eXpress: Guided Path Exploration for Regression Test Generation

Kunal Taneja[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2], Wolfram Schulte[2]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC, 27695, USA
[2]Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
[1]{ktaneja, txie}@ncsu.edu, [2]{nikolait, jhalleux, schulte}@microsoft.com

ABSTRACT

Regression test generation aims at generating a test suite that can detect behavioral differences between the original and the new versions of a program. Regression test generation can be automated by using Dynamic Symbolic Execution (DSE), a state-of-the-art test generation technique, to generate a test suite achieving high structural coverage. DSE explores paths in the program to achieve high structural coverage but exploration of all these paths can often be expensive. However, if our aim is to detect behavioral differences between two versions of a program, we do not need to explore all these paths in the program since not all these paths are relevant for detecting behavioral differences. In this paper, we propose an approach on guided path exploration that avoids exploring irrelevant paths in terms of detecting behavioral differences. Hence, behavioral differences are more likely to be detected earlier in path exploration. In addition, our approach leverages the existing test suite (if available) for the original version to efficiently execute the changed regions of the program and infect program state. Experimental results on 72 versions (in total) of four programs show that our approach requires about 53% fewer runs (i.e., explored paths) on average to cause the execution of a changed region and 44% fewer to cause program-state differences after its execution than exploration without using our approach. In addition, our approach requires 71% fewer runs to cover all the changed regions by exploiting an existing test suite than exploration without using the test suite.

1. INTRODUCTION

Regression test generation aims at generating a test suite that can detect behavioral differences between the original and the new versions of a program. A behavioral difference between two versions of a program can be reflected by the difference between the observable outputs produced by the execution of the same test (referred to as a difference-exposing test) on the two versions. Developers can inspect these behavioral differences to determine whether they are intended or unintended (i.e., regression faults).

Regression test generation can be automated by using Dynamic Symbolic Execution (DSE) [11, 23, 4], a state-of-the-art test generation technique, to generate a test suite achieving high structural coverage. DSE explores paths in a program to achieve high structural coverage, and exploration of all these paths can often be expensive. However, if our aim is to detect behavioral differences between two versions of a program, we do not need to explore all these paths in the program since not all these paths are relevant for detecting behavioral differences.

To formally investigate irrelevant paths for exposing behavioral differences, we adopt the Propagation, Infection, and Execution (PIE) model [28] of error propagation. According to the PIE model, a fault can be detected by a test if a faulty statement is executed (E), the execution of the faulty statement infects the state (I), and the infected state (i.e., error) propagates to an observable output (P). A change in the new version of a program can be treated as a fault and then the PIE model is applicable for effect propagation of the change. Many paths in a program often cannot help in satisfying any of the conditions P, I, or E of the PIE model.

In this paper, we present an approach[1] `eXpress` and its implementations that uses DSE to detect behavioral differences based on the notion of the PIE model.

Our approach first determines all the branches (in the program under test) that cannot help in achieving any of the conditions E and I of the PIE model in terms of the changes in the program. To make test generation efficient, we develop a new search strategy for DSE to avoid exploring these irrelevant branches (including which can lead to an irrelevant path[2]). In particular, our approach guides DSE to avoid from flipping branching nodes[3], which on flipping execute some irrelevant branch.

In addition, our approach can exploit the existing test suite (if available) for the original version by seeding the tests in the test suite to the program exploration. Our technique of seeding the exploration with the existing test suite can be used to efficiently augment an existing test suite so that various changed parts of the program (that are not covered by the existing test suite) are covered by the augmented test suite.

This paper makes the following major contributions:

[1]An earlier version of this work [25] is described in a four-page paper that appears in the NIER track of ICSE 2009. This version significantly extends the previous work in the following major ways. First, in this paper, we develop techniques for efficiently finding irrelevant branches that cannot execute any change. Second, we develop techniques for exploiting the existing test suite for efficiently generating regression tests. Third, we automate our approach by developing a tool. Fourth, we conduct extensive experiments to evaluate our approach.

