Symbolic Verification of Regular Properties

Hengbiao Yu[1,2*], Zhenbang Chen[1*], Ji Wang[1,2*], Zhendong Su[3], Wei Dong[1]
[1]College of Computer, National University of Defense Technology, Changsha, China
[2]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
[3]Department of Computer Science, University of California, Davis, USA
{hengbiaoyu,zbchen,wj}@nudt.edu.cn,su@cs.ucdavis.edu,wdong@nudt.edu.cn

ABSTRACT

Verifying the regular properties of programs has been a significant challenge. This paper tackles this challenge by presenting *symbolic regular verification* (SRV) that offers *significant speedups* over the state-of-the-art. SRV is based on dynamic symbolic execution (DSE) and enabled by novel techniques for mitigating path explosion: (1) a regular property-oriented path slicing algorithm, and (2) a synergistic combination of property-oriented path slicing and guiding. Slicing prunes redundant paths, while guiding boosts the search for counterexamples. We have implemented SRV for Java and evaluated it on 15 real-world open-source Java programs (totaling 259K lines of code). Our evaluation results demonstrate the effectiveness and efficiency of SRV. Compared with the state-of-the-art — pure DSE, pure guiding, and pure path slicing — SRV achieves average speedups of more than 8.4X, 8.6X, and 7X, respectively, making symbolic regular property verification significantly more practical.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

Regular property; Verification; Dynamic Symbolic Execution; Slicing; Guiding

ACM Reference Format:
Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic Verification of Regular Properties. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180227

1 INTRODUCTION

Regular properties are ones that can be specified by finite state machines (FSMs) [22]. They are widely used for property specification in software analysis and verification (*e.g.*, model-based testing [36], typestate analysis [21], model checking [14], and performance analysis [33]). However, scalable regular property verification is difficult, and practical verification of regular properties of real-world programs is a significant software engineering research challenge.

Two main lines of research exist on regular property verification: *static* and *dynamic* verification. Static verification (such as [17, 20, 21]) *soundly* abstracts programs for verification, which usually has high code coverage, but suffers from false alarms. Dynamic verification (such as [2, 12]), in contrast, executes the program and monitors program executions online. Hence, dynamic verification ensures *completeness*, *i.e.*, every discovered violation is real. However, dynamic approaches only verify a program's behavior under specific inputs, thus may miss bugs.

Symbolic execution [10, 23, 30] achieves trade-offs between static and dynamic verification by using symbolic values for program execution. A key step in symbolic execution is to explore all possible cases when encountering a branch via forking states or re-executing the program. Compared with static and dynamic approaches, symbolic execution achieves better precision or coverage, respectively.

Our goal is to develop a practical technique for symbolic regular property verification. At the high-level, it works as follows. For a regular property φ and program P, an *event* in φ's FSM represents the execution of one or more statements of P. For example, a method invocation may produce an event. Hence, *w.r.t.* φ, an execution path p of P generates an event sequence, denoted as $Seq(p)$. If $Seq(p)$ is empty, p is an *irrelevant path*; otherwise, p is *relevant*. To verify that P satisfies φ, we adopt symbolic execution to explore P's path space. If there exists a path p that violates φ, *i.e.*, $Seq(p)$ is accepted by the FSM of $\neg\varphi$ (denoted by $\text{FSM}_{\neg\varphi}$), a violation is found, and p is a *counterexample path*. Otherwise, P satisfies φ.

However, symbolic execution is hindered by the problem of *path explosion* — exponential path space *w.r.t.* the number of branches in the program. Thus, how to steer symbolic execution to (1) completely explore the path space and (2) find counterexamples *as soon as possible* is critical. This paper tackles these challenges and introduces a scalable verification technique, called *symbolic regular verification* (SRV), for regular properties via dynamic symbolic execution (DSE) [23, 41]. SRV is inspired by two key observations. First, there usually exist a large number of irrelevant paths in P *w.r.t.* the regular property φ. Second, many of the relevant paths in P are equivalent, *i.e.*, the paths having identical event sequence *w.r.t.* φ. Therefore, during DSE, it is desirable to (1) prune both irrelevant and equivalent relevant paths, and (2) explore counterexample paths as early as possible. Doing these can boost symbolic verification to find counterexamples and finish path exploration more promptly.

To verify a regular property φ for a program, the *key idea* of SRV is to (1) slice a path *w.r.t.* related statements of φ, which results in

*Zhenbang Chen and Ji Wang are the corresponding authors. Hengbiao Yu and Zhenbang Chen contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180227

pruned paths related to the sliced branches, and (2) use $\neg\varphi$ to guide the selection of branches to explore for finding counterexample paths early. The *main technical novelty* of SRV is the design of a slice algorithm and a verification framework. The algorithm is a regular property-oriented slicing algorithm based on path slicing [27]. Besides control and data dependencies, our algorithm also exploits property-related program information to perform slicing. The algorithm can slice the branches along which no counterexample path exists; besides, the branches along which each counterexample path has an equivalent previously explored path can also be sliced. The verification framework combines regular property-oriented path slicing and the property-oriented guiding technique [49] in a synergistic manner, in which the combined techniques complement and also boost each other.

We have implemented SRV for Java utilizing a regular property guided symbolic execution engine [47] and a dynamic slicing tool Javaslicer [15]. SRV has been extensively evaluated on 15 real-world open-source Java programs using regular properties involving both single or multiple objects. The evaluation results demonstrate SRV's effectiveness and efficiency for regular property verification.

This paper makes the following main contributions:

- A property-oriented path slicing algorithm that can prune paths for verification *w.r.t.* regular properties. The explored paths using our slicing algorithm is two orders of magnitude less than that using path slicing [27][16].
- A method that enhances regular property guided DSE [49] for supporting multi-object regular properties.
- A DSE-based framework that integrates slicing and guiding for practical regular property verification.
- A prototype implementation for Java that significantly outperforms the state-of-the-art: (1) successfully verified 30 out of 39 verification tasks on a total of 259K lines of code within 1 hour, while pure DSE, guiding, and path slicing verified 22/22/23 tasks, respectively; and (2) on the 30 successfully verified tasks, offered more than 8.4X/8.6X/7X average speedups over pure DSE, guiding, and path slicing, respectively.

The rest of this paper is organized as follows. Section 2 motivates and illustrates symbolic regular verification (SRV) via a concrete example, and Section 3 presents the details of SRV. Section 4 explains our implementation and empirical evaluation of SRV. Finally, we survey related work (Section 5) and conclude (Section 6).

2 ILLUSTRATING EXAMPLE

This section uses an example to motivate and illustrate our symbolic regular property verification technique. Figure 1 shows a Java program snippet that uses an `Iterator` to access an `ArrayList`. The input parameters are an `ArrayList` `arr` and an integer variable `m`. First, we increase `m` by 1 if `m` is greater than 10. Then, we obtain `arr`'s iterator, and assign it to `iter`. The following `for` loop (Lines 6-9) finds the maximum value of the first half of `arr`, and stores it in `max`. Next, `max` will be removed from `arr` if its value equals 100. The following `while` loop (Lines 12-16) iterates `arr` by using the iterator `iter`. During the iteration, we update the value of `max` to the value of an element if the element is bigger than `max`. Finally, the addition of `m` and `max` is returned.

```
1  public int test(ArrayList<Integer> arr, int m){
2    if(m > 10) //{q0}
3      m++; //{q0}
4    int max=0; //{q0}
5    Iterator iter=arr.iterator(); //{q1 ~ q4}
6    for(int i = 0; i < arr.size()/2; i++){ //{q1 ~ q4}
7      if(arr.get(i).intValue() > max) //{q1 ~ q4}
8        max = arr.get(i).intValue(); //{q1 ~ q4}
9    }//{q1 ~ q4}
10   if(max == 100) //{q1 ~ q4}
11     arr.remove(max); //{q3, q4}
12   while(iter.hasNext()){ //{q1, q3, q4}
13     int temp = iter.next(); //{q3, q4}
14     if(temp > max) //{q3, q4}
15       max = temp; //{q3, q4}
16   } //{q4}
17   return m+max; //{q4}
18 }
```

Figure 1: An example program.

