PANDA: Simultaneous Predicate Abstraction and Concrete Execution

Jakub Daniel and Pavel Parízek

Charles University in Prague, Faculty of Mathematics and Physics,
Department of Distributed and Dependable Systems

Abstract. We present a new verification algorithm, PANDA, that combines predicate abstraction with concrete execution and dynamic analysis. Both the concrete and abstract state spaces of an input program are traversed simultaneously, guiding each other through on-the-fly mutual interaction. PANDA performs dynamic on-the-fly pruning of those branches in the abstract state space that diverge from the corresponding concrete trace. If the abstract branch is actually feasible for a different concrete trace, PANDA discovers the covering trace by exploring different data choices. Candidate spurious errors may also arise, for example, due to overapproximation of the points-to relation between heap objects. We eliminate all the spurious errors using the well-known approach based on lazy abstraction refinement with interpolants. Results of experiments with our prototype implementation show that PANDA can successfully verify programs that feature loops, recursion, and manipulation with objects and arrays. It has a competitive performance and does not report any spurious error for our benchmarks.

1 Introduction

Program verification techniques based on predicate abstraction and iterative refinement have been the subject of extensive research. The set of popular approaches includes counterexample-guided abstraction refinement (CEGAR) [11] and lazy abstraction with interpolants [1,16,18], which are implemented in tools such as BLAST [6] and CPACHECKER [8]. Although these approaches are successful in verifying programs with predominantly acyclic control-flow, programs containing loops with many iterations and programs with arrays pose a challenge to them. The initial abstraction is usually too coarse to capture only the feasible executions of a loop. Therefore, these kinds of approaches are forced to repeatedly refine the abstraction and effectively unroll the loop. Many of the unrollings are incomplete, and the corresponding traces are spurious because they exit the loop prematurely.

Each step of abstraction refinement is considerably costly because it usually involves expensive SMT calls, and therefore use of refinement makes a verification procedure rather inefficient in this setting. More recent techniques (e.g., SMASH [14]) complement the abstraction refinement with some kind of underapproximating analysis (e.g., testing) in order to rule out spurious traces and to focus directly on the complete unrollings with proper number of loop iterations. Some of the recent approaches, such as DASH [5] implemented in the tool YOGI [20], in fact alternate between predicate

abstraction and concrete execution. Tests always explore a feasible number of loop iterations, and therefore spurious traces that would otherwise cause refinement are avoided, saving many calls to SMT. In general, the combination of abstraction with testing preserves the benefits of each approach while mitigating their respective weaknesses.

Example 1. Consider the small example program in Figure 1. The function findGreater searches the array a of integer values and returns the index of the first value that is greater than t. If no such value is present in a, then the length of the array is returned instead. The program further contains a procedure main that asserts the correct behavior of findGreater. The function loadUnknownArray creates an array of arbitrary integer values with a statically given length and stores it into the variable a. After the call of findGreater(a, 10), the procedure main asserts the desired property of the returned value.

```
1  void main(String[] args) {
2      int[] a = loadUnknownArray();
3      int i = findGreater(a, 10);
4      assert i == a.length || a[i] > 10;
5  }
6
7  int findGreater(int[] a, t) {
8      for (int j = 0; j < a.length; j++) {
9          if (a[j] > t) return j;
10     }
11     return a.length;
12 }
```

Figure 1. Example program

Both CEGAR and lazy abstraction, as implemented for example in BLAST [6], would struggle analyzing the loop at lines 8-10 in Figure 1 provided the array was large enough. They would iteratively discover spurious traces that exit the loop prematurely, and rule out the traces one by one in separate refinement steps by deriving predicates that relate j to a specific constant. On the other hand, approaches like DASH employ testing in order to find the correct number of loop iterations. A run of a test always represents a feasible execution and thus never yields spurious behavior. Furthermore, concrete execution is typically cheap because it does not use expensive SMT calls.

Based on the same observations, we introduce a new technique that combines predicate abstraction with concrete execution. We propose a verification algorithm PANDA, which performs abstract state space traversal that is augmented with simultaneous concrete execution in order to eliminate spurious abstract traces on-the-fly. The predicate abstraction and the concrete execution guide each other during the traversal. Usage of concrete execution enables PANDA to faithfully capture the behavior of programs written in mainstream object-oriented languages, and to support features of such programs that are hard to model with abstraction predicates. It also allows PANDA to prune infea-

sible abstract traces that arise due to the overapproximating predicate abstraction, and thus greatly reduces the number of necessary refinement steps.

The state space is constructed on-the-fly during the systematic traversal by unrolling the control-flow graph of the program. In each state, all possible outgoing transitions are determined using the overapproximate abstract information, and then every transition is explored using both concrete and abstract execution. The complete reachable state space of a given program is covered in this way.

Although pruning based on concrete execution eliminates some spurious traces, it may not prune everything for two reasons: (1) consistency between abstract and concrete executions is checked only locally, and (2) concrete execution still allows for nondeterminism (see Section 3). Therefore, the state space traversal procedure may still report a spurious error. To address this problem, PANDA uses the well-known approach of lazy abstraction with iterative refinement that is based on interpolants computed for the spurious counterexample [16].

In the case of our example program, PANDA eliminates the traces that are spurious due to an infeasible number of the loop body unrollings (at line 8) without resorting to iterative refinement. The algorithm explores all the feasible traces — more specifically, one trace returning from the function findGreater at line 9 for every value of the index j between 0 and the length of the array a, and one trace returning from findGreater through line 11.

