Regular Property Guided Dynamic Symbolic Execution

Yufeng Zhang*†, Zhenbang Chen†§, Ji Wang*†, Wei Dong† and Zhiming Liu‡
*State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
†College of Computer, National University of Defense Technology, Changsha, China
‡Centre for Software Engineering, Birmingham City University, Birmingham, UK
Email: {yufengzhang, zbchen, wj, wdong}@nudt.edu.cn, zhiming.liu@bcu.ac.uk
§Corresponding author

***Abstract*—A challenging problem in software engineering is to check if a program has an execution path satisfying a regular property. We propose a novel method of dynamic symbolic execution (DSE) to automatically find a path of a program satisfying a regular property. What makes our method distinct is when exploring the path space, DSE is guided by the synergy of static analysis and dynamic analysis to find a target path as soon as possible. We have implemented our guided DSE method for Java programs based on JPF and WALA, and applied it to 13 real-world open source Java programs, a total of 225K lines of code, for extensive experiments. The results show the effectiveness, efficiency, feasibility and scalability of the method. Compared with the pure DSE on the time to find the first target path, the average speedup of the guided DSE is more than 258X when analyzing the programs that have more than 100 paths.**

I. Introduction

A regular property of a program can be represented by a Finite State Machine (FSM). The FSM is an abstract model of the executions of the program. A label (or *event*) of a transition corresponds to the execution of one or more statements in the program *relevant to the property*. FSM can describe many kinds of program properties including resource usage (*e.g.*, file usage [1]), memory safety (*e.g.*, memory leak [2]), communication protocol [3], *etc*. In software engineering, regular properties and FSMs are widely used in different techniques, such as model-based testing [4], typestate analysis [5], and specification mining and synthesis [6]. These techniques encounter the common problem of *checking whether there exists a path of a program that satisfies a regular property*. Therefore, finding effective solutions to this problem and their implementations are essential to many applications.

Static analysis and *dynamic analysis* are the two effective approaches used for checking regular properties. Static analysis, *e.g.*, [7], [1], [5], usually enjoys the advantage of high coverage and good scalability. But its users are often bothered by false alarms, due to the extra behaviors introduced by over approximation. The dynamic approach [8], [9] has the advantage of no false alarm, and can provide precise information (such as the input) for replay. However, it is confined by the problem of limited input coverage as only the program executions of the selected inputs can be checked. For example, traditional software testing [4] has low coverage. Furthermore, techniques pursuing coverage, such as automatic test generation [10], are difficult to scale up.

Dynamic symbolic execution (DSE) [11], [12] enhances traditional symbolic execution [13] by combing concrete execution and symbolic execution. DSE repeatedly runs the program both concretely and symbolically. After each run, all the branches off the execution path, called the *off-path-branches*, are collected, and then one of them is selected to generate new inputs for the next run to explore a new path. Hence, DSE improves the coverage through symbolic execution, and avoids false alarms by actually running the program. More importantly, DSE can use information of the concrete execution to simplify symbolic reasoning and handle environment modeling.

When DSE is applied to checking a program against a regular property, an execution path *satisfies* the property *if* the sequence of the events in the path is accepted by the FSM of the property, and we call this path an *accepted path*. However, the number of paths is exponential with the number of branch statements executed during DSE, which brings the path explosion problem. Therefore, how to guide DSE to find a path satisfying the regular property *as soon as possible* is a challenging problem.

We propose a novel DSE approach, called *regular property guided DSE*, to find the program paths satisfying a regular property as soon as possible. Our approach is based on the key insight that only the paths with specific sequences of events can satisfy the regular property. The portion of the accepted paths is often very small. It is desirable not to explore the *irrelevant* paths (*i.e.*, the paths not containing any event in the FSM) and the *relevant* paths not satisfying the property. However, it is impossible to avoid all these paths. What we propose is to explore the off-path-branches along which the paths are *most likely* to satisfy the property.

The novelty of the guided DSE is the design of the algorithm to evaluate the possibility of an off-path-branch along which there exists an accepted path. To this end, the evaluation uses the history and future behaviors of the off-path-branch. For an off-path-branch b, the *history* of b is described by the states of the FSM that the current execution path has reached up to b. The *future* of b is described by the states of the FSM from which the final state can be reached by the execution of the

program after b. We calculate the history in the DSE process and the future statically using backward dataflow analysis. If the intersection of the history and future of b is not empty, there is likely a path along b satisfying the regular property.

We have implemented the guided DSE for Java programs, based on JPF-JDart [14] and WALA [15]. The tool is applied to analyze 13 real-world open source Java programs, a total of 225K lines of code, against representative regular properties. The experimental results indicate that the guided DSE is more efficient than the pure DSE with a lower time overhead and an acceptable memory overhead.

The main contributions of this paper are as follows:

- The algorithm for guiding DSE with respect to regular properties: with the synergy of static analysis and dynamic analysis, the algorithm can find the program paths satisfying a regular property effectively and efficiently.
- The prototype tool of the guided DSE for Java programs: it is applied successfully to analyze 13 real-world Java programs with respect to regular properties.
- Extensive experiments: the results show that the guided DSE has 1) an average >1880X speedup on the iterations needed to find the first accepted path, and 2) an average >258X time speedup for finding the first accepted path on the programs whose path space is bigger than 100. Furthermore, for 3 out of the 13 programs, guided DSE successfully finds an accepted path in one hour, reproducing a known typestate bug; whereas, the pure DSE fails for these 3 programs in 24 hours.

The rest of this paper is as follows. Section II motivates our approach. Section III elaborates the details. Section IV presents the implementation and evaluation. We discuss related work in Section V, and conclusion in Section VI.

II. Motivating Example

To motivate the guided DSE, Fig. 1 shows a program that uses an `InputStreamReader` to read a file. Procedure `foo` has three integer parameters and two loops in lines 6−11 and lines 17−22, respectively. If `tag` is 0, the reader `w`, an instance of `InputStreamReader`, is closed before the second loop (line 14), and a `java.io.IOException` may be thrown at line 19. For a reader, we are concerned with the regular property that method `read` is performed after method `close`. Fig. 2 is the FSM, denoted by M_r, of this property, where `init` (object initialization, *e.g.*, line 3), `close` and `read` method invocations are the events. We suppose that an event is atomic, *i.e.*, no other events can be observed during its execution.