For the motivation program, we are interested in the correct usage of a collection's `Iterator`, *i.e.*, the collection cannot be modified while being iterated and the iterator should invoke method *hasNext* before *next*. Note that such a safety property involves two objects.The property can be specified as a regular property φ, and FSM$_{\neg\varphi}$ is shown in Figure 2, where a and i represent an `ArrayList` object and the corresponding iterator object, respectively. For brevity, we use $a.update$ to represent adding an element to a or removing an element from a. Event $a.iterator$ denotes the accessing of a's iterator. We use $i.hasNext$ and $i.next$ to represent invoking the method *hasNext* and *next*, respectively. Obviously, when the first half of `ArrayList` has an element that equals 100, a violation of φ occurs, *i.e.*, the `ArrayList` removes an element while being iterated. In this paper, we assume that every event is atomic, *i.e.*, no other events may be generated during its execution.

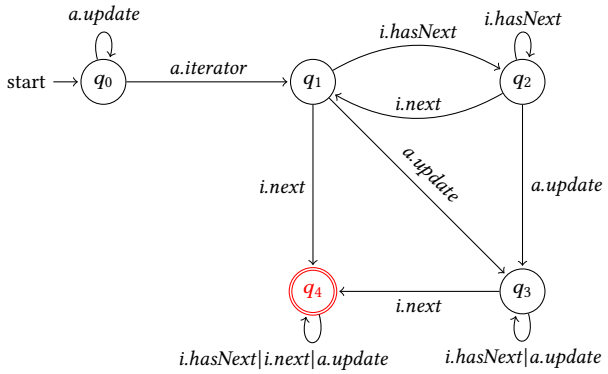

Figure 2: The FSM of iterator bug involving multi-objects.

When using pure DSE to verify the program *w.r.t.* φ, it requires many iterations to unfold the two loops. In addition, the branch at Line 2 doubles the huge path space. Symbolic regular verification (SRV) aims to finish the path exploration and find counterexample paths as soon as possible. For the example program and the property φ, SRV (1) needs only *two iterations* to finish the path exploration and (2) finds the first counterexample path in the *second iteration.*

SRV procedure consists of two stages. At the first stage, the program is statically analyzed *w.r.t.* FSM$_{\neg\varphi}$ through a backward

data flow analysis to calculate the future behavior information, called *Postset*, of every program location. A *Postset* of a location *loc* contains some states of FSM$_{\neg\varphi}$, and each state q indicates that there *may* exist a subsequent path p after *loc* and *Seq*(p) can drive FSM$_{\neg\varphi}$ from q to an accepted state. The comment of each line in Figure 1 shows the *Postset* of the location *after* the line. For brevity, we use $\{q_j{\sim}q_k\}$ to represent $\{q_l \mid j \leq l \leq k\}$. For example, after Line 10, there exists a subsequent path p that *Seq*(p)=$\langle$*update*, *hasNext*, *next*$\rangle$, which can drive q_1 to the accepted state. Hence, the *Postset* of the location after Line 10 contains q_1.

At the second stage, the program will be analyzed via DSE. During DSE, the *Postset* information calculated earlier will be used to select the paths to explore. Because DSE also runs the program concretely, we can use the available runtime information to calculate certain history information, called *Preset*, of a branch to be explored. Same as *Postset*, the *Preset* of a branch b also contains some states of FSM$_{\neg\varphi}$. A state q in *Preset* indicates that q can be reached from the initial state via the path from the beginning of the program to b. Based on the *history* and *future* information, we can (1) prune the redundant program paths, which include irrelevant paths, non-counterexample paths and the equivalent paths of previously explored paths; and (2) evaluate the possibility of a branch for generating a counterexample path.

When a path p is explored by DSE, property-oriented path slicing is employed on p to slice the branches *w.r.t.* φ. Slicing uses static dependence analysis [27] and the *history* and *future* information to reason about possibly accepted event sequences along a branch. If a branch b's intersection of the *Preset* and *Postset* is empty, it means no counterexample path along b exists. Besides, if all the counterexample paths along b have equivalent paths explored previously, it is also no need to explore b. Under both of these conditions, b will be pruned, which results in pruning all the paths along b. On the other hand, we also use the intersection of the *Preset* and *Postset* to calculate the heuristic value of b. If the size of the intersection is larger, the possibility of having a counterexample path along b is considered higher. Hence, b will be selected with a higher priority.

Consider the program in Figure 1 and the FSM$_{\neg\varphi}$ in Figure 2. To analyze the program via DSE, the two input parameters are made symbolic variables. We assume that the `Arraylist arr` has a fixed length and contains two elements $arr[0]$ and $arr[1]$. Suppose the initial input to `test` is ⟨`arr`={1, 2}, `m`=3⟩. The first iteration generates the event sequence ⟨*iterator*, *hasNext*, *next*, *hasNext*, *next*, *hasNext*⟩, and is not a counterexample path. Figure 3 shows the execution tree after the first iteration, where *dashed* states are candidate states to explore. The *Preset* and *Postset* of a candidate branch are above and below the branch, respectively. For example, b_3 corresponds to the true branch of Line 10, and b_3's *Preset* and *Postset* are $\{q_1\}$ and $\{q_1, q_2, q_3, q_4\}$, respectively. The *pruned* candidate branches are *dashed* and *grey*, while the remaining are *dotted* and *black*.

The slicing of the first path prunes branches b_1, b_4, and b_5, which means the paths along these pruned branches are redundant for verification. Branch b_1 can be pruned according to path slicing [27]. The reason is that there is no events transitively data or control depend on `m`, which means changing the value of `m` cannot generate new event sequences. On the other hand, the reason for pruning b_4 and b_5 is the intersection of *Preset* and *Postset* is empty,

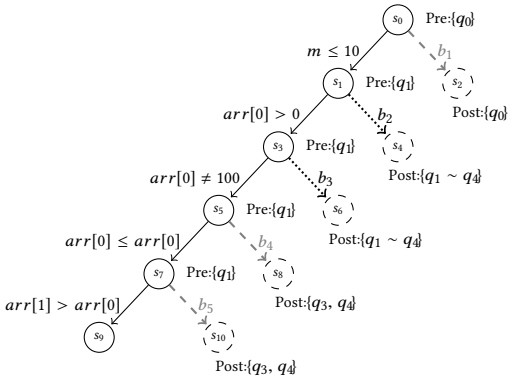

Figure 3: Execution tree after the first iteration.

which implies no counterexample paths along these branches exist. Both of b_2 and b_3 have the same result of intersecting *Preset* and *Postset*. In this situation, SRV will select the deeper branch, *i.e.*, b_3, to explore. The path condition for generating the next input is $m \leq 10 \wedge arr[0]{>}0 \wedge arr[0] == 100$. Through solving the new path condition, we assume that the generated input is ⟨`arr` = {100, 2}, `m` = 3⟩. The second iteration generates an accepted event sequence ⟨*iterator*, *update*, *hasNext*, *next*⟩. Thus, the path is a counterexample path.

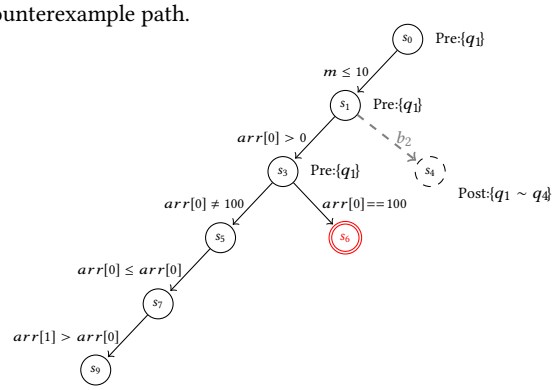

Figure 4: Execution tree after the second iteration.