We implemented the PANDA approach in a tool with the same name. Unlike most of the tools we target Java and not C. PANDA builds on concrete state space traversal provided by Java Pathfinder [25], and simultaneously computes predicate abstraction in such a way that the systematic exploration of a concrete state space and the predicate abstraction can interact. We also performed experimental evaluation of PANDA on small examples from our previous work [21] and benchmarks taken from the Competition on Software Verification [26], and compared its performance with other tools. Results show that the proposed approach is promising — our prototype implementation has a competitive performance and does not report spurious errors.

2 Preliminaries

Here we define more formally basic concepts that are used in the rest of this paper, and the important terminology.

Program. We model programs using control-flow automata (CFA). A *program* P is a tuple $(\mathcal{C}, l_{init}, l_{err})$, where $\mathcal{C}$ is a set of control-flow automata representing individual methods in the program, l_{init} is the initial location of the whole program, and l_{err} is the error location. The *control-flow automaton* C for a method m is a tuple (L, A, l_{en}) that encodes a directed graph with a single root node and labeled edges. Nodes of the graph correspond to the set L of program locations in the method m, and edges correspond to the set A of actions between locations. An action $a \in A$ from the location l to the location l', written as (l, a, l'), is represented by a graph edge that is labeled with the program statement corresponding to a. The location $l_{en} \in L$ is the *entry point* of the method m. We use the symbol $vars(C)$ to denote the set of local variables that appear in statements that correspond to actions of the control-flow automaton C.

The initial location l_{init} of the whole program corresponds to the entry point l_{en} of some method m_{init}, which is modeled by $C_{init} \in \mathcal{C}$. Any two distinct CFA's may have only the error location l_{err} in common. It is the destination location of every action that triggers a possible runtime error.

A program statement can be either an assume, an assignment, a procedure call, or a return from the current procedure. The assume statements are used to model the intra-procedural control flow, such as branching and loops. If there are more actions defined at one location, they all have to be assume statements. We allow only variables (fields, array elements) of an integer type and references to heap objects.

Abstraction. The symbol abs denotes a global mapping from program locations to sets of abstraction predicates. For a given location l, the set $abs(l)$ contains all predicates associated with the location l, i.e. the set of predicates whose scope includes l.

States. A *program state* s is a pair $(\mathcal{H}, \mathcal{S})$, where $\mathcal{H}$ denotes the heap and $\mathcal{S}$ is the call stack. The heap $\mathcal{H}$ is a directed graph. Inner nodes of the graph represent objects, classes, and arrays. Leaf nodes are associated with the concrete values of object fields and array elements that have an integer type. In general, edges in the graph capture the points-to relation between heap objects, and associate objects with values of their fields, respectively arrays with values of their elements. An edge (o, f, v) connects a node that represents a heap object o with a node that represents the possible value v of the field f. Similarly, an edge (o, n, v) connects a node that represents an array object o with a node that represents the value v of an element with the index n.

The call stack is a sequence of tuples (l_i, σ_i, Φ_i) that represent method frames. The symbol l_i denotes the current program location within the corresponding method, σ_i is the assignment of values to all local variables, and Φ_i is the valuation of all abstraction predicates in $abs(l_i)$. Possible values of each predicate are $\bot$, $\top$, and $*$ representing *false*, *true*, and *unknown*, respectively.

We assume that a program P has a single initial concrete state, and reads input from the environment during its execution. The initial state s_0 has an empty heap and stack with a single frame. This frame contains the initial location l_{init} where the program execution starts, initial values σ_{init} of local variables in the scope of the entry CFA, and the initial valuation Φ_{init} of abstraction predicates (i.e., *unknown*). More formally, $\sigma_{init} = \{v \mapsto 0 \mid v \in vars(C_{init})\}$ and $\Phi_{init} = \{p \mapsto * \mid p \in abs(l_{init})\}$.

Reachability graph. We use reachability graphs, defined over the set S of program states and the set T of transitions between states, to model the program behavior and state space. A *reachability graph* $R(P)$ for the program P is a possibly infinite directed graph $R = (S, T)$. A transition $\tau \in T$ is an edge (s, a, s') labeled with action a in some CFA. We use a single monolithic reachability graph for the whole program P. It is constructed inductively as fixpoint of a monotonic sequence $R_i, i \geq 0$ of finite approximations, which starts in $R_0 = (\{s_0\}, \{\})$. An approximation R_{k+1} extends R_k with a new state s' and a transition $\tau = (s, a, s')$ such that s is a state already present in R_k but unexpanded and a is the action executed by τ. The order of state expansions can be arbitrary, although we use only the depth-first order in this paper for simplicity.

Note that $R(P)$ is always specific to a given set of abstraction predicates. The sets S and T, and therefore also the shape of the reachability graph, are changed upon

refinement. For brevity, we use the symbol R to denote a finite approximation of $R(P)$ in the rest of the paper.

Alternative interpretations. Each statement of an input program P is executed both concretely and abstractly. The abstract execution of a single statement may give rise to multiple *alternative interpretations*.