Consider the initial input $I_1 = (m = 1, n = 1, tag = 1)$. The first iteration of DSE is shown by Fig. 3. In the first execution path, shown by the leftmost path of the graph, the body of each loop is executed once only. This execution path generates the sequence "*init*, *read*, *read*" of events, driving M_r to state q_1. For this path, the path condition PC_1 is the conjunction of five conditions: $m > 0$, $m \leq 1$, $tag \neq 0$, $n > 0$ and $n \leq 1$, denoted by c_1−c_5. Here, c_1 and c_2 are from statement st_1 `while(k++<m)`, c_3 from st_2

```
1  int foo(int m, int n, int tag) { //{q0}
2     File file = new File("..."); //{q0}
3     InputStreamReader w = new InputStreamReader(
          new FileInputStream(file)); //{q1,q2,q3}
4     int result = 0; //{q1,q2,q3}
5     int k = 0; //{q1,q2,q3}
6     while (k++ < m) //{q1,q2,q3}
7     { //{q1,q2,q3}
8        int i = w.read(); //{q1,q2,q3}
9        if (i == -1) break;//{q1,q2,q3}
10       result += i; //{q1,q2,q3}
11    } //{q1,q2,q3}
12    if (tag == 0) //{q1,q2,q3}
13    { //{q1,q2,q3}
14       w.close(); //{q2,q3}
15    } //{q2,q3}
16    k = 0; //{q2,q3}
17    while (k++ < n) //{q2,q3}
18    { //{q2,q3}
19       int i = w.read(); //{q2,q3}
20       if (i == -1) break;//{q2,q3}
21       result -= i; //{q2,q3}
22    } //{q3}
23    return result;
24 }
```

Fig. 1. An example program

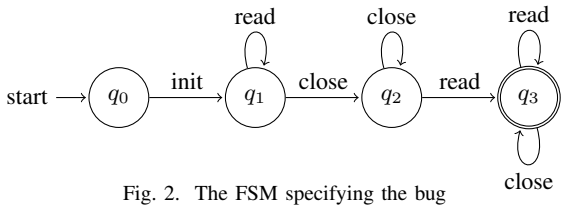

Fig. 2. The FSM specifying the bug

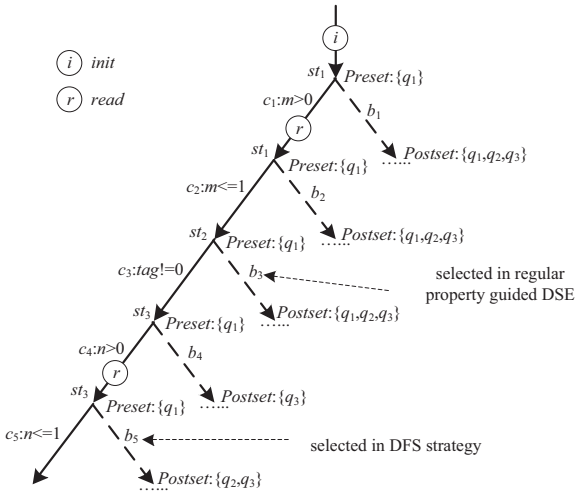

Fig. 3. Difference between guided DSE and pure DSE

`if(tag==0)` and c_4, c_5 from st_3 `while(k++<n)`. Let $b_1 \ldots b_5$ be the corresponding off-path-branches of $c_1 \ldots c_5$, respectively. Clearly, to reach state q_3 of M_r, we need an input that can steer the execution to cover line 14, which lies immediately after b_3. We can see b_3 needs the least iterations to get an accepted path.

With depth first search strategy, b_5 will be selected and we have $PC_1' = \langle m > 0 \wedge m \leq 1 \wedge tag \neq 0 \wedge n > 0 \wedge n > 1 \rangle$. Consider the generated input from PC_1' is $I_2 = (m = 1, n = 2, tag = 1)$. Then, in the second iteration, the path condition is $\langle m > 0 \wedge m \leq 1 \wedge tag \neq 0 \wedge n > 0 \wedge n > 1 \wedge n \leq 2 \rangle$, and

the body of the second loop is executed twice. Similarly, in the subsequent iterations, the second loop would be unrolled repeatedly until the upper bound is reached. This implies that line 14 will not be executed until the conditions from the second loop (line 17) are drained. If we use breath first search strategy, we still need many iterations before reaching line 14. The reason is that the two loops generate a plenty of conditions during DSE, and the search procedure does not know it is necessary to execute the statement at line 14 to reach q_3.

With the same initial input, the guided DSE needs only *two iterations* to find the target path. The guided DSE first calculates the future behavior information, denoted by $Postset$, of each program point pt using a context sensitive dataflow analysis. For each state q in $Postset$, there exists at least one static path[1] along pt that can drive the FSM from q to a final state. We use the comment of a line in Fig. 1 to show the $Postset$ of the program point below that line. For example, the comment in line 11 indicates that the program after line 11 (not including line 11) may reach q_3 from state q_1, q_2 or q_3. Also, the guided DSE calculates the history behavior, denoted by $Preset$, of an off-path-branch b. The $Preset$ of b contains the states of the FSM that can be reached by the execution up to b. Fig. 3 shows the $Preset$ of b_3 is $\{q_1\}$. This is because the event sequence "*init*, *read*" before b_3 reaches state q_1. Fig. 3 displays the $Preset$ and $Postset$ of $b_1 \dots b_5$.

The guided DSE uses $Preset$ and $Postset$ to select the off-path-branch to explore preferentially. If the intersection of the $Preset$ and $Postset$ of an off-path-branch b is *empty*, there will be no accepted path along b, and b is given a low priority to be selected for exploration[2]. For example, b_5 is not selected for the next iteration, as the $Preset$ and $Postset$ of b_5 are $\{q_1\}$ and $\{q_2, q_3\}$, respectively.

In the case when the intersection of the $Preset$ and $Postset$ of b is not empty, the paths along b are possible to satisfy the property. For different b and b', if the interaction of the $Preset$ and $Postset$ of b contains a larger number of states than that of b', the guided DSE assigns a higher priority to b. When the interactions of the $Postset$ and $Preset$ of b and b' have the same number of states, a higher priority is given to the deeper branch to limit the repetition of path exploration. In our example, b_3 is selected and $PC'_1 = \langle m > 0 \wedge m \leq 1 \wedge tag = 0 \rangle$. The execution with the new input generated by solving PC'_1 covers line 14 to find an accepted path in only two iterations.

III. Regular Property Guided DSE

In this section, we elaborate the details of regular property guided DSE.

A. DSE Algorithm

Algorithm 1 shows the worklist based search procedure. The inputs are the target program P, the FSM M_p and the initial input I_0. The algorithm explores the path space of P to find the paths that can be accepted by M_p.

[1]A static path is only a static path segment, but not necessarily feasible.

[2]In Section III-B, we explain why b is not deleted.

Algorithm 1: Regular Property Guided DSE

```
   Data: program P, FSM M_p, initial input I_0
1  begin
2      worklist ← ∅;
3      I ← I_0;
4      ComputePostset(P, M_p);
5      while true do
6          runAndMonitor(I, M_p);
7          if reach final state then
8              report path;
9          saveOffPathBranches(worklist);
10         if worklist = ∅ then
11             exit;
12         PC ← getFirstSat(worklist);
13         I ← solver(PC);
```

The *worklist* stores the off-path-branches that are yet to be explored in order. Initially, the *worklist* is empty, and the input I is set to the initial input I_0, which is generated randomly or manually (line 3). The function *ComputePostset*(P, M_p) (line 4) computes the $Postset$ of each location in P against M_p (elaborated in Section III-D). Function *runAndMonitor*(I) feeds P with input I and executes P both symbolically and concretely (line 6). During the execution, M_p is used to check whether the sequence of the events in the execution path satisfies the regular property. When a final state of M_p is reached, the current input I and the path being explored are reported (line 8). Since DSE executes the program concretely, Algorithm 1 does not generate false alarms. Then, the function *saveOffPathBranches*(*worklist*) saves all the off-path-branches to *worklist* in the order of their priorities (discussed in Section II and elaborated in Section III-B).