Figure 4 shows the execution tree after the second iteration. For brevity, we omit the pruned candidate branches and states generated in the first iteration. Note that the second path terminates in the accepted state s_6, because a runtime property violation happens. For branch b_2, with the help of our *property-oriented slicing* method, we can infer that all possible event sequences along b_2 accepted by FSM$_{\neg\varphi}$ are equivalent to the one explored in the second iteration, *i.e.*, ⟨*iterator*, *update*, *hasNext*, *next*⟩. Hence, b_2 can be pruned.

In total, SRV needs two iterations to explore the full path space and finds the counterexample path in the second iteration regardless of `arr`'s size. If we use depth-first search (DFS) or breadth-first search (BFS), the exploration will get stuck due to unfolding the two loops, failing to quickly find the counterexample path. If we use pure path slicing [27], only branch b_1 can be pruned, and the exploration will also get stuck due to the two loops. If we use pure regular property guiding [49], it will find the counterexample path during the second iteration, but no path pruning happens, hence it fails to complete the path exploration.

3 SRV: SYMBOLIC REGULAR VERIFICATION

This section presents the technical details of SRV. It first presents the overall synergic verification framework, then the two combined techniques, and finally discusses SRV.

3.1 Synergic Verification Framework

SRV's key insight is to use slicing *w.r.t.* φ to prune redundant program paths, and the guiding method in [49] to find counterexamples earlier. More precisely, in addition to the synergy between slicing and guiding, (1) SRV's property-oriented slicing method can prune additional paths through exploiting the guiding information, *i.e.*, *Preset* and *Postset*, compared with path slicing [27]; and (2) SRV enhances the guiding method [49] with the support of multi-object regular properties. SRV aims for *full verification*, which means exploring the program's whole path space to successfully verify the program or find all inequivalent violations of the property.

Algorithm 1: DSE-based Regular Property Verification

```
   SRV(P, M_¬φ, I_0)
   Data: program P, FSM M_¬φ and an initial input I_0
1  begin
2      worklist, X ← ∅; PC ← true; I ← I_0;
3      Postset ← ComputeFutureInfo(P, M_¬φ);
4      while true do
5          (PC, path_c, Preset) ← runAndMonitor(I, M_¬φ);
6          if accept(M_¬φ, Seq(path_c)) then
7              X ← X ∪ {LSeq(path_c)};
8              Report a counterexample path;
9          R_s ← Slice(P, path_c, M_¬φ, Preset, Postset);
10         update(worklist, R_s, PC);
11         if worklist = ∅ ∨ Timeout then
12             exit;
13         PC ← Select(worklist, Preset, Postset);
14         I ← Solve(PC);
```

Algorithm 1 shows the overall framework of SRV. The input to SRV consists of a program P, an FSM $M_{\neg\varphi}$ for the negation of the regular property φ to be verified and an initial input I_0 to P. The algorithm first computes the *Postset* information of P *w.r.t.* $M_{\neg\varphi}$ (Line 3, *cf.* Section 3.2) that will be used by slicing and guiding later. It uses a *worklist* to store the branches to be explored and $\mathcal{X}$ to store the accepted event sequences. During each iteration, the algorithm runs P and checks the property on the fly (Line 5, *cf.* Section 3.3). Besides the path condition PC, the current path $path_c$ is also collected along with DSE. At the same time, the *Preset* information is also calculated for each branch along $path_c$ *w.r.t.* $M_{\neg\varphi}$ (*cf.* Section 3.3). If $path_c$ is a counterexample path (Line 6), we add $LSeq(path_c)$, *i.e.*, the generated event sequence with *program location* [37] information, to $\mathcal{X}$ and report $path_c$. Once a path is terminated, the property-oriented path slicing algorithm Slice (*cf.* Algorithm 2) is invoked to prune branches along the path (Line 9). Then, *update* is invoked to save new branches to *worklist* and prune the branches in *worklist* according to the slicing result. Based on the heuristic value of each branch (*cf.* Section 3.5), Select selects a branch to generate the path condition for the next iteration (Line 13). The inputs of the next iteration can be generated by invoking a backend SMT solver (Line 14). The algorithm repeats this process until the *worklist* becomes empty or timeout (Lines 11&12).

3.2 Statically Compute Future Information

For slicing and guiding, we calculate the *Postset* for each static program location. We improve the *Postset* calculation method in [49] in two dimensions: (1) extending the flow functions in IFDS to support multi-object regular properties; and (2) enhancing the data facts and flow functions in IFDS to record encountered event sequences for a program location. For each location l, the *Postset* contains two types of information: (1) from which states the rest program after l can drive $M_{\neg\varphi}$ to an accepted state; and (2) the generated event sequences after l that can drive a state to an accepted state.

More precisely, we first construct the reversed FSM (denoted by $\overleftarrow{M_{\neg\varphi}}$) [22] of $M_{\neg\varphi}$, which accepts the reversed ones of $M_{\neg\varphi}$'s accepted paths. For example, Figure 5 shows $\overleftarrow{M_{\neg\varphi}}$ of the FSM in Figure 2, and $\overleftarrow{M_{\neg\varphi}}$ accepts $\langle i.next, a.iterator\rangle$ that is the reverse of $\langle a.iterator, i.next\rangle$ accepted by the FSM in Figure 2. Observe that one state of $\overleftarrow{M_{\neg\varphi}}$ may correspond to a set of states of $M_{\neg\varphi}$. For example, Figure 5's FSM has a state $\{q_1, q_3, q_4\}$, to which there exists a transition from state $\{q_4\}$. The transition means there exists a transition from state q_1 to state q_4 in the FSM in Figure 2.

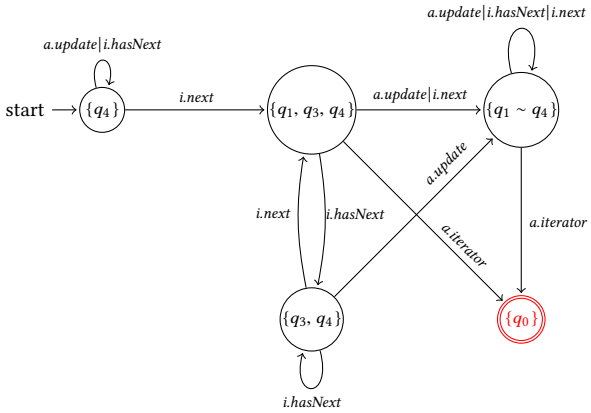

Figure 5: The reversed FSM of iterator bug.

To calculate the *Postset*, we update the data facts during exploring the program statements according to the transitions in $\overleftarrow{M_{\neg\varphi}}$ and merge the data facts at merging points in the control flow graph. Take the program in Figure 1 for example, and there exists a data fact $((o_1, o_2), \{q_3, q_4\})$ before exploring the statement at Line 11, where o_1 and o_2 correspond to the `ArrayList` object and `Iterator` object, respectively. According to the transitions in Figure 5, the data fact becomes $((o_1, o_2), \{q_1{\sim}q_4\})$ after exploring $o_1.remove$. Actually, the data fact $((o_1, o_2), \{q_3, q_4\})$ also explores the false branch that has no effect on the fact, because no events can be encountered in the branch. Hence, there exist two data facts at Line 10 through merging the data facts from the true and false branches, *i.e.*, $((o_1, o_2), \{q_1{\sim}q_4\})$ and $((o_1, o_2), \{q_3, q_4\})$.