The symbol $alt(R, s, a)$ denotes the set $\{s' \mid (s, a, s') \in R\}$ of all alternative interpretations for an action a in the state s in R. The set contains all the already explored transitions from s. Note that although in general there may be infinitely many alternative interpretations of an action in any given state in the entire $R(P)$, e.g. interpretations of `x = unknown()`, the set $alt(R, s, a)$ is always finite for a given approximation R and it is initially empty.

The symbol alt^* denotes the set of potential alternative interpretations that will be expanded later (in future); it is initially defined as:

$$alt^*(s,a) = \begin{cases} \emptyset & a \text{ is not an action at the current location of } s \text{ in } \mathcal{C} \\ \{s'\} & a \text{ is } \boxed{\texttt{x = unknown()}};\ s' \text{ is successor for } \boxed{\texttt{x = 0}} \\ \{s'_1, \ldots, s'_n\} & a \text{ is } \boxed{\texttt{x = e}};\ s'_i \text{ are successors for valuations of } \boxed{\texttt{e}} \\ \{s'\} & a \text{ is } \boxed{\texttt{m(x)}}, \text{ or } \boxed{\textbf{return}\ \texttt{x}};\ s' \text{ is the only successor} \\ \emptyset \text{ or } \{s'\} & a \text{ is } \boxed{\textbf{assume}\ \texttt{c}};\ s' \text{ augments } s \text{ with the condition } c \end{cases}$$

There are no interpretations defined for actions that are not enabled in the given state. The initial interpretation of `x = unknown()` is such that `x` is assigned the value 0. The interpretation of a regular assignment may not be deterministic due to heap abstraction, e.g. in the case of `x = a[i]`, and therefore we consider the set of alternative interpretations to contain all the choices. Interpretation of procedure calls and returns is straightforward and affects the call stack component of a program state. The interpretation of an assume statement depends on the valuation of the assumed condition c in the state s. If the fact is satisfiable there is one interpretation s', otherwise there is none.

The new alternatives to be expanded later are discovered on-demand, and in the majority of cases only a small finite subset of alternatives needs to be expanded. An expansion of an action a in s effectively moves the corresponding interpretation s' from $alt^*(s, a)$ to $alt(R \oplus (s, a, s'), s, a)$. For convenience, we define a set $unexp(s) = \{a \mid alt^*(s, a) \neq \emptyset\}$ of actions that are not completely expanded. We assume the presence of a special unique state s_{end}, for which the set $unexp(s_{end})$ is always empty.

Traces. An *execution trace* tr of the program P is a finite path in the reachability graph $R(P)$ that starts in s_0 and can be viewed as an alternating sequence of states and actions $(s_0, a_1, s_1, \ldots, a_n, s_n)$. Every such trace tr is associated with a *trace formula* φ_{tr} that captures the execution of the program P along the trace. The trace formula φ_{tr} is a conjunction of constraints that express the semantics and effects of all executed statements (corresponding to actions $a_1, \ldots, a_n$). Each constraint is defined using the static-single-assignment form. We say that an execution trace tr is *feasible* if the corresponding trace formula φ_{tr} is satisfiable. A trace tr that reaches the error state s_{err} is called an *error trace* or a *counterexample*.

3 PANDA Algorithm

We describe the core PANDA algorithm in the first part of this section, and then we provide more details on selected aspects in the following subsections.

The core algorithm is shown in Figure 2. It takes a program $P = (\mathcal{C}, l_{init}, l_{err})$ and the initial map abs as input, and constructs the monolithic reachability graph R for P through iterative unrolling of control-flow graphs in the set $\mathcal{C}$. Note that the map abs is usually empty at the start, but the user can provide some predicates for specific locations in this way. The reachability graph R is iteratively unrolled in the function UNROLL by means of an overloaded function $advance$ and a dual function $backtrack$ that carry out key steps of the search. When the error location l_{err} is reached by the last transition τ', the PANDA algorithm checks feasibility of the counterexample cex. If the error is real then it is reported to the user; otherwise PANDA performs abstraction refinement in order to eliminate the spurious counterexample and then restarts the state space traversal. The verification of a program terminates when all the reachable states are processed. This happens when PANDA backtracks over the initial state s_0 and the current trace tr becomes empty (line 4). Note also that the verification algorithm does not perform state matching. Our definition of the verification algorithm in Figure 2 contains several other auxiliary functions ($scopes$, $locs$, and itp) that are described later in this section, and also the function TRANS that we first explain as a black box and then provide more details in Section 3.1.

The function TRANS(R, tr, s, a) performs simultaneous concrete and abstract execution of a given action a in the state s at the end of the trace tr in R. It returns some transition $\tau' = (s, a, s')$ for some candidate successor state $s' \in alt^*(s, a)$. There is always at least one successor state, otherwise $a \notin unexp(s)$. See Section 3.1 for more details on the selection of s'. New valuation of abstraction predicates in s' after the execution of the action a is computed using the standard approach based on weakest preconditions and decision procedures. In addition, the abstract interpreter uses knowledge of the abstract heap to determine more precise valuation of predicates that capture aliasing between reference variables. However, predicates that help maintain the aliasing relation among variables still have to be introduced through refinement. Effects of the action a on the concrete part of the program state s are determined by concrete semantics of the statement corresponding to a.