[2]An irrelevant path is a path that cannot help in achieving P, I, and E of the PIE model.

[3]A branching node in the execution tree of a program is an instance of a conditional statement in the source code. A branching node consists of two sides (or more than two sides for a `switch` statement): the true branch and the false branch. Flipping a branching node is flipping the execution of the program from the true (or false) branch of the branching node to the false (or true) branch. Flipping a branching node representing a switch statement is flipping the execution of the current branch to another unexplored branch.

Path Exploration for Regression Test Generation. We propose an approach called `eXpress` that uses DSE for efficient generation of regression unit tests. To the best of our knowledge, ours is the first approach that guides path exploration on the new version specifically for regression test generation.
Incremental Exploration. We develop a technique for exploiting an existing test suite, so that path exploration focuses on covering the changes rather than starting from scratch. To the best of our knowledge, ours is the first technique that leverages an existing test suite for automated regression test generation.
Implementation. We have implemented our `eXpress` approach in a tool as an extension for Pex [26], an automated structural testing tool for .NET developed at Microsoft Research. Pex has been previously used internally at Microsoft to test core components of the .NET architecture and has found serious bugs [26]. The current Pex has been downloaded for thousands of times in industry.
Evaluation. We have conducted experiments on 72 versions (in total) of four programs. Experimental results show that our approach requires about 53% fewer runs (i.e., explored paths) on average to cause the execution of a changed region and 44% fewer to cause program-state differences after its execution than exploration without guidance. In addition, our approach requires 71% fewer runs to cover all the changed regions (i.e, infection) by exploiting an existing test suite than exploration without using the test suite.

2. DYNAMIC SYMBOLIC EXECUTION

In our approach, we use Pex [26] as an engine for Dynamic Symbolic Execution (DSE). Pex starts program exploration with some default inputs. Pex then collects constraints on program inputs from the predicates at the branching statements executed in the program. We refer to these constraints at branching statements as branch conditions. The conjunction of all branch conditions in the path followed during execution of an input is referred to as a path constraint. Pex keeps track of the previous executions to build a dynamic execution tree. Pex, in the next run[4], chooses one of the unexplored branch in the execution tree (dynamically built thus far). Pex flips the chosen branching node in the dynamic execution tree to generate a new input that follows a new execution path. Pex uses various heuristics [30] for choosing a branching node (to flip next) in the execution tree using various search strategies with an objective of achieving high code coverage fast. We next present definitions of some of the terms that we use in the rest of this paper.
Discovered node. We refer to all the branching nodes that are executed in the current DSE run but were not executed in previous runs as discovered branching nodes (in short as discovered nodes).
Instance of a branching node. A branching node c_i in a Control Flow Graph (CFG) of a program can be present multiple times in the dynamic execution tree (of the program) due to loops in the program. We refer to these multiple branching nodes in the tree as instances of c_i.

3. EXAMPLE

In this section, we illustrate different components of the `eXpress` approach with an example. `eXpress` takes as input two versions of a program and produces as output a regression test suite, with the objective of detecting behavioral differences (if any exist) between the two versions of program under test. Although `eXpress` analyzes assembly code of C# programs, in this section, we illustrate the `eXpress` approach using program source code.

Consider the example in Figure 1. Suppose that the statements at Lines 12 and 13 of `TestMe` are added in the new version. We next describe our approach using the example.

[4]A run is an exploration iteration.

```
     static public int TestMe(string[] c){
1      int state = 0;
2        for(int i=0; i< c.Length; i++){
3          if(c[i] == "[")
4            state =1;
5          else if(state == 1 && c[i] == "{")
6            state =2;
7          else if(state == 2 && c[i] == "(")
8            state =3;
9          else if(state == 3 &&  c[i] == "<")
10           state =4;
11         else if(state == 4 && c[i] == "*"){ state =5;
12           if(c.Length==15) //Added in new version
13             break; //Added in new version
           }
14         if(c[i]==' ')
15           break;
16         if(state==-1)
17           return state;
         }
       }
18     return state;
     }
```