To make the *Postset* analysis inter-procedural, we carry out a data flow analysis on the program's inter-procedural control flow

graph (ICFG) *w.r.t.* $\overleftarrow{M_{\neg\varphi}}$. The data flow analysis is implemented by employing the IFDS framework [40].

For a multi-object property φ involving k objects, a data flow fact in IFDS is an element (T_a, q_r, b, s) in the domain $\bigcup_{1\le n\le k} O^n \times S \times B \times E$ (denoted by D), where O is the set of the identities of static objects, *i.e.*, the static locations of object creations [37], S is the state set of the *reversed* FSM $\overleftarrow{M_{\neg\varphi}}$ of $\neg\varphi$, B is the set of the basic blocks in the program, and E is the set of event sequences. For example, a data fact $((o_1, o_2), \{q\}, b, s)$ of a program location l means that 1) from state q the program after l can drive $M_{\neg\varphi}$ to an accepted state, 2) the event sequence s can be generated by o_1 and o_2 after l, and 3) s can drive $M_{\neg\varphi}$ to an accepted state from q. The relationship between an event and its corresponding object can be obtained through checking whether the class or interface of the event corresponds to the type of the object. For example, there exists a data fact $((o_1, o_2), \{q_1{\sim}q_4\}, b, \langle update, hasNext, next\rangle)$ in the *Postset* of Line 10 in the example program in Figure 1, where o_1 and o_2 represent the identity of the `ArrayList` and `Iterator` objects, and b is the corresponding basic block of Line 10. Obviously, $\langle update, hasNext, next\rangle$ can drive state q_1 to the accepted state of the FSM in Figure 2, and *update* is related to the `ArrayList` object, while *hasNext* and *next* are related to the `Iterator` object.

Two different typed static objects o_1 and o_2 are *related* if their types are specified by φ. Take the motivation program for example, an `Iterator` object is related with an `ArrayList` object according to the property specification in Figure 2. Without loss of generality, we assume φ specifies two objects for brevity. There are four kinds of flow functions in IFDS: *call-to-start*, *exit-to-return*, *call-to-return* and *normal* functions. The normal and exit-to-return functions do not have any influence on calculating *Postset*, and both of them are identity functions. If a method invocation statement does not produce any event, its call-to-start function is the identity function; otherwise, it is a *killall* function [40] that kills all the data facts. The *call-to-return* function $f_{cr} : D \to D$ is the main one that drives the calculation of *Postset*.

For a method invocation statement `obj.meth(...)`, if the statement does not produce an event, its f_{cr} is the identity function. Otherwise, suppose the identity set of `obj` is O_s, the produced event is e_1 that can make the transition from q_1 to q_2 in $\overleftarrow{M_{\neg\varphi}}$, and the block of the statement is b_n, then f_{cr} is the smallest function [29] satisfying the following conditions, where $s_1 \circ s_2$ represents the concatenation of sequences s_1 and s_2:

- For each $d \in domain(f_{cr})$ and d is $((o_1), q_1, b, s)$: (1) if $o_1 \notin O_s$ and no element in O_s is related to o_1, then $d \in range(f_{cr})$; (2) if $o_1 \in O_s$, then $((o_1), q_2, b_n, \langle e_1\rangle \circ s) \in range(f_{cr})$; (3) if there exists $o_2 \in O_s$, and o_2 is related to o_1, then $((o_1, o_2), q_2, b_n, \langle e_1\rangle \circ s) \in range(f_{cr})$ and also $d \in range(f_{cr})$. On the other hand, if d is $((o_1, o_2), q_1, b, s)$ and $\{o_1, o_2\} \cap O_s$ is not empty, then $((o_1, o_2), q_2, b_n, \langle e_1\rangle \circ s) \in range(f_{cr})$. All possible cases of related objects are considered.
- If the initial state of $\overleftarrow{M_{\neg\varphi}}$ can make a transition to q_e via e_1, then for each $o \in O_s$, we have $((o), q_e, b_n, \langle e_1\rangle) \in range(f_{cr})$.
- If `obj` points to multiple static objects, *i.e.*, $|O_s| > 1$, then we have $domain(f_{cr}) \subseteq range(f_{cr})$. We adopt weak update [1] to achieve an over-approximation.

When calculating the *fixed point* in IFDS, the *first three parts* (T_a, q, b) of a data fact constitute the *key* that identifies the data fact. Hence, considering the numbers of states, static objects, and basic blocks are finite, the termination of IFDS is guaranteed. Given a static location, there may exists multiple data facts with a same key, and these facts have different event sequences. Each of the sequences is a possibly produced event sequence after the location.

For example, in Figure 1, the *Postset* below Line 10 contains four data facts with a same key $((o_1, o_2), \{q_1{\sim}q_4\}, b)$ but having the following four event sequences: $\langle$U, H, N$\rangle$, $\langle$U, H, N, H$\rangle$, $\langle$U, H, N, H, N$\rangle$, $\langle$U, H, N, H, N, H$\rangle$, where U, H and N denote *update*, *hasNext* and *next*, respectively, o_1 and o_2 are the identities of the `ArrayList` and `Iterator` objects, and b is the basic block of Line 10.

For the future information of a branch br (denoted by $Postset(br)$) whose location is l, its soundness means that the *Postset* of l includes the data facts of all the possible cases that can reach an accepted state after executing the remaining program after l. The following theorem gives a *sufficient* condition that makes *Postset* sound.

Theorem 3.1. *Given a program P and a regular property φ, if φ is only parametric with objects, and the related objects of φ when running P are not data-dependent on the inputs, Postset is sound.*

Proof. *Postset* is computed by IFDS framework, and the flow functions result in an over-approximation when the event information is computed soundly. If there are some events of the property in which the requirements of values exist, *e.g.*, the return value of a method invocation must be true, it is beyond the capability of our static data flow analysis. Because the property φ is only parametric with objects, and we use pointer-to analysis to get the information of static objects, the event information computed by our static analysis is static object sensitive. Furthermore, the related objects of φ when running P do not depend on the inputs. Hence, we can obtain sound results of computing events, and thus *Postset* is sound. □

3.3 Compute History Information

Different from *Postset*, *Preset* is calculated dynamically. The *Preset* of a location contains the states that are reached via the execution from the program entry to the location. Similar to the guiding method [49], we adopt runtime verification [31] to calculate the *Preset* information. Note that we enhance the calculation method to support multi-object properties.

A runtime object is a *sensitive object* if the corresponding class or interface is specified by the property φ, *e.g.*, the sensitive objects of the property given by Figure 2 include the `ArrayList` object and the `Iterator` object. We create a monitor to monitor the method invocations of every *sensitive object*, and record the current state of each monitored object. Note that for a multi-object property, the monitors of related objects will be merged according to the property specification. *Preset* is a set of monitors. Formally, we use (I_s, q) to represent a monitor, where I_s is a set of object identities, and q is a state of the FSM $M_{\neg\varphi}$. If any object's method invocation results in an event of $M_{\neg\varphi}$, the monitor performs a state transition *w.r.t.* $M_{\neg\varphi}$ to compute *Preset*, which we formalize as

$$(I_s, q_1) \xrightarrow{o.e_1} (I_s, q_2) \tag{1}$$

where $o \in I_s$, and state $q_1 \in M_{\neg\varphi}$ is transited to state q_2 by event e_1 generated by the method invocation on the object identified by o.

For a multi-object property, a monitor's sensitive objects are not created simultaneously. Suppose sensitive object o produces event e_1, if o is related to the objects in I_s *w.r.t.* the property φ and $o \notin I_s$, following rule is used, where e_1 makes a transition from q_1 to q_2:

$$(I_s, q_1) \xrightarrow{o.e_1} (I_s \cup \{o\}, q_2) \quad (2)$$

Take the second iteration of the motivation program (*cf.* Figure 4) for example, and suppose o_1 and o_2 represent the identities of the `ArrayList` and `Iterator` objects, respectively. Obviously, the encountered event sequence at Line 11 is $\langle o_1.iterator, o_1.remove\rangle$. As a result, according to the FSM in Figure 2, the *Preset* of location line 11 is $((o_1, o_2), q_3)$.

