Formal Methods in Computer-Aided Design 2024

Combining Symbolic Execution with Predicate Abstraction and CEGAR

Martin Jonáš
Masaryk University
Brno, Czechia
martin.jonas@mail.muni.cz

Jan Strejček
Masaryk University
Brno, Czechia
strejcek@fi.muni.cz

Alberto Griggio
Fondazione Bruno Kessler
Trento, Italy
griggio@fbk.eu

***Abstract*—The paper presents a simple, yet effective program verification technique that combines symbolic execution with implicit predicate abstraction and CEGAR. The technique can prove correctness of many programs that are beyond the reach of the standard symbolic execution because their symbolic execution tree is prohibitively large or even infinite. The technique has been implemented in the software model checker KRATOS. Our experimental evaluation shows that it also decides correctness of some programs that were decided neither by the standard symbolic execution nor by IC3 with predicate abstraction (all implemented in KRATOS).**

***Index Terms*—Program verification, symbolic execution, predicate abstraction, CEGAR.**

I. Introduction

Symbolic execution [Kin76] is a powerful and popular technique for static program analysis. It consists in exploring the behaviours of the program by traversing its control flow graph one path at the time, accumulating the constraints visited during such traversal in formulas called *path conditions*, which are then checked with a constraint solver (e.g., a SAT or SMT solver) for feasibility. Symbolic execution has been applied effectively to different program analysis tasks, including automated test generation [Kin76], software verification [JNS11], input filtering [CCZ+07], program debugging [QRLV12], and program repair [NQRC13], [MYR16]. Although primarily aimed at finding feasible paths satisfying a desired condition (e.g., reaching a target location, or traversing a specific set of locations), symbolic execution can also be used to prove unreachability of some error locations, by exhaustively enumerating all the feasible paths. In practice, however, this often diverges, because the number of such paths in many programs is prohibitively large or infinite (a simple example is shown in Fig. 1).

In this paper, we present a simple technique for improving the effectiveness of symbolic execution at proving unreachability. The main idea is to integrate *implicit predicate abstraction* [JM07], [Ton09] in the enumeration of paths, so as to ensure that the (abstract) symbolic execution tree is always finite (for a given set of predicates). This is done by setting up *abstraction locations* covering all program loops, assigning to each such location a finite set of *abstraction predicates*, and then restricting the symbolic exploration to *abstract simple paths*, i.e., paths in which all abstract states can occur at most once. To better control when the abstraction is applied, we also assign to each abstraction location an *abstraction threshold* saying that the abstraction is not applied in this location before the number of occurrences of the location in the current path exceeds the threshold. This can help avoiding the imprecise abstraction for loops with a small number of iterations. We show how these ideas can be integrated in a standard symbolic execution algorithm and included in a standard CEGAR loop [CGJ+03] with little effort, and demonstrate its effectiveness by evaluating our implementation in the KRATOS [GJ23] software model checker on a benchmark set obtained from the latest *Competition on Software Verification* SV-COMP [Bey24]. In particular, our results show that the new technique significantly improves the peformance of the symbolic execution engine of KRATOS on safe benchmarks (i.e., where the error location is unreachable), and it also can prove correctness of some programs that could not be verified by the other compared engines of KRATOS (within the given resource bounds), thus contributing to its overall performance on the benchmark set.

Paper outline: The rest of the paper is organized as follows. We introduce general background notions in Section II and standard symbolic execution in Section III. Our combination of symbolic execution, implicit abstraction, and CEGAR is described in Section IV. We present experimental evaluation in Section V and discuss related work in Section VI. Finally, we draw conclusions and discuss future directions in Section VII.

II. Preliminaries

Logic: We work in the setting of standard first-order logic. We use the standard notions of theory, satisfiability, and validity of a formula. For each term t and an assignment μ to variables and possibly uninterpreted function and relation symbols, $\mu(t)$ denotes the result of the evaluation of t under μ. Similarly, for a formula φ, we denote as $\mu(\varphi)$ the result of the evaluation of φ under μ. If the formula evaluates to *true*, we say that μ is a model of φ and write $\mu \models \varphi$. We assume that we work over a theory whose quantifier-free fragment

M. Jonáš and J. Strejček have been supported by the Czech Science Foundation grant GA23-06506S. A. Griggio has been supported by the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU, and by the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

https://doi.org/

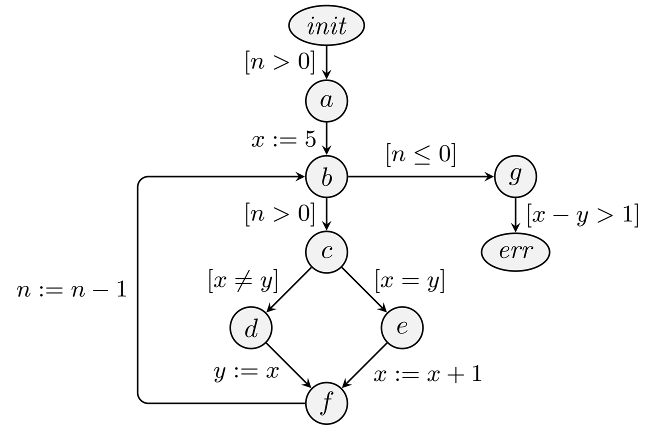


Fig. 1. A safe CFA where symbolic execution runs forever.

is decidable, i.e., there is a computable function $\textsc{IsSAT}(\varphi)$ that returns *true* if φ is satisfiable and *false* otherwise. For presentation purposes, all the examples in the paper are over the theory of linear integer arithmetic, but the concepts work for any theory with decidable quantifier-free fragment.

Mathematical notation: For each function $f\colon A \to B$ and $a \in A$, $b \in B$, we denote by $f[a \leftarrow b]$ the function that maps a to b and x to $f(x)$ for all x in $A \smallsetminus \{a\}$. The domain of a (partial) function f is denoted as $\mathrm{dom}(f)$. For each set A, we denote as A^+ the set of all non-empty sequences of elements from A. If $u, v \in A^+$, we denote their concatenation as $u.v$ (or just uv, if it is clear from the context). We denote the set of Booleans as $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$.

