Symbolic Execution with CEGAR

Dirk Beyer[1] and Thomas Lemberger[2]

[1] LMU Munich, Germany
[2] University of Passau, Germany

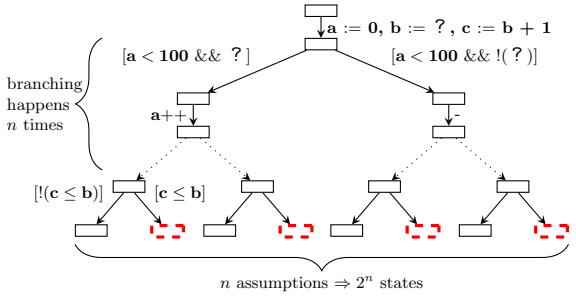
Abstract. Symbolic execution, a standard technique in program analysis, is a particularly successful and popular component in systems for test-case generation. One of the open research problems is that the approach suffers from the path-explosion problem. We apply abstraction to symbolic execution, and refine the abstract model using counterexample-guided abstraction refinement (CEGAR), a standard technique from model checking. We also use refinement selection with existing and new heuristics to influence the behavior and further improve the performance of our refinement procedure. We implemented our new technique in the open-source software-verification framework CPACHECKER. Our experimental results show that the implementation is highly competitive.

1 Introduction

Symbolic execution was introduced in 1976 for program testing and verification [27]. By extending the interpreter of a programming language to handle symbolic values without changing the program syntax, programs can be executed in such interpreter using symbolic values as input. If a fork in the program's control flow occurs, e.g., due to a branching statement for which both branches are possible, the execution splits into two separate executions, recording the particular branching condition. Each such execution represents the execution of the program for a *set* of concrete input values, which can be derived based on all recorded branching conditions of an execution. This way, a lot fewer *symbolic* executions are necessary for reaching a certain test coverage in comparison to executions with concrete input values. The main problem of symbolic execution is that the number of separate executions is exponential in the number of branching statements in the program. Because every visit of a loop head can be seen as a branching statement, the number of separate executions for a single program may easily exceed feasible amounts. This is known as the *path-explosion problem*. Figure 1 demonstrates this via a simple example program. It uses a function `?` that returns a non-deterministic, arbitrary value at every call. In a real-world application this could be, for example, a system call or user input. The program counts a program variable `a` from 0 to 100 in a non-deterministic number of loop iterations (caused by the non-deterministic assumption in the loop body). After that, it checks whether the non-deterministic, but unchanged value stored in `b` is still smaller than its increment stored in `c`. This is always true. Although the number of iterations through the loop has no influence on this property, an eager

```
1  a := 0;
2  b := ?;
3  c := b + 1;
4  while a < 100 do
5    if ? do
6      a++;
7  if c <= b do
8    error;
```

(a) Example program



(b) Example analysis tree

Fig. 1: A simple program demonstrating the path-explosion problem

approach like symbolic execution will explore the complete state-space before proving the property violation (symbolized by the dashed nodes) as infeasible. We propose a lazy approach that, in this example, ignores the loop conditions if they are not necessary to verify a given property or to satisfy a given coverage criterion. Different approaches exist to mitigate the path-explosion problem, but none of them tries to weaken the *precision* of the symbolic execution. In formal software verification, abstraction is a widely-used technique to reduce the state-space, with counterexample-guided abstraction refinement (CEGAR) [18] being a popular and successful approach for computing an abstract model.

We present SYMEX$^+$, a combination of symbolic execution with abstraction, which automatically refines the abstract model using CEGAR [18] in a lazy manner [22]. The automatic precision adjustment [7] with lazy CEGAR allows us to use a precision as weak as possible (to manage the state-space) and as strong as necessary (to prove a program safe or find a bug). Considering the example above, an analysis using CEGAR will not track the value of program variable `a`, because it is not necessary for proving that the error is not reachable. This idea makes it possible to prove the property after only one iteration of CEGAR and without unrolling the loop (and thus keeping the state-space significantly smaller).

Further, we are able to benefit from improvements to CEGAR. As a first improvement, we apply *refinement selection* [11] to our precision adjustment to have control over the choice of precision from multiple candidate precisions, and using this method, we can better control the overall symbolic-execution process.

Since symbolic execution is the composition of two abstract domains, namely (1) tracking explicit and symbolic values of variables and (2) tracking constraints over symbolic values, we apply CEGAR to two abstract domains at once instead of only to one abstract domain, as in previous work. We extend CEGAR to refining several abstract domains simultaneously by using a composite strongest-post operator of the configurable program analysis (CPA) [6]. This special case of using CEGAR can be generalized to any composition of abstract domains, allowing novel applications. Until now, CEGAR was applied to only one single abstract domain independently for one error path, even in composite setups [7].

Availability. We implemented symbolic execution with CEGAR in the open software-verification framework CPACHECKER [8]. All experimental results are available on our supplementary web page.[3]

Structure. After clarifying the preliminaries in Sect. 2, we formalize the application of CEGAR and interpolation to symbolic execution in Sect. 3. In Sect. 4, we perform a thorough evaluation to show the applicability and high competitiveness of our approach to reachability analysis in software verification. The results show a major speed-up compared to traditional symbolic execution for most verification tasks.

Related Work. There are four major ways to address the path-explosion problem of symbolic execution: (1) search heuristics for achieving a high level of branch or path coverage as fast as possible, (2) compositional execution, creating summaries of functions or paths, and reuse them instead of recomputing already explored states, (3) handling of unbounded loops, and (4) using interpolants for tracking reasons why a certain path is infeasible. While many concepts are presented in the context of testing, they can be applied to verification as well.