For a given branch br, the soundness of history information means that *Preset* contains all the reached states from the initial state via the path from the beginning of the program to branch b.

Theorem 3.2. *Given a program P and a regular property φ, if the sensitive objects of P are not data-dependent on the inputs, Preset is sound.*

Proof. Recall that we compute the history information along with DSE, and *Preset* is calculated as the reached state set of the other branch of br, denoted by $\neg br$, executed in the current path. Suppose the corresponding event sequence to $\neg br$ is $\langle\rangle$ or $\langle e_0, \ldots, e_i\rangle$ where $i \geq 0$. Because the sensitive objects do not depend on the inputs, the event sequence of br is the same as that of $\neg br$. Hence, *Preset* of br contains the same reached states of $\neg br$, which implies the soundness of *Preset*. □

A static object is identified by its creation location. We relate a runtime object with a static object by their creation locations. Given the *Preset* and *Postset* of a branch br, there *intersection*, denoted by $Preset(br) \sqcap Postset(br)$, is defined as follows, where $S_t(I_s)$ represents the set of the creation locations of the objects identified by I_s, $S(T_a)$ denotes the set of all elements in tuple T_a, and $M(X_1, X_2)$ denotes $X_1 \subseteq X_2 \vee X_2 \subseteq X_1$.

$$\{q \mid \exists (I_s, q) \in Preset, (T_a, q_r, b, s) \in Postset \bullet M(S_t(I_s), S(T_a)) \wedge q \in q_r\} \quad (3)$$

Take the candidate branch b_3 in the first iteration of the motivation program (*cf.* Figure 3) for example. There exists a data fact, *i.e.*, $((o_1, o_2), \{q_1 \sim q_4\}, b, \langle \mathsf{U}, \mathsf{H}, \mathsf{N}\rangle)$, in the *Postset* below Line 10, where o_1 and o_2 correspond to the identities of the static `ArrayList` and `Iterator` objects, respectively. The computed *Preset* at Line 10 is $((o_1', o_2'), q_1)$, where o_1' and o_2' represent the `ArrayList` object and `Iterator` object, respectively. Clearly, the set of the creation locations of o_1' and o_2' is equal to (o_1, o_2). Besides, since $q_1 \in \{q_1 \sim q_4\}$, we have q_1 in the intersection of *Preset* and *Postset*.

3.4 Regular Property-Oriented Path Slicing

We now describe the algorithm for regular property-oriented path slicing, which is based on path slicing [27]. Specifically, our slicing algorithm enhances path slicing through exploiting the guiding information, *i.e.*, *Preset* and *Postset*, to prune additional branches.

Before elaborating the slicing algorithm, we first give the definition of the equivalence relation of event sequences. An FSM M is a triple $(\Sigma, Q, q_0, \delta, F)$, where Σ is the event alphabet, Q is the state set, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and F is the set of accepted states. An event sequence $s = \langle e_1, \ldots, e_n\rangle$ $(n \geq 1)$ is accepted by M if for each e_i, where $1 \leq i \leq n$, there exists $q_i \in Q$ such that $(q_{i-1}, e_i, q_i) \in \delta$, and $q_n \in F$. We use $\mathcal{R}(s)$ to denote the event sequence after removing any event e_k in s that makes a self-circled transition, *i.e.*, $(q_{k-1}, e_k, q_k) \in \delta$ and q_{k-1} is equal to q_k.

Definition 3.1. Given an FSM M and two accepted event sequence s_1 and s_2 of M, s_1 and s_2 are *equivalent* (denoted by $s_1 \equiv_M s_2$) iff $\mathcal{R}(s_1) = \mathcal{R}(s_2)$.

For example, the sequences $\langle iterator, update, hasNext, next\rangle$ and $\langle iterator, update, hasNext, next, hasNext\rangle$ are equivalent *w.r.t.* the FSM in Figure 2.

Algorithm 2: Regular Property-Oriented Path Slicing

```
   Slice(P, path_c, M_¬φ, Preset, Postset)
   Data: program P, a path path_c, and FSM M_¬φ
1  begin
2      i ← tail(L(path_c)); S ← ∅;
3      while i ≠ null do
4          if i is a branch instruction then
5              if Preset(¬i) ⊓ Postset(¬i) ≠ ∅ then
6                  T←Concatenate(¬i);
7                  if ∃s1 ∈ T, ∀s2 ∈ X • ¬(s2 ≡_{M^L_¬φ} s1) then
8                      if any e ∈ Seq(path_c) depends on i
                        ∨ ¬i can reach any new event
9                      then
10                         S ← S ∪ {i};
11         else
12             if any e ∈ Seq(path_c) depends on i then
13                 S ← S ∪ {i};
14         i ← before(i);
15     return S;
```

Algorithm 2 gives our regular property-oriented path slicing. The input to the algorithm consists of the program P under verification, the current path $path_c$, and the FSM $M_{\neg\varphi}$ corresponding to the negation of the regular property to be verified. The algorithm processes the instructions in $path_c$ in a backward manner, where $\mathcal{L}(path_c)$ denotes the instruction list of $path_c$. Finally, all the remaining instructions are stored in $\mathcal{S}$ and returned. The branches not in S are pruned.

For a branch instruction i in $\mathcal{L}(path_c)$, we use $\neg i$ to represent i's branch not explored in $path_c$. To decide whether i can be pruned, we exploit *Preset* and *Postset* in two ways. First, considering that both $Preset(\neg i)$ and $Postset(\neg i)$ are sound, the emptiness of their intersection implies that there exists no path along $\neg i$ that can violate the property. Hence, we can slice branch i if $Preset(\neg i) \sqcap Postset(\neg i)$ is empty (Lines 5). Second, we also prune branch i if all possible accepted event sequences of $\neg i$ have an equivalent accepted event sequence explored before (Lines 6&7). The Concatenate operation concatenates the event sequence before $\neg i$ (recorded during running) and the possible event sequences after $\neg i$ (calculated in *Postset*), and produces an *accepted* event sequence set T. If every sequence in T has an equivalent sequence in $\mathcal{X}$ (*cf.* Algorithm 1), which means that all the possible accepted event sequences along $\neg i$ have been explored in an equivalent manner, i can be sliced. We also use the criteria in path slicing [27] to further slice the instructions

that cannot alter the event sequence of the current path $path_c$, *i.e.*, a branch instruction i can be sliced if there is no event in $Seq(path_c)$ transitively data- or control- depends on it and no new event can be reached along the direction of $\neg i$ (Lines 8-10), similar rules also apply to the remaining types of instructions (Lines 12&13).

Path slicing [27] also slices a program execution path in a backward manner. More precisely, it keeps track of a set of variables (called *live set*) that determines the feasibility of the suffix of the path's event sequence and the latest remained instruction (called *step location*). A branch instruction i will be remained if one of the following three conditions is satisfied. (1) $\neg i$ can bypass the *step location*; and (2) there exists a path in the direction of $\neg i$ that can modify the sensitive variables in *live set*; and (3) there exists a path in the direction of $\neg i$ that can reach a new event. Actually, the first two conditions correspond to transitive data and control dependence, respectively. On the other hand, a normal instruction will be remained if it can modify the variables in *live set*.