Figure 1: An example program

Difference Finder. The Difference Finder component of compares the original and the new versions of the program under test to find differences between each corresponding method of the two program versions. For the program in Figure 1, Difference Finder detects that the statements at Lines 12 and 13 are added in the new version.
Graph Builder. The Graph Builder component then builds a Control-Flow Graph (CFG) of the new version of the program under test and marks the changed vertices in the graph. The left side of Figure 2 shows the CFG of the program in Figure 1. The labels of vertices in the CFG denote the corresponding line numbers in Figure 1. The red (dark) vertices denote the newly added statements at Lines 12 and 13. The gray vertices denote the branching nodes (i.e., these vertices denote the branching statements in the program), while the white vertices denote the other statements in the program.
Graph Traverser. The Graph Traverser component traverses the CFG to find each branch[5] b in a program such that if b is taken, the program execution cannot reach any red (dark) vertex at Line 12 or 13. On traversing the CFG in Figure 2, the Graph Traverser detects that taking the branches $< 2, 18 >$[6], $< 14, 15 >$, $< 16, 17 >$, and $< 12, 14 >$ (dotted red/dark red edges in Figure 2.), the program execution cannot reach either of the vertices at Lines 12 or 13. Since, after taking these branches, the execution cannot reach the changed statements at Lines 12 and 13, the execution of these branches cannot help in executing the changed statements. Since behavioral differences between two versions of the program cannot be detected without executing the changed statements, Dynamic Test Generator (described later in this section) component Hence, does not explore these branches while generating regression tests for the program under test.
Instrumenter. The Instrumenter component transforms the two versions of the program code such that the transformed program code is amenable to regression testing. In particular, the Instrumenter component instruments both versions of the program under test so that program states can be compared (to determine state infection) after the execution of the added statements at Lines 12 and 13.
Dynamic Test Generator. To cover the changed statement at Line 13, DSE needs at least 6 DSE runs (starting from an empty input

[5]A branch is an outgoing edge of a branching node.
[6]$< x, y >$ represents a branch from node x to node y.

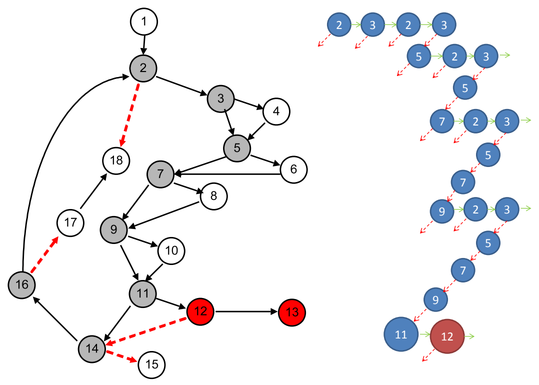

Figure 2: **The left side shows the Control-Flow Graph for the program in Figure 1, while the right side shows part of execution tree of the program for c = {"[", "{", "(", "<", "*" }.**

array c). However, the number of runs depends on the choice of branches that DSE flips in each run. In each run, DSE has the choice of flipping 7 branches in the program. For the program in Figure 1, Pex takes 441 DSE runs to cover the added statement at Line 13 (to infect the program state). The number of runs can be much more if the number of branching statements in the program increases. To reduce the search space of DSE, the Dynamic Test Generator component adopts the PIE model [28] of error propagation described in Section 1. In particular, the Dynamic Test Generator prunes all the following branches from the search space of DSE so that DSE has fewer branches to flip in each run.

- **Branches not satisfying E**. The branches $< 2, 18 >$, $< 14, 15 >$, $< 16, 17 >$, and $< 12, 14 >$ found to be irrelevant by the Graph Traverser component cannot help in executing the changed statement at Line 12 or 13. Hence, these branches are not explored by the Dynamic Test Generator component.

- **Branches not satisfying I**. If the statement 12 is executed but program state is not infected (i.e., execution does not take the $< 12, 13 >$ branch), the branching nodes (in the execution path) that are after the last instance[7] of the changed node are not explored. For the example in Figure 1, the branches $< 14, 16 >$, and $< 16, 12 >$ are pruned from exploration.