Search Heuristics. Burnim and Sen [14] propose three different heuristics for reaching a target region or uncovered branches faster in state-space exploration, in contrast to the standard depth-first search. KLEE [15] is a tool for automatic test-case generation that runs one symbolic execution for each branch separately. The implementation uses two different heuristics to decide at a certain program location which execution to continue first. While heuristics can assist in speeding up the process of finding an error, they do not mitigate the problem of path-explosion for proving that a program is error-free.

Compositional Execution. Compositional symbolic execution [20] tests functions in isolation in order to create summaries of the functions for reuse. It is implemented in SMART, an extension of the symbolic-execution-based testing tool DART [21]. Demand-driven compositional symbolic execution extends compositional symbolic execution by *lazy* and *relevant exploration* [1].

Handling of Unbounded Loops. Lazy Annotation [29] tackles potentially infinite analyses that are caused by loops, by computing inductive invariants for loops. A major downside of this approach is that it will only terminate if such invariants can be found. Based on this insight, loop invariants can be computed inductively, in order to speed up computation by using speculative loop invariants [25]. Strongest possible invariants are used to keep the analysis as eager as possible, while keeping the analysis tree finite. If invariants are too coarse to prove the infeasibility of an inconclusive counterexample, a refinement procedure similar to CEGAR (restricted to loop headers) is used. This is a compromise between performing eager symbolic execution and lazy CEGAR when encountering unbounded loops. Compact symbolic execution [30] analyzes cyclic paths in the control-flow automaton (CFA) and computes a so called *template* for each one, describing all possible program states that may leave the cycle after any number of iterations.

[3] `http://www.sosy-lab.org/~dbeyer/cpa-symexec/`

This mitigates the path-explosion problem considerably, because no more loops exist in the execution. However, due to quantifiers in formulas, the complexity of formulas that have to be solved is increased significantly. The experimental evaluation shows that despite this trade-off, the analysis performance is still considerably improved compared to the previous approaches.

Using CEGAR with symbolic execution, as proposed in this work, may also avoid the problem of path-explosion in the presence of unbounded loops, because information altered by loops is not always necessary for reasoning about programs. Since we implemented symbolic execution in the verification framework CPACHECKER, we are able to make use of the possibility to combine several different analyses that are implemented in the framework. For handling unbounded loops, an analysis that is specialized on this could be used in parallel.

Interpolation. The technique of interpolation [19,28] is often used to identify reasons for path infeasibility. If a path is found to be infeasible, an interpolant is computed for each program location on the path and stored as "precision" of the analysis. If such a program location is re-visited on a different path, it is checked whether the interpolant is implied by the current abstract state. If it is, execution on this path may halt, if it is known that the path is infeasible based on the interpolant. This concept was used in the context of the constraint-logic programming scheme [23,26]. There were experiments on using weakest pre-conditions instead of strongest post-conditions for the computation of weaker interpolants [25], on various search heuristics [24], and on adding the notion of *laziness* to symbolic execution [17]. Instead of computing interpolants immediately after a path is determined as infeasible, the symbolic execution might continue on this path ignoring the infeasibility, in order to be able to learn better interpolants.

Lazy Annotation [29] uses interpolants to store conditions for nodes and edges on the CFA under which no target region is reachable from this node or using this edge. Instead of annotating all edges on an infeasible path with interpolants in a separate procedure, interpolants are computed bottom-up during state-space exploration.

A main difference that persists between symbolic execution with CEGAR and symbolic execution using interpolants is the amount of information stored. CEGAR is lazy, starting with a coarse precision and refining it, while traditional symbolic execution is eager, tracking all information and computing interpolants for subsuming new states only. Using CEGAR, refinements to compute the needed level of abstraction and iterative analysis replace unnecessary state-space exploration. This pays off if only few program variables or constraints have to be tracked, or only few possible error paths exist.

2 Background

Control-Flow Automaton, State, Path, Semantics, and Precision. For presentation, all theoretical concepts are based on a simplified programming language where all operations are either variable assignments or assumptions.[4] All values in this language are integers of arbitrary range. We represent a program by a *control-flow automaton* (CFA) [13]. A CFA $A = (L, l_0, G)$ is a directed graph whose nodes L represent the program locations of the program, the initial node $l_0 \in L$ represents the program entry, and the set $G \subseteq L \times Ops \times L$ represents all edges of the graph. An edge $g \in G$ exists between two nodes if a program statement exists that transfers control between the program locations represented by the nodes. Each edge is labeled with the operation that transfers the control. The set X is the set of all program variables occurring in the program. A *concrete state* (l, c) consists of a program location $l \in L$ and a concrete variable assignment $c : X \to \mathbb{Z}$, which assigns to a program variable $x \in X$ an integer value from $\mathbb{Z}$ (the set of integer numbers). The set of all concrete states of a program is C. A set $r \subseteq C$ is called a *region*. The region of concrete states that violate a given specification is called *target region* r^t. For a partial function $f : M \rightarrow\!\!\!\!\!\rightarrow N$ for two sets M and N, we denote the *definition range* as $\mathtt{def}(f) = \{x \mid \exists y : (x, y) \in f\}$ and the *restriction* to a new definition range M' as $f_{|M'} = f \cap (M' \times N)$. An abstract state $s \in \mathbb{A}$ is an element of an arbitrary type $\mathbb{A}$ that depends on the analysis. Abstract state s represents the region $[\![s]\!]$ of concrete variable assignments. The special value $\bot$ with $[\![\bot]\!] = \varnothing$ is part of every abstract-state type. An abstract variable assignment $v : X \rightarrow\!\!\!\!\!\rightarrow \mathbb{V}$ is a partial function that assigns to a program variable from X a value from the set $\mathbb{V}$, which consists of arbitrary values. The strongest-post operator $\mathtt{SP}_{op} : \mathbb{A} \to \mathbb{A}$ defines the semantics of an operation $op \in Ops$, i.e., $\mathtt{SP}_{op}(a) = a'$ expresses that abstract state a' represents the set of concrete variable assignments that are reachable by executing op from concrete variable assignments represented by abstract state a.

