(2n+1) & 1 \leq n, flag \neq 1 \end{cases}$$

(b)

Fig. 2. An illustrating example.

of most widely used numerical abstract domains [40]. And the non-linear behaviors also make most SMT solvers hard or even unable to verify the assertion. Hence, it is also difficult to verify the assertion by using abstraction and SMT based techniques, such as CEGAR based software model checking [17].

We now illustrate step by step how VAT verifies the assertion ψ in P. First, since there is no constraint over the input variables n and $flag$, VAT starts from the initial abstract input $X^\sharp = \top$. Suppose we use here the octagon abstract domain [41] to perform abstract interpretation. Abstract execution with abstract input $X^\sharp = \top$ will result in the following invariant (namely $Inv^{X^\sharp}$) at the assertion location (at Line 10): $n - x \leq 0 \wedge x \geq 0$. Then our analysis performs property checking and finds that this invariant is not strong enough to prove ψ or $\neg\psi$. Thus we perform a backward analysis starting from the assertion location assuming that $y == 1225$ (that is the negation of the original assertion). However, this round of backward analysis results in $\top$ at program entry and does not help in refining the necessary precondition to falsify the assertion. Then we partition the current abstract input $\top$ into several sub-inputs. Here, we use a predicate based partitioning strategy (which will be described in Sect. 3.3) to partition $\top$ into the following 6 abstract sub-inputs: $\{X_1^\sharp : n \leq 0 \wedge flag \leq 0; X_2^\sharp : n \leq 0 \wedge flag == 1; X_3^\sharp : n \leq 0 \wedge flag \geq 2; X_4^\sharp : n \geq 1 \wedge flag \leq 0; X_5^\sharp : n \geq 1 \wedge flag == 1; X_6^\sharp : n \geq 1 \wedge flag \geq 2\}$. Then verifying ψ on abstract input $X^\sharp$ boils down to verifying ψ on each abstract sub-input.

For $X_1^\sharp$, we perform forward abstract execution and get the invariant $\{Inv^{X_1^\sharp} : y == 0 \wedge \dots\}$ at the assertion location. After performing property checking, we find that $Inv^{X_1^\sharp} \Rightarrow \psi$, which imply that ψ holds for the abstract input $X_1^\sharp$. Similarly, for the other abstract inputs $X_2^\sharp$, $X_3^\sharp$, $X_4^\sharp$ and $X_6^\sharp$, the invariants generated by abstract execution at the assertion location are respectively $\{Inv^{X_2^\sharp} : y == 0 \wedge \dots; Inv^{X_3^\sharp} : y == 0 \wedge \dots; Inv^{X_4^\sharp} : y \leq -1 \wedge \dots; Inv^{X_6^\sharp} : y \leq -1 \wedge \dots\}$, which implies that ψ holds for all these abstract sub-inputs.

Now we consider the more complicated case, i.e., the abstract sub-input $X_5^\sharp$. We perform abstract execution on $X_5^\sharp$ and get invariant $Inv^{X_5^\sharp} : \{n - 1 \geq 0 \wedge -n + y \geq 0 \wedge x - 1 \geq 0 \wedge -x + y \geq 0\}$. Unfortunately, $Inv^{X_5^\sharp}$ is not precise

enough to prove ψ or $\neg\psi$. Then we perform backward analysis assuming that $y == 1225$ at the assertion location and get a refined abstract input $X_{51}^\sharp : \{n \geq 1 \wedge n \leq 1225 \wedge flag == 1\}$. In other words, inside $X_5^\sharp$, only those inputs in $X_{51}^\sharp$ may cause the assertion ψ to fail, thus we only need to check $X_{51}^\sharp$ for the case $X_5^\sharp$. Since now the abstract sub-input $X_{51}^\sharp$ contains only 1225 concrete inputs, we employ the bounded exhaustive testing to check the case $X_{51}^\sharp$. Then we will find the concrete counter-example input $n = 49, flag = 1$ in $X_{51}^\sharp$ that falsifies the assertion ψ.

To summarize, for the illustrating example shown in Fig. 2(a), we totally partition the whole input space $X^\sharp$ into 6 abstract sub-inputs such that $X^\sharp = X_1^\sharp \cup \ldots \cup X_6^\sharp$, where $X_1^\sharp, X_2^\sharp, X_3^\sharp, X_4^\sharp, X_6^\sharp$ are verified by abstract testing and we use bounded exhaustive testing to find a concrete counter-example in $X_{51}^\sharp$ which is a refinement substitution of $X_5^\sharp$.

3 Property-Oriented Iterative Abstract Testing

In this section, we formalize the main idea of iterative abstract testing. Sect. 3.1 gives the background of abstract testing. Sect. 3.2 introduces our framework of iterative abstract testing. Sect. 3.3 presents the algorithm of partitioning on abstract input.

3.1 Abstract Testing

With the abstract semantics, sound program invariants can be computed automatically in finite steps by forward abstract interpretation [18] (denoted as Forward_AI) and backward abstract interpretation [21] (denoted as Backward_AI) respectively. The computation with abstract interpretation is parameterized by abstract domains specifying the considered approximated properties. Note that backward abstract execution also makes use of the invariants generated by the aforementioned forward abstract execution. In this paper, we combine forward and backward abstract execution to generate for each program location those constraints that describe states which are reachable from the abstract input and may cause the target assertion fail.