Incremental Exploration. Our approach can reuse an existing test suite for the original version so that changed parts of the program can be explored efficiently due to which test generation is likely to find behavior differences earlier in path exploration. Assume that there is an existing test suite covering all the statements in the old version of the program in Figure 1. Suppose that the test suite has a test input I = {"[", "{", "(", "<", "*" }. The input covers all the statements in the new version of `TestMe` except the newly added statement at Line 13. If we start the program exploration from a scratch (i.e., with default inputs), Pex takes 441 runs to cover the statement at Line 13. However, we can reuse the existing test suite for exploration to cover the new statement efficiently. Our approach executes the test suite to build an execution tree for the tests in the test suite. Our approach then starts program exploration using the dynamic execution tree built by executing the existing test suite instead of starting from an empty tree. Some of the branches in the tree may take many runs for Pex to discover if starting from an empty tree. The right side of Figure 2 shows part of the execution tree for the input I. A dotted red/dark edge in the tree indicates the false side of a branching node while a solid green (horizontal) edge indicates the true side. To generate an input for the next DSE runs, Pex flips a branching node b in the tree whose the other side has not yet been explored and generates an input so that program execution takes the unexplored branch of b. Pex chooses such branch for flipping using various heuristics for covering changed regions of the program. It is likely that Pex chooses the branching node 12 (colored dark), which on execution covers the statement at Line 13. When starting the program exploration from scratch, Pex takes 420 runs before it reaches(discovers) the red/dark branching node in the right side of Figure 2. Using our approach of seeding the tests from the existing test suite, Pex takes 39 runs to flip the branching node and cover the statement at Line 13.

[7]Multiple instances of a node in the CFG (of a program) can be present in the execution tree due to loops in the program.

4. APPROACH

`eXpress` takes as input the assembly code of two versions $v1$ (original) and $v2$ (new) of the program under test. In addition, `eXpress` takes as input, the name and signature of a parameterized unit test (PUT)[8] [27]; such a PUT serves as a test driver for path exploration. `eXpress` consists of five main components.

The Difference Finder component of finds the set of differences between each of the corresponding method pairs in the two versions of the program. The Graph Builder component builds an inter-procedural graph G with the input PUT as the starting method. The Graph Traverser component traverses the graph to find all the branches B that need not be explored for executing the changed regions[9] (found by the Graph Traverser). The Instrumenter component instruments both versions of the program under test so that program states can be compared (to determine state infection). The Dynamic Test Generator then generates tests for the input PUT, pruning branches B from exploration. When an existing test suite is available for the original version, `eXpress` conducts incremental exploration that exploits the test suite for generating tests for the new version. We next discuss in detail the main components in `eXpress` and its incremental exploration.

4.1 Difference Finder

The Difference Finder component takes the two versions $v1$ and $v2$ as input and analyzes the two versions to find pairs $< M_{i1}, M_{i2} >$ of corresponding methods in $v1$ and $v2$, where M_{i1} is a method in $v1$ and M_{i2} is a method in $v2$. A method M is defined as a triple $< FQN, Sig, I >$, where FQN is the fully qualified name[10] of the method, Sig is the signature[11] of the method, and I is the list of assembly instructions in the method body. Two methods $< M_{i1}, M_{i2} >$ form a corresponding pair if the two methods M_{i1} and M_{i2} have the same FQN and Sig. Currently our approach considers as different methods the methods that have undergone Rename Method or Change Method Signature refactoring. A refactoring detection tool [6] can be used to find such corresponding methods. For each pair $< M_{i1}, M_{i2} >$ of corresponding methods, the Difference Finder finds a set of differences Δ_i between the list of instructions $I_{M_{i1}}$ and $I_{M_{i2}}$ in the body of Methods M_{i1} and M_{i2}, respectively. Δ_i is a set of instructions such that each in-

[8]A PUT is a test method with parameters. A test generation tool (such as Pex) can generate values for the parameters to explore different feasible paths in the program under test.