A *path* σ is a sequence $\langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ of operations and their corresponding target program locations. A path σ is a *program path* if σ represents a syntactic walk through the CFA, that is, for every $1 \leq i \leq n$, a CFA edge $g = (l_{i-1}, op_i, l_i)$ exists and l_0 is the initial program location. Every path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ defines a *constraint sequence* $\gamma_\sigma = \langle op_1, \ldots, op_n \rangle$ [9]. The *conjunction* of two constraint sequences $\gamma = \langle op_1, \ldots, op_n \rangle$ and $\gamma' = \langle op'_1, \ldots, op'_n \rangle$ is defined as their concatenation: $\gamma \wedge \gamma' = \langle op_1, \ldots, op_n, op'_1, \ldots, op'_n \rangle$.

The *semantics of a path* $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ is defined as the successive application of the strongest-post operator to each operation of the corresponding constraint sequence γ_σ, that is, $\mathtt{SP}_{\gamma_\sigma}(a) = \mathtt{SP}_{op_n}(\ldots \mathtt{SP}_{op_1}(a))$. A program path σ is *feasible* if $\mathtt{SP}_{\gamma_\sigma}(\varnothing)$ is not contradicting, that is, $\mathtt{SP}_{\gamma_\sigma}(\varnothing) \neq \bot$. Otherwise, it is *infeasible*. An *error path* is a path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ for which $\mathtt{SP}_{\gamma_\sigma}(\varnothing)$ represents at least one concrete variable assignment c_{γ_σ} for which (l_n, c_{γ_σ}) is part of the target region r^t. A program is *safe* if no feasible error path exists. The *precision* $\pi : L \to 2^\Pi$ assigns to each program location some

[4] Our implementation in CPAchecker is based on the language C.

Algorithm 1 $CEGAR(\mathbb{D}, e_0, \pi_0)$, adapted from [7]

Input: a CPA $\mathbb{D}$ with dynamic precision adjustment, an initial abstract state $e_0 \in E$ with precision $\pi_0 \in \Pi$
Output: the verification result TRUE or FALSE
Variables: the sets `reached` and `waitlist` of elements of $E \times \Pi$, an error path σ

```
 1: reached := {(e_0, π_0)}
 2: waitlist := {(e_0, π_0)}
 3: π := π_0
 4: while true do
 5:     (reached, waitlist) := CPA(𝔻, reached, waitlist)
 6:     if waitlist = ∅ then
 7:         return TRUE
 8:     else
 9:         σ := extractErrorPath(reached)
10:         if isFeasible(σ) then
11:             return FALSE
12:         else
13:             π := π ∪ refine(σ)
14:             reached := {(e_0, π)}
15:             waitlist := {(e_0, π)}
```

information that defines the level of abstraction of the analysis. The information type Π depends on the abstract domain of the analysis. For an explicit-value domain for example, the set Π is a set X of program variables, and the precision defines the program variables that should be tracked at the respective location.

Counterexample-guided Abstraction Refinement (CEGAR). CEGAR [18] is a technique to construct an abstract model that contains as few information as possible while retaining enough information to prove or disprove the correctness of a program. The technique starts the analysis with a coarse abstraction and refines it based on infeasible error paths. An error path is a witness of a property violation. If no error path is found by the analysis, it terminates and reports that no property violation exists. If an error path is found, it is checked whether the path is feasible, e.g., by repeating the analysis with full precision $\pi(l) = \Pi$ for all $l \in L$. If the path is feasible, the analysis terminates and reports the found property violation. If the error path is infeasible, then it was due to a too coarse abstract model (too low precision). To eliminate this infeasible error path from future state-space explorations, the precision is increased (which refines the abstract model) using information extracted from the infeasible error path. Afterwards, the analysis starts again, using the new precision. Algorithm 1 uses a configurable program analysis (CPA) with dynamic precision adjustment $\mathbb{D}$ [7] and an initial state e_0 with initial precision π_0 (usually $\pi_0(l) = \varnothing$ for all $l \in L$) to perform the state-space exploration.

The CPA algorithm operates on a set of reached abstract states (`reached`) and a subset of this set that contains all reached abstract states that have not been handled yet (`waitlist`). If `waitlist` is empty, the CPA algorithm has

handled all reachable states without encountering any target state. If this is the case, no property violation was found and the algorithm can return TRUE. Otherwise, an error path is extracted from the reached set. If the error path is reported as feasible, a property violation exists and the algorithm returns FALSE. If the error path is infeasible, the current precision is too coarse. The precision is refined based on the infeasible error path by using function $\texttt{refine} : \Sigma \to 2^\Pi$ with Σ being the type of all infeasible error paths. The function assigns to an infeasible error path a precision that is sufficient to prove the infeasibility of the error path and eliminate this infeasible path from future explorations. After this, the reached set and waitlist are reset to their initial values and the algorithm continues the analysis with the refined precision. It is important to note that the return type of `refine` has to be equal to the precision type 2^Π used in $\mathbb{D}$. Because of this, analyses are in general not exchangeable without changing the refinement component as well. Since the problem of finding the coarsest possible refinement for a given abstract model based on an infeasible error path is NP-hard [18], heuristics have to be used to find suitable refinements [11].