The process of *Abstract testing* (denoted as AbstractTesting()) is built on top of Forward_AI and Backward_AI, as defined in Algorithm 1. Abstract testing takes a program P, a target property ψ to verify, a chosen abstract domain D and an abstract input $X^\sharp$. Abstract testing first calls Forward_AI (at Line 1), which computes the invariants (denoted as $Inv^{X^\sharp}$) on program P with initial state $X^\sharp$. To check whether the property ψ holds, abstract testing extracts $Inv^{X^\sharp}(l_\psi)$ of $Inv^{X^\sharp}$ at the location (i.e., l_ψ) before the assertion $assert(\psi)$. Three cases may arise after the checking: (a) the property ψ is surely true (Line 3 in Alg. 1) ; (b) the property ψ is surely false (Line 7); (c) whether the property ψ holds or not can not be determinated yet by $Inv^{X^\sharp}(l_\psi)$ (Line 10). In the third case, a backward abstract execution Backward_AI is launched to refine the abstract

Algorithm 1 Abstract Testing Algorithm

Input: program P, property ψ, abstract input $X^\sharp$, abstract domain D
Output: result res, refined abstract input $X'^\sharp$, program invariant $Inv_b^{X^\sharp}$

1: $Inv^{X^\sharp} \leftarrow$ Forward_AI$(P, X^\sharp, D)$
2: $Inv_b^{X^\sharp} \leftarrow Inv^{X^\sharp}$
3: **if** $Inv^{X^\sharp}(l_\psi) \Rightarrow \psi$ **then**
4: res $\leftarrow$ True
5: $X'^\sharp \leftarrow \bot$
6: **else**
7: **if** $Inv^{X^\sharp}(l_\psi) \Rightarrow \neg\psi$ **then**
8: $res \leftarrow$ False
9: $X'^\sharp \leftarrow \bot$
10: **else**
11: $Inv_b^{X^\sharp} \leftarrow$ Backward_AI$(P, Inv^{X^\sharp}(l_\psi) \sqcap \neg\psi, D)$
12: **if** $X^\sharp \sqcap Inv_b^{X^\sharp}(l_{ent}) == \bot$ **then**
13: $res \leftarrow$ True
14: $X'^\sharp \leftarrow \bot$
15: **else**
16: $res \leftarrow$ Unknown
17: $X'^\sharp \leftarrow X^\sharp \sqcap Inv_b^{X^\sharp}(l_{ent})$
18: **end if**
19: **end if**
20: **end if**
21: **return** $res, X'^\sharp, Inv_b^{X^\sharp}$

input $X^\sharp$. Backward_AI takes the program and the error state $Inv^{X^\sharp}(l_\psi) \sqcap \neg\psi$ as input and computes backward the necessary pre-condition $Inv_b^{X^\sharp}$ that may cause the property to fail. If $X^\sharp \sqcap Inv_b^{X^\sharp}(l_{ent})$ (where l_{ent} is the entry location of P) is $\bot$ (which means there is no concrete input in $X^\sharp$ that violates ψ), ψ must be true (Lines 12). Otherwise, whether ψ holds is still unknown (Line 15) within $X^\sharp$, and in this case a refined input $X'^\sharp$ is generated as a refinement substitution of $X^\sharp$ (Line 17).

3.2 Algorithm of Iterative Abstract Testing

One iteration of abstract testing may fail to verify the given property due to the over-approximation. In this paper, we propose to partition the input space to refine the computed invariants on demand.

Definition 1 *(Partition of abstract input) A partition of an abstract input $X^\sharp$ is a set of sub-inputs $\{X_1^\sharp, X_2^\sharp, \ldots, X_n^\sharp\}$ such that*

$$\begin{cases} \forall i \in \{1,2,...,n\}, X_i^\sharp \neq \bot \\ \forall i,j \in \{1,2,...,n\}, i \neq j \Rightarrow X_i^\sharp \sqcap X_j^\sharp = \bot \\ \sqcup_{i\in\{1,2,...,n\}} X_i^\sharp = X^\sharp. \end{cases}$$

Let $P(X^\sharp)$ denote the program P, whose initial state of input variables are constrained by $X^\sharp$. Then we have the following proposition.

Proposition 1 *(Soundness of verification by partitioning) Let* $\{X_1^\sharp, X_2^\sharp, \ldots, X_n^\sharp\}$ *be a partition of abstract input* $X^\sharp$*. If for any* $i \in \{1, 2, ..., n\}$*, assertion* ψ *is proved to be true in program P(*$X_i^\sharp$*), then* ψ *must be true in P(*$X^\sharp$*).*

Proof. We assume ψ is false in $P(X^\sharp)$, then there must exists a concrete input x that satisfies the constraint of $X^\sharp$, but makes ψ false. From Definition 1, we know $\sqcup_{i\in\{1,2,...,n\}} X_i^\sharp = X^\sharp$, thus there exists $k \in \{1, 2, ..., n\}$ such that $x \in X_k^\sharp$, which means ψ is false in $P(X_k^\sharp)$. This conflicts with the assumption. Thus ψ is true in P($X^\sharp$). □

Algorithm 2 Iterative Abstract Testing Algorithm

Input: program P, property ψ, abstract domain D
Output: True or False or Timeout
1: worklist $L \leftarrow \{\top\}$
2: **while** $L \neq \varnothing$ **do**
3: $X^\sharp \leftarrow$ Remove(L) //get and remove an element from L
4: $(res, X'^\sharp, Inv_b^{X^\sharp}) \leftarrow$ AbstractTesting($P, \psi, X^\sharp, D$)
5: **if** $res ==$ False **then**
6: Terminate with counter-example in $X^\sharp$
7: **else**
8: **if** $res ==$ True **then**
9: *skip*
10: **else**
11: $X\_list \leftarrow$ Partition($X'^\sharp, Inv_b^{X^\sharp}$)
12: $L \leftarrow$ Insert(L, X_list)
13: **end if**
14: **end if**
15: **if** Timeout **then**
16: Terminate with Timeout
17: **end if**
18: **end while**
19: Terminate with ψ proved