[9]A changed region is a minimal set of statements that contains all the changed statements in a method.

[10]The fully qualified name of a method m is a combination of the method's name, the name of the class c declaring m, and the name of the namespace containing c.

[11]Signature of a method m is the combination of parameter types of m and the return type of m.

struction ι in Δ_i is an instruction in $I_{M_{i2}}$ (or in $I_{M_{i1}}$ for a deleted instruction), and ι is added, modified, or deleted from list $I_{M_{i1}}$ to form $I_{M_{i2}}$.

In its other components, `eXpress` analyzes Version $v2$ of the program while using the differences obtained from the Difference Finder component to efficiently generate regression tests. Note that Version $v1$ can also be used instead of $v2$ in path exploration.

4.2 Graph Builder

The Graph Builder component makes an inter-procedural control flow graph (CFG) of the program version v_2. The Graph Builder component starts the construction of the inter-procedural CFG from the Parametrized Unit Test (PUT) τ provided as input. The inter-procedural CFG is used by the Graph Traverser component to find branches (in the graph) via which the execution cannot reach any vertex containing a changed instruction in the graph. Since a moderate-size program can contain millions of method calls (including those in its dependent libraries), often the construction of inter-procedural graph is not scalable to real-world programs. Hence, we build a minimal inter-procedural CFG for which our purpose of finding branches that cannot reach some changed region in the program can be served. The pseudo code for building the inter-procedural CFG is shown in Algorithm 1. Initially, the algorithm `InterProceduralCFG` is invoked with the argument as the PUT τ. The algorithm first constructs an intra-procedural CFG g for method τ. For each method invocation vertex[12] (invoking Method c) in g, the algorithm `InterProceduralCFG` is invoked recursively with the invoked method c as the argument (Line 22 of Algorithm 1), after adding c to the call stack (Line 21). After the control returns from the recursive call, the method c is removed from the call stack (Line 23) and added to the set of visited methods (Line 24). The inter-procedural graph `cg` (with c as an entry method) resulting from the recursive call at Line 22 is merged with the graph g (Line 25). The algorithm `InterProceduralCFG` is not invoked recursively with c as argument in the following situations:

Algorithm 1 $InterProceduralCFG(\tau)$

Input: A test method τ.
Output: The inter-procedural Control Flow Graph (CFG) of the program under test.

```
1:  $Graph\ g \leftarrow GenerateIntraProceduralCFG(\tau)$
2:  for all Vertex $v \in g.Vertices$ do
3:      if v.Instruction = MethodInvocation then
4:          $c \leftarrow getMethod(v.Instruction)$
5:          if $c \in MethodCallStack$ then
6:              $goto$ Line 2 To avoid loops
7:          end if
8:          if $c \in ReachableToChangedRegion$ then
9:              $g \leftarrow GraphUnion(ChangedMethod, g, v)$
10:             $goto$ Line 2
11:         end if
12:         if $c \in Visited$ then
13:             $goto$ Line 2
14:         end if
15:         if $c \in ChangedMethods$ then
16:             $ChangedMethod \leftarrow c$
17:             for all Method $m \in MethodCallStack$ do
18:                 $ReachableToChangedRegion.Add(m)$
19:             end for
20:         end if
21:         $MethodCallStack.Add(c)$
22:         $cg \leftarrow InterProceduralCFG(c)$
23:         $MethodCallStack.Remove(c)$
24:         $Visited.Add(c)$
25:         $g \leftarrow GraphUnion(cg, g, v)$
26:     end if
27: end for
28: return $g$
```

c is in call stack. If c is already in the call stack, `InterProceduralCFG` is not recursively invoked with c as argument (Lines 5-6). This technique ensures that our approach is not stuck in a loop in method invocations. For example, if method A invokes method B, and method B invokes A, then the construction of the inter-procedural graph stops after A is encountered the second time.