Programs: We consider programs represented by *control-flow automata (*CFA*)*. Let *Vars* be a fixed set of program variables. A control flow automaton is a tuple $A = (L, \mathit{init}, \mathit{err}, E)$, where L is a finite set of program locations, $\mathit{init} \in L$ is the initial location, $\mathit{err} \in L \smallsetminus \{\mathit{init}\}$ is the error location, and $E \subseteq L \times \mathit{Ops} \times (L \smallsetminus \{\mathit{init}\})$ is a finite set of edges between program locations that are labeled by operations. We assume that *init* has only outgoing edges. Each operation $o \in \mathit{Ops}$ has one of the three following forms:

1) an *assumption* $[\varphi]$, where φ is a formula over *Vars*,
2) an *assignment* $x := t$, where $x \in \mathit{Vars}$ and t is a term over *Vars*, or
3) a *nondeterministic assignment* $x := *$, where $x \in \mathit{Vars}$.

We assume that if a single location has multiple outgoing edges, all of them are assumptions. A CFA used as a running example can be found in Fig. 1.

A *(control-flow) path* π leading to a location $l_k \in L$ is a nonempty finite sequence of consecutive edges $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k) \in E^+$ where $l_1 = \mathit{init}$. The path is called *error path* if $l_k = \mathit{err}$.

A program state is a pair (l, μ), where $l \in L$ is a program location and μ is an assignment of values to program variables. There is a transition $(l, \mu) \xrightarrow{(l,o,l')} (l', \mu')$ between two states along the edge $(l, o, l') \in E$ if one of the following holds:

1) $o = [\varphi]$, $\mu \models \varphi$, and $\mu = \mu'$, or
2) $o = (x := t)$ and $\mu' = \mu[x \leftarrow \mu(t)]$, or
3) $o = (x := *)$ and $\mu'(v) = \mu(v)$ for all $v \in \mathit{Vars} \smallsetminus \{x\}$.

A path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$ is *feasible* if there exists a sequence of assignments $\mu_1, \mu_2, \ldots, \mu_k$ satisfying $(l_i, \mu_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, \mu_{i+1})$ for all $1 \leq i < k$. For example, the path $\pi = (\mathit{init}, [n > 0], a)(a, x := 5, b)$ in our running example is feasible, but the path $\pi.(b, [n \leq 0], g)$ is not due to the contradictory assumptions on the value of n.

A CFA is called *unsafe* if there is a feasible error path and it is called *safe* otherwise. In the rest of the paper, we assume that $(L, \mathit{init}, \mathit{err}, E)$ is an arbitrary fixed CFA and we are interested in proving whether the CFA is safe or unsafe. We also assume that *Vars* contains only the program variables used in the CFA.

III. Symbolic Execution

Symbolic execution [Kin76] is a technique that systematically explores all feasible paths of a given CFA. The main idea is that instead of concrete input values, symbolic execution uses variables representing arbitrary input values. Consequently, values of program variables are terms over the input variables. When symbolic execution evaluates an assumption $[\varphi]$, the program variables in φ are replaced by the corresponding terms and the resulting formula is added to the so-called *path condition*. The path condition is satisfiable if and only if the corresponding path is feasible. When the path condition becomes unsatisfiable, symbolic execution explores another path. Now we present symbolic execution formally.

Let *Inputs* be a countably infinite set of variables that represent inputs of the program. A *symbolic state* is a pair (pc, m), where pc is a formula over *Inputs* called a *path condition* and m is a *symbolic memory* which assigns to each program variable $x \in \mathit{Vars}$ a term $m(x)$ over *Inputs* that represents the current value of x. We extend m to arbitrary terms and formulas. Namely, if t is a term over *Vars*, $m(t)$ is the result of simultaneously replacing each program variable x in t by $m(x)$. Analogously, $m(\varphi)$ is the formula over *Inputs* obtained from a formula φ over *Vars* in the same way. Given a symbolic state s, by $s.pc$ we denote its path condition and by $s.m$ its symbolic memory.

We assume that there is a function *fresh*() whose every call returns a fresh variable from *Inputs* and a function *freshMem*() whose every call returns a symbolic memory that assigns to each program variable a fresh input variable.

We define a function *next* that for each symbolic state s and each operation $o \in \mathit{Ops}$ returns the successor symbolic state $\mathit{next}(s, o)$. For a state $s = (pc, m)$ and an operation o, we set

$$\mathit{next}(s, o) = \begin{cases} (pc \wedge m(\varphi), m), & \text{if } o = [\varphi], \\ (pc, m[x \leftarrow m(t)]), & \text{if } o = (x := t), \\ (pc, m[x \leftarrow \mathit{fresh}()]), & \text{if } o = (x := *). \end{cases}$$

Algorithm 1 presents standard symbolic execution formulated as a recursive function exploring the tree of all feasible paths in a depth-first manner.

The function $\textsc{Symex}(l, s, h)$ has three arguments: the current location l, the current symbolic state s, and the sequence h

Algorithm 1 Standard symbolic execution

```
function SYMEX($l, s, h$)
    $h \leftarrow h.(l, s)$                              ▷ update history
    if $l = err$ then
        return (UNSAFE, $h$)

    for $(l, o, l') \in E$ do                       ▷ for each outgoing edge
        $s' \leftarrow next(s, o)$                       ▷ successor state
        if $o$ is an assumption then
            if not ISSAT($s'.pc$) then
                continue                          ▷ the path is not feasible
        $h' \leftarrow h.o$                        ▷ new history with the operation
        if SYMEX($l', s', h'$) = (UNSAFE, $h''$) then
            return (UNSAFE, $h''$)                 ▷ feasible error path

    return SAFE          ▷ all outgoing feasible paths are safe
```

tracking the *history* of the current path (i.e., h stores the visited pairs of a location and a symbolic state interleaved with the performed operations). To symbolically execute a given CFA, we call SYMEX($init, s_0, \varepsilon$), where $s_0 = (true, freshMem())$ is a symbolic state with path condition $true$ and a fresh symbolic memory. The function first extends the history with (l, s). If the current location is err, then it returns UNSAFE and the current history representing the detected feasible error path. Otherwise, the function processes the edges leading from the current location l one by one. The operation of the edge is evaluated and if it changes the current path condition into an unsatisfiable one, the path is infeasible and we terminate its exploration. Otherwise, the operation is added to the history and symbolic execution is recursively called from the location and symbolic state after the operation. If this recursive call detects a feasible error path, the function produces the same verdict. If the recursive call finishes without finding any feasible error path, we continue with the next edge. If all edges are processed without finding any feasible error path, the function returns SAFE.

The biggest disadvantage of standard symbolic execution is its unability to show that a system with an infinite number of feasible paths is safe. This is, for example, the case of the CFA in Fig. 1: for each $j > 0$, the path going through locations $init.a(bcdfbcef)^j bg$ is feasible. Symbolic execution of such a path leads to the symbolic state with memory $m(n) = v_n - 2j$, $m(x) = 5 + j$, and $m(y) = 5 + (j - 1)$ and path condition equivalent to $v_n = 2j \wedge v_y \neq 5$, where $v_n, v_y \in Inputs$ represent the initial values of program variables n, y, respectively.