A boolean formula Γ is a Craig interpolant [19] for two boolean formulas γ^- (called prefix) and γ^+ (called suffix), if the following three conditions are fulfilled:

a) The prefix implies Γ, that is, $\gamma^- \Rightarrow \Gamma$.
b) Γ contradicts the suffix, that is, $\Gamma \wedge \gamma^+$ is contradicting.
c) Γ only contains variables occurring in *both* prefix γ^- and suffix γ^+.

It is proven that such an interpolant always exists in the domain of abstract variable assignments [9] as well as in the theory of linear arithmetics [19]. A Craig interpolant describes information that is sufficient for proving a remaining path, i.e., the suffix, infeasible (contradicting). This information can be used to derive a new precision for abstraction refinement.

Refinement Selection. Usually, several different Craig interpolants exist for a single infeasible path. Each of them may represent a different reason for infeasibility. When using interpolants for abstraction refinement in CEGAR, the choice of interpolant for an infeasible path, and as such the tracked reason, may greatly influence the further course of the analysis. Traditional abstraction refinement does not account for the differences between these interpolants and just takes arbitrarily the interpolants that the interpolation engine returns (based on the heuristics inside the interpolation engine). In contrast, *refinement selection* [11] tries to select the interpolant that promises the best verification progress for a given infeasible path. It looks at various possible interpolants, e.g., by using sliced path prefixes [12], and chooses the most promising one based on a selected heuristic. Some heuristics proposed in other work [11] include:

- Selection of the shortest prefix (called *short*).
- Selection based on a score computed from the domain type [2] of program variables, with easy/small types like boolean and integer being preferred (called *domain good*).
- Selection of the interpolant with the most narrow width (called *width narrow*). The width of an interpolant is defined by the number of locations on an error

path for which the interpolant is not *false* and not *true*, i.e., the number of locations at which additional information must be tracked.

Several heuristics may be applied sequentially, in case one heuristic alone is not able to choose a single best interpolant.

3 Symbolic Execution using CEGAR and Interpolation

Abstract Domain and Abstract Semantics. Our new approach SymEx$^+$ is the combination of traditional symbolic execution with Cegar. An abstract state (v, γ) in symbolic execution consists of an abstract variable assignment v and a sequence $\gamma = \langle [\rho_1], \ldots, [\rho_n] \rangle \in \widehat{\mathbb{S}}$ of constraints $[\rho_i]$ over symbolic values from $\mathbb{S}$. The abstract variable assignment $v : X \to\!\!\!\!\oplus\, \mathbb{V}$ used in symbolic execution assigns to a program variable from X either a concrete integer value from $\mathbb{Z}$, a symbolic value from $\mathbb{S}$, or the special value $\bot$, which represents a contradicting assignment, i.e., $\mathbb{V} = \mathbb{Z} \cup \mathbb{S} \cup \{\bot\}$. An abstract state (v, γ) represents the set $[\![(v, \gamma)]\!]$ of concrete variable assignments, which is formally defined as follows: $[\![\bot]\!] = \varnothing$ and

$$[\![(v,\gamma)]\!] = \{c \mid \forall x \in \mathtt{def}(v) : \; v(x) \in \mathbb{Z} \implies v(x) = c(x) \\ \wedge \exists s : \bigwedge_{[\rho] \in \gamma} \rho \wedge \; \forall x \in \mathtt{def}(v) : v(x) \in \mathbb{S} \implies v(x) = s(v(x)) = c(x) \; \}$$

where $s : \mathbb{S} \to \mathbb{Z}$ maps symbolic to concrete values. The strongest-post operator $\widehat{\mathtt{SP}}_{op} : X \times \widehat{\mathbb{S}} \to X \times \widehat{\mathbb{S}}$ is defined as follows:

1. For an assignment operation $x := exp$ we have

$$\widehat{\mathtt{SP}}_{x:=exp}((v,\gamma)) = \big(v_{|X \setminus \{x\}} \cup \{(x,y)\},\;\; \gamma\big)$$

   with

$$y = \begin{cases} d & \text{if } d \in \mathbb{Z} \cup \mathbb{S} \text{ is the evaluation of arithmetic expression } exp_{/v} \\ e & \text{if } exp_{/v} \text{ can not be evaluated and } e \text{ is a new symbolic value } e \in \mathbb{S} \end{cases}$$

   and $exp_{/v}$ is the interpretation of expression exp for the abstract variable assignment v. If exp contains a program variable of X that is not in the definition range $\mathtt{def}(v)$, then $exp_{/v}$ can not be evaluated. If $exp_{/v}$ contains a symbolic value of $\mathbb{S}$, the evaluation of $exp_{/v}$ equals $exp_{/v}$ and $exp_{/v} \in \mathbb{S}$.
2. For an assume operation $[p]$ we have

$$\widehat{\mathtt{SP}}_{[p]}((v,\gamma)) = \begin{cases} \bot & \text{if } p_{/(v,\gamma)} \text{ is unsatisfiable} \\ (v \cup v_p,\; \gamma \wedge \langle [p_{/(v \cup v_p)}] \rangle) & \text{otherwise} \end{cases}$$

   with new abstract variable assignments

$$v_p = \{(x,e) \in (X \setminus \mathtt{def}(v) \times \mathbb{S}) \mid x \text{ occurs in } p \text{ and } \\ e \text{ is a new symbolic value } e \in \mathbb{S}\}$$

that assign a new symbolic value to every unknown program variable occurring in p, the interpretation $p_{/(v \cup v_p)}$ of p for the abstract variable assignment $v \cup v_p$ and

$$p_{/(v,\gamma)} = p \wedge \bigwedge_{x \in \mathtt{def}(v)} x = v(x) \wedge \bigwedge_{[\rho] \in \gamma} \rho \ .$$

If $p_{/(v,\gamma)}$ is satisfiable, an assignment to a new symbolic value is added to the abstract variable assignment for every unknown program variable occurring in p and the assume operation $[p_{/(v \cup v_p)}]$ is appended to the existing constraints sequence.