The function *getFirstSat*(*worklist*) fetches the first feasible off-path-branch from *worklist*, according to which the next input is generated. The `while` loop (from line 5 to line 13) iteratively explores different paths until *worklist* is empty (line 10). More details, such as divergence handling, are omitted for brevity.

The main procedure of Algorithm 1 differs from the standard worklist based search procedure of DSE. Firstly, each execution is monitored to check whether the current path can be accepted by M_p. Secondly, the $Preset$ for each off-path-branch is calculated and maintained during execution. Thirdly, the evaluation of an off-path-branch is new, which is discussed in the following subsection.

B. Evaluation Function

To find the paths that can be accepted by M_p with the least number of iterations, the evaluation function should be designed to assign the highest priority to the off-path-branch b such that: 1) there is a path along b that can be accepted by M_p, and 2) the distance (measured by the number of branch statements) between b and the final event before reaching the final state is the shortest.

There are two difficulties in deciding the priority of an off-path-branch b. The first difficulty is how to evaluate the

history of b. If we consider all the possible sequences of the events in the static path prefix of b, the history of b contains all the states that can be reached by the sequences of the events. We use $Preset_{ideal}(pt_b, C_b)$ to denote this ideal history, where pt_b and C_b are respectively the program point and the call string [16] of b. However, it is in general too costly to compute the ideal history for an off-path-branch. We decide to compute an approximation, denoted by $Preset(pt_b, C_b)$, by only considering the current execution up to b, obtained in the DSE procedure (elaborated in Section III-C).

The other difficulty is to evaluate the future of b. In theory, computing the future behavior after arbitrary branch b can be reduced to the halting problem. Inspired by static typestate analysis [5], [17], we propose an efficient and yet precise enough method. As for the history, we define $Postset_{ideal}(pt_b, C_b)$ to be the set of the states in M_r from which a final state of M_r can be reached by the execution of the program after b under the context C_b. Let

$$h^+(pt_b, C_b) = |Preset_{ideal}(pt_b, C_b) \cap Postset_{ideal}(pt_b, C_b)|$$

Then, $h^+(pt_b, C_b) > 0$ *iff* there exists at least one path along b that can be accepted by M_p. We can prune b if $h^+(pt_b, C_b) = 0$. Therefore, $h^+(pt_b, C_b)$ reflects the possibility of the paths along b to be accepted by M_p. Note that, for two off-path-branches b_1 and b_2, $h^+(pt_{b_1}, C_{b_1}) > h^+(pt_{b_2}, C_{b_2})$ does not mean less iterations are needed to explore b_1. This is because we do not know how many iterations after b_1 or b_2 are needed to find an accepted path.

As $Postset_{ideal}(pt_b, C_b)$ is in general not computable, we compute an approximation, denoted by $Postset(pt_b, C_b)$. For this, we use a call strings based context sensitive and interprocedural dataflow analysis (elaborated in Section III-D). Let

$$h(pt_b, C_b) = |Preset(pt_b, C_b) \cap Postset(pt_b, C_b)| \quad (1)$$

Static analysis does not consider the feasibility of a path. Thus, $Postset(pt_b, C_b)$ may contain the states that do not occur in $Postset_{ideal}(pt_b, C_b)$, and $h(pt_b, C_b) > 0$ does not guarantee the existence of an accepted path along b. We tend to compute an over-approximation for $Postset$, *i.e.*, $Postset_{ideal}(pt_b, C_b) \subseteq Postset(pt_b, C_b)$. However, due to the limitations of static analysis used in our approach, we cannot guarantee the result is always an over-approximation. To ensure the soundness, we assign b a very low priority if $h(pt_b, C_b) = 0$ instead of pruning it for possible exploration.

Another factor affecting the priority of an off-path-branch b is the distance between b and the target point where a final state of M_p can be reached. For b_2 and b_3 in Fig. 3, we have $h(pt_{b_2}, C_{b_3}) = h(pt_{b_3}, C_{b_3})$. But the distance between b_2 and the statement whose execution can drive M_r to the final state (line 6 in Fig. 1) is larger. Thus, it would be better to select b_3 over b_2, as the search procedure is likely to need fewer iterations to reach the target. If $h(pt_b, C_b) = h(pt_{b'}, C_{b'})$, the deeper branch is assigned a higher priority. This strategy is based on the fact that in many cases a deeper branch is a descendant of the branch with a shorter depth.

Based on the above analysis, we define the evaluation function:

$$H(b, C_b) = h(pt_b, C_b) + depth(b)/Max \quad (2)$$

where $depth(b)$ represents the depth of b, and Max is a large constant that is greater than the depth of any branch. Note that $depth(b)/Max$ is less than 1, thus the depths of branches decide the priorities of b and b' when $h(pt_b, C_b) = h(pt_{b'}, C_{b'})$. Algorithm 2 decides the order of $worklist$. The priorities of elements in the worklist are based on $H(b, C_b)$ and the strategy discussed earlier.

Algorithm 2: Compare the Evaluation Value

```
Data: off-path-branches b1, b2 with program points pt_{b1}, pt_{b2}
      and call strings C_{b1}, C_{b2}
1  begin
2      H1 = |Preset(pt_{b1}, C_{b1}) ∩ Postset(pt_{b1}, C_{b1})|;
3      H2 = |Preset(pt_{b2}, C_{b2}) ∩ Postset(pt_{b2}, C_{b2})|;
4      if H1 > H2 then
5          return b1;
6      if H1 < H2 then
7          return b2;
8      else
9          return (depth(b1) > depth(b2)?b1 : b2);
```

C. Computing Preset

The $Preset$ is computed on-the-fly during DSE. We use the FSM of the regular property in the same way as a *monitor* in runtime verification [18]. The monitor inspects the *sensitive objects*, that are the runtime objects of the class or interface specified in the FSM. When a *sensitive object* is created, its corresponding monitor will also be initiated. When a *method* of the object is invoked, an event is generated if the invocation corresponds to an event in the FSM, and the event is sent to the monitor for carrying out a state transition.

Without loss of generality, we suppose that there is only one regular property that is analyzed at a time. We maintain a map $\mathcal{D} : S_D \rightarrow S_{ID}$ from the identities of sensitive objects to the identities of the monitor instances. An event from a sensitive object o is only monitored by the monitor instance identified by $\mathcal{D}(o)$. For example, when monitoring the property in Fig. 2, the events of different objects of *InputStreamReader* are monitored separately by their corresponding instances of M_r.

In correspondence, $Preset(pt_b, C_b)$ is represented as a set of pairs $\{s \mid s = (ID, q)\}$, where *ID* is the identify of the sensitive object created when the execution reaches pt_b in the context of C_b, and q is the current state of the monitor identified by $\mathcal{D}(ID)$. For example, the $Preset$ of b_3 in Figure 3 is $\{(ID_w, q_1)\}$, where ID_w is the identity of the object created at line 3.

D. Computing Postset

We now describe how to compute the $Postset$. We use a backward dataflow analysis to check a regular property on the control flow graph (CFG) of the program. Inspired by the backward analysis in [17], we use the reversed FSM as the monitor during the backward dataflow analysis.