Intuitively, our framework partitions an abstract input $X^\sharp$ when one iteration of abstract testing cannot prove whether the property holds or not, and then applies abstract testing further on the partitioned sub-inputs separately. The overall iterative algorithm is shown in Algorithm 2, which fits into a conventional worklist algorithm. In the beginning, the only element in the worklist L is the initial input space I. For the sake of simplicity, in this paper we assume $I = \top$ (which means there is no restriction over the input). Then the algorithm copes with the abstract inputs in the worklist L one by one (in Lines 2-18). For each abstract input $X^\sharp$ in the worklist L (Line 3), the algorithm calls AbstractTesting(), which is detailed in Algorithm 1, trying to prove the property ψ with respect to the abstract input $X^\sharp$. If it fails, AbstractTesting() will return a refined abstract input $X'^\sharp$. Then Algorithm 2 partitions $X'^\sharp$ at Line 11 (where the Partition() procedure will be detailed in Algorithm 3), and puts all the newly split abstract sub-inputs into the worklist and repeats the process again (starting from Line 2). Until a counter-example is found or the property is proved true over all abstract inputs in the worklist, or a time limit is reached, the algorithm terminates.

Algorithm 3 Partitioning Algorithm

Input: abstract input $X^\sharp$, program invariant $Inv_b^{X^\sharp}$
Output: abstract input list X_list
1: $PS_0 \leftarrow \varnothing$; $X\_list = \{X^\sharp\}$
2: **for** each $l \in Lc(P)$ **do**
3: $\quad PS_0 \leftarrow PS_0 \bigcup \text{Project}(Inv_b^{X^\sharp}, l)$
4: **end for**
5: **for** each $p_0 \in PS_0$ **do**
6: $\quad p \leftarrow \text{Rename}(p_0)$
7: $\quad$ **for** each $X \in X\_list$ **do**
8: $\quad\quad X\_list \leftarrow (X\_list \setminus X) \cup \{X \wedge p, X \wedge \neg p\}$
9: $\quad$ **end for**
10: **end for**
11: **if** $X\_list = \{X^\sharp\}$ **then**
12: $\quad Itvs \leftarrow \text{Interval\_Hull}(X^\sharp)$
13: $\quad v \leftarrow \text{Var\_of\_Largest\_Range}(Itvs)$
14: $\quad X\_list \leftarrow \{X^\sharp \wedge v_{inf} \leq v \leq \frac{v_{inf}+v_{sup}}{2}, X^\sharp \wedge \frac{v_{inf}+v_{sup}}{2} < v \leq v_{sup}\}$
15: **end if**
16: **return** X_list

3.3 Partitioning

Input partitioning plays an important role in the iterative abstract testing. Depending on the target programs, we employ two strategies for dynamic input partitioning as shown in Algorithm 3: predicate based strategy (from Lines 2 to 10) and dichotomy strategy (from Lines 11 to 15).

Predicate-based strategy. The main idea of this strategy is to first derive a set of predicates over the symbolic initial values of the input variables and then to partition the input space $X^\sharp$ into a set of sub-inputs based on these predicates. To this end, first, as a preprocessing step, for every input parameter (e.g., x) in the program P, we introduce a symbolic input variable (e.g., x_0) and insert an assignment statement (e.g., $x_0 = x$;) to symbolically record its initial value.

AbstractTesting() (at Line 4 of Algorithm 2) analyzes the instrumented program and returns the program invariant $Inv_b^{X^\sharp}$, which records all the constraints between the original variables of P and the introduced symbolic input variables. In Algorithm 3, to derive interesting predicates, we only consider predicates at those program locations after conditional tests (which are represented as *Lc(P)* at Line 2). Moreover, we are only interested in predicates over symbolic input variables, the set of which is denoted as PS_0. Then for each program location l in *Lc(P)*, we project out all other variables (except symbolic input variables) from the computed invariants $Inv_b^{X^\sharp}$ by function Project() at Line 3. Project() returns a set of predicates over symbolic input variables, which are all collected into PS_0. Note that the projection operator is a default operator in each abstract domain and implemented efficiently using algorithms tailored to the specific constraint representation of the abstract domain. Then for each predicate p_0 in PS_0, we rename all the symbolic input variables (e.g., x_0) as the original input variables (e.g., x) by function Rename() at Line 6. It returns a splitting predicate p on input variables, which is used to split all the abstract inputs in X_list (from Lines 7 to 9). For each X in X_list, our algorithm first deletes X from

X_list, then splits X into two abstract inputs (i.e., $X \wedge p$, $X \wedge \neg p$), and adds them into X_list at Line 8. Note that, in the worst case, 2^n abstract inputs can be generated based on n predicates. To prevent partition explosion, we need to bound the number of predicates used for partitioning. Our immediate idea chooses a limited number of those predicates that emerge early in the forward AI analysis.