A non-deterministic choice in the state space is created when the result of abstract execution of a given action a cannot be determined precisely using information from the program state. Possible sources of non-determinism include especially predicate valuations at branching statements (e.g., when the condition is *unknown*) and overapproximating points-to relation due to weak update. The effect of an assignment to a reference variable is modeled by weak update whenever the destination cannot be determined precisely, and in that case the variable may point to multiple heap objects. When processing an access to array, PANDA can make choices at two levels to consider all the possibly affected elements and values — first it has to determine all concrete indices that satisfy constraints encoded by abstraction predicates, and then for each index it has to find all the array element values based on the points-to relation.

When executing a procedure call, PANDA computes initial valuation of abstraction predicates of the new stack frame (i.e., in the callee scope) using predicates over the

actual arguments of the call. Upon return, valuation of abstraction predicates in the scope of the caller procedure is updated using valuation of predicates over the actual arguments of a reference type and predicates over the returned value.

```
 1: function PANDA(P, abs)
 2:     R ← ({s_0}, {})
 3:     tr ← (s_0)
 4:     while tr ≠ () do
 5:         (R, tr, cex) ← UNROLL(P, R, tr)
 6:         if cex ≠ ⊥ then
 7:             if REAL(cex) then return cex
 8:             abs ← REFINE(P, abs, cex)
 9:             (R, tr) ← RESET(R, tr, cex)
10:     return safe

11: function REFINE(P, abs, cex)
12:     (C, l_init, l_err) ← P
13:     for φ_scp ∈ scopes(φ_cex) do
14:         L_scp ← locs(cex, φ_scp, C)
15:         for l ∈ L_scp do
16:             p ← itp(φ_scp, φ_cex, l)
17:             if p ∉ abs(l) then
18:                 abs(l) ← abs(l) ∪ {p}
19:     return abs

20: function UNROLL(P, R, tr)
21:     (C, l_init, l_err) ← P
22:     cex ← ⊥
23:     s ← last state of tr
24:     if ∃a ∈ unexp(s) then
25:         τ' ← TRANS(R, tr, s, a)
26:         R ← advance(R, τ')
27:         tr ← advance(tr, τ')
28:     else
29:         tr ← backtrack(tr)
30:     if tr reaches l_err then
31:         cex ← tr
32:     return (R, tr, cex)

33: function REAL(cex)
34:     return is φ_cex satisfiable?

35: function RESET(R, tr, cex)
36:     return (({s_0}, {}), (s_0))
```

Figure 2. PANDA algorithm

New abstraction predicates are derived from a spurious counterexample cex in the function REFINE by the means of interpolation. We use a variant of the standard approach based on computing an interpolation-sequence [23] over the trace formula φ_{cex}. The trace formula is obtained as a conjunction of clauses that encode individual statements, heap manipulation (via read and write), and the non-deterministic choices made during their execution (e.g., choice of a concrete array index when processing statements like `x = a[i]`). In our case, interpolants are generated separately for each method call in φ_{cex}. To ensure a proper scope of interpolants within each individual method call on the given trace, PANDA uses a procedure similar to *nested interpolants* [15]. The function $scopes$ divides the whole trace formula φ_{cex} into many fragments, where each of them corresponds to the scope of execution of some method call. The function $locs$ returns a list L_{scp} of locations that appear in the given fragment of the trace formula, i.e. in the corresponding scope. Note that if a method m is executed several times in cex, then the function $scopes$ will return a separate fragment φ_{scp} for each execution of m, and similarly a location can appear multiple times in L_{scp} (e.g., due to a loop

in the code). Actual interpolants for every fragment φ_{scp} are computed by the function $itp(\varphi_{scp}, \varphi_{cex}, l)$, which calls an interpolating solver. This approach respects method call boundaries and variable scopes. In particular, interpolants generated for locations inside a method m contain only symbols that represent local variables of m.

3.1 Dynamic Pruning and Discovery of Feasible Covering Paths

A consequence of the simultaneous concrete and abstract execution is that a transition may reach an inconsistent combined state. This situation occurs when an action a allows for non-deterministic expansion, i.e. when the abstract pre-state s induces multiple alternative interpretations in $alt^*(s, a)$, while there is usually a single successor in the concrete state space. In such a case, the concrete successor is consistent only with one of the abstract successors.

Figure 3 illustrates *dynamic pruning and discovery of feasible covering paths*, the strategy that we propose for resolving such situations. The function TRANS$_{pruning}$ is the implementation of TRANS from Section 3. First, at line 2, it selects and executes one interpretation of action a in the state s (there must be at least one due to the check at line 24 in Figure 2) and marks it as processed at line 4. Further, if the abstract part of s' overapproximates the concrete part, i.e. when there are no inconsistencies, the function returns a transition leading to s'. Otherwise, PANDA is bound to prune the current trace by returning transition to s_{end} at line 10, because the currently analyzed interpretation is not consistent. Then the main algorithm is forced to backtrack in the next iteration, i.e. in the next call to UNROLL. However, there may still exist a concrete trace that captures different values returned by the unknown statements and conforms to the same abstract trace. Its existence is checked at lines 6 and 7 by means of generating a model for the related trace formula, and the corresponding branch will be explored by PANDA under a different combined trace in R. Although we omit this from the pseudocode of TRANS in Figure 3, when PANDA searches for the alternative concrete trace, it first tries to reuse the already discovered values that were returned in statements x = unknown() (in order to minimize backtracking) and only then it explores new models. PANDA extracts new interpretations (i.e., new values of all unknown statements) from the model and adds them into respective sets alt^*, so that they are explored later (line 9). Here the operator $\cdot\,[x \leftarrow e]$ produces a state that differs from its operand only in valuation of the variable x, which is fixed to the value e. Note that a single value returned by some unknown may prevent further execution of a trace in combination with specific values of other unknown statements along the trace, and it may permit the execution in combination with different values.