[12]A method invocation vertex is a vertex representing a call instruction.

c is already visited. If c is already visited, `InterProcedural CFG` is not recursively invoked with c as argument (Line 23). This technique ensures that we do not build the same subgraph again.

c is in `ReachableToChangedRegion`. The set `ReachableToChanged Region` is populated whenever a changed method[13] is encountered. In particular, if a changed method is encountered, the methods currently in the call stack are added to the set `ReachableToChanged Region` (Lines 17-19). If c is in `ReachableToChangedRegion`, `InterProceduralCFG` is not recursively invoked with c as argument, while merging CFG of some changed method with g (Line 8-11). Note that if a method `m` can reach a changed method `cm`, all the branches in this method `m` may not be able to reach `cm` (for example, due to $return$ statements). Branches that cannot reach any changed region (irrelevant branches) are found by the Graph Traverser component.

If a branching node b can reach a changed region in Inter-Procedural CFG $g1$ built without using the preceding optimization, the branching node b can reach a changed region (maybe a different one) in graph $g2$ built using the preceding optimizations. Since our aim of building the intra-procedural CFG is to find irrelevant branches, those in the graph via which the execution cannot reach any changed region, the preceding three optimizations help achieve the aim while reducing the cost of building the inter-procedural CFG. In addition, the size of the inter-procedural CFG is reduced resulting in reduction in the cost of finding irrelevant branches.

4.3 Graph Traverser

The Graph Traverser component takes as input the inter-procedural CFG g constructed by the Graph Builder and a set V of changed vertices in the CFG g (found by Difference Finder). The Graph Traverser traverses the graph to find a set of branches B via which the execution cannot reach any of the branches in V. A branch b in CFG g is an edge $e =< v_i, v_j >: e \in g$, where v_i is a vertex in g with an outgoing degree of more that one. The vertex v is referred to as a branching node. The pseudo code for finding the set of branches B is shown in Algorithm 2.

Algorithm 2 $FindUnreachableBranches(g, V)$

Input: A Graph g and a set V of vertices in Graph g.
Output: A set of branches B in Graph g that do not have a path to any vertex $v \in V$.

```
1:  $Reachable \leftarrow Reachable \cup V$
2:  for all Vertex $v \in g.Vertices$ such that $v.degree > 1$ do
3:      if $Reachable.Contains(v)$ then
4:          goto Line 2
5:      end if
6:      $\rho \leftarrow FindPathUsingDFS(v, V, Reachable)$
7:      if $\rho$ is empty then
8:          for all Vertex $n \in v.OutVertices$ do
9:              $B \leftarrow B\cup < v, n >$
10:         end for
11:         goto Line 2
12:     end if
13:     for all Vertex $pv \in \rho$ such that $v.degree > 1$ do
14:         $Reachable \leftarrow Reachable \cup pv$
15:     end for
16:     while $v.OutVertices \neq \phi$ do
17:         $R \leftarrow R\cup < v, v_j >: v_j \in \rho$
18:         $g \leftarrow g.RemoveEdge(v, v_j)$
19:         $\rho \leftarrow FindPathUsingDFS(v, V, Reachable)$
20:         if $\rho$ is empty then
21:             for all Vertex $o \in v.OutVertices$ do
22:                 $B \leftarrow B\cup < v, o >$
23:             end for
24:             $g \leftarrow g.AddEdges(R)$
25:             goto Line 2
26:         end if
27:     end while
28:     $g \leftarrow g.AddEdges(R)$
29: end for
30: return $B$
```

For each Vertex v in g where $degree(v) > 1$, the Graph Traverser performs a depth-first search (DFS) from Vertex v (Line 6)

[13]A changed method M_i is a method for which the set $\Delta_i \neq \phi$.

to find a path between v and some vertex $c \in V$. If no path is found, all branches $b_i =< v, v_i >$ are added to the set B (Lines 7-10) since none of these branches have a path to any of the vertices in V. If a path ρ is found (Lines 16-27) by the DFS, there may still be a branch $b_i =< v, v_i >$ that cannot reach any vertex $c \in V$. To find such a branch, we remove each edge $e_j =< v, v_j >$: $e_j \in \rho$ one at a time from the graph and perform DFS again starting from v. We repeat the preceding steps until we either find no path ρ or all the edges from v are removed from the graph. If no path is found, the remaining edges (branches) from v are added to the set B (Lines 21-23).