Take our illustrating program in Fig. 2(a) for example. First, as shown in Fig. 3, at Line 2, our preprocess defines two symbolic input variables for the input parameters n and $flag$, and assigns them with the initial values of n and $flag$. Then abstract testing generates invariants as well as necessary precondition of property violation for the program using the Octagon abstract domain. After projecting out other variables, the invariants on n_0 and $flag_0$ are derived, which is shown as annotations in Fig 3 at Lines 4, 6 and 11. After renaming, our analysis collects two meaningful atomic predicates $\{n \geq 1, flag = 1\}$. Based on them, the following 6 abstract sub-inputs are generated: $\{X_1^\sharp : n \leq 0 \wedge flag \leq 0; X_2^\sharp : n \leq 0 \wedge flag = 1; X_3^\sharp : n \leq 0 \wedge flag \geq 2; X_4^\sharp : n \geq 1 \wedge flag \leq 0; X_5^\sharp : n \geq 1 \wedge flag = 1; X_6^\sharp : n \geq 1 \wedge flag \geq 2\}$.

```
1  void fun(int n, int flag){
2    int n0=n, flag0=flag;
3    int x = 0,y = 0;
4    while(x < n){  // n0 ≥ 1 ∧
5      x = x + 1;
6      if(flag == 1)  // n0 ≥ 1 ∧ flag0=1
7        y = y + x;
8      else
9        y = y − x * x;
10   }
11   assert(y != 1225);  // top }
```

Fig. 3. The illustrating example with predicates annotated.

Given an abstract input $X^\sharp$, if no useful predicate on the symbolic input variables can be found, predicate based partitioning would fail (i.e., the condition at Line 11 in Algorithm 3 holds). In this case, our framework employs the dichotomy strategy (from Lines 12 to 14).

Dichotomy strategy. This strategy projects every input variable in the abstract input $X^\sharp$ into an interval (e.g., we treat $\top$ as [-max_int,max_int-1] for Integer type) by function Interval_Hull() at Line 12, which returns interval hulls of all input variables. Then it chooses the variable of the largest range as the variable (i.e., v) to be split, which is done by function Var_of_Largest_Range() at Line 13. After that, it conducts splitting evenly on the interval range of v, and $X^\sharp$ is split into two abstract sub-inputs at Line 14, where v_{inf} and v_{sup} represent the lower bound and upper bound of v respectively.

3.4 Sorting of Abstract Inputs

This subsection elaborates the sorting of the elements in the worklist L in Algorithm 2 in the framework of iterative abstract testing. This operation is necessary in the sense that, if an abstract input that violates the property ψ can be put in the front of the worklist, then the verification process can terminate earlier.

To perform sorting, we define a fitness function for each abstract input $X^\sharp$ as $fit(X^\sharp) = \frac{|\gamma(Inv_{l_\psi}^{X^\sharp} \sqcap \neg\psi)|}{|\gamma(Inv_{l_\psi}^{X^\sharp} \sqcap \psi)|}$, where $Inv_{l_\psi}^{X^\sharp}$ is the invariant at the assertion location computed by Forward_AI, and γ is a concrete function mapping abstract states to concrete states soundly. Here we assume that if the value returned by $fit(X^\sharp)$ is larger, then it is more likely to find a property violation within $X^\sharp$. Since $|\gamma(Inv_{l_\psi}^{X^\sharp} \sqcap \neg\psi)|$ and $|\gamma(Inv_{l_\psi}^{X^\sharp} \sqcap \psi)|$ are usually too costly to compute, in practice, we use $fit'(X^\sharp) = \frac{|\gamma(\text{Interval\_Hull}(Inv_{l_\psi}^{X^\sharp} \sqcap \neg\psi))|}{|\gamma(\text{Interval\_Hull}(Inv_{l_\psi}^{X^\sharp} \sqcap \psi))|}$, where $|\gamma(\text{Interval\_Hull}(Y^\sharp))|$ represents the number of points in the interval hull of $Y^\sharp$. The value of $|\gamma(\text{Interval\_Hull}(Y^\sharp))|$ is computed by projecting each input variable y into its interval bound $[a_y, b_y]$, and multiplying the widths of all these intervals. In other words, the value of $|\gamma(\text{Interval\_Hull}(Y^\sharp))|$ is computed as the volume of the interval hull of $Y^\sharp$.

In iterative abstract testing, our algorithm computes $fit'(X^\sharp)$ for each generated abstract input $X^\sharp$, and adds them into the worklist satisfying the decreasing order according to their fitness values (in Line 12 of Algorithm 2).

4 Combination with Bounded Exhaustive Testing

When the considered abstract input $X^\sharp$ is bounded and of small size (which means that the number of concrete inputs inside $X^\sharp$ is small), a good alterative of verifying the program on such an abstract input $X^\sharp$ is to use bounded exhaustive testing (BET) [11, 38], which is complete, sound, and able to find counter-examples if they exist. In this section, we extend our framework by combining with bounded exhaustive testing.

4.1 Synergic Verification Framework

Our synergic verification framework (denoted by SynergicVerification()) is based on Algorithm 2, while the only change is replacing function AbstractTesting() (at Line 4 in Algorithm 2) with SynergicTesting() described in Algorithm 4. Compared with Algorithm 2 that uses solely abstract testing, the key difference in Algorithm 4 is that we add a decision module SizeChecking() to decide whether to take abstract testing or bounded exhaustive testing (at Line 1) and a module BETesting() to conduct bounded exhaustive testing (at Line 2).

Algorithm 4 Synergic Testing Algorithm

Input: program P, property ψ, abstract input $X^\sharp$, abstract domain D
Output: result res, refined abstract input $X'^\sharp$, program invariant $Inv_b^{X^\sharp}$
1: **if** SizeChecking($X^\sharp$) **then**
2: res = BETesting($P, X^\#$)
3: $X'^\# \leftarrow \bot$
4: $Inv_b^{X^\sharp} \leftarrow \bot$
5: **else**
6: $\langle res, X'^\#, Inv_b^{X^\sharp} \rangle$ = AbstractTesting($P, \psi, X^\#, D$)
7: **end if**
8: **return** $res, X'^\#, Inv_b^{X^\sharp}$

Decision module Function SizeChecking() is implemented by just checking the size of the abstract input (i.e., $|\gamma(X^\sharp)|$): If the size is under a threshold, we adopt bounded exhaustive testing, otherwise we use abstract testing. In practice, $|\gamma(X^\sharp)|$ could be hard to be precisely computed, and hence we utilize $|\gamma(\text{Interval\_Hull}(X^\sharp))|$ as a compromise.