Example 2. Now consider the situation depicted in Figure 4 that illustrate the whole process. On the left, there is a short code snippet, which first stores a non-deterministic value into the variable x and then compares this variable with the constant 1. The rest of the figure shows combined concrete and abstract traces that are explored by PANDA during analysis of the code snippet. Dashed circles represent abstract states, solid dots represent concrete states, tubes depict abstract transitions, and finally solid arrows stand for concrete transitions. Each state label always applies to both the concrete and abstract part of a state, and the same is true for transitions.

1: **function** TRANS$_{pruning}(R, tr, s, a)$
2: $s' \leftarrow$ any successor in $alt^*(s, a)$
3: $\tau \leftarrow (s, a, s')$
4: remove s' from $alt^*(s, a)$
5: **if** s' is consistent **then return** τ
6: $M \leftarrow$ model of $\varphi_{tr \oplus \tau}$
7: **if** $M \neq \bot$ **then**
8: **for all** $(s_1, b, s_2) \in tr$ and b is `x = unknown()` **do**
9: augment $alt^*(s_1, b)$ with $s_2[x \leftarrow M(x)]$
10: **return** (s, a, s_{end})

Figure 3. The implementation of TRANS within PANDA

The process of pruning inconsistent traces and discovering feasible alternative traces that cover the pruned behavior is divided into three phases. Each of the phases is illustrated with a subfigure to the right of the code snippet in Figure 4.

```
1  x = unknown();
2  if (x > 1) {
3      ...
4  }
```

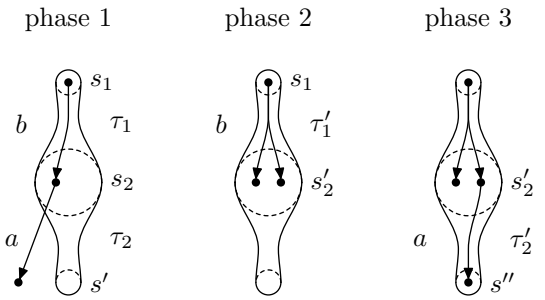

Figure 4. On-the-fly discovery of feasible covering paths

Phase 1. PANDA expanded the action b corresponding to `x = unknown()` in state s_1 to produce the transition τ_1 and reach the state s_2. The default interpretation of b is equivalent to `x = 0` (recall the definition of alt^*). At this point, the *then*-branch is selected first and TRANS$(R, tr, s_2,$ `assume x > 1`$)$ yields the state s', which is not consistent because the abstract state satisfies $x > 1$ while the concrete state assigns 0 to x. This is the reason why the solid dot is not included in the dashed circle for the state s', and therefore the solid arrow leaves the tube — representing the inconsistency between concrete and abstract interpretation of the action a in the state s_2. However, a different interpretation of b in s_1 exists that would produce a consistent transition. It is extracted from the model of $\varphi_{(s_1,b,s_2,a,s')}$. We suppose, for the purpose of the example, that the discovered interpretation of b is equivalent to `x = 2`, although many other integer values could be returned from `unknown()`. The new interpretation is added to the set $alt^*(s_1, b)$ before TRANS returns s_{end} and forces PANDA to backtrack to s_1.

Phase 2. After the backtrack, $b \in unexp(s_1)$ as it was reintroduced in the previous phase, and so it is selected for expansion. In the middle subfigure, the alternative interpretation s'_2 of the action b is expanded by PANDA in TRANS(R, tr, s_1, b). As a result, the state s'_2 is added to the reachability graph R.

Phase 3. The search now continues from s'_2. In the right-most subfigure, TRANS explores the interpretation of a in state s'_2. This time, it is consistent and yields the transition τ'_2 and the state s''. Thus the abstractly reachable *then*-branch is covered also by the concrete execution, although first it has been discovered with a concrete trace that had no feasible extension entering the branch.

In general, dynamic pruning eliminates many infeasible traces from the abstract state space based on the knowledge of concrete states. That is an important benefit of the simultaneous concrete and abstract execution. Note, however, that usage of pruning does not guarantee that all the infeasible abstract branches are eliminated, because it handles only choices introduced by actions that read non-deterministic values. Although only the feasible concrete execution traces will be explored for many input programs, iterative abstraction refinement still may be necessary in the case of choices caused by non-determinism of other kinds (e.g., imprecise heap abstraction).

3.2 Soundness and Termination

In this section, we discuss soundness of the proposed PANDA algorithm, and why it may not terminate in general. We show that dynamic pruning and discovery of feasible covering paths is sufficient to guarantee exploration of all the feasible behaviors of the given program.

We say that a reachability graph R is *complete* if every reachable state of the program P is directly contained in R, and that R is *precise* if it does not contain a spurious trace reaching s_{err}.

The proof of soundness of our verification procedure PANDA is based on the following theorem.

Theorem 1. *The program P is safe if and only if the error state s_{err} is not contained in a complete and precise reachability graph R constructed for P.*

Proof. We show the two directions of the equivalence separately for some complete precise reachability graph R for P.