Let M be an FSM, and L be the language accepted by M. The reverse of M, denoted by $\overleftarrow{M}$, accepting the mirror language [19] of L is obtained as follows: first, swap initial and final states, and reverse all the edges; and then, determinize the FSM [19]. Note that one state of $\overleftarrow{M}$ may correspond to a set of states in M.

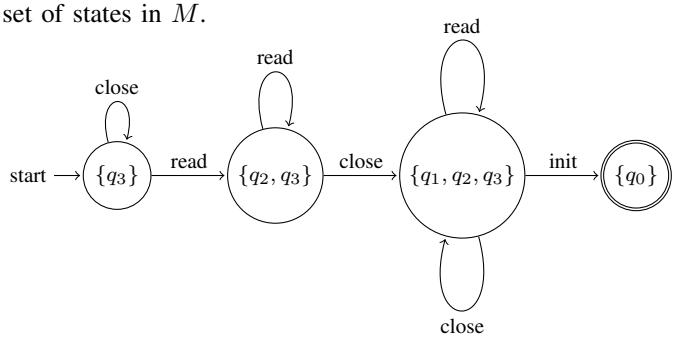

Fig. 4. The reversed FSM of M_r in Fig. 2

For example, Fig. 4 shows $\overleftarrow{M_r}$ of M_r in Fig. 2. The second state in $\overleftarrow{M_r}$ is $\{q_2, q_3\}$, which means executing event `read` from q_2 or q_3 can reach a final state of M_r.

In general, when we carry out the backward dataflow analysis for a program against an FSM M, $\overleftarrow{M}$ is used to monitor the state of a *static object* [16] (calculated by pointer analysis) until a fixed-point of $Postset$ is reached. The $Postset(pt_b, C_b)$ is represented by a set of pairs (ID, q), where *ID* is the identity of a static object, and q is the state in M from which the execution of the program after pt_b under the call string C_b can reach a final state of M.

For example, the program in Fig. 1 has only one static object of `InputStreamReader` (line 3). We denote its identity by o_r, and use $\overleftarrow{M}$ as the corresponding instance monitor. In the initial configuration of the backward dataflow analysis, the monitor $\overleftarrow{M}$ is in state $\{q_3\}$. This is because the execution of line 23 does not involve any event of $\overleftarrow{M}$. When the first iteration of dataflow analysis procedure progresses to the statement at line 19, the state of the monitor changes to $\{q_2, q_3\}$, and the state information at the location between line 18 and line 19 becomes stable with $Postset = \{(o_r, q_2), (o_r, q_3)\}$. When the dataflow analysis reaches the branch statement at line 12, the two branches are merged (by a join operation [16]) to change the $Postset$ of the program point above line 12 to $\{(o_r, q_1), (o_r, q_2), (o_r, q_3)\}$.

Note that the program in Fig. 1 is not interprocedural. For an interprocedural program, we use the IFDS framework [20] to encode our problem to enable an interprocedural backward dataflow analysis.

E. IFDS Encoding

For the dataflow analysis of a program with method invocations, IFDS works on the interprocedural control flow graph (ICFG) of the program that connects the CFGs of the individual procedures. In IFDS, the semantics of each node in ICFG is interpreted as a flow function, whose input is the facts that hold before the node, and output is the facts that hold after executing the statement of the node. In principle, IFDS converts an interprocedure dataflow analysis problem to a graph reachability problem with a *polynomial* time solution. Encoding an IFDS problem contains the definitions of four kinds of flow functions: *call-to-start*, *exit-to-return*, *call-to-return* and *normal* functions. For more details about the IFDS framework, readers can refer to [20].

In the calculation of $Postset$ for a program and an FSM M, a dataflow fact is an element in $D = O \times S$, where O is the set of the identities of the static objects in the program, and S is the state set of $\overleftarrow{M}$. Note that D is finite. The dataflow facts are mainly manipulated by the call-to-return flow functions.

Let f_{cr} be the call-to-return function of a method invocation statement `objectRef.methodName(...)` (denoted by m). If the execution of m does not correspond to any event of $\overleftarrow{M}$, f_{cr} is the identity function. Otherwise, let e_m be the event executing m, O_m the set of the identities of the static objects of the reference `objectRef` (calculated by pointer analysis), $init$ the initial state of $\overleftarrow{M}$, and $succ(q, e)$ the successor of state q in $\overleftarrow{M}$ by the transition made by the event e. Then $f_{cr} : D \to D$ is the smallest function that satisfies the following three conditions:

- For each $(o, q) \in domain(f_{cr})$, if $o \notin O_m$, $(o, q) \in range(f_{cr})$; otherwise, $(o, succ(q, e_s)) \in range(f_{cr})$. This means the events of a static object change the state of the monitor.
- If $succ(init, e_m)$ exits, we have $(o, succ(init, e_m)) \in range(f_{cr})$ for each o in O_m. This implies a new monitor instance is created for a static object o.
- If `objectRef` points to multiple static objects, *i.e.*, $|O_m| > 1$, $domain(f_{cr}) \subseteq range(f_{cr})$. Alias is handled conservatively by keeping all the possible facts.

All normal functions and exit-to-return functions are the identity function because their corresponding statements do not affect dataflow facts. If the execution of an invocation statement s_m is an event, the call-to-start function f_{cs} of s_m is $killall$, which is a special flow function that kills all the facts [20]. The reason is that an event is atomic and there is no need to analyze the internal statements. If the execution of s_m is not an event, f_{cs} is the identity function.

The flow functions defined above are distributive over the union operator on sets of facts, which is an essential requirement of IFDS framework [20]. After getting the dataflow facts, we can directly transform the facts to the corresponding $Postset$ by using the state mapping between $\overleftarrow{M}$ and M. For example, the set of dataflow facts between line 11 and line 12 in Fig. 1 is $\{(o_s, \{q_2, q_3\}), (o_s, \{q_1, q_2, q_3\})\}$, and the corresponding $Postset$ is $\{(o_s, q_1), (o_s, q_2), (o_s, q_3)\}$.

In theory, for a call strings based context sensitive dataflow analysis, a larger bound of call string improves the precision, but also increases the overhead of the analysis. A practical analysis often seeks an appropriate bound to get a balance between precision and efficiency (*c.f.* Section IV-C).

Furthermore, there is no accepted path of the program when an over-approximation is obtained by the analysis and $Postset(pt_e, \varepsilon)$ does not contain the initial state of M, where

pt_e is the start point of the program and ε is the empty call string. In this case, there is no need to perform DSE anymore. However, due to the limitations of static analysis, it is sometimes hard to get a non-trivial over-approximation.

F. Discussion

Our guided DSE combines the complementary advantages for precision and scalability. The static analysis phase mainly contains the two procedures that construct ICFG and compute $Postset$, respectively. The former is commonly used, and the latter is essentially the static typestate checking by backward dataflow analysis [17]. The worst complexity of the algorithm solving IFDS problems is $O(|E| \times |D|^3)$ [20], where $|E|$ is the number of the edges in ICFG, and $|D|$ is the size of the fact domain. The dynamic analysis uses the object-sensitive runtime checking [18] in runtime verification.

Our approach has the following advantages:

- *Scalability*: The main procedure of our approach is DSE, which is more scalable than traditional symbolic execution [13] because of utilizing concrete execution. The dynamic and static analyses for calculating guiding information are also scalable.
- *Absence of false positive*: Compared to static analysis techniques, our approach does not produce any false alarm, since DSE executes programs concretely.
- *Replay*: Our approach can generate the paths satisfying the regular property, while static analysis does not.