Bounded exhaustive testing module Bounded exhaustive testing aims to achieve exhaustive coverage of all the concrete inputs in the given abstract input $X^\sharp$. We exhaustively generate concrete input cases not directly from $X^\sharp$, but from its interval hull, and then filter out those that are not in $\gamma(X^\sharp)$ by checking whether they satisfy the constraints representing $X^\sharp$. During bounded exhaustive testing, once a concrete input is found as a counter-example that violates the target property, we will terminate the whole verification process.

4.2 Soundness discussion

Theorem 1. *(Soundness of the synergic verification) Suppose that we use the synergic verification algorithm (i.e., Algorithm 2 wherein Line 4 is replaced with SynergicTesting()) to verify whether an assertion ψ holds in program P. If the algorithm terminates with ψ proved (or terminates with a counter-example found), then ψ must be true (or false) in program P with any (or some) inputs.*

Proof. First, we consider the case that Algorithm 2 combined with bounded exhaustive testing terminates with a counter-example found in abstract input $X^\sharp$. Since the counter-example must belong to the initial input space (i.e., $\top$), thus it is also a counter-example input to make P violate ψ.

The more complicated case is when Algorithm 2 terminates with ψ proved. To prove ψ must be true in P, we first briefly explain the soundness of abstract testing and bounded exhaustive testing. (1) The soundness of abstract testing (Algorithm 1) is guaranteed by the soundness of abstract interpretation [18], since it applies forward and backward abstract interpretation, which essentially compute over-approximations of the concrete reachable states. (2) The soundness of bounded exhaustive testing is obvious since BET has tested all the possible input cases (within the given bounded abstract input space) under the most

precise (i.e., concrete) semantics. In Algorithm 2, the initial input space is partitioned into a set of sub-inputs, and each sub-input is proved by abstract testing or BET, or further partitioned into smaller sub-inputs, which are further coped with by Algorithm 2. According to Proposition 1 and the soundness of abstract testing and BET, we can conclude that ψ is true in P. □

Note that in this paper, as normal abstract interpretation-based verification [28, 32], we assume the absence of undefined behaviors and runtime errors in statements before the assertion location. The unsoundness of verification due to undefined behaviors and runtime errors has been handled in [16], which is orthogonal to our work. Moreover, in this paper, a non-deterministic variable is treated as an interval of its whole valid input range (e.g., [-∞, +∞]) in abstract testing and a fresh random value in concrete testing. For non-deterministic programs, bounded exhaustive testing is used to verify false assertions only when it finds a counter-examples (but not used to provide proof for true assertions).

5 Experiments and Evaluation

We have implemented a prototype tool, namely VATer, based on our verification approach utilizing both iterative abstract testing (IAT) and bounded exhaustive testing (BET). We will evaluate VATer along the following three experimental questions (EQs). VATer can verify programs through refining abstract interpretation by dynamic input partitioning. We want to know whether VATer can prove more true assertions than abstract interpretation based tools (which may use other refinement techniques) (EQ1). We also hope to know the performance of VATer comparing with other widely used verification techniques in practice (EQ2). VATer utilizes bounded exhaustive testing to help abstract interpretation to prove "corner cases" or generate counter-examples. We should know whether the use of bounded exhaustive testing can help verify more assertions in practice (EQ3).

5.1 Experimental Setup

VATer is constructed based on the APRON numerical abstract domain library [36] (which includes the abstract domains of intervals [18], octagons [41], polyhedra [22], and linear congruence abstract domain [27]), and the Fixpoint Solver Library [1]. Moreover, VATer syntactically supports both C programs (by using the front-end CIL [42]) and the SPL language (by using the static analyzer Interproc [35] as the front-end). The upper bound of the number of splitting predicates chosen each time is 10, and the threshold for BET is set as 2500, which are both set according to our timeout bound empirically.

To address these experimental questions, we run our tool VATer, abstract interpretation involved tools (e.g., Interproc [35], SeaHorn [32]) and three tools participating in SV-COMP'18 [2] (i.e., VeriAbs [14], ULTIMATE Taipan [28] (which obtained the highest scores in the *ReachSafety-Loops* category), CPAchecker [9] (which won the gold medal overall in SV-COMP'18)). We use three numerical loop benchmark sets: (1) all the 46 programs from HOLA [24]; (2) all the

35 programs from C4B [13]. (3) all the 152 programs from the verification tasks of six folders in ReachSafety-Loops category of SV-COMP'18 [2]. Note that we used the version of HOLA and C4B from [23].

Different sets of benchmarks and tools are chosen to answer different experimental questions (EQ). To answer EQ1, benchmarks (C4B and Hola) with only true assertions and tools involving abstract interpretation technique (i.e., Interproc, SeaHorn, ULTIMATE Taipan) are used. To answer EQ2 and EQ3, benchmarks from SV-COMP'18 (which contain both true and false assertions) are more suitable, and we chose to compare with three state-of-the-art tools (i.e., ULTIMATE Taipan, VeriAbs, CPAchecker) for EQ2.

All the experiments are carried out with a timeout limit of 900 seconds for each benchmark program on a machine with Ubuntu 16.04 which has 16GB RAM and a 3.6 GHz octa-core Intel® Core™ i7-7700U host CPU.