$\Leftarrow$) The state s_{err} is not in R, and since R is complete it contains all the reachable states of P. Therefore, s_{err} is not reachable in P and by definition P is safe.

$\Rightarrow$) Assume that P is safe. Then, s_{err} is unreachable in any execution of P, and any abstract trace reaching s_{err} is spurious. Because the given R is precise, it cannot contain spurious abstract trace reaching s_{err} and thus s_{err} is excluded from R. □

What remains to be shown is that when PANDA does not report an error in P, it either terminates with a complete precise reachability graph $R(P)$ or does not terminate at all. Precision of R follows from the abstraction refinement step of the main algorithm.

PANDA either does not terminate or there are finitely many refinement steps, and thus the resulting reachability graph may not contain spurious error traces and it is precise.

Now assume that PANDA terminates on P and the reachability graph R is not complete, i.e. there is a reachable state s of P that is not contained in R. In that case, there must be a trace tr from s_0 to s and a state $\overset{\circ}{s}$ that is the first state on that trace not contained in R. Let $(\overset{\bullet}{s}, a, \overset{\circ}{s})$ be the transition reaching $\overset{\circ}{s}$ on the trace tr for the first time, which means that $\overset{\bullet}{s} \in R$. The only reason for a consistent reachable state $\overset{\circ}{s}$ to be excluded from R is that it was never included in $alt^*(\overset{\bullet}{s}, a)$. Since the heap abstraction and computation of abstract successors are overappoximating, the sets of alternative interpretations for assignment statements, branching, looping, function calls, and returns are overappoximating as well, and they never exclude any abstractly reachable successor unless it is pruned. Every abstract successor that is being pruned is analyzed (lines 6-9 in Figure 4) for feasibility and appropriate enabling interpretations of actions along the trace are added to alt^*, so that they can be explored later. Consequently, if the algorithm terminated without processing the alternative that reaches $\overset{\circ}{s}$, it could not have been feasible and R is, in fact, complete.

Theorem 2. PANDA *soundly verifies safety of programs.*

Proof. Follows directly from the discussion above. □

The whole PANDA algorithm may not terminate. The reachability graph may be infinite due to unbounded loops and recursion that admit infinite number of concrete traces of different lengths. Also, the abstraction refinement loop may diverge for input programs with possibly infinite state spaces [16].

4 Implementation

We implemented the proposed verification algorithm in the tool called PANDA, which is built upon Java Pathfinder (JPF) [25] and accepts programs in Java. JPF is responsible for concrete execution of Java bytecode instructions and systematic traversal of the concrete state space, and it also provides concrete values taken from dynamic program states. Predicate abstraction and lazy refinement are performed with the help of SMT solvers. The current version of PANDA uses CVC4 [4] and Z3 [19]. The complete source code of our implementation, including examples and benchmark programs, is available at `https://github.com/d3sformal/panda`.

In the rest of this section, we describe several optimizations of the core algorithm in Figure 2 that apply to the restart of state space traveral after refinement.

The basic variant of the function RESET backtracks to the initial state, and drops all information about the state space fragment explored before the spurious error was hit. However, in this case PANDA would explore again the fragment of the program state space that has already been proven safe. A more efficient approach, heavily inspired by lazy abstraction [16] used in BLAST [6], is the following: (1) determine which locations and states on the spurious error trace are affected by the refinement, (2) backtrack only to the last state of the longest unaffected prefix of the error trace, and (3) then

resume state space exploration from that point with the refined abstraction. Location l is affected by the refinement when new predicates were added to $abs(l)$.

Another limitation of the basic PANDA algorithm is repeated exploration of certain safe fragments of the program state space. We designed an optimization that is based on recording information about explored state space branches. During the traversal, PANDA remembers all safe branches for each choice on the current trace, and when the traversal resumes with the more precise abstraction it skips the recorded branches.

5 Evaluation

We performed experiments on three groups of Java programs in order to evaluate PANDA. A brief description of each group of benchmarks follows.

The first group contains 7 benchmarks from the categories *loops* and *arrays* of the Competition on Software Verification (SV-COMP) [26]. Four benchmarks in this group (Array, Invert String, Password, and Reverse Array) use arrays whose content is based on non-deterministic input, Eureka 01 computes aggregate properties of data structures based on the values of corresponding elements of multiple arrays, TREX 03 involves loops with a possibly large number of iterations but without a single explicit control variable, and the benchmark Two Indices maintains a relation over array elements at different indices. We had to translate all of them from C into Java, and we also reduced the sizes of arrays in both language variants, because the current version of PANDA is not yet optimized for programs with large arrays.

The second group contains 4 example programs that we used in previous work [21], namely Data-flow Analysis, Cycling Race, Image Rendering, and Scheduler. These benchmark programs are more realistic; they involve manipulation with arrays (sorting), field accesses on heap objects, and loops.

The third group contains variants of two benchmarks from the CTC repository [24]: Alarm Clock and Producer-Consumer. We translated the original concurrent programs into sequential programs using an approach similar to context-bounded reduction [22].

As the benchmark programs in the second and third group are relatively larger, we used them to find whether PANDA is competitive in terms of scalability. Note also that source code of all the programs contains assertions but the corresponding error states are not reachable.

We ran PANDA and selected other tools – namely BLAST [6], CPACHECKER [8], UFO [1], and WOLVERINE [17] – on all the benchmark programs in order to find whether our proposed approach is competitive with respect to the ability of verifying program safety and the running time. We used CPACHECKER in the version from SV-COMP'15, BLAST and UFO in the versions from SV-COMP'14, and WOLVERINE from the year 2012. Table 1 contains results of the experiments.