There are sources of imprecision in the guided DSE: 1) the imprecision of $Preset$ and $Postset$ (in comparison with $Preset_{ideal}$ and $Postset_{ideal}$), and 2) the imprecision of the interoperation of dynamic and static analyses. The imprecision will be empirically evaluated in Section IV-C.

When evaluate the history of an off-path-branch b, we actually use the prefix of the current path up to b, rather than that of an unexplored path along b. This may cause imprecision when the event sequence in the prefix varies with the input. The source of the imprecision of $Postset$ includes the following three aspects: 1) the current state-of-art of static analysis, *e.g.*, complex mechanisms of programming languages bring obstacles to static analysis; 2) the intrinsic imprecision of dataflow analysis, where the feasibility of paths is not considered; 3) the static analysis for $Postset$ does not consider the return values of invocations for efficiency. Consider the program in Fig. 5. The `next` method may be inappropriately invoked when the `hasNext` invocation returns `false`. However, our static analysis considers that the program obeys the contract of `Iterator`, *i.e.*, a `next` invocation should be preceded by an invocation of `hasNext` with `true` return value.

```
1 boolean result = iterator.hasNext();
2 result = result && ...;
3 if(!result)
4       iterator.next();
```

Fig. 5. Example of the unsoundness of static analysis

The interoperation of dynamic and static analyses is also imprecise. $Preset$ is calculated for runtime objects, while $Postset$ for static objects. In Equation (1), a runtime object and a static object are considered as equivalent if they originate from the same statement. However, an object creation statement inside a loop may create multiple runtime objects, all of which are related to one static object in static analysis.

IV. Evaluation

We evaluate the guided DSE on the following two questions:

- **Effectiveness and efficiency**. Can the guided DSE effectively find the paths satisfying a regular property? How efficient is it compared with the pure DSE?
- **Overhead**. Is it costly to compute the information for guiding DSE? The overhead should be acceptable compared with the resources needed for DSE.

A. Implementation

We have implemented the guided DSE for Java programs based on the DSE engine JPF-JDart [14], JDart for short, and the static analysis platform WALA [15]. Our implementation has improved many modules of JDart, including the core control module, the core data structures, *etc*. These improvements substantially enhance the efficiency, scalability, feasibility and robustness of JDart. For example, we use a tree to store the explored part of the path space, and Glazed Lists library [21] to implement the worklist. To enhance the feasibility of JDart further, we have integrated JPF-nhandler [22] into JDart, which helps SPF/JPF [23], [24] to handle the environment problem automatically. In the implementation of the static analysis, the values of $Postset$ are stored in a cache so that they can be obtained instantaneously during DSE. The dynamic analysis is implemented as a listener in JPF.

B. Experimental Setup

Table I lists the programs in our experiments. The thirteen programs are all real-world open source Java programs. The number of lines of code (LOC) is counted by Metrics [25].

TABLE I
The programs used in the experiments

Program	LOC	Brief Description
rhino-a	19799	Javascript interpreter
schroeder	11092	Sampled audio editor
soot-c	32358	Static analysis tool
toba-s	5720	Java bytecode to C compiler
jlex	4400	Lexical analyzer
bloat	45375	Java bytecode optimization
bmpdecoder	531	BMP file decoder
ftpclient	2436	FTP client in Java
htmlparser	21830	HTML parser in Java
fastjson	20223	JSON library from alibaba
udl	26896	UDL [26] language library
jep	28892	Mathematics library
sixpath	5927	XPath library
Total	**225479**	**13 open source programs**

The first four programs `rhino-a`, `schroeder`, `soot-c` and `toba-s` are from the Ashes suite[3] benchmark. `Jlex` is a lexical analyzer for Java. `Bloat` is from the DaCapo

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

benchmark [27]. `BMPDecoder` is a widely used Java library for decoding BMP files, in which many bit operations are used. `Ftpclient` is an FTP client program with often used parameters of FTP operations. The rest of the programs are library programs, such as `htmlparser` for parsing HTML pages and `fastjson` for JSON strings.

TABLE II
BUG PROPERTIES USED IN THE EXPERIMENTS

Property	Meaning
Enumeration	Call hasMoreElements before nextElement
Iterator	Call hasNext before next
Reader	Do not read a closed stream
Writer	Do not write a closed stream
Socket	Do not use a closed socket

The regular properties in our experiments can be classified into two categories. The properties in Table II are for "bug finding" (noted as "bug regular properties" in the rest of this paper). We use these properties to check the first eight programs listed in Table I. These properties are also widely used in the literature on typestate analysis. The second category of properties are application specific and defined manually. For example, for `htmlparser`, we specify a user-defined regular property that requires the input to be a JSP string (*i.e.,* formatted as "`<%...%>`").

We create a test driver for each library program to provide the entry point for analysis. The initial inputs for DSE are constructed randomly. For `soot-c`, `bloat` and `ftpclient`, we set the input arguments to be symbolic variables. This is why the path spaces of these three programs are small. For the other programs, we set each input byte to be symbolic. For each program/property combination, we carry out both the pure DSE and the guided DSE for the purpose of comparison. Since exploring all the paths of each program is usually infeasible, we limit the time for each analysis to *24 hours*. All the experiments are carried out on a server with 256GB memory and four 2.13GHz XEON CPUs using JDK 1.7. We run each analysis with 10GB heap memory of JVM.

C. Effectiveness and Efficiency

Table III shows parts of the experimental results of both the pure DSE (denoted as *pure* in the table) and the guided DSE (denoted as *guided* in the table). The "**#iters**" column shows the number of the iterations for finding the first accepted path; the "**Time(s)**" column shows the time needed for finding the first accepted path, where the numbers in the brackets are the time for static analysis. The four columns in the "**Paths**" column show the numbers of the *accepted* paths, the *relevant* paths (with ratio), the *irrelevant* paths and the *total* paths explored during each analysis, respectively.

We have automatically found two known typestate bugs in `rhino-a` and `bloat`, which are reported originally in [5] and [17] as potential bugs but confirmed manually. For the other six programs checked against bug properties, we find no bug within 24 hours. Since the violations of the properties listed in Table II are often serious bugs. We believe that most of these bugs are fixed during development, where our tool can be used appropriately as a bug finder.

To evaluate our approach further, we randomly inject bugs into the target programs as follows. We collect all the branch statements that make the execution tree of a program during the pure DSE; then we randomly select *three* branches to inject an event statement with respect to the property, *e.g.*, an invocation of method `close` for Socket property. In Table III, programs with injections are tagged with "-bugX". In fact, an injection may not cause a bug. This is also validated by the experimental results (*e.g.*, `schroeder-bug3` and `toba-s-bug2`). The programs without bugs are not displayed in the table.

Table III shows, for 20 out of 21 combinations (95.2%), the guided DSE successfully finds at least one path satisfying the property, while the pure DSE fails for 4 combinations in 24 hours. For `jlex-bug3`, both methods fail to find a bug. This result shows the effectiveness of our guided DSE in finding the paths satisfying a regular property.