5.2 EQ1: Does VATer strengthen the ability of proving true assertions over abstract interpretation based techniques?

Table 1 shows the verification results on the HOLA and C4B benchmarks. It lists the number of verified programs (sub-column "#V") together with the total time for verified programs in seconds (column "#T(s)") for each tool. For VATer, it also lists the number of times using abstract testing (sub-column "#AT") and bounded exhaustive testing (sub-column "#BET") for the verified programs. We compare VATer with three available abstract interpretation based verification tools, i.e., Interproc (based on pure abstract interpretation), SeaHorn (combining Horn-clause solving and abstract interpretation) and ULTIMATE Taipan (combining CEGAR based software model checking and abstract interpretation).

Table 1. Comparison results with abstract interpretation involved tools.

	Interproc		SeaHorn		UTaipan		VATer			
Benchmark	#V	T(s)	#V	T(s)	#V	T(s)	#V	T(s)	#AT	#BET
HOLA(46)	17	2.9	34	298.5	38	805.2	**44**	14.1	64	1
C4B(35)	2	0.1	24	274.9	17	1277.4	**32**	2.3	85	0
Total(81)	19	3.0	58	573.4	51	2082.6	**76**	16.4	149	1

All the specified assertions hold in the programs from HOLA and C4B. Table 1 shows that VATer can correctly verify 76 programs out of 81 with total 16.4 seconds consumed (0.22s per program on average). Comparing with Interproc (one pass forward analysis), which can verify 19 programs with average 0.16s, VATer achieves significant improvements on proving true assertions without too much extra time overhead. It indicates that our technique strengthens the ability

Table 2. Comparison results with state-of-the-art verification tools.

Folder	P	IAT		VATer(IAT+BET)				VeriAbs		UTaipan		CPAChecker	
		#V	T(s)	#V	T(s)	#AT	#BET	#V	T(s)	#V	T(s)	#V	T(s)
Loops(67)	T(35)	21	4.2	23	5.0	23	2	26	749.9	25	400.3	**27**	1281.6
	F(32)	7	2.4	18	55.0	145	11	24	416.4	25	1080.7	**29**	550.3
Loop-new(8)	T(8)	4	4.7	**7**	5.8	7	3	2	21.9	4	187.1	2	710
	F(0)	0	0	0	0	0	0	0	0	0	0	0	0
Loop-lit(16)	T(15)	9	2.3	13	2.7	15	4	12	304.9	**14**	388.5	6	27.1
	F(1)	0	0	**1**	0.2	1	1	**1**	13.4	**1**	4.5	**1**	3.9
Loop-inv(19)	T(18)	**15**	32.6	**15**	32.6	16	0	8	144.7	10	253.4	5	440.6
	F(1)	0	0	**1**	11.9	86	1	**1**	17.1	**1**	7.7	**1**	5.4
Loop-craft(7)	T(6)	2	0.2	**4**	0.6	4	2	3	33.7	2	9.2	3	520.5
	F(1)	0	0	0	0	0	0	1	555.8	1	4.5	1	4.2
Loop-acc(35)	T(19)	9	0.9	**16**	96.2	78	38	12	132.9	13	510.7	10	663.9
	F(16)	1	0.1	**16**	9.0	43	15	13	290.1	6	84.2	8	946.8
Total(152)	T(101)	60	44.9	**78**	142.9	143	49	63	1388	68	1749.2	53	3643.7
	F(51)	8	2.5	36	76.1	275	28	**40**	1292.8	34	1181.6	**40**	1510.6

of proving true assertions over standard abstract interpretation. And considering the iterations of abstract testing and BET, we find that this strengthening mainly comes from the iterative abstract testing through dynamic input partitioning.

VATer can verify 18 (31%) and 25 (49%) more programs than SeaHorn and ULTIMATE Taipan, respectively. Concerning timing, for those verified programs, VATer (on average 0.22s) has an average 46X, 190X speedups over SeaHorn (on average 9.9s) and ULTIMATE Taipan (on average 40.8s) respectively. This improvement is achieved since most of the programs in HOLA and C4B have complex input-data dependent loops with disjunctive or non-linear properties. They are difficult to verify as a whole. While VATer utilizes partitioning techniques to simplify the program behaviors for each abstract input. This result reflects that VATer performs more effectively and efficiently than abstract interpretation based tools that use forward analysis only (e.g., SeaHorn and ULTIMATE Taipan) on these benchmarks.

5.3 EQ2 – How does VATer work comparing with other verification techniques

We compare VATer with three state-of-the-art verification tools participating in SV-COMP'18: VeriAbs, ULTIMATE Taipan, CPAchecker. Table 2 shows the comparison result. The column "Folder" shows the names of all the folders and the number of programs included in each folder. The column "Property" distinguishes the programs with true assertions and false assertions in each folder.

For the programs with true assertions (totally 101 programs), VATer can verify 15 (24%), 10 (15%) and 25 (47%) more programs than VeriAbs, ULTIMATE

Taipan and CPAchecker respectively. This improvement is achieved mainly due to the fact that most of these programs have input-data dependent loops and loop-result dependent branches. These program characteristics may result in infinite number of program states and make the precise (enough) loop invariants difficult to find by software model checking based tools. However, for these programs, VATer can always utilize input partitioning to get refined abstract inputs to help forward and backward abstract interpretation to generate sound invariants and necessary preconditions, which may be precise enough to prove the assertions finally. For the programs with false assertions (totally 51 programs), VATer finds counter-examples for 36 programs, which is less than VeriAbs (40 programs), CPAchecker (40 programs) but more than ULTIMATE Taipan (33 programs). We have further investigated those false-assertion programs for which other tools succeed but VATer fails. We found that for false-assertion programs we rely on bounded exhaustive testing to ensure the soundness of VATer, but if iterative abstract testing cannot reduce the search space into a small-size region then bounded exhaustive testing may take too much overhead to do dynamic testing exhaustively. For all the programs with true or false assertions (totally 152 programs), compared with these alternatives, VATer achieves 11%, 13%, 22% improvement respectively. Concerning timing, for those verified programs, VATer (on average 1.9s) at least has an average 13.6X, 15.2X, and 29.2X speedups over VeriAbs (on average 26.0s), ULTIMATE Taipan (on average 28.8s), CPAChecker (on average 55.4s) respectively. The results indicate that VATer also significantly outperforms other three tools on efficiency for these benchmarks.