Concatenation. The insight of concatenation is to infer the possible accepted event sequences of a branch br based on $Preset(br)$ and $Postset(br)$. For an element (T_a, q_r, b, s) in $Postset(br)$, we use $Aes(br, q)$ to represent the set of statically calculated event sequences that can drive q to an accepted state, which can be derived from *Postset* as follows, where D_c is the set of calculated data facts during data flow analysis and $D_c \subseteq D$.

$$Aes(br, q) = \{s_i \mid (T_a, q_r, b, s_i) \in D_c \wedge q \in q_r\} \quad (4)$$

Then, we define Concatenate(br) as the following set

$$\{s_1 \circ s_2 \mid \exists\, q \in Preset(br) \sqcap Postset(br) \bullet s_2 \in Aes(br, q)\} \quad (5)$$

where s_1 is the event sequence before br. A natural way to obtain the accepted event sequences along br is to concatenate the event sequence produced before br and the event sequences in $Aes(br, q)$.

In principle, to ensure the soundness of slicing, both s_1 and $Aes(br, q)$ need to be context and flow sensitive [37]. In practice, $Aes(br, q)$ is flow-sensitive, but not context-sensitive. We check the equivalence relation *w.r.t.* a *location sensitive* variant of $M_{\neg\varphi}$, denoted by $\mathcal{M}^L_{\neg\varphi}$. The intuition is that the reasons of the bugs caused by the same statement under different contexts tend to be the same. $\mathcal{M}^L_{\neg\varphi}$ can be computed according to the program P in two steps. First, we collect all possible static locations for every event by an inter-procedural control flow analysis. Second, we replace every transition with the transitions of location sensitive events.

In the illustration example, after the second iteration (*cf.* Figure 4), the accepted sequence $\langle iterator_5, update_{11}, hasNext_{12}, next_{13}\rangle$ is added to $\mathcal{X}$. $\mathcal{M}^L_{\neg\varphi}$ is the FSM after replace the event of each transition in Figure 2 by the event with location information. Then, according to the *Preset* and *Postset* of b_2, $Preset(b_2) \sqcap Postset(b_2)$ only contains q_1. Based on the example in Section 3.2, $Aes(b_2, q_1)$ has four event sequences. Besides, the sequence before b_2 is $\langle iterator_5\rangle$. Hence, Concatenate($b_2$) contains four accepted event sequences, each of which is equivalent to $\langle iterator_5, update_{11}, hasNext_{12}, next_{13}\rangle$ *w.r.t.* $\mathcal{M}^L_{\neg\varphi}$. Therefore, b_2 is pruned.

3.5 Branch Selection

For a regular property φ, only the relevant paths with specific event sequences can violate φ. It is desirable to evaluate a branch's probability of generating the accepted paths *w.r.t.* $M_{\neg\varphi}$. Then, after slicing, the branch with a higher probability will be selected first, in order to find counter-example paths earlier. Same as the regular property guided DSE [49], we use the size of $Preset \sqcap Postset$ as the main heuristic value for evaluating a branch. When two branches have the same size of $Preset \sqcap Postset$, the deeper one will be selected for efficient exploration.

3.6 Discussions

In principle, slicing and guiding are the orthogonal techniques that are synergistically combined in SRV. Slicing prunes irrelevant and equivalent relevant paths during DSE, while guiding boosts finding counterexample paths. Both are important for boosting verification. Since slicing is used when a path is completed, the effectiveness of slicing is related to how fast completed paths are generated. For example, if there exist very short paths in a program, BFS may be a good choice. On the other hand, guiding is insensitive to the search strategy. Without any knowledge of the shape of the path space, we integrate these two techniques with the DFS strategy.

In addition to their compatibility, slicing can boost the efficiency of property guiding. Finding accepted paths may be hindered by exploring relevant paths due to the imprecision of the guiding method. With the help of slicing, after one path is explored, the equivalent ones of the path can be pruned. Therefore, compared with pure guiding, SRV tends to find counterexamples faster.

Since slicing performs static analysis, its overhead may be high, *e.g.*, when the path is long and the control flow graph is complex. To improve the performance, we perform slicing selectively based on the history results of slicing. The intuition is the locality of program paths, *i.e.*, if a branch cannot be pruned, it tends not to be pruned either along the nearby paths. If the branch of any constraint in the current path is pruned at last time or first encountered, slicing would be carried out. With the help of such a lightweight dynamic prediction, we can reduce the effort of slicing, achieving a good balance between path pruning and slicing overhead.

Theorems 3.1&3.2 give sufficient conditions for the soundness of $Preset$ and $Postset$, respectively. Even though, there still exist a large number of properties and programs satisfying such conditions, e.g., the ones used in our experiment. However, considering our approach for computing the event sequences in $Postset$ is not context sensitive, SRV is not sound, *i.e.*, SRV may miss bugs resulting from different contexts but having same static location. We believe SRV is practical, because the root causes for the bugs of the same statement under different contexts tend to be the same.

4 IMPLEMENTATION AND EVALUATION

We have realized a prototype of SRV based on a regular property guided DSE tool [47], which is implemented on the DSE engine JPF-JDart [26] and WALA [24] static analysis platform. We have developed a property-oriented path slicer for Java bytecode based on Javaslicer [15], *i.e.*, a dynamic slicing [45] tool.

We evaluate SRV along two dimensions:

- **Effectiveness and efficiency of SRV**. Can SRV effectively verify regular properties for real-world Java programs? How efficient is SRV compared with DFS, pure guiding and pure slicing?
- **Synergy between slicing and guiding**. Can slicing boost guiding in finding counterexample paths? How significant is the improvement?

4.1 Evaluation Setup

Table 1 lists the programs in our evaluation, which are all real-world open-source Java programs, totaling 259K lines of code (LOC). `Rhino-a` and `soot-c` come from the Ashes benchmark[1] suite. `Jlex` is a Java lexical analyzer. `Bloat` is from the DaCapo benchmark suite [6]. `BMPDecoder` is a decoder for BMP files. `Ftpclient` is an FTP client. The six programs, `pobs`, `jpat`, `jericho`, `nano-xml`, `htmlparser` and `xml`, are parser programs. The remaining, *i.e.*, `fastjason`, `jep` and `udl`, are library programs.

Table 1: Programs in the experiments

Program	LOC	Brief Description
rohino-a	19799	Javascript interpreter
soot-c	32358	Static analysis editor
jlex	4400	Lexical analyzer
bloat	45357	Java bytecode optimization
bmpdecoder	531	BMP file decoder
ftpclient	2436	FTP client in Java
pobs	5488	Java parser objects
jpat	3254	Java string parser
jericho	25657	Jericho HTML Parser
nano-xml	3317	Non-validating XML parser
htmlparser	21830	HTML parser in Java
xml	5138	XML parser in Java
fastjson	20223	JSON library from alibaba
jep	42868	Mathematics library
udl	26896	UDL language library
Total	**259642**	**15 open source programs**

As Table 2 shows, we applied SRV to verify widely-used regular properties, including both single- and multi-object ones. Properties with superscript $*$ are multi-object ones; the remaining are single object properties. In addition, we also verify some user-defined properties. For example, the property we defined for `htmlparser` requires that the input string is in the JSP format, *i.e.*, " $<\%\ldots\%>$ ".

Table 2: Regular properties in the experiments

Property	Meaning
Enumeration	Call hasMoreElements before nextElement
Iterator	Call hasNext before next
	Do not update the collection while iterating*
Reader	Do not read a closed stream
	No read if dependent input stream closed*
Writer	Do not write a closed stream
	No write if dependent output stream closed*
Socket	Do not use a closed socket

Since most of the programs are violation free, to further evaluate our method, we mutate [28] the programs. First, we collect all the branch statements along DSE, then we randomly select a branch to automatically inject an event, *e.g.*, a `close` operation for the `Reader` property. We generate three mutants for each program, except for those with user-defined properties. Note that such injections may not necessarily lead to real violations.