Using these operations, the conditions ρ of the assume operations $[\rho] \in \gamma$ contain symbolic values from $\mathbb{S}$, but no program variables from X.

Precision and Interpolation. For our symbolic-execution domain, the set Π (for defining a precision) is a composition of the set X of program variables and the set $\widehat{\mathbb{S}}$ of constraint sequences, i.e., $\Pi = X \times \widehat{\mathbb{S}}$. The precision defines the program variables and the constraints that should be tracked at each location.

We base our refinement procedure for the precision of symbolic execution on the refinement procedure for the precision of abstract variable assignments [9], using Craig interpolants to derive the precision. Algorithm 2 shows our computation of interpolants for a prefix γ^- and a suffix γ^+. Since we want to create an interpolant Γ that contains all information necessary for proving that $\widehat{\mathtt{SP}}_{\Gamma \wedge \gamma^+}$ is contradicting, we have to consider not only abstract variable assignments but also constraints. First, the algorithm computes the strongest-post condition (v, γ) for the prefix γ^- based on the initial abstract state $(\varnothing, \varnothing)$. We then eliminate all constraints from γ that are not necessary for proving that γ^+ is contradicting. Next, we remove every mapping of a program variable to a value from v that is not required. This way we try to get the weakest interpolant possible. We then build the interpolant from all constraints left in γ and all assignments left in v.

Refinement of Abstract Model. Algorithm 3 defines the complete refinement procedure used in the CEGAR algorithm. It starts with an initial, empty interpolant Γ and empty precision π with $\pi(l) = (\varnothing, \varnothing)$ for all $l \in L$. For each location (l_i, op_i) on the infeasible error path, the suffix γ^+ for this location is set and the interpolant is computed from the previous interpolant in conjunction with the current operation (i.e., $\Gamma \wedge \langle op_i \rangle$) and the suffix (line 5). The full prefix $\langle op_1, \ldots, op_i \rangle$ must not be used for interpolation, because multiple reasons for the infeasibility of a path may exist; if the full prefix is available for interpolation, then different reasons for infeasibility might be used for consecutive interpolants on a path, resulting in a precision that is not able to prove the path infeasible in further analysis iterations. Because of this, inductive interpolants that are derived from the same reason for the infeasibility of the given infeasible error path must be computed by reusing the previous interpolant as part of the prefix.

In the next step, a precision for the current program location is extracted from the interpolant using `extractPrecision`. The extracted precision for symbolic execution is an object of type $X \times \widehat{\mathbb{S}}$; such a pair consists of the set of all program

Algorithm 2 $\texttt{interpolate}(\gamma^-, \gamma^+)$

Input: two constraint sequences γ^- and γ^+, with $\gamma^- \wedge \gamma^+$ contradicting
Output: a constraint sequence Γ, which is an interpolant for γ^- and γ^+
Variables: an abstract variable assignment v and a constraints sequence γ

1: $(v, \gamma) := \widehat{\mathtt{SP}}_{\gamma^-}((\varnothing, \varnothing))$
2: **for each** $[p] \in \gamma$ **do**
3: **if** $\widehat{\mathtt{SP}}_{\gamma^+}((v,\ \gamma \setminus [p]))$ is contradicting **then**
4: $\gamma := \gamma \setminus [p]$
5: **for each** $x \in \mathtt{def}(v)$ **do**
6: **if** $\widehat{\mathtt{SP}}_{\gamma^+}((v_{|\mathtt{def}(v)\setminus\{x\}},\ \gamma))$ is contradicting **then**
7: $v := v_{|\mathtt{def}(v)\setminus\{x\}}$
8: $\Gamma := \gamma$
9: **for each** $x \in \mathtt{def}(v)$ **do**
10: $\Gamma := \Gamma \wedge \langle x := v(x) \rangle$
11: **return** Γ

Algorithm 3 $\texttt{refine}(\sigma)$

Input: infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$
Output: precision $\pi : L \to X \times \widehat{\mathbb{S}}$
Variables: interpolant constraint sequence Γ

1: $\Gamma := \langle\rangle$
2: $\pi(l) := (\varnothing, \varnothing)$ for all program locations l
3: **for** $i := 1$ to $n - 1$ **do**
4: $\gamma^+ := \langle op_{i+1}, \ldots, op_n \rangle$
5: $\Gamma := \mathtt{interpolate}(\Gamma \wedge \langle op_i \rangle, \gamma^+)$
6: $\pi(l_i) := \mathtt{extractPrecision}(\Gamma)$
7: **return** π

variables that occur in an assignment operation in the interpolant and all assume operations that occur in the interpolant, formally:

$$\mathtt{extractPrecision}(\Gamma) = (\mathtt{def}(v),\ \gamma) \ ,$$

where $\widehat{\mathtt{SP}}_\Gamma(\varnothing, \varnothing) = (v, \gamma)$.

Refinement Selection. We apply refinement selection based on sliced path prefixes analogously to the application for the domain of abstract variable assignments [11,12]. In addition to the existing heuristics, we define heuristics that select an interpolant based on the amount of assumptions in it. We call the heuristic selecting the interpolant with most assumptions *assumptions – most*.