For PANDA, we report the total running time (t), size of the reachable state space ($|S|$), number of refinement steps, maximum number of abstraction predicates at some location, and the total number of satisfiability queries executed by PANDA. For the other tools, we report only the total running time in case the respective tool provided a correct answer. Other possible outcomes are expressed by specific symbols. We use the symbol ✗ to denote that a tool reported a spurious error (i.e., a wrong answer), the symbol **?** to

Table 1. Experimental results and comparison with other tools

Benchmark	PANDA					BLAST	CPA	UFO	WOLVERINE
	t	$\|S\|$	#ref	$\|abs\|$	#sat				
Array	4 s	38	0	7	1802	2 s	2 s	1 s	1 s
Eureka 01	23 s	741	0	53	11462	✗	?	✗	timeout
TREX 03	21 s	1425	0	9	14371	✗	✗	1 s	1 s
Invert String	6 s	126	0	18	2728	✗	6 s	✗	9 s
Password	22 s	870	0	19	12837	23 s	3 s	✗	4 s
Reverse Array	5 s	135	0	18	2358	✗	3 s	✗	3 s
Two Indices	4 s	55	0	15	1921	✗	2 s	✗	1 s
Data-flow Analysis	379 s	508	0	64	8159	?	?	✗	✗
Cycling Race	5 s	87	0	28	2151	6 s	3 s	2 s	2 s
Image Rendering	timeout					-	44 s	-	✗
Scheduler	5 s	108	0	35	2185	?	4 s	✗	4 s
Alarm Clock	970 s	21200	0	20	87628	?	✗	✗	-
Producer-Consumer	timeout					?	✗	-	✗

indicate that a tool says ”don’t know”, and the character ”-” when a tool fails for some other reason (e.g., missing support for a particular language feature). We put the limit of two hours on the running time for all experiments.

The results show that PANDA did not have to perform abstraction refinement in the case of all our benchmarks for which verification finished before the time limit. In addition, PANDA did not report a spurious error for any benchmark program, unlike some of the other tools. This observation supports our claim that simultaneous abstract and concrete execution is very precise and avoids spurious behaviors.

Regarding performance, the results are mixed — PANDA is faster than other tools for some of the programs and slower in other cases, but its running times are competitive for all the benchmarks. Data for the benchmarks Alarm Clock, Image Rendering, and Producer-Consumer show that PANDA has limited scalability, but the other tools failed on these benchmarks with the exception of CPACHECKER on Image Rendering. By manual inspection of execution logs, we found the following main reasons for the long running times and state explosion in the case of these three programs.

1. Each trace contains many non-deterministic data choices (unknown statements) for which multiple concrete values have to be explored.
2. Some of the more complex SMT queries executed by PANDA, in particular those used to derive new return values for unknown statements, take a very long time to answer — for example, even up to 200 seconds in the case of Image Rendering.

On the other hand, PANDA successfully verified the programs Alarm Clock, Data-flow Analysis, and Eureka 01, for which all the other tools failed or reported a wrong answer.

6 Related Work

Many verification techniques based on the CEGAR principle [11] have been proposed in the past. However, we are not aware of any existing approach that combines abstraction with concrete execution in the same way as PANDA does. We provide details about selected techniques and highlight the main differences.

The PANDA algorithm extends the approach to lazy predicate abstraction, which was originally proposed by Henzinger et al. [16] and implemented in BLAST [6]. Simultaneous combination of abstraction with concrete execution allows PANDA to prune many infeasible execution paths and spurious errors on-the-fly during the state space traversal, thus avoiding many expensive steps of abstraction refinement. In the more recent work of McMillan [18] and Alberti et al. [3], lazy abstraction is done using only interpolants without predicate abstraction, but in this case it is more difficult to check whether a given state was already covered during traversal. UFO [1] is another verification technique that combines abstraction, unrolling of a control flow graph, and interpolants. It captures multiple error traces with a single formula in order to reduce the number of necessary refinement steps.

CPACHECKER [8] is a tool that performs multiple custom analyses simultaneously, using the framework proposed by Beyer et al. [7]. For example, it enables users to combine predicate abstraction with shape analysis. The definition of each program analysis consists of an abstract domain, transfer relation, merge operator, and an operator that performs the covering check. It might be possible to implement the PANDA algorithm in CPACHECKER, assuming that different analyses can exchange the necessary information during a run of the tool. Concrete execution would have to be expressed as one of the analyses.

Charlton [9] proposed another framework that supports combination of multiple analyses and verification techniques. The analyses are executed in steps by the overall worklist algorithm. In each iteration, they exchange computed facts about the program behavior using logic formulas, and they can also query each other.

The DASH algorithm [5] combines testing with abstraction in an iterative manner to achieve better precision and performance. In each iteration, it explores the current abstract state space in order to search for a possible error trace. Then, if there is an abstract error trace, DASH attempts to find a corresponding concrete trace by creating and running new tests. Based on their results, it can either confirm the presence of a real error or extend the current forest of tests. Only when such a test cannot be found, the abstraction is refined by predicates that are derived from the first infeasible transition on the given error trace. Like in the case of PANDA, use of concrete execution (testing) saves many refinement steps and helps to avoid many SMT queries, especially if the input program contains loops with many iterations. The main difference is that DASH performs the individual phases, i.e. concrete execution and changes of the abstraction, consecutively (in turns), while PANDA unrolls the reachability graph on-the-fly using both concrete execution and predicate abstraction simultaneously (in tandem). This enables PANDA to refine multiple regions of the abstraction in each iteration, achieving faster convergence.