Table III also shows that 19 out of the 20 program/property combinations (95%) need the same or fewer iterations for the guided DSE to find the first accepted path. For the combinations on which an accepted path is found, the average speedup of the guided DSE on the number of the iterations for finding the first accepted path is more than 1880X. The time used for static analysis ranges from 4.24s (`bmpdecoder`) to 99.5s (`soot-c`). The speedup on the time for finding the first accepted path of the programs whose path space contains more than 100 paths is more than 258X in average. For those programs with small path spaces (such as `bloat`, `soot-c` and `ftpclient`), the guided DSE does not outperform the pure DSE. This is because the path spaces is so small that the static analysis dominates the time of the whole analysis. Overall, compared with the pure DSE in finding the first accepted path, the guided DSE 1) needs much less iterations, and 2) is much more efficient on the analysis time for programs with large path spaces.

We further inspect the guiding ability of the guided DSE from the following three perspectives:

- The percentage and the distribution of the relevant paths during analyses. We not only want to explore more relevant paths, but also earlier.
- The number of state transitions along a path. If there are no state transitions along a path, the path will not be accepted, even if there are sensitive objects created along the path (except for the trivial case in which the initial state is also a final state).
- The shortest distance to the final state at the end of a path. To some extent, this perspective reflects how close a path is to be accepted. The distance is 0 if a path is already accepted, ∞ if there is no sensitive object generated during the path and otherwise the shortest distance from the current state to the final state of the monitor when the path is finished.

Table III lists the numbers of different kinds of paths explored during analysis and the percentage of the relevant paths in each analysis. For 19 out of the 21 programs, the guided DSE has a higher percentage of relevant paths than

TABLE III
EXPERIMENTAL RESULTS (**S.A.**: STATIC ANALYSIS)

Program (Property)	Mode	First Path		Paths			
		#iters	Time(s)(S.A.)	Accepted	Relevant(ratio)	Irrelevant	Total
rhino-a (Enumeration)	pure	>5922	>24hours	0	1 (0.02%)	5921	5922
	guided	11126	325.24 (9.69)	6	198191 (59.98%)	132251	330442
bloat (Iterator)	pure	2	5.04	20	42 (100%)	0	42
	guided	2	18.18 (13.39)	20	42 (100%)	0	42
schroeder-bug1 (Reader)	pure	1	2.38	1	35 (100%)	0	35
	guided	1	7.4 (5.11)	1	35 (100%)	0	35
schroeder-bug2 (Reader)	pure	656	5839.26	2	659 (100%)	0	659
	guided	5	12.57 (5.72)	3	659 (100%)	0	659
soot-c-bug1 (Writer)	pure	7	9.82	3	15 (57.69%)	11	26
	guided	2	104.81 (99.5)	3	15 (57.69%)	11	26
soot-c-bug2 (Writer)	pure	10	11.66	1	13 (54.17%)	11	24
	guided	2	89.93 (84.85)	1	13 (54.17%)	11	24
soot-c-bug3 (Writer)	pure	4	7.81	3	15 (57.69%)	11	26
	guided	2	103.03 (97.94)	3	15 (57.69%)	11	26
toba-s-bug1 (Reader)	pure	1	1.99	1	26 (100%)	0	26
	guided	1	7.67 (5.86)	1	26 (100%)	0	26
toba-s-bug3 (Reader)	pure	15	8.38	1	1339 (100%)	0	1339
	guided	2	9.58 (6.11)	1	1339 (100%)	0	1339
jlex-bug1 (Reader)	pure	1346	27.28	1	526993 (100%)	0	526993
	guided	1346	37.78 (8.43)	115	3376633 (100%)	0	3376633
jlex-bug2 (Reader)	pure	73620	2317.92	8	549792 (100%)	0	549792
	guided	2	10.44 (7.73)	14565	1131821 (100%)	0	1131821
jlex-bug3 (Reader)	pure	>546866	>24hours	0	546866 (100%)	0	546866
	guided	>4618215	>24hours (7.98)	0	4618215 (100%)	0	4618215
bmpdecoder-bug2 (Reader)	pure	28	88.55	4	104 (100%)	0	104
	guided	2	6.88 (4.69)	4	107 (100%)	0	107
bmpdecoder-bug3 (Reader)	pure	4	3.41	10	132 (100%)	0	132
	guided	6	12.85 (4.24)	10	133 (100%)	0	133
ftpclient-bug1 (Socket)	pure	3	4.46	8	40 (100%)	0	40
	guided	3	23.39 (19.23)	8	40 (100%)	0	40
ftpclient-bug2 (Socket)	pure	17	8.44	2	34 (100%)	0	34
	guided	3	23.3 (19.35)	2	34 (100%)	0	34
htmlparser (UserDefined)	pure	214	18.04	21726	196495 (75.16%)	64926	261421
	guided	9	14.22 (9.79)	151	81888 (72.09%)	31709	113597
fastjason (UserDefined)	pure	>462323	>24hours	0	1160 (0.25%)	461163	462323
	guided	929	43.68 (13.62)	2753	2573596 (100%)	1	2573597
udl (UserDefined)	pure	1204	266.16	1	1 (0.03%)	3430	3431
	guided	142	31.06 (16.85)	1	1 (0.03%)	3430	3431
jep (UserDefined)	pure	26327	1250.07	106	9202 (26.21%)	25903	35105
	guided	288	21.96 (8.97)	106	9186 (26.18%)	25903	35089
sixpath (UserDefined)	pure	>31786	>24hours	0	19 (0.06%)	31767	31786
	guided	3790	945.4 (8.94)	27	29374 (87.85%)	4063	33437

the pure DSE. The only two exceptions are `htmlparser` and `jep`, for which the percentage of relevant paths of our method is slightly lower. In addition to the percentage, we also want to explore relevant paths earlier so that the target paths can be found earlier. For this, we select the program/property combinations for which a plenty of relevant and irrelevant paths are explored, and observe the distribution of the relevant paths in the first hour of analyses.

The guided DSE explore relevant paths earlier than irrelevant paths. Fig. 6 shows the distribution of the relevant paths during the first one hour. The X-axis represents the time in seconds during analysis. The Y-axis represents the number of the relevant paths that are explored at each second. We can see that the guided DSE explores more relevant paths at the beginning, and the number starts to decrease from about 250 seconds. In contrast, the number of the relevant paths explored by the pure DSE increases after 1300 seconds. Besides, Fig. 6 also shows the guided DSE tends to explore more relevant paths than the pure DSE.