IV. Extending Symbolic Execution with Predicate Abstraction

One of the techniques used to reduce the number of states and paths of programs is *predicate abstraction* [BR02], [BHJM07]. Given a CFA A with program variables $Vars$ and a set $\mathbb{P}$ of formulas over $Vars$ called the *predicates*, the predicate abstraction is used to construct an abstract system $\widehat{A}_{\mathbb{P}}$ such that if the error state is unreachable in $\widehat{A}_{\mathbb{P}}$, it is also unreachable in the original system A. The system $\widehat{A}_{\mathbb{P}}$ has Boolean variables $Vars_{\mathbb{P}} = \{x_P \mid P \in \mathbb{P}\}$ that correspond to the predicates and its states are thus pairs $(l, \mu_{\mathbb{P}})$ of a location l and an assignment $\mu_{\mathbb{P}} \colon Vars_{\mathbb{P}} \to \mathbb{B}$ representing the current values of the predicates. There is a relation $H(\mu, \mu_{\mathbb{P}})$ between assignments to variables of the original and the abstracted system that holds if and only if $\mu(P) = \mu_{\mathbb{P}}(x_P)$ for all $P \in \mathbb{P}$. There is a transition $(l, \mu_{\mathbb{P}}) \xrightarrow{(l,o,l')} (l', \mu'_{\mathbb{P}})$ in $\widehat{A}_{\mathbb{P}}$ if and only if there exists a transition $(l, \mu) \xrightarrow{(l,o,l')} (l', \mu')$ in A such that $H(\mu, \mu_{\mathbb{P}})$ and $H(\mu', \mu'_{\mathbb{P}})$. The predicate abstraction can be further refined by assigning different sets of predicates to different program locations or by abstracting only in a subset of the locations [BKW10]. The computation of the transition relation in the abstract system is potentially expensive as it needs many SMT queries or alternative approaches with quantified SMT queries or AllSMT queries. This potentially expensive computation can be avoided by *implicit predicate abstraction* [JM07], [Ton09], where the abstraction itself is embedded in SMT queries asking for the existence of a certain path in the abstract system.

In the rest of this section, we present the main contribution of the paper, which is extending the symbolic execution with predicate abstraction and CEGAR. We do this in three conceptual steps. First, in Section IV-A we formalize the considered abstractions, define feasibility of paths in the abstract system, and introduce the *simplicity* of these paths which intuitively means that a path cannot pass the same abstract state twice. Section IV-B then presents our algorithm for symbolic execution extended with implicit predicate abstraction. Finally, Section IV-C incorporates the algorithm in a CEGAR loop that checks feasibility of the obtained abstract counterexamples and refines the abstraction.

A. Precision Function, Feasible and Simple Abstract Paths

First of all, we define *precision functions* that specify where, when, and what abstraction to use. Let $\mathcal{F}$ denote the set of formulas over program variables. A *precision function* is an arbitrary partial function $p \colon L \to \mathbb{N}_0 \times \mathcal{P}_{fin}(\mathcal{F})$ that assigns to a program location l a pair $p(l) = (c, \mathbb{P})$ of a non-negative integer c called *threshold* and a finite set $\mathbb{P}$ of *predicates*. Locations in $\mathrm{dom}(p)$ are called *abstraction locations* and the abstraction will be used only there. For an abstraction location l, the value $p(l) = (c, \mathbb{P})$ says that the abstraction in location l is applied only when the current path visits l at least c times and the abstraction uses the formulas of $\mathbb{P}$ as abstraction predicates. We refer to c and $\mathbb{P}$ assigned to l by p with $p(l).c$ and $p(l).\mathbb{P}$, respectively. In the following, we always assume that p denotes some precision function.

Given a path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$, we say that l_i is an *abstraction point* on π if it is an abstraction location that appears at least $p(l_i).c$ times in $l_1, l_2, \ldots, l_{i-1}$. Given an abstraction location l, we say that two assignments μ, μ' are $p(l)$*-equivalent* if they satisfy the same predicates assigned to l by p, i.e., for each $P \in p(l).\mathbb{P}$ it holds that

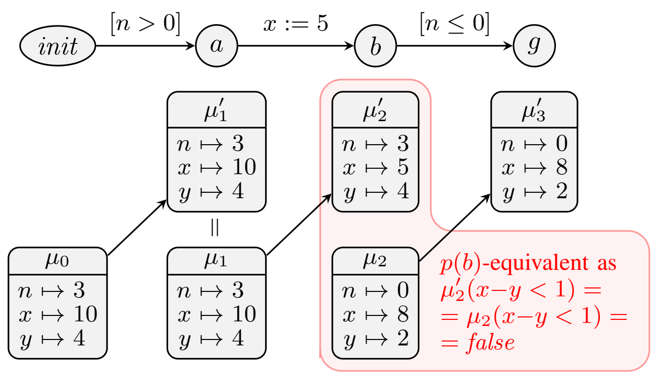


Fig. 2. A p-feasible path for $p(b) = (0, \{x - y < 1\})$ that is not feasible.

$\mu(P) = \mu'(P)$. The values $\mu(P)$ of the abstraction predicates $P \in p(l).\mathbb{P}$ form the *abstract state* associated to l.

Definition 1 (p-feasible path)**.** *We say that a control-flow path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3)\dots(l_{k-1}, o_{k-1}, l_k)$ is p-feasible if there exist assignment sequences $\mu_1, \mu_2, \dots, \mu_{k-1}$ and $\mu'_2, \mu'_3, \dots, \mu'_k$ such that for each edge (l_i, o_i, l_{i+1}) there is a transition $(l_i, \mu_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, \mu'_{i+1})$ and for each $1 < i < k$ it holds that if l_i is an abstraction point on π then μ_i, μ'_i are $p(l_i)$-equivalent and $\mu_i = \mu'_i$ otherwise.*

Theorem 1. *Each feasible path is also p-feasible for each precision function p.*

Proof. Let $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3)\dots(l_{k-1}, o_{k-1}, l_k)$ be a feasible path. As the path is feasible, there exist assignments $\mu_1, \mu_2, \dots, \mu_k$ such that $(l_i, \mu_i) \xrightarrow{(l_i,o,l_{i+1})} (l_{i+1}, \mu_{i+1})$ for each $1 \le i < k$. The path is also p-feasible as the assignment sequences $\mu_1, \mu_2, \dots, \mu_{k-1}$ and $\mu'_2 = \mu_2, \mu'_3 = \mu_3, \dots, \mu'_k = \mu_k$ clearly satisfy all the conditions in Definition 1. □

Note that the other implication does not hold. For example, the path $(init, [n > 0], a)(a, x := 5, b)(b, [n \le 0], g)$ of the running example is not feasible as mentioned in Section II, but Fig. 2 shows that it is p-feasible for precision function with $\mathrm{dom}(p) = \{b\}$ and $p(b) = (0, \{x - y > 0\})$.