5.4 EQ3: Does BET help VATer generate counter-examples over abstract testing?

In Table 2, the column "IAT" gives results where only iterative abstract testing is used. Comparing IAT and VATer, we can see that: (1) Considering the 101 programs with true assertions, IAT can verify 60, while VATer can successfully verify 78 programs, achieving a 30% improvement. We have inspected those programs with true assertions that only VATer successfully verified, and found that all of them have "corner cases" that cannot be verified by abstract testing solely, while testing can handle them quickly. (2) Considering 51 programs with false assertions, IAT finds counter-examples for 8 programs, while VATer generates counter-examples for 36 programs. The result indicates that bounded exhaustive testing makes significant contribution to generate counter-examples for programs with false assertions. (3) Overall, VATer can verify 114 programs with the average time of 1.9s, while IAT can verify 68 programs with average time of 0.7s. Hence, VATer achieves 68% improvement on IAT without much extra time overhead. This achievement mainly owes to the fact that VATer combines the advantages of both iterative abstract testing and bounded exhaustive testing. Considering the "#AT" and "#BET" column, VATer can verify 46 programs more than IAT with 77 (average 1.7) times of BET used. It indicates that, for these programs, iterative abstract testing has restricted the unverified input

spaces into small sizes, thus only a few numbers of BET are conducted to prove the true assertions or generate counter-examples for false assertions.

6 Related work

Abstract interpretation based verification. Many efforts [6, 30, 31, 37] have been devoted to combine the strengths of over and under approximation. They mainly used model checking based techniques as over approximation engines. While this paper has used abstract interpretation, which can handles loops automatically in a terminate and sound way. Abstract interpretation is one of the fundamental techniques for automatic program verification [20, 25]. Many recent approaches and tools for program verification use abstract interpretation. SeaHorn [32] combines Horn-clause solving techniques with abstract interpretation based analyzer IKOS [12], where IKOS is mainly used to supply program invariants to other techniques. ULTIMATE Taipan [28] is a CEGAR-based software model checker for verifying C programs, where abstract interpretation is used to derive loop invariants for the path program corresponding to a given spurious counterexample. A series of works have used interpolation technique to recover the imprecision due to widening and improved the verification ability of abstract interpretation based techniques, such as DAGGER [29], VINTA [4], UFO [5]. Unlike the above works, we use input space partitioning to refine abstract interpretation on-demand iteratively, and use bounded exhaustive testing to complement abstract interpretation.

Recently, combining abstract interpretation with dynamic analysis has received increasing attention. Most of these works combine abstract interpretation with symbolic execution [3,15,16,26], which mainly combine them in a two-stage manner and use (non-iterative) abstract interpretation as a black-box. While our work aims at program verification by performing abstract interpretation in an iterative way and makes use of the results of dynamic testing to complement abstract interpretation. Quite interestingly, Toman et al. [43] have recently presented the Concerto system for analyzing framework-based applications by combining concrete and abstract interpretation, which analyzes framework implementations using concrete interpretation and analyzes application code using abstract interpretation. Compared with their work, our work uses dynamic testing and abstract interpretation to verify the same code, rather than different parts of the target program. The closest work to ours is [19], in which Cousot and Cousot propose first the notion of "abstract testing", and compare abstract testing with classical program debugging and model checking. Our work is inspired by this work [19], but we further propose to conduct abstract testing iteratively with respect to the target property with the help of dynamic input partitioning. Moreover, we also propose to combine abstract testing with bounded exhaustive testing.

Partitioning techniques. There exist several partitioning techniques in the context of abstract interpretation. Bourdoncle [10] presents a partitioning method in the context of analyzing recursive functions, to allow the dynamic determi-

nation of interesting abstract domains using data structures built over simpler domains. Jeannet [34] proposes a method to dynamically select a suitable partitioning according to the property to be proved, which relies on the use of a new abstract lattice combining Boolean and numerical properties. These partitioning techniques belong to state partitioning, while this paper only partitions input and then conducts abstract interpretation separately for each partition. Mauborgne and Rival [39] propose a systematic framework to utilize trace partitioning for managing disjunctions. Their trace partitioning techniques rely on heuristics or annotations to specify partition creation and merge points, while our approach only chooses program entries as the partitioning points, which makes our partitioning strategy fully automatic and easier to deploy. Another benefit of input space partitioning lies in that it can help to recover the precision loss as early as possible during generating invariants. Thus it can generate more precise invariants than partitioning intermediate states. Conditional model checking [7, 8] combines the verification abilities of several different model checkers. Each model checker generates a condition to describe the successfully verified state space. Thus, utilizing this condition, the later verifiers only focus on verifying the yet unverified state space. These conditions generated by model checkers can be considered as a partition of state space. Compared with their work, we perform partitioning dynamically and iteratively according to the need of the current verification task and we only consider partitioning the inputs at the entry point of a program.