Refinement for Compositions of Abstract Domains. In the same way as demonstrated above, a composite precision of any composition of analyses can be refined and used with CEGAR. While CEGAR has been used with a composition of analyses before (c.f. [9]), the precision of only one analysis was refined in each step, first refining a less expensive analysis' precision, and only if not

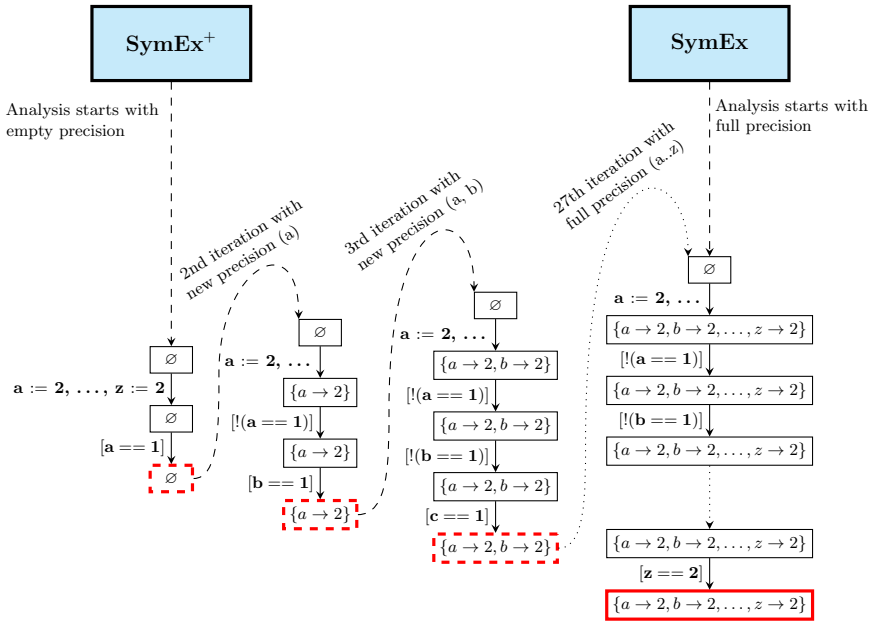


Fig. 2: An example for which the lazy analysis of SymEx$^+$ performs worse than the traditional eager analysis of SymEx. Highlighted nodes represent error locations. Every dashed rectangle represents an error location that is infeasible when using full precision.

avoidable refining a more expensive analysis' precision. Due to the nature of the previous method, only analyses without an interdependency can be refined and no information exchanged between analysis is considered. In contrast, using our new approach, any composition of analyses of arbitrary number and possibly information exchange between them can be used to extract a composite precision.

Discussion. Using Cegar, we change the symbolic execution from being eager to being lazy, while keeping its potential expressiveness. For an arbitrary verification task, it is difficult to say upfront whether the lazy or the eager approach is better suited. While a lazy approach may keep the state-space potentially smaller if only little information is necessary (many operations can be abstracted away), its refinement iterations can be time-consuming if the abstraction is not effective enough. On the other hand, an eager approach suffers from the path-explosion problem, but may stop the analysis at unreachable branches and avoid unnecessary computation. Figure 2 shows the analysis of such a program for both lazy symbolic execution with Cegar (SymEx$^+$) and traditional eager symbolic execution (called SymEx). The highlighted nodes are error locations. The program first initializes the program variables **a** to **z** with value 2. Afterwards, it checks whether program variables **a** to **y** are initialized with a value different from 1 and whether **z** is initialized with a value different from 2. If one of these conditions is wrong, an

Table 1: Comparison of different refinement-selection heuristics in SymEx$^+$

Verdict	unsolved	solved	correct TRUE	correct FALSE	incorrect TRUE	incorrect FALSE
No preference	4341	2336	1737	443	0	**156**
Domain good – width narrow	4444	2233	1702	531	0	171
Domain good – short	3906	**2771**	**2042**	567	0	162
Assumptions most – short	4028	2491	1892	**599**	0	158

error location is reached. Since all program variables are initialized with value 2 at the beginning of the program, only the last error location can be reached. Despite this, the Cegar algorithm visits one error location after the other, always refining the precision to track only one additional variable and then restarting from the beginning of the program with the adjusted precision. This lazy approach performs many computations that are unnecessary. The eager approach does not visit the infeasible target locations and reaches the only feasible property violation at the end in one single execution path.

4 Evaluation

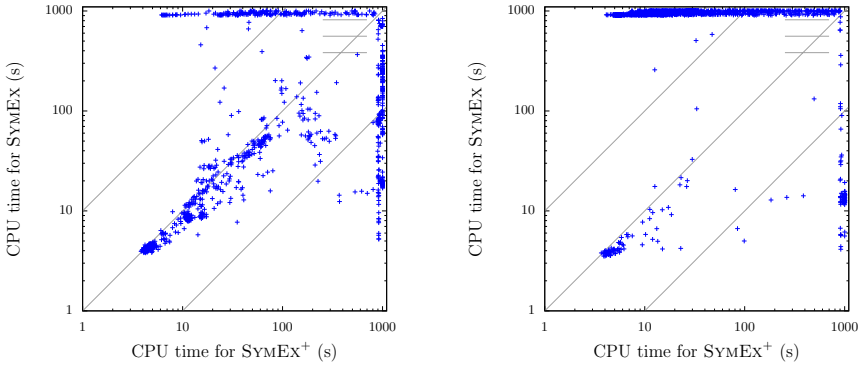
Experimental Setup. We perform our experimental evaluation on a cluster of machines with Intel Xeon E5-2650 v2 CPUs at 2.60 GHz and 135 GB of memory. Each verification task can use 2 CPU cores and 15 GB of memory. We use a time limit of 900 s of CPU time. After 1000 s, a task is terminated if it has not shut down yet. To get a statistically significant result, we run our implementation against the complete set of verification tasks [5] of the 5th International Competition on Software Verification (SV-COMP'16) [5]. To guarantee a reliable and accurate evaluation, we use BenchExec [10] to run our benchmarks. Our implementation is available in CPAchecker under tag `cpachecker-1.6-isola16`. [6]