SMASH [14] combines may analysis (abstraction) with must analysis (concrete execution in the form of dynamic test generation) using a compositional approach based

on procedure summaries. In each step, it can update either the may summary of some procedure or the must summary, but not both of them simultaneously. The key feature of SMASH is the alternation (interplay) of testing and abstraction such that intermediate analysis results are exchanged between the two. Both the DASH and SMASH algorithms are implemented in the YOGI tool [20].

PANDA resembles also mixed symbolic and concrete execution, implemented in tools such as DART [13] and KLEE [10]. However, in PANDA the concrete execution and predicate abstraction are performed simultaneously in such a way that they guide each other, while in DART, for example, they do not interact during the traversal of one path. In addition, PANDA uses predicates that are more expressive than path constraints in DART, because it generates new predicates by applying interpolation to trace formulas (i.e. not just by extraction from the program code). It is also more efficient because it can prune several infeasible paths in one step. The main practical limitation of symbolic execution is that users must put a bound on the number of explored paths and their depth. Tools based on this approach are therefore used mainly for dynamic test generation and bug hunting, while PANDA can explore all paths in the reachability graph of a given program to check whether it is safe.

Some work has been done also on combining symbolic execution with predicate abstraction and iterative refinement. The approach proposed by Albarghouthi et al. [2] uses symbolic execution to explore the underapproximation of a program behavior, and in each iteration checks whether the abstract model created by symbolic execution is also an overapproximation of the concrete state space. Abstraction refinement is performed to add new predicates that would enable the verification procedure to cover more feasible execution paths.

7 Conclusion

In this paper we presented the PANDA algorithm that combines predicate abstraction with simultaneous concrete execution. Dynamic pruning, the method that we proposed for solving inconsistencies between concrete and abstract execution, eliminates many spurious execution paths on-the-fly. A consequence of this combination is a higher analysis precision that allows PANDA to keep the number of necessary refinement steps to a minimum. Specifically, PANDA did not have to perform abstraction refinement for any of the benchmark programs that we used in our experiments.

In future, we plan to optimize our prototype implementation and we would also like to use a different abstract representation of the program heap. Our long term goals include support for data containers, concurrency, and predicates over data shared between threads, most probably through adaptation of some already known techniques [12, 21].

Acknowledgements. This work was partially supported by the Grant Agency of the Czech Republic project 13-12121P and by Charles University institutional funding SVV-2015-260222.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In Proceedings of TACAS 2012, LNCS, vol. 7214.
2. A. Albarghouthi, A. Gurfinkel, O. Wei, and M. Chechik. Abstract Analysis of Symbolic Executions. In Proceedings of CAV 2010, LNCS, vol. 6174.
3. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy Abstraction with Interpolants for Arrays. In Proceedings of LPAR 2012, LNCS, vol. 7180.
4. C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In Proceedings of CAV 2011, LNCS, vol. 6806.
5. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from Tests. In Proceedings of ISSTA 2008, ACM.
6. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. STTT, 9(5-6), 2007.
7. D. Beyer, T. A. Henzinger, and G. Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In Proceedings of CAV 2007, LNCS, vol. 4590.
8. D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Proceedings of CAV 2011, LNCS, vol. 6806.
9. N. Charlton. Program Verification with Interacting Analysis Plugins. Formal Aspects of Computing, 19(3), 2007.
10. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of OSDI 2008, USENIX.
11. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In Proceedings of CAV 2000, LNCS, vol. 1855.
12. A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs. In Proceedings of CAV 2011, LNCS, vol. 6806.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In Proceedings of PLDI 2005, ACM.
14. P. Godefroid, A. Nori, S.K. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In Proceedings of POPL 2010, ACM.
15. M. Heizmann, J. Hoenicke, and A. Podelski. Nested Interpolants. In Proceedings of POPL 2010, ACM.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In Proceedings of POPL 2002, ACM.
17. D. Kroening and G. Weissenbacher. Interpolation-Based Software Verification with Wolverine. In Proceedings of CAV 2011, LNCS, vol. 6806.
18. K. McMillan. Lazy Abstraction with Interpolants. In Proc. of CAV 2006, LNCS, vol. 4144.
19. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In Proceedings of TACAS 2008, LNCS, vol. 4963.
20. A. Nori, S.K. Rajamani, S. Tetali, and A. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In Proceedings of TACAS 2009, LNCS, vol. 5505.
21. P. Parizek and O. Lhotak. Predicate Abstraction of Java Programs with Collections. In Proceedings of OOPSLA 2012, ACM.
22. S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In Proc. of PLDI 2004, ACM.
23. Y. Vizel and O. Grumberg. Interpolation-Sequence Based Model Checking. In Proceedings of FMCAD 2009, IEEE.
24. Concurrency Tool Comparison, `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison`
25. Java Pathfinder, `http://babelfish.arc.nasa.gov/trac/jpf`
26. Competition on Software Verification, `http://sv-comp.sosy-lab.org/2015/`