7 Conclusion

We have presented a property-oriented verification approach based on iterative abstract testing, to verify properties of numerical programs. Our approach iterates forward abstract execution (to compute invariants) and backward abstract execution (to compute necessary pre-condition for property violation) to verify the target property. The key point behind our iterative mechanism is the utilization of dynamic input space partitioning to split an abstract input that needs further exploration into sub-inputs such that each sub-input involves less program behaviors and may be easier to verify. The partitioning is conducted dynamically (on demand) according to the needs of the sub-goal of the verification. Moreover, the partitioning enables the verification to be achieved via bounded exhaustive testing over an abstract sub-input of small size, which complements the abstract testing and is able to generate counter-examples when the property does not hold. Finally, we have shown promising experimental results comparing against several state-of-the-art program verification tools.

For future work, we plan to investigate other dynamic analysis techniques to complement our abstract testing, especially for the cases that the property to be checked does not hold. Also, our approach is highly parallelizable by nature thanks to the partitioning, and thus we plan to develop a parallel version for speedup.

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), the NSFC Program (Nos. 61872445, 61532007), and the NSF under Grants CNS-1446511 and CCF-1617717. This work is also supported by the Hunan Key Laboratory of Software Engineering for Complex Systems, China.

References

1. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/index.html
2. SV-COMP'18 (7th international competition on software verification). https://sv-comp.sosy-lab.org/2018/
3. Alatawi, E., Søndergaard, H., Miller, T.: Leveraging abstract interpretation for efficient dynamic symbolic execution. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 619–624. IEEE Press (2017)
4. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: International Static Analysis Symposium. pp. 300–316. Springer (2012)
5. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction-and interpolation-based software verification. In: International Conference on Computer Aided Verification. pp. 672–678. Springer (2012)
6. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S.D., Thakur, A.V.: Proofs from tests. IEEE Transactions on Software Engineering **36**(4), 495–508 (2010)
7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. p. 57. ACM (2012)
8. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1182–1193. ACM (2018)
9. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: International Conference on Computer Aided Verification. pp. 184–190. Springer (2011)
10. Bourdoncle, F.: Abstract interpretation by dynamic partitioning. Journal of Functional Programming **2**(4), 407–435 (1992)
11. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on *Java* predicates. In: ACM SIGSOFT Software Engineering Notes. vol. 27, pp. 123–133. ACM (2002)
12. Brat, G., Navas, J.A., Shi, N., Venet, A.: Ikos: A framework for static analysis based on abstract interpretation. In: International Conference on Software Engineering and Formal Methods. pp. 271–277. Springer (2014)
13. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. ACM SIGPLAN Notices **50**(6), 467–478 (2015)
14. Chimdyalwar, B., Darke, P., Chauhan, A., Shah, P., Kumar, S., Venkatesh, R.: Veriabs: Verification by abstraction (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 404–408. Springer (2017)

15. Christakis, M.: On narrowing the gap between verification and systematic testing. it-Information Technology **59**(4), 197–202 (2017)
16. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proceedings of the 38th International Conference on Software Engineering. pp. 144–155. ACM (2016)
17. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5), 752–794 (2003)
18. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM (1977)
19. Cousot, P., Cousot, R.: Abstract interpretation based program testing. In: Proceedings of the SSGRR 2000 Computer & eBusiness International Conference. Scuola Superiore G. Reiss Romoli L'Aquila, Italy (2000)
20. Cousot, P., Cousot, R.: On abstraction in software verification. In: International Conference on Computer Aided Verification. pp. 37–56. Springer (2002)
21. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 128–148. Springer (2013)
22. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 84–96. ACM (1978)
23. Cyphert, J., Breck, J., Kincaid, Z., Reps, T.: Refinement of path expressions for static analysis. Proceedings of the ACM on Programming Languages **3**(POPL), 45 (2019)
24. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: Acm Sigplan Notices. vol. 48, pp. 443–456. ACM (2013)
25. D'silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(7), 1165–1178 (2008)
26. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: dynamic symbolic execution guided with static verification results. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 992–994. ACM (2011)
27. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics **30**(3-4), 165–190 (1989)
28. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Taipan: Trace abstraction and abstract interpretation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 399–403. Springer (2017)
29. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 443–458. Springer (2008)
30. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 117–127. ACM (2006)
31. Gunter, E., Peled, D.: Model checking, testing and verification working together. Formal Aspects of Computing **17**(2), 201–221 (2005)
32. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: International Conference on Computer Aided Verification. pp. 343–361. Springer (2015)

33. Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the costs of bounded-exhaustive testing. In: International Conference on Fundamental Approaches to Software Engineering. pp. 171–185. Springer (2009)
34. Jeannet, B.: Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. Formal Methods in System Design **23**(1), 5–37 (2003)
35. Jeannet, B.: Interproc analyzer for recursive programs with numerical variables. INRIA, software and documentation are available at the following URL: http://pop-art. inrialpes. fr/interproc/interprocweb. cgi. Last accessed pp. 06–11 (2010)
36. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: International Conference on Computer Aided Verification. pp. 661–667. Springer (2009)
37. Kroening, D., Groce, A., Clarke, E.: Counterexample guided abstraction refinement via program execution. In: International Conference on Formal Engineering Methods. pp. 224–238. Springer (2004)
38. Marinov, D., Khurshid, S.: Testera: A novel framework for automated testing of Java programs. In: Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on. pp. 22–31. IEEE (2001)
39. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: European Symposium on Programming. pp. 5–20. Springer (2005)
40. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages (FnTPL) **4**(3–4), 120–372 (2017)
41. Miné, A.: The octagon abstract domain. Higher-order and symbolic computation **19**(1), 31–100 (2006)
42. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. In: International Conference on Compiler Construction. pp. 213–228. Springer (2002)
43. Toman, J., Grossman, D.: Concerto: a framework for combined concrete and abstract interpretation. Proceedings of the ACM on Programming Languages **3**(POPL), 43 (2019)