Refinement Selection. We compare different heuristics for refinement selection to find the one suited best for our approach. Table 1 shows three selected heuristics, *domain good* combined with *width narrow*, *domain good* combined with *short*, and *assumptions most* combined with *short*. The best heuristic for proving tasks safe is *domain good – short*, raising the number of tasks that can be proven safe by 305 (which equals an increase by almost 18 %). The best heuristic for finding errors is *assumptions most – short*, which allows us to raise the number of tasks correctly found erroneous by 156 (which equals an increase by 35 %). Using one of the two heuristics performs significantly better for both proving tasks safe and finding errors, compared to using no refinement selection (*no preference*). The combination *domain good – width narrow* performs worse for proving tasks safe and better for finding errors than the use of no refinement selection.

[5] `https://github.com/sosy-lab/sv-benchmarks/tree/svcomp16`
[6] `https://svn.sosy-lab.org/software/cpachecker/tags/`

Table 2: Comparison of classical symbolic execution (SymEx) to SymEx$^+$ (both implemented in CPAchecker) and Symbiotic (an external tool)

Verdict	unsolved	solved	correct TRUE	correct FALSE	incorrect TRUE	incorrect FALSE
SymEx	5756	921	171	**634**	1	115
SymEx$^+$	3906	**2771**	**2042**	567	0	162
Symbiotic	5388	1289	769	503	2	**15**



(a) CPU time to find an error (b) CPU time to prove a program safe

Fig. 3: Runtime performance of SymEx$^+$ and SymEx in comparison

This is especially notable since for two other analyses, predicate analysis and explicit value analysis, the combination *domain good – width narrow* yields the best results compared to other heuristics [11]. This shows that the best choice of a heuristic not only depends on the task, but also on the analysis that is used.

Comparison to Other Tools. We compare our implementation to the implementation of symbolic execution in CPAchecker that does not use Cegar (SymEx) and to the mature symbolic-execution tool Symbiotic 3 [16] in version 3.0.1 (Symbiotic participated in SV-COMP in 2013, 2014 and 2016 [3,4,5]). For this evaluation, we use SymEx$^+$ with refinement selection, using the heuristics *domain good – short*. Our benchmarks show the competitiveness of SymEx$^+$ (Table 2). Figure 3a underlines the already mentioned contrast between eager SymEx and lazy SymEx$^+$. It shows the CPU time required for both approaches to find a (possibly non-existent) error. For a significant amount of tasks in our task set, only one of SymEx and SymEx$^+$ is able to find an error within 900 seconds. These cases are represented by the points at the right border (SymEx$^+$ reaches the time limit) and upper border (SymEx reaches the time limit) of the plot. For proving the safety of a program, SymEx$^+$ performs significantly better, showing bad performance for only few programs, due to its laziness (Fig. 3b). The high

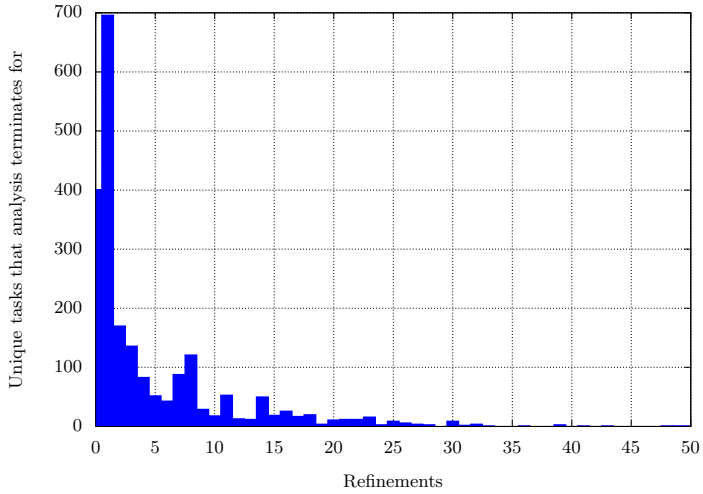


Fig. 4: Number of refinements performed by SymEx$^+$ for tasks that it can solve correctly and SymEx can not. Note that relatively few refinements are necessary for most of these tasks.

Table 3: Difference between tools for proving tasks safe. Each value describes the number of tasks that the tool on the left can correctly prove safe and the tool on the top can not.

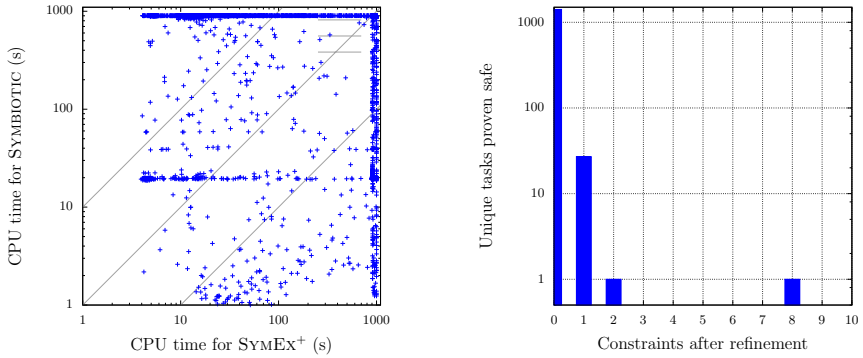
	¬ SymEx	¬ SymEx$^+$	¬ Symbiotic		correct TRUE
SymEx	-	83	62	of	171
SymEx$^+$	1954	-	1444	of	2042
Symbiotic	660	171	-	of	769

precision of eager symbolic execution is often unnecessary to correctly decide whether a program is safe or unsafe. This is underlined by the number of refinements that are necessary for SymEx$^+$ to analyze a task. For most tasks for which SymEx$^+$ is able to compute a result for, and for which SymEx is not able to, a small number of refinements are necessary (Fig. 4). For an unsafe program, this implies that no or only few infeasible error paths have to be explored before a feasible error path is found (which eager analysis could not explore in the time limit at all). For a safe program, this implies that only few information must be tracked to prove all error paths infeasible.