An evaluation task comprises a program and a property. A task was run in four modes: DFS (D), pure guiding (G), pure path slicing (S) and SRV (S′). Under each mode, the time limit is 1 hour. All the experiments are carried out on the identical servers, each of which has 256GB RAM and four 2.13GHz XEON CPUs with 32 cores.

[1] http://www.sable.mcgill.ca/ashes/

4.2 Evaluation Results

Table 3 lists evaluation results. The first column gives the verification tasks, including the names of the programs and the verified properties, where a multi-object property has a superscript $*$. The second column **Type** indicates whether an analyzed program has been mutated or not, with O represents the original program, and bug_i the ith mutant. The column **Total Time(s)** lists time consumed for each verification task in four modes, where TO represents time-out. In our evaluation, completing a verification means having explored all the path space. The time for finding the first counterexample is shown in the column **First Violation Time(s)**, where NO means no counterexample path and NA represents unknown due to timeout.

Table 3 shows that SRV completes 30 tasks in 39, while DFS (D), pure guiding (G) and pure slicing (S) complete 22, 22 and 23 tasks, respectively. Compared with these alternatives, SRV achieves 36%, 36% and 30% improvement, respectively. For the successfully verified 30 tasks, SRV at least has an average 8.4X, 8.6X, and 7X speedups over DFS, pure guiding and pure slicing, respectively.[2] We inspected the programs that SRV fails to verify, *i.e.*, `jlex`, `rhino-a`, `htmlparser` and `udl`, and found that those programs have complex and sensitive control flow. Most paths of those programs are relevant, and only a few paths are counterexample paths. As a result, the overhead of static analysis used in slicing and guiding becomes very high, and only a small part of path space can be pruned.

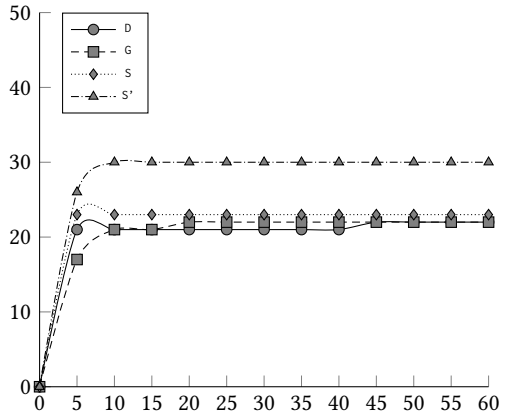

Figure 6: Completed tasks under a time threshold.

Figure 6 shows the relationship between the number of completed verification tasks and the time threshold. The X-axis varies the time threshold from five minutes to one hour, while the Y-axis is the number of completed verification tasks. SRV can complete the most tasks under a given time threshold. In addition, all the completed 30 tasks by SRV are completed within 5 minutes, demonstrating SRV's efficiency.

To show the effectiveness of property guiding, we also collect the time for finding the first counterexample path. For the tasks in which violations exist, pure guiding is the most efficient to find the first violation. Due to the overhead of slicing, SRV is slower than pure guiding in most tasks, but has the same order of magnitude in time. Both SRV and pure guiding achieve orders of magnitude

[2] A timeout is counted as 1 hour.

Table 3: Experiment results of analysis time (D: DFS, G: pure guiding, S: pure path slicing, S′: SRV)

Program (Property)	Type	Total Time(s)				First Violation Time(s)			
		D	G	S	S′	D	G	S	S′
soot-c (Writer)	O	28.43	374.23	95.14	398.98	NO	NO	NO	NO
	bug1	28.59	354.26	101.32	413.83	18.41	343.72	85.01	413.71
	bug2	27.61	358.01	97.06	389.1	19.39	346.6	81.62	389.09
	bug3	26.71	369.87	104.37	469.51	15.12	358.99	83.77	469.39
soot-c (Writer*)	O	27.2	177.91	91.86	214.85	NO	NO	NO	NO
	bug4	29.82	187.74	97.91	219.26	NO	NO	NO	NO
	bug5	27.9	187.41	98.15	218.9	15.71	176.27	79.4	216.47
	bug6	29.2	174.32	103	206.73	NO	NO	NO	NO
bloat (Iterator)	O	24.49	48.1	57.56	66.97	10.13	36.02	35.2	50.14
bloat (Iterator*)	O	27.1	71.75	54.23	90.54	NO	NO	NO	NO
	bug1	29.2	71.81	102.07	128.85	NO	NO	NO	NO
	bug2	25.85	70.02	60.74	92.19	25.85	70.02	42.86	92.19
	bug3	26.63	72.05	64.57	96.88	26.63	70.95	40.11	78.92
bmpdecoder (Reader)	O	8.65	16.97	21.71	16.71	NO	NO	NO	NO
	bug1	9.18	17.45	19.79	22.04	NO	NO	NO	NO
	bug2	9.15	17.92	21.01	18	7.93	12.54	20.48	17.96
	bug3	9.26	17.88	20.97	23.26	NO	NO	NO	NO
ftpclient (Socket)	O	12.44	37.08	37.83	49.78	NO	NO	NO	NO
	bug1	14.12	41.48	42.12	55.1	9.19	36.44	38.89	54.55
	bug2	13.57	37.66	37.47	50.55	11.73	33.96	37.2	50.55
	bug3	15.52	40.45	40.29	53.39	NO	NO	NO	NO
jlex (Reader)	O	TO	TO	TO	29.48	NA	NA	NA	NO
	bug1	TO	TO	TO	TO	12.75	23.35	400.71	63.97
	bug2	TO	TO	TO	TO	NA	14.58	NA	27.04
	bug3	TO	TO	TO	TO	NA	NA	NA	NA
jlex (Reader*)	O	TO	TO	TO	29.81	NA	NA	NA	NO
	bug4	TO	TO	TO	TO	NA	20.07	NA	52.24
	bug5	TO	TO	TO	TO	217.56	38.18	NA	109.39
	bug6	TO	TO	TO	TO	51.33	146.88	NA	NA
rhino-a (Enumeration)	O	TO	TO	TO	TO	NA	NA	NA	NA
jpat (UserDefined)	O	TO	TO	TO	46.94	NA	23.36	NA	43.99
nano-xml (UserDefined)	O	TO	TO	TO	19.18	NA	14.02	NA	19.16
pobs (UserDefined)	O	TO	TO	21.44	26.31	NA	14.96	20.79	23.07
jericho (UserDefined)	O	TO	TO	53.7	27.66	NA	19.6	53.33	27.66
fastjason (UserDefined)	O	TO	TO	TO	102.6	NA	NA	NA	102.52
jep (UserDefined)	O	2590.38	1090.05	TO	167.87	1439.06	29.72	NA	167.84
htmlparser (UserDefined)	O	TO	TO	TO	TO	27.62	50.95	NA	106.03
udl (UserDefined)	O	TO	TO	TO	TO	NA	2829.57	NA	NA
xmlparser (UserDefined)	O	TO	TO	TO	24.89	NA	18.25	NA	24.89

speedups over DFS and pure path slicing in finding the first counterexamples. When a violation is very deep and there possibly exist a large number of relevant paths, it cannot be detected without slicing. For example, for `fastjason`, pure guiding fails to detect a violation within one hour, while SRV needs only 102.6 seconds. Within one hour, guiding and SRV can find a counterexample for 23 and 22 programs respectively, while DFS and pure path slicing can only find 15 and 13, indicating the effectiveness and efficiency of guiding.

Pruning branches with positive heuristic values can boost finding counterexamples. For the 24 tasks with counterexamples found, slicing can boost guiding by reducing the number of iterations for finding the first counterexample in 7 (29%) tasks. Notably, for `fastjason`, SRV needs only 5 iterations, but all the other modes fail to detect a violation after thousands of iterations. To inspect the boosting of slicing to guiding further, we collect the information of the pruned branches with positive heuristic values.