It would not be useful to modify the symbolic execution to explore all p-feasible paths instead of all feasible paths as Theorem 1 implies that the number of p-feasible paths can only be higher. The key observation for our approach is that we do not have to explore *all* p-feasible paths, but only the *paths that do not contain two abstraction points with the same location and abstract state.* We call such paths p-simple. Formally, this is stated by the following definition and theorem.

Definition 2 (p-simple path)**.** *A control-flow path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3)\dots(l_{k-1}, o_{k-1}, l_k)$ is called p-simple if it is p-feasible and there exist assignment sequences from the definition of p-feasibility that additionally satisfy the following: for all $1 \le i < j \le k$ such that l_i is an abstraction point on π and $l_i = l_j$ it holds that μ_i, μ'_j are **not** $p(l_i)$-equivalent.*

Theorem 2. *If there exists a feasible error path, then for each precision function p there is a p-simple error path.*

Proof. Let $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3)\dots(l_{k-1}, o_{k-1}, l_k)$ be a feasible path leading to $l_k = err$ and p be a precision function. Because π is feasible, there exist assignments $\nu_1, \nu_2, \dots, \nu_k$ such that $(l_i, \nu_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, \nu_{i+1})$ for each $1 \le i < k$. We show by induction that for each $1 < i \le k$ there exists a path ρ leading to l_i and assignment sequences $\mu_1, \mu_2, \dots, \mu_{|\rho|}$ and $\mu'_2, \mu'_3, \dots, \mu'_{|\rho|+1}$ showing that ρ is p-simple and

1) $\nu_i = \mu'_{|\rho|+1}$ or
2) the last location on ρ is an abstraction point and $\nu_i, \mu'_{|\rho|+1}$ are $p(l_i)$-equivalent

Note that for $i = k$ this proves the statement.

Base case ($i = 2$) Consider the path $\rho = (l_1, o_1, l_2)$ and assignments $\mu_1 = \nu_1$ and $\mu'_2 = \nu_2$. The path is p-feasible as $(l_1, \mu_1) \xrightarrow{(l_1,o_1,l_2)} (l_2, \mu'_2)$. It is also p-simple as $l_1 = init$ has no incoming edges and thus $l_1 \ne l_2$.

Induction step ($i > 2$) The induction hypothesis gives us a path ρ' leading to l_{i-1} and assignment sequences $\mu_1, \mu_2, \dots, \mu_{|\rho'|}$ and $\mu'_2, \mu'_3, \dots, \mu'_{|\rho'|+1}$ showing that ρ' is p-simple and

1) $\nu_{i-1} = \mu'_{|\rho'|+1}$ or
2) the last location on ρ' is an abstraction point and $\nu_{i-1}, \mu'_{|\rho'|+1}$ are $p(l_{i-1})$-equivalent.

Consider the path $\rho'' = \rho'.(l_{i-1}, o_{i-1}, l_i)$ and the assignment sequences prolonged with $\mu_{|\rho'|+1} = \nu_{i-1}$ and $\mu'_{|\rho'|+2} = \nu_i$. Note that ρ'' is p-feasible as ρ' is p-simple, $(l_{i-1}, \mu_{|\rho'|+1}) \xrightarrow{(l_{i-1},o_{i-1},l_i)} (l_i, \mu'_{|\rho'|+2})$, and

1) $\mu_{|\rho'|+1} = \nu_{i-1} = \mu'_{|\rho'|+1}$ or
2) the last but one location on ρ'' is an abstraction point, and $\mu_{|\rho'|+1} = \nu_{i-1}, \mu'_{|\rho'|+1}$ are $p(l_{i-1})$-equivalent.

If ρ'' is also p-simple, we can simply set $\rho = \rho''$ as $\nu_i = \mu'_{|\rho'|+2} = \mu'_{|\rho''|+1}$.

If ρ'' is not p-simple, it has to be because of adding the last edge as ρ' is p-simple. Hence, there exists an abstraction point $l_{i'}$ on ρ' such that $l_{i'} = l_i$ and $\mu_{i'}, \mu'_{|\rho''|+1}$ are $p(l_i)$-equivalent. However, then we can set ρ to be the prefix of ρ' ending with $l_{i'}$. Such ρ leads to l_i, it is p-simple, its last location is an abstraction point and $\nu_i = \mu'_{|\rho'|+2} = \mu'_{|\rho''|+1}, \mu_{i'} = \mu_{|\rho|+1}$ are $p(l_i)$-equivalent. □

The important benefit of restricting the attention to p-simple paths is that for a suitable choice of the precision function p, there are only finitely many p-simple paths. In particular, we want to use a precision function p such that every cycle in the CFA contains at least one abstraction location. This is formalized by the following theorem, which will guarantee termination of the symbolic execution with predicate abstraction formulated in the next subsection.

Theorem 3. *Let p be a precision function such that each control-flow cycle contains at least one abstraction location $l \in \mathrm{dom}(p)$. Then the set of p-simple paths is finite.*

Algorithm 2 Symbolic execution with predicate abstraction

```
function SYMEXPA($l, s, h, p$)
    $h \leftarrow h.(l, s)$                                   ▷ update history
    if $l = err$ then
        return (UNSAFE, $h$)

    if ABSTRACT?($l, h, p$) then                 ▷ should we abstract?
        $m_A \leftarrow freshMem()$
        $pc_A \leftarrow s.pc \wedge eq(p(l).\mathbb{P}, s.m, m_A)$
        $pc_A \leftarrow pc_A \wedge simple(p(l).\mathbb{P}, l, m_A, h)$
        $s \leftarrow (pc_A, m_A)$

    for $(l, o, l') \in E$ do                    ▷ for each outgoing edge
        $s' \leftarrow next(s, o)$                          ▷ successor state
        if $o$ is an assumption or ABSTRACT?($l, h, p$) then
            if not ISSAT($s'.pc$) then
                continue                            ▷ the path is not p-simple
        $h' \leftarrow h.o$                   ▷ new history with the operation
        if SYMEXPA($l', s', h', p$) = (UNSAFE, $h''$) then
            return (UNSAFE, $h''$)       ▷ p-simple error path

    return SAFE      ▷ all outgoing p-simple paths are safe
```