To make the traversal efficient, whenever a path ρ is found, all the vertices $r : degree(r) > 1$ are added to the set of $Reachable$. This technique can help in making the future runs of DFS efficient. Whenever a vertex in the set $Reachable$ is encountered, we stop the DFS and return the current path.

4.4 Instrumenter

The Instrumenter component uses Sets Δ_i (differences between method M_{i1} and M_{i2}) produced by the Difference Finder. For each changed method pair $< M_{i1}, M_{i2} >$ for which $\Delta_i \neq \phi$, the Instrumenter component finds a region δ_i containing all the changed instructions in the program. δ_i is a minimal list of continuous instructions such that all the changed instructions in the method M_i are in the region δ_i. Hence, there can be a maximum of one changed region in one method. At the end of each changed region δ_i, the Instrumenter component inserts instructions to save the program state. In particular, the Instrumenter inserts the corresponding instructions for `PexStore` statements for each variable (and field) defined in the changed region. The `PexStore` statement for a variable x results in an assertion statement `PexAssert.IsTrue( "uniqueName", x == currentX)` in the generated test, where `currentX` is the value of x in the new version. The Dynamic Test Generator component generates tests for the new version $v2$. After the Dynamic Test Generator component generates a test that executes a changed region (note that in $v1$, we do not have a changed region), the the component executes the test on Version $v1$ to compare program states after the execution of the changed region with the ones captured in the execution of Version $v2$. The instrumentation enables us to perform only one instance of DSE on the new version instead of performing two instances of DSE: one on the original and the other on the new program version. Performing two instances of DSE can be technically challenging since we have to perform the two DSE instances in a controlled manner such that both versions are executed with the same input and the execution trace is monitored for both the versions by a common exploration strategy to decide which branching node to flip next in the two versions.

4.5 Dynamic Test Generator

The Dynamic Test Generator component performs Dynamic Symbolic Execution (DSE) [5, 15, 11, 23, 4] to generate regression tests for the two given versions of a program. DSE iteratively generates test inputs to cover various feasible paths in the program under test (the new version in our approach). In particular, DSE flips some branching node discovered in previous executions to generate a test input for covering a new path. The node to be flipped is decided by a search strategy such as depth-first search. The exploration is quite expensive since there are an exponential number of paths with respect to the number of branches in a program. However, the execution of many branches often cannot help in detecting behavioral differences. In other words, covering these branches does not help in satisfying any of the condition E or I in the PIE model described in Section 1. Therefore, we do not flip such branching nodes in our new search strategy for generating test inputs that detect behavioral differences between the two given versions of a program. Recall that, we refer to such branches as irrelevant branches. These branches are found using the Graph Traverser component. We next describe the two categories of paths that our approach avoids exploring, and then describe their corresponding branches. We finally describe our incremental exploration technique when an existing test suite is available for the original version.

4.5.1 *Path Pruning*

Our approach avoids exploring the following categories of paths:
Category E. Let $V = v_1, v_2, .., v_n$ be the set of all vertices in CFG g such that $v_i \in g$ and $v_i.degree > 1$. Let $E_i = e_{i1}, e_{i2}, ..., e_{in}$ be the set of outgoing edges (branches) from v_i. Let C be the set of changed vertices in g, $\rho(v_i, C, e_{ij})$ denotes a path (if exists) between a vertex v_i to any vertex in set C that takes the branch e_{ij} of Vertex v_i. For all vertices $v_i \in V$, our approach avoids from exploring all the branches $e_{ij} \in E_i : \rho(v_i, C, e_{ij}) = \phi$ (such branches are found using the Graph Traverser component). In other words, if a branching node v_i is in the built dynamic execution tree[14], such that the branch e_{ij} is not explored yet, our approach does not flip v_i to make program execution take branch e_{ij}. As an effect, our approach helps reduce the number of paths to be explored by avoiding the paths