Tables 3 and 4 show the number of tasks that one tool can solve while another can not. As already shown, the higher performance of SymEx$^+$ for many tasks in comparison to SymEx results in more tasks that can be successfully solved within the time limit. But due to the existing limitations of our analysis (e.g., pointer arithmetic), some of these are bound to be wrong, resulting in more tasks

Table 4: Difference between tools for finding errors in tasks. Each value describes the number of tasks that the tool on the left correctly finds unsafe and the tool on the top does not.

	¬ SymEx	¬ SymEx$^+$	¬ Symbiotic		correct FALSE
SymEx	-	213	318	of	634
SymEx$^+$	146	-	302	of	567
Symbiotic	187	238	-	of	503



(a) CPU time to analyze a task

(b) Number of constraints necessary to prove tasks that Symbiotic can not prove

Fig. 5: Comparison of SymEx$^+$ and Symbiotic

that are incorrectly declared as FALSE. Compared to Symbiotic, the difference in solved tasks is even higher than compared to SymEx, which can be accounted to its different optimizations and its implementation outside of CPAchecker. It is obvious that the strengths of both analyses are different, as they follow a lazy and an eager approach. Figure 5a illustrates this. It displays the CPU time that each analysis takes for every task of our task set. It can be seen that both Symbiotic and SymEx$^+$ have significantly different behavior for the same tasks. Because of its laziness, SymEx$^+$ is still able to correctly prove a significant amount of more tasks safe and declare a few more tasks unsafe in the given environment. For most of the safe tasks, no constraints on symbolic values have to be tracked at all (Fig. 5b). Thanks to Cegar, SymEx$^+$ ignores these unnecessary constraints and keeps the state-space small.

5 Conclusion

By transferring the lazy approach of CEGAR to the domain of symbolic execution, we were able to mitigate the path-explosion problem of symbolic execution considerably. We implemented our proposed concepts in the open-source verification framework CPACHECKER and created a generic refinement procedure based on Craig interpolants which allows compositional refinement of precisions that is independent from the analyses' domain. In addition, we applied refinement selection based on sliced path prefixes and implemented new heuristics for it. Our evaluation shows the significant improvement that can be gained by using CEGAR with refinement selection and the impact that different heuristics can have on the analysis. By comparing our implementation with an implementation of the classical approach within the same tool, and with the external symbolic execution tool SYMBIOTIC, we were able to illustrate the differences between eager and lazy approaches. Our experimental study shows the competitiveness of our proposed concepts on a representative task set. Given the many existing orthogonal approaches to mitigate the path-explosion problem of symbolic execution, future work could focus on combining SYMEX$^+$ with suitable other approaches and evaluating their impact on our approach.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. TACAS*, LNCS 4963, pages 368–381. Springer, 2008.
2. S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. v. Rhein. Domain types: Abstract-domain selection based on variable usage. In *Proc. HVC*, LNCS 8244, pages 262–278. Springer, 2013.
3. D. Beyer. Second competition on software verification. In *Proc. TACAS*, LNCS 7795, pages 594–609. Springer, 2013.
4. D. Beyer. Status report on software verification. In *Proc. TACAS*, LNCS 8413, pages 373–388. Springer, 2014.
5. D. Beyer. Reliable and reproducible competition results with BENCHEXEC and witnesses. In *Proc. TACAS*, LNCS 9636, pages 887–904. Springer, 2016.
6. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.
7. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
8. D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
9. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
10. D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015.
11. D. Beyer, S. Löwe, and P. Wendler. Refinement selection. In *Proc. SPIN*, LNCS 9232, pages 20–38. Springer, 2015.

12. D. Beyer, S. Löwe, and P. Wendler. Sliced path prefixes: An effective method to enable refinement selection. In *Proc. FORTE*, LNCS 9039, pages 228–243. Springer, 2015.
13. D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. IMPACT. In *Proc. FMCAD*, pages 106–113. FMCAD, 2012.
14. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446. IEEE, 2008.
15. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX*, volume 8, pages 209–224. USENIX, 2008.
16. Marek Chalupa, Martin Jon, Jiri Slaby, Jan Strejek, and Martina Vitovsk. SYMBIOTIC 3: New slicer and error-witness generation (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 946–949. Springer, 2016.
17. D. Chu, J. Jaffar, and V. Murali. Lazy symbolic execution for enhanced learning. In *Proc. RV*, LNCS 8734, pages 323–339. Springer, 2014.
18. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
19. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
20. P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54. ACM, 2007.
21. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
23. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) language and system. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 14(3):339–395, 1992.
24. J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proc. FSE*, page 48. ACM, 2013.
25. J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded symbolic execution for program verification. In *Proc. RV*, LNCS 7186, pages 396–411. Springer, 2012.
26. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Proc. CP*, LNCS 5732, pages 454–469. Springer, 2009.
27. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
28. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
29. K. L. McMillan. Lazy annotation for program testing and verification. In *Proc. CAV*, LNCS 6174, pages 104–118. Springer, 2010.
30. J. Slaby, J. Strejcek, and M. Trtik. Compact symbolic execution. In *Proc. ATVA*, LNCS 8172, pages 193–207. Springer, 2013.