Proof. We show that the length of each p-simple path is bounded from above by a constant. Let $L_A = \mathrm{dom}(p)$ be the set of abstraction locations, $L_N = L \smallsetminus L_A$ be the set of all non-abstraction locations, and π be a p-simple path. Since π is p-simple, it contains each abstraction location l at most $p(l).c + 2^{|p(l).\mathbb{P}|}$ times. Overall, it contains at most $b = \sum_{l \in L_A} p(l).c + 2^{|p(l).\mathbb{P}|}$ abstraction locations. Since each control-flow cycle contains at least one abstraction location, the path π does not contain more than $|L_N|$ consecutive locations from L_N. There are at most $b + 1$ consecutive segments of locations from L_N (initial, terminal, and between each two abstraction locations). The length of the path is thus at most $b + (b+1)|L_N|$. □

B. Symbolic Execution with Implicit Predicate Abstraction

The symbolic execution with predicate abstraction is computed by Algorithm 2. It is a modification of Algorithm 1 (the different parts are in red) that explores p-simple paths instead of feasible paths. In particular, Algorithm 2 builds path conditions that are satisfiable iff the corresponding path is p-simple.

The function ABSTRACT?(l, h, p) returns *true* iff l is an abstraction location that has already been visited at least $p(l).c$ times by the current path (i.e., we are in an abstraction point). If this is not the case, the next step proceeds as in the standard symbolic execution. If we are in an abstraction point with a location l and a symbolic state s, we perform the abstraction before processing the next step. The abstraction resets the symbolic memory to a fresh memory m_A. To ensure that m_A represents assignments that are $p(l)$-equivalent with assignments represented by $s.m$, we add the formula $eq(p(l).\mathbb{P}, s.m, m_A)$ to the path condition, where

$$eq(\mathbb{P}, m, m') = \bigwedge_{P \in \mathbb{P}} (m(P) \leftrightarrow m'(P)).$$

To ensure p-simplicity, we add to the path condition the formula $simple(p(l).\mathbb{P}, l, m_A, h)$ satisfied by assignments where the memory m_A is not $p(l)$-equivalent with any memory previously visited by the path in an abstraction point with the same location l. Formally, we define

$$simple(\mathbb{P}, l, m, h) = \bigwedge_{\substack{(l', s') \in aPoints(h) \\ l' = l}} \neg eq(\mathbb{P}, m, s'.m)$$

where $aPoints(h)$ is the set of abstraction points and their corresponding symbolic states in the history h, i.e., $aPoints(h)$ contains the pairs (l', s') such that l' is an abstraction location appearing in h at least $p(l).c$ times before the pair (l', s').

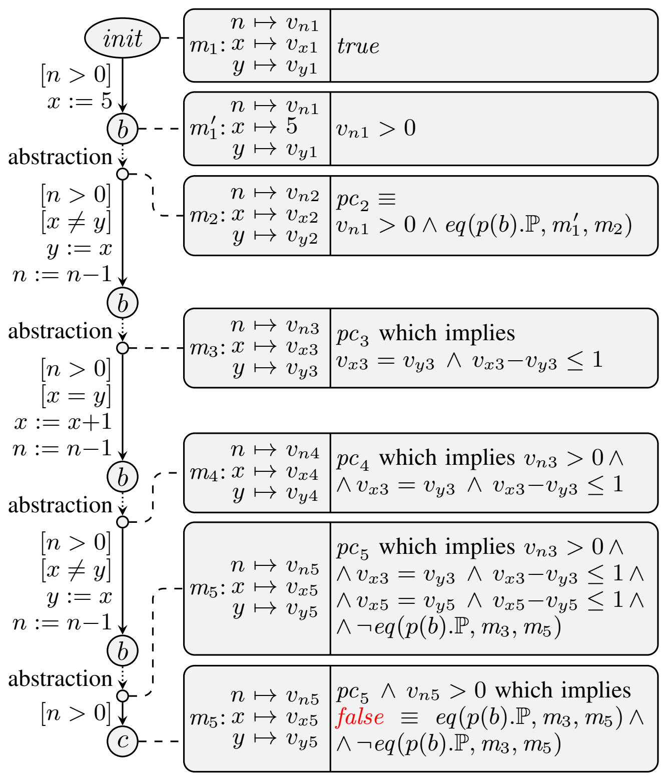
Theorem 2 implies that the algorithm is sound, i.e., if it returns SAFE, there is no feasible error path in the CFA. On the other hand, if the algorithm returns (UNSAFE, h), the p-simple error path represented by the history h can be infeasible. Theorem 3 implies that the algorithm terminates for each precision function p that defines at least one abstraction location on each control-flow cycle. This is true as there is only a finite number of p-simple paths and the algorithm checks satisfiability of the path condition that enforces p-simplicity one step after each abstraction point.

Algorithm 2 proves that our running example is correct if we use the precision function that specifies the only abstraction location b with $p(b) = (0, \{n > 0, x - y > 1, x = y\})$. We do not show the full computation due to space limits. Figure 3 sketches the computaiton along the path through locations $init.ab.cdfb.cefb.cdfb.c$ that ends with an unsatisfiable path condition meaning that the path is not p-simple. The figure fully presents the initial symbolic state, the symbolic state after the first two operations, and the symbolic state after the first abstraction (corresponding to the first values of m_A and pc_A in the algorithm). From the symbolic states after the next three abstractions and the final symbolic state we show only the symbolic memories and some imporant consequences of the path conditons.

C. Abstraction Refinement Loop

Algorithm SYMEXPA can be integrated into the standard CEGAR loop that checks the returned counterexamples for feasibility and iteratively refines the precision until SYMEXPA decides that the system is safe or a feasible error path is found. An implementation of this loop is presented in Algorithm 3. The algorithm uses three external functions:

- INITIALPRECISION() returns an initial set of abstraction locations with their thresholds and abstraction predicates, chosen either heuristically or by the user.
- ISFEASIBLE(cex) checks feasibility of the path given by cex. This can be done by performing the standard symbolic execution along the path and checking satisfiability of the path condition.