Figure 7 shows the improvement by synthesizing the results of all the tasks, where the X-axis is the path order for the first 2000 paths, and the Y-axis is the number of the pruned branches with a positive heuristic value for guiding. As shown in the figure, much of the boosting happens during the early stage, *i.e.*, in the first 1500 paths, which indicates the necessity of selective slicing.

In addition, we collect the information about iterations, and the results show that SRV uses the fewest iterations to complete path exploration. Specifically, the iterations using our slicing algorithm is two orders of magnitude less than that using path slicing [27]. Furthermore, we adjust the time threshold to 24 hours for the failed tasks, and found that all the tasks were still failed to be verified, except that program `jep` can be verified in pure slicing mode.

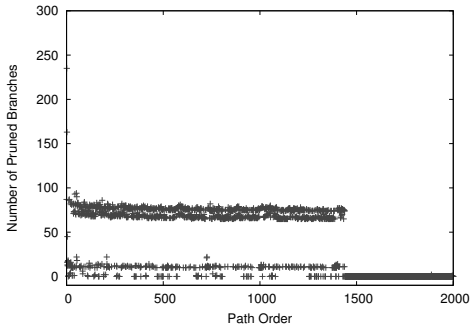

Figure 7: Branch pruning of each path.

Threats to Validity. Threats to the validity of our results are mainly external. The programs are representative because (1) the programs are of various types, such as parser and network manipulation; (2) the programs are of different sizes, *i.e.*, from 0.5K to 45K LOC, and 259K in total; and (3) they are commonly used for evaluating Java program analyses [7, 21, 49]. Furthermore, the verified regular properties are mainly common contracts [18] of Java programs. User-defined properties have practical meanings *w.r.t.* functionalities. Although SRV is implemented and evaluated for Java, it is general and can be applied to programs in other languages, such as C and C++. Finally, we set the time threshold to be 1 hour, which is fairly enough for a verification task. Increasing the time threshold to 24 hours only results in one more completed task.

5 RELATED WORK

The closest related work to SRV is regular property guided DSE [49] and Woodpecker [16]. Different from the objective of [49], *i.e.*, finding an accepted path *as soon as possible*, SRV aims to quickly complete the path exploration of the program by employing slicing to prune redundant paths, and the slicing can also reduce the iterations for finding counterexample paths. Compared with Woodpecker [16], which uses path slicing [27] to prune redundant paths for verifying system rules via symbolic execution, as demonstrated by the evaluation results (*cf.* Section 4.2), SRV is more scalable because it can prune more paths and find violations faster.

Meta Compilation (MC) [19, 20] is a scalable static approach to detecting violations of properties specified by a state machine language. MC is neither sound nor complete. ESP [17] is a path-sensitive static verifier for regular properties. ESP achieves strong scalability by merging symbolic states. However, ESP may produce false alarms due to imprecise modeling of program statements. In [21], a staged static typestate property [43] verification framework is proposed based on a parametric abstract domain. The false alarms can be eliminated gradually by the staged analysis. Clara [7] employs forward and backward data flow analysis to remove instrumentations for runtime monitoring of typestate properties. Our guiding method makes the backward analysis of Clara to be interprocedural for calculating *Postset*. Compared with static approaches, SRV enjoys completeness by trading scalability because it executes the program under verification.

Dynamic methods are mainly from runtime verification [31]. The basic procedure is to generate a monitor for verification from a property, and the monitor is usually implemented via instrumentations to the program. The verification takes place at runtime based on the information collected by instrumentations. Hence, dynamic approaches verify a single program path. JavaMOP [12] and Tracematches [2] are representative tools for runtime verification of Java programs. The calculation of *Preset* uses the idea of monitoring in runtime verification, and the monitoring is implemented at the virtual machine level. Compared with dynamic approaches, SRV employs DSE to explore the path space of the program systematically, which improves code coverage and finds more bugs.

Software model checking has also been used for regular property verification. SLAM [4] uses predicate abstraction [5] to obtain an abstract model of a program. Then, at the model level, SLAM uses model checking to verify regular properties. When a counterexample is found by model checking, it is reported when it is a real violation; otherwise, the counterexample is used to refine the abstract model. YOGI [38] improves SLAM by integrating DSE to speed up model refinement and finding real counterexamples. Compared with these approaches, SRV is lightweight and scalable because it adopts efficient static analysis to boost verification.

Furthermore, guiding and pruning are commonly investigated for improving the scalability of symbolic execution. For guiding symbolic execution, different methods are proposed *w.r.t.* different goals, including improving code coverage [9, 10, 32, 42, 44, 46], reaching a program location [3, 11, 34, 48], targeting the differences between two program versions [35, 39], aiming at the unverified path space [13], and generating a path satisfying a regular property [49]. On the other hand, pruning path space is also an effective method to mitigate path explosion. Same as guiding, the existing work on pruning also differs in their perspectives to decide redundance, such as read-write information [8], assertion violation [25], and rule violation [16]. SRV extends the existing work by the synergy of guiding and pruning for verifying regular properties.

6 CONCLUSION

This paper presents symbolic regular verification, a practical DSE-based technique for verifying regular properties. To improve scalability, we introduce a synergistic combination of property-oriented path slicing and guiding. SRV's property-oriented path slicing prunes redundant paths, while guiding helps finding counterexamples quickly. The two combined techniques not only complement, but also strengthen each other. We have developed a prototype of SRV for Java and evaluated it on real-world programs *w.r.t.* widely-used regular properties. Our extensive evaluation demonstrates that SRV is effective and efficient, and outperforms the state-of-the-art significantly for regular property verification. Interesting future work includes (1) techniques to further reduce slicing overhead and (2) further improvements to our tool's usability and feasibility for releasing to and benefiting the community.

ACKNOWLEDGEMENT

This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and NSFC Program (No. 61472440, 61632015, 61690203, and 61532007). Zhendong Su was supported in part by the United States National Science Foundation (NSF) Grants 1528133 and 1618158, and by a Google Faculty Research Award.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools* Addison-Wesley, 1986.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[3] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, pages 12–22. ACM, 2011.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[7] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE*, pages 5–14, 2010.

[8] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*, pages 351–366, 2008.

[9] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE* pages 443–446, 2008.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[11] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.

[12] F. Chen and G. Rosu. MOP: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[13] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155, 2016.

[14] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[15] Clemens Hammacher, Martin Burger, and Valentin Dallmeier. JavaSlicer. https://www.st.cs.uni-saarland.de/javaslicer/, 2008.

[16] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS*, pages 329–342, 2013.

[17] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[19] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[20] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.

[21] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

[22] A. Gill et al. Introduction to the theory of finite-state machines. 1962.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[24] IBM. T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net/.

[25] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *FSE*, pages 48–58, 2013.

[26] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NASA Formal Methods*, pages 121–125, 2009.

[27] R. Jhala and R. Majumdar. Path slicing. In *PLDI*, pages 38–47, 2005.

[28] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

[29] U. P. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009.

[30] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[31] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[32] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32, 2013.

[33] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[34] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, pages 95–111, 2011.

[35] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, pages 716–726, 2012.

[36] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[37] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2015.

[38] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The YOGI project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181, 2009.

[39] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[40] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[41] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[42] H. Seo and S. Kim. How we get there: a context-guided search strategy in concolic testing. In *FSE*, pages 413–424, 2014.

[43] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, (1):157–171, 1986.

[44] N. Tillmann and J. De Halleux. Pex – white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[45] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 1995.

[46] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pages 359–368, 2009.

[47] H. Yu, Z. Chen, Y. Zhang, J. Wang, and W. Dong. RGSE: a regular property guided symbolic executor for java. In *FSE*, pages 954–958, 2017.

[48] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.

[49] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. In *ICSE*, pages 643–653, 2015.