Fig. 7 shows the difference between the numbers of state transitions of the guided DSE and the pure DSE. The X-axis represents the order of the paths in which they are explored, and Y-axis is the difference between the numbers of state transitions of the path under the guided DSE and that of the path under the pure DSE. For the ith path p_i, the value on Y-axis of p_i is calculated as follows.

$$Value(i) = \sum_{c \in Combinations} ST_g^c(p_i) - \sum_{c \in Combinations} ST_p^c(p_i)$$

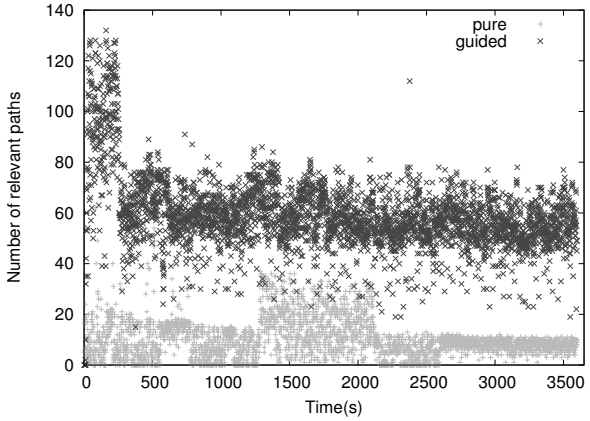

Fig. 6. Relevant path distribution

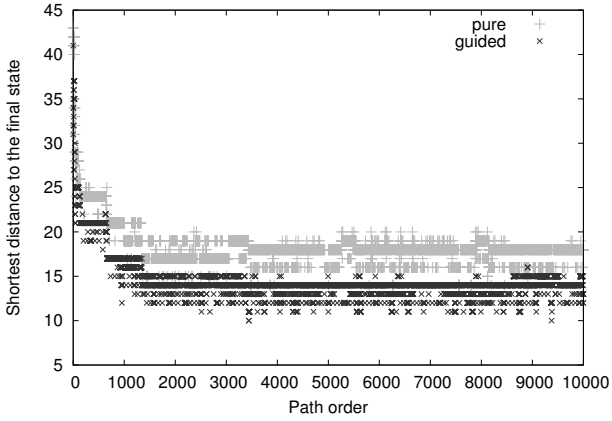

Fig. 8. Shortest distance of each path

where $ST_g^c(p_i)$ and $ST_p^c(p_i)$ are the numbers of the state transitions in p_i when analyzing combination c with the guided DSE and the pure DSE, respectively. Fig. 7 shows the results of the first 10 thousands paths. For 97.05% of the first 4000, the guided DSE causes more state transitions than the pure DSE. Although for some of the rest 6000 paths the pure DSE causes more state transitions, there are still 91.55% of the points above 0 on Fig. 7. This shows the guided DSE causes more state transitions.

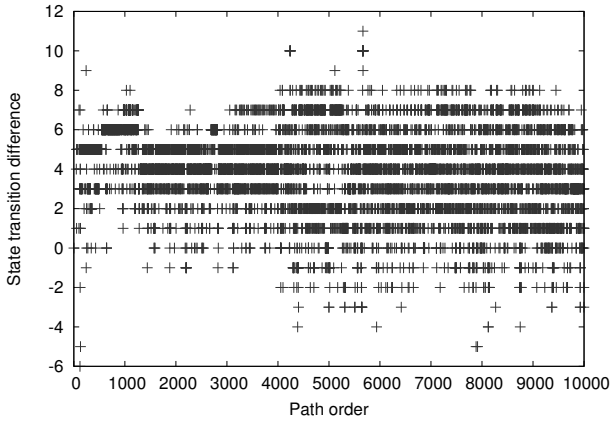

Fig. 7. State transition difference in each path

We also inspect the guided DSE from the perspective about the shortest distance in a similar way. Fig. 8 displays the sum of the distance of the ith path in each combination. We can see for 99.6% of the paths, the distance to the final state under the pure DSE is larger than that under the guided DSE. This indicates the guided DSE tends to explore the paths that drive the FSM closer to a final state.

Call string bound. In theory, the larger the bound of the call string, the more precise the analysis result will be. In our experiments, we have tried different call string bounds (1 to 4) for all the combinations. One of the findings shows that to complete the analysis in an acceptable time, the call string bounds are often confined to be less than 5 due to the scale of the programs. For example, for `soot-c` and `udlparser`, the static analysis phase does not finish in one hour if the call string bound is larger than 2. Another result of the experiment shows that the guided DSE gains no improvement in the analysis precision with increased call string bounds. The reason is that real-world programs usually have complex call graphs. Hence, in our experiments, the call string bounds are set uniformly to 1.

The precision of guiding. We evaluate the precision of the guiding information as follows. During DSE, we directly *prune* away the off-path-branches for which the intersections of $Preset$ and $Postset$ are empty, and compare the results with those in Table III. We can see the same results for 18 out of the 20 (90%) combinations. This indicates that the guiding information for DSE is pretty precise for the combinations.

D. Overhead

In general, the time/memory overhead varies with the scale and complexity of the target program. Compared with the pure DSE, the time overhead of the guided DSE is mainly caused by the static analysis. In our experiments, the time overhead is very low (less than 6%) for most programs. For programs `soot-c`, `bloat` and `ftpclient`, however, the time overhead is bigger than 30%. The reason is that static analysis dominates the whole analysis time due to the small path spaces of these programs.

The extra memory overhead consists of 1) the memory used for static analysis and 2) the memory used to store the call string and $Preset$ for each off-path-branch during DSE. For 1), we use jConsole to monitor the memory overhead during static analysis. The memory used for static analysis ranges from 461MB (`schroeder`) to 2GB (`soot-c`). It is necessary to point out this result may change under a different configuration because of the memory management mechanisms of JVM. For 2), we calculate the memory for storing call string and $Preset$ by the memory size calculation tool Jamm [28] for Java. The results show this part of memory overhead is less than 55MB in most cases. The only exception is `schroeder-bug2` which needs 580MB. The reason is that too many symbolic variables are introduced during DSE. Overall, the memory overhead is acceptable.

E. Threats to Validity

The threats to the validity of our experiments are mainly external because of the limited samples of program/property combinations. We have used 13 real-world open source Java programs, and the LOCs of these programs vary from 0.5K to 45K. Five of these 13 programs are from Java program analysis benchmarks, and these five programs are also used in the existing work of static typestate analysis [5][17]. These programs cover different types of applications and libraries, including parsing, optimization, network, *etc*. Some of these programs are used widely, such as `fastjson`. These programs are quite representative for evaluating DSE methods. In addition, although our target programs in this paper are Java programs, our approach can be used for analyzing the programs in other languages, such as C and C++. For the properties used in the experiments, the properties in Table II are representative contracts of Java libraries, and are also frequently used in typestate analysis. The application specific user-defined properties are defined by ourselves. These properties are not complex FSMs, and are easy to define. We believe that user-defined properties are an important part during the application of our guided DSE in practice.

V. Related work

The methods of guiding symbolic execution have been proposed for different goals, including improving coverage [29], [30], [31], [32], [33], reaching a statement [34], [35], [36], [37], checking a rule [38], [39] and exploring the difference between versions of a program [40], [41], *etc*.

Both approaches presented in [38] and [39] slices programs or paths against a rule to prune paths during symbolic execution. The method in [39] first slices the program after the instrumentations for checking a regular property, and then feeds the sliced program into KLEE [29] for symbolic execution to detect bugs. Hence, there is *no guiding support* during the symbolic execution for checking the regular property. The rule-directed symbolic execution [38] is most related to our work. There, path slicing is used to slice the irrelevant parts of the current path to prune irrelevant paths or the relevant paths equivalent to the current. The guided DSE and the path slicing based approach in [38] complement to each other. Our guided DSE can provides a priority order for the off-path-branches to be explored; whereas, the slicing based approach prunes redundant off-path-branches. In addition, when the current explored path is *irrelevant*, the approach in [38] prunes the off-path-branches along which there will be no event. The guided DSE not only gives a lower priority to the pruned branches, but also gives different priorities for the remaining branches. In our experimental results (*c.f.* Table III), the portions of the irrelevant paths in many programs under the guided DSE are not low.