$init$
$[n > 0]$
$x := 5$
b
abstraction
$[n > 0]$
$[x \neq y]$
$y := x$
$n := n-1$
b
abstraction
$[n > 0]$
$[x = y]$
$x := x+1$
$n := n-1$
b
abstraction
$[n > 0]$
$[x \neq y]$
$y := x$
$n := n-1$
b
abstraction
$[n > 0]$
c
$m_1 : n \mapsto v_{n1}, x \mapsto v_{x1}, y \mapsto v_{y1}$ | $true$
$m_1' : n \mapsto v_{n1}, x \mapsto 5, y \mapsto v_{y1}$ | $v_{n1} > 0$
$m_2 : n \mapsto v_{n2}, x \mapsto v_{x2}, y \mapsto v_{y2}$ | $pc_2 \equiv v_{n1} > 0 \land eq(p(b).\mathbb{P}, m_1', m_2)$
$m_3 : n \mapsto v_{n3}, x \mapsto v_{x3}, y \mapsto v_{y3}$ | pc_3 which implies $v_{x3} = v_{y3} \land v_{x3} - v_{y3} \leq 1$
$m_4 : n \mapsto v_{n4}, x \mapsto v_{x4}, y \mapsto v_{y4}$ | pc_4 which implies $v_{n3} > 0 \land \land v_{x3} = v_{y3} \land v_{x3} - v_{y3} \leq 1$
$m_5 : n \mapsto v_{n5}, x \mapsto v_{x5}, y \mapsto v_{y5}$ | pc_5 which implies $v_{n3} > 0 \land \land v_{x3} = v_{y3} \land v_{x3} - v_{y3} \leq 1 \land \land v_{x5} = v_{y5} \land v_{x5} - v_{y5} \leq 1 \land \land \neg eq(p(b).\mathbb{P}, m_3, m_5)$
$m_5 : n \mapsto v_{n5}, x \mapsto v_{x5}, y \mapsto v_{y5}$ | $pc_5 \land v_{n5} > 0$ which implies $false \equiv eq(p(b).\mathbb{P}, m_3, m_5) \land \land \neg eq(p(b).\mathbb{P}, m_3, m_5)$

Fig. 3. A sketch of the run of SYMEXPA($init$, ($true$, $freshMem()$), ε, p) on the path going through locations $init.ab.cdfb.cefb.cdfb.c$ with the precision function $p(b) = (0, \{n > 0, x - y > 1, x = y\})$

Algorithm 3 Symbolic execution with predicate abstraction and CEGAR

```
1: function SYMEXPA-CEGAR()
2:     p ← INITIALPRECISION()
3:     s ← (true, freshMemory())
4:     while SYMEXPA(init, s, ε, p) = (UNSAFE, cex) do
5:         if ISFEASIBLE(cex) then
6:             return (UNSAFE, cex)        ▷ real counterexample
7:         p ← REFINE(p, cex)
8:     return SAFE                         ▷ abstract system is safe
```

- REFINE(p, cex) generates new predicates that block the spurious counterexample. Here, we treat it as a black-box that can be implemented by any existing technique for predicate generation. As a back-up solution for the case when predicate generation fails, the abstraction threshold can be increased for all locations on the path.

Similarly to BLAST [BHJM07], Algorithm 3 can be improved by not restarting the symbolic execution from scratch after a refinement. The symbolic execution can simply backtrack to the highest location whose precision was increased and restart from there with the new precision.

V. Experimental Evaluation

A. Implementation

We implemented the algorithm proposed in Section IV-C, including the symbolic execution backtracking after a refinement, in the KRATOS [GJ23] software model checker. The changes overall amounted to 778 lines of C++ code, including new user options related to the algorithm, logging, and statistics computation. The abstraction is performed only at loop heads. Refinement is implemented by computing sequence interpolants at the loop heads from the unsatisfiable feasibility query. The implementation relies on the SMT solver MathSAT5 [CGSS13] both for constraint solving and for interpolant computation. Because the proposed technique does not support function calls, the implementation first eagerly inlines all functions (and thus does not support unbounded recursion). Note that all engines of KRATOS, including the newly implemented one, support dynamic memory by modeling the heap and pointers using the theory of arrays.

The implementation is closed-source, but the binary is publicly available for academic and non-commercial use from https://www.fi.muni.cz/~xjonas/papers/fmcad24_symexecia/.

B. Experimental setup

For evaluation, we considered all the C programs from the *ReachSafety* category of the 2024 edition of the annual software verification competition SV-COMP [Bey24]. The category consists of 11 222 C programs divided into 15 benchmark families. We compare the standard symbolic execution implemented in KRATOS (*symexec*) and its proposed extension with implicit predicate abstraction and CEGAR using initial abstraction thresholds 0, 1, and 100 (*symexecia-0*, *symexecia-1*, *symexecia-100*, respectively). As external reference points, we execute the benchmarks using IC3 with implicit predicate abstraction [CGMT16] implemented in KRATOS (*IC3IA*), symbolic execution with CEGAR implemented in CPACHECKER [BL16] (*CPA-symexec+*), and finally SYMBIOTIC 10 [JKN$^+$24] (*Symbiotic*) as a well performing participant of SV-COMP based on the state-of-the-art symbolic executor KLEE [CDE08].

The experiments were performed on several identical PCs equipped with Intel Core i7-8700 CPU @ 3.20 GHz and 32 GiB of RAM. Each execution was limited to use a single CPU core, 5 minutes of wall time, and 8 GiB of RAM. For reliable benchmarking, all experiments were executed using BENCHEXEC [BLW19].

We observed that some of the tools produced *false positives*, i.e., returned *unsafe* for benchmarks marked as *safe*. In particular, *CPA-symexec+* has 44 false positives (23 in *Arrays*, 1 in *Fuzzle*, and 20 in *Heap*), *IC3IA* has 5 false positives (all in *Hardness*), *Symbiotic* has 1 false positive (in *Fuzzle*), *symexec* has 3 false positives (2 in *Hardness* and 1 in *Hardware*), and *symexecia-100* has 1 false positive (in *Hardware*). We do not consider these results in the rest of the evaluation and focus only on the correctly solved benchmarks.

C. Results

We first compare the results of the standard symbolic execution implemented in KRATOS with the proposed symbolic execution with predicate abstraction and CEGAR. The numbers of correctly solved benchmarks are shown in Table I.

TABLE I

NUMBERS OF CORRECTLY SOLVED UNSAFE (U) AND SAFE (S) BENCHMARKS BY STANDARD SYMBOLIC EXECUTION AND SYMBOLIC EXECUTION WITH PREDICATE ABSTRACTION AND CEGAR WITH VARIOUS INITIAL ABSTRACTION THRESHOLDS.