The existing work [34], [35], [36] for guiding symbolic execution toward reaching one statement uses the information of the distance to the target statement for guiding. In [37], a backward symbolic analysis method is proposed to infer the weakest precondition for reaching a statement. The work in [42] proposes a static analysis method to generate a shortest path to cover multiple statements as more as possible. Our work can be used to achieve the same goal of reaching one or multiple statements, because FSM is expressive enough to specify reachability properties.

The methods for statically analyzing programs against regular properties differ mainly in soundness, precision and scalability. Sound methods [5], [17], [1], [43], usually used for verification, abstract the target program using different abstract domains [44] to have a tradeoff between precision and scalability. Some methods, *e.g.*, [7], [45], [46], are neither sound nor complete, but they have been validated to perform well on bug finding. Dynamic methods, such as runtime verification [8], [9], complements the static methods by improving scalability and precision, but with the sacrifice of soundness. We combine static analysis and dynamic analysis to analyze a program against regular properties.

Finally, the tools of software model checking, such as SLAM [47], use static analysis to extract a model from a program, and check the model against regular properties to find bugs or verify the program. YOGI [48], [49], [50] improves SLAM by the synergy of DSE and predicate abstraction to make the analysis faster [51]. Compared with the work in YOGI, the guiding method of our guided DSE uses a static analysis with a polynomial complexity, which leads to a better scalability.

VI. Conclusions and Future work

This paper proposes a new DSE method, called regular property guided DSE, to automatically find the program paths satisfying a regular property. With the synergy of static analysis and dynamic analysis, the guided DSE is effective and scalable in finding the paths satisfying a regular property. We have implemented the guided DSE for Java programs. The results of the extensive experiments on real-world programs show that our method is highly efficient for steering DSE against regular properties. Future work will be in several directions: (1) Until now, no regular property involving multiple objects is supported. We will investigate how the guided DSE can be extended to solve this problem. (2) We are also interested in developing a guiding method for other types of properties, such as *context-free* properties. (3) The precision and efficiency of our guiding method can be improved by using more advanced alias analysis techniques [52], [53]. (4) Furthermore, we plan to work on the applications and optimizations of our method on some specific software engineering topics, *e.g.*, typestate analysis and path-oriented test case generation.

Acknowledgments

We thank Prof. Xiaoguang Mao and Prof. Jingling Xue for the discussions and suggestions to our work. This work was supported in part by National 973 Program (2014CB340703) and National Natural Science Foundation (61120106006, 61472440, 61272140) of China.

REFERENCES

[1] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *PLDI*, pp. 57–68, ACM, 2002.

[2] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *PLDI*, pp. 480–491, ACM, 2007.

[3] G. V. Bochmann, "Finite state description of communication protocols," *Computer Networks*, vol. 2, no. 4, pp. 361–372, 1978.

[4] G. Myers, C. Sandler, and T. Badgett, *The art of software testing*. Wiley, 2011.

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

[6] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, pp. 371–382, IEEE, 2009.

[7] D. R. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI*, pp. 1–16, USENIX Association, 2000.

[8] 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*, pp. 345–364, ACM, 2005.

[9] F. Chen and G. Rosu, "MOP: an efficient and generic runtime verification framework," in *OOPSLA*, pp. 569–588, ACM, 2007.

[10] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, *et al.*, "Combining test case generation and runtime verification," *Theoretical Computer Science*, vol. 336, no. 2, pp. 209–234, 2005.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, pp. 213–223, ACM, 2005.

[12] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *FSE*, pp. 263–272, ACM, 2005.

[13] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for java," in *NFM*, pp. 121–125, Springer, 2009.

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

[16] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

[17] E. Bodden, "Efficient hybrid typestate analysis by determining continuation-equivalent states," in *ICSE*, pp. 5–14, ACM, 2010.

[18] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[19] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2003.

[20] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, pp. 49–61, ACM, 1995.

[21] Jesse Wilson, "Glazed Lists Library." http://www.glazedlists.com/, 2014.

[22] N. Shafiei and F. van Breugel, "Automatic handling of native methods in java pathfinder," in *SPIN*, pp. 97–100, Springer, 2014.

[23] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of java bytecode," in *ASE*, pp. 179–180, ACM, 2010.

[24] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *ISSTA*, pp. 15–26, ACM, 2008.

[25] Metrics. http://metrics.sourceforge.net, 2014.

[26] AOST, "Tellurium UID Description Language (UDL)." http://code.google.com/p/aost/wiki/TelluriumUIDDescriptionLanguage, 2014.

[27] S. M. Blackburn and et al., "The dacapo benchmarks: java benchmarking development and analysis," in *OOPSLA*, pp. 169–190, ACM, 2006.

[28] Jamm, "Java agent for memory measurements." https://github.com/jbellis/jamm, 2014.

[29] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, pp. 209–224, USENIX Association, 2008.

[30] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, pp. 443–446, IEEE, 2008.

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

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

[33] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *FSE*, ACM, 2014.

[34] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS*, pp. 95–111, Springer, 2011.

[35] D. Babic, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA*, pp. 12–22, 2011.

[36] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *EuroSys*, pp. 321–334, 2010.

[37] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *PLDI*, pp. 363–374, 2009.

[38] H. Cui, G. Hu, J. Wu, and J. Yang, "Verifying systems rules using rule-directed symbolic execution," in *ASPLOS*, pp. 329–342, ACM, 2013.

[39] J. Slabỳ, J. Strejček, and M. Trtík, "Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution," in *FMICS*, pp. 207–221, Springer, 2012.

[40] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *PLDI*, pp. 504–515, ACM, 2011.

[41] P. D. Marinescu and C. Cadar, "make test-zesti: A symbolic execution solution for improving regression testing," in *ICSE*, pp. 716–726, IEEE, 2012.

[42] A. Lal, J. Lim, M. Polishchuk, and B. Liblit, "Path optimization in programs and its application to debugging," in *ESOP*, pp. 246–263, 2006.

[43] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," in *CCS*, pp. 235–244, ACM, 2002.

[44] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, pp. 238–252, ACM, 1977.

[45] Y. Xie and A. Aiken, "Scalable error detection using boolean satisfiability," in *POPL*, pp. 351–363, ACM, 2005.

[46] D. Babic and A. J. Hu, "Calysto: scalable and precise extended static checking," in *ICSE*, pp. 211–220, ACM, 2008.

[47] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *PLDI*, pp. 203–213, ACM, 2001.

[48] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: a new algorithm for property checking," in *FSE*, pp. 117–127, ACM, 2006.

[49] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *ISSTA*, pp. 3–14, ACM, 2008.

[50] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *POPL*, pp. 43–56, 2010.

[51] A. V. Nori and S. K. Rajamani, "An empirical study of optimizations in YOGI," in *ICSE*, pp. 355–364, 2010.

[52] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, "Fast algorithms for Dyck-CFL-reachability with applications to alias analysis," in *PLDI*, pp. 435–446, 2013.

[53] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with CFL-reachability," in *CC*, pp. 61–81, 2013.