	Total		symexec		symexecia-0		symexecia-1		symexecia-100	
Family	U	S	U	S	U	S	U	S	U	S
Arrays	113	320	5	**10**	57	4	**59**	4	53	8
BitVectors	15	34	**11**	21	10	23	10	24	**11**	**27**
Combinations	430	241	**55**	**10**	0	2	0	2	4	7
ControlFlow	29	37	3	3	**4**	**15**	3	7	3	6
ECA	480	783	18	0	25	**51**	**29**	44	28	0
Floats	268	804	**39**	**202**	10	200	10	200	10	200
Fuzzle	0	15	0	0	0	0	0	0	0	0
Hardness	0	4005	0	824	0	**2315**	0	1646	0	833
Hardware	497	727	62	0	47	44	50	**49**	**71**	32
Heap	73	166	20	**52**	20	48	20	49	**21**	**52**
Loops	201	528	**114**	282	73	194	84	195	111	**286**
ProductLines	265	332	178	86	129	**156**	128	104	**213**	86
Recursive	54	102	0	0	0	0	0	0	0	0
Sequentialized	400	184	4	0	3	**4**	4	3	**6**	0
XCSP	59	60	**47**	**49**	**47**	**49**	**47**	**49**	**47**	**49**
Total	2884	8338	556	1539	425	**3105**	444	2376	**578**	1586

TABLE II

NUMBERS OF CORRECTLY SOLVED UNSAFE (U) AND SAFE (S) BENCHMARKS BY COMPETING TOOLS AND THE BEST CONFIGURATION OF SYMBOLIC EXECUTION WITH PREDICATE ABSTRACTION AND CEGAR.

	Total		CPA-symexec+		IC3IA		Symbiotic		symexecia-0	
Family	U	S	U	S	U	S	U	S	U	S
Arrays	113	320	68	1	66	1	**86**	**65**	57	4
BitVectors	15	34	11	11	12	**27**	**13**	16	10	23
Combinations	430	241	122	0	60	**24**	**211**	0	0	2
ControlFlow	29	37	9	**15**	4	**15**	**18**	4	4	**15**
ECA	480	783	38	279	145	**347**	**270**	0	25	51
Floats	268	804	**33**	156	31	78	21	**335**	10	200
Fuzzle	0	15	0	0	0	0	0	0	0	0
Hardness	0	4005	0	6	0	**3160**	0	98	0	2315
Hardware	497	727	38	9	**114**	**202**	48	0	47	44
Heap	73	166	50	26	20	46	**70**	**119**	20	48
Loops	201	528	110	110	57	130	**132**	**306**	73	194
ProductLines	265	332	128	271	249	**286**	**265**	94	129	156
Recursive	54	102	0	1	0	0	**50**	**44**	0	0
Sequentialized	400	184	82	7	7	10	**244**	**51**	3	4
XCSP	59	60	11	0	46	**50**	38	**50**	**47**	49
Total	2884	8338	700	892	811	**4376**	**1466**	1182	425	3105

The proposed technique significantly improves the number of decided *safe* benchmarks (1539 vs 3105 with initial threshold 0) and also slightly improves the number of decided *unsafe* benchmarks (556 vs 578 with initial threshold 100). The improvements occur among multiple benchmark families. Different initial abstraction thresholds provide different benefits and downsides (see for example *safe* benchmarks from *BitVectors* and *Loops* or *unsafe* benchmarks from *ProductLines*). Intuitively, the chosen thresholds determine the numbers of loop unrollings after which the abstraction is applied. Therefore, if some loops of the program need only a small number of iterations, a higher threshold allows exploring them precisely by the standard symbolic execution without applying the abstraction. The experiments show that this might be cheaper and beneficial in some cases.

Out of the 3530 benchmarks decided by *symexecia-0*, 2673 were decided without any refinements. Additionally, 149 benchmarks were decided after 1 refinement and required at most 8 predicates per abstraction location, 52 after 2 refinements with at most 28 predicates, 114 after 3 refinements with at most 14 predicates, 69 after 4 refinements with at most 101 predicates. The remaining 473 benchmarks required at least 5 refinements and at most 204 predicates per location.

Table II presents a comparison of the best-performing configuration of our algorithm, *symexecia-0*, with other competing tools. It can be seen that our algorithm outperforms other symbolic-execution-based competitors, *Symbiotic* and *CPA-symexec+*, on several families of *safe* benchmarks and also on some families of *unsafe* benchmarks. On the other hand, *symexecia-0* is outperformed by the other engine of KRATOS, *IC3IA*. However, we note that *symexecia* can be easily implemented into virtually any existing symbolic execution engine, whereas this is not the case of *IC3IA* as it uses a significantly different and more complex algorithm.

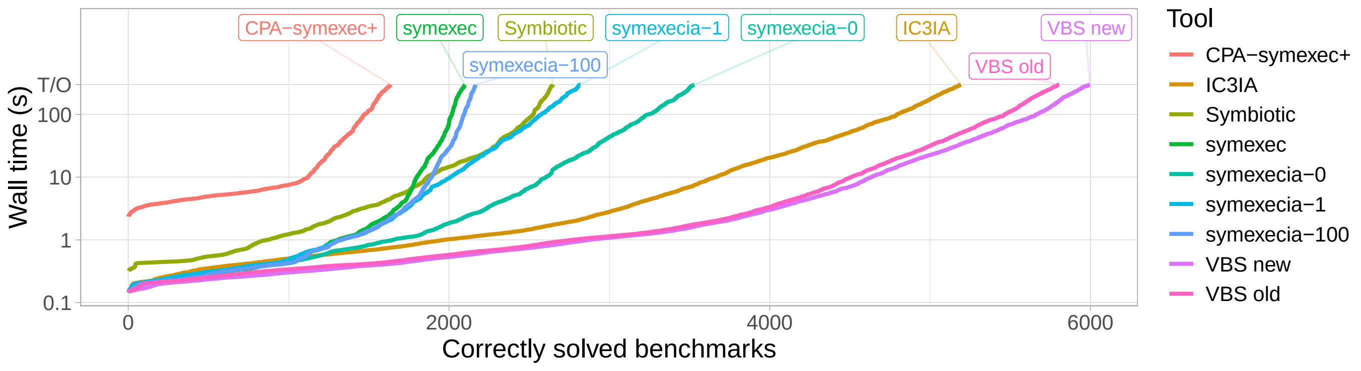


Fig. 4. The cactus plot of benchmarks solved by each tool. For each tool, the corresponding line shows the number of benchmarks (x-axis) solved in at most the given number of seconds of wall time (y-axis).

TABLE III
NUMBERS OF CORRECTLY SOLVED UNSAFE (U) AND SAFE (S) BENCHMARKS BY VIRTUAL-BEST SOLVER OF IC3IA + SYMEXEC IMPLEMENTED IN KRATOS AND THE VIRTUAL-BEST SOLVER OF IC3IA + SYMEXEC + ALL VARIANTS OF SYMEXECIA.

Family	VBS old U	VBS old S	VBS new U	VBS new S	Gain
Arrays	71	11	**72**	**12**	2
BitVectors	12	31	12	31	0
Combinations	68	26	68	**27**	1
ControlFlow	5	15	5	15	0
ECA	156	347	**159**	347	3
Floats	41	225	41	225	0
Fuzzle	0	0	0	0	0
Hardness	0	3208	0	**3369**	161
Hardware	134	202	134	202	0
Heap	22	52	22	52	0
Loops	127	358	**129**	**368**	12
ProductLines	258	315	258	**327**	12
Recursive	0	0	0	0	0
Sequentialized	7	10	7	10	0
XCSP	47	50	47	50	0
Total	948	4850	**954**	**5035**	191

To see whether the proposed technique brings any additional benefit to KRATOS compared to a simple parallel portfolio combination of predicate abstraction implemented in *IC3IA* and standard symbolic execution, we also compare the virtual-best solver composed of *IC3IA+symexec* (denoted as *VBS old*) and the virtual-best solver that also includes all variants of *symexecia-** (denoted as *VBS new*). The results are shown in Table III. *Symexecia* brings 6 newly solved *unsafe* benchmarks and 185 *safe* benchmarks across multiple benchmark families.

We also compared the runtime of all the tools. The number of solved benchmarks depending on the time-out is presented in the cactus plot in Figure 4. The plot supports all of the quantitative results from the tables and the previous paragraphs.

Additional plots and tables and all the logfiles from our experiments and scripts used for their analysis can be found at https://www.fi.muni.cz/~xjonas/papers/fmcad24_symexecia/.

Overall, despite its simplicity, our algorithm significantly outperforms symbolic-execution-based competitors on *safe* benchmarks and can solve benchmarks that can be solved neither by standard symbolic execution nor by more advanced approaches as IC3 with predicate abstraction.

VI. Related Work

Our procedure can be seen as an instance of a more general family of techniques combining exploration of CFA paths with abstraction and refinement, using (lazy) predicate abstraction [BHJM07], [BKW10] and/or interpolants [McM06], [McM10], [BW12], possibly combined with symbolic execution and invariant inference [JNS11], [McM10]. All such approaches work by constructing abstract reachability graphs, in which nodes correspond to abstract states representing an overapproximation of states that are reachable by following some specific program paths, and rely on node coverage, i.e., showing that all the states represented by a given node n are contained within the states represented by a previously-visited node m, to ensure that the constructed abstract graph is finite. Our method, on the other hand, does not require the explicit computation of abstract states, and it relies only on (abstract) simple path constraints for making the abstract space finite. This is conceptually much simpler to integrate in a standard symbolic execution algorithm than approaches based on abstract states and covering such as [JNS11]; it should however be acknowledged that the technique of [JNS11] can potentially result in more compact abstract graphs.

There are other techniques that combine symbolic execution and abstraction, but in a different way and with a different aim than our procedure. For example, [APV09] extends symbolic execution with memory abstraction, but explicitly stores the visited abstract states, computes underapproximations of feasible paths, the abstract domain is fixed beforehand, there is no refinement, and the technique requires a dedicated algorithm for subsumption check. In [RMV09], the authors propose to use the abstract counterexample obtained by other means to guide the computation of standard symbolic execution on the original program towards the error location.

Another recent technique for combining symbolic execution with CEGAR and interpolation-based refinement is proposed in [BL16]. The difference with our approach is that in [BL16] the abstraction consists in tracking only a subset of the program variables and CFA constraints precisely (with interpolation-based refinement used to increase the set of variables and constraints to track), but the core symbolic execution algorithm is not modified; in particular, the technique does not guarantee that only finite abstract spaces are explored during each iteration of the CEGAR loop.

VII. Conclusions and Future Work

We presented a program verification technique that combines symbolic execution with implicit predicate abstraction and CEGAR, and implemented it the software model checker KRATOS. Our experimental evaluation showed that, despite its simplicity, the technique is effective in improving not only the proving capabilities of symbolic execution, but also the overall performance of KRATOS, by solving some benchmarks that could not be decided by its other verification engines.

As future work, we plan to extend the technique with interprocedural analysis, i.e., to handle function calls without relying on inlining. We also want to investigate additional ways of computing predicates during refinement, instead of relying on interpolation, and additional strategies for exploring the symbolic execution tree besides the current depth-first search.

References

[APV09] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.*, 11(1):53–67, 2009.

[Bey24] Dirk Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In *TACAS (3)*, volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *Int. J. Softw. Tools Technol. Transf.*, 9(5-6):505–525, 2007.

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *FMCAD*, pages 189–197. IEEE, 2010.

[BL16] Dirk Beyer and Thomas Lemberger. Symbolic execution with CEGAR. In *ISoLA (1)*, volume 9952 of *Lecture Notes in Computer Science*, pages 195–211, 2016.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In John Launchbury and John C. Mitchell, editors, *POPL*, pages 1–3. ACM, 2002.

[BW12] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. In *FMCAD*, pages 106–113. IEEE, 2012.

[CCZ$^+$07] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *OSR*, pages 117–130. ACM, 2007.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGMT16] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods Syst. Des.*, 49(3):190–218, 2016.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[GJ23] Alberto Griggio and Martin Jonáš. Kratos2: An SMT-based model checker for imperative programs. In *CAV (3)*, volume 13966 of *Lecture Notes in Computer Science*, pages 423–436. Springer, 2023.

[JKN$^+$24] Martin Jonáš, Kristián Kumor, Jakub Novák, Jindřich Sedláček, Marek Trtík, Lukáš Zaoral, Paulína Ayaziová, and Jan Strejček. Symbiotic 10: Lazy memory initialization and compact symbolic execution - (competition contribution). In *TACAS (3)*, volume 14572 of *Lecture Notes in Computer Science*, pages 406–411. Springer, 2024.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. *Log. Methods Comput. Sci.*, 3(4), 2007.

[JNS11] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded symbolic execution for program verification. In *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2011.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[McM10] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, pages 691–701. IEEE, 2016.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *ICSE*, pages 772–781. IEEE Press, 2013.

[QRLV12] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *TOSEM*, page 19, 2012.

[RMV09] Neha Rungta, Eric G. Mercer, and Willem Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In Corina S. Pasareanu, editor, *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*, volume 5578 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2009.

[Ton09] S. Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM*, pages 89–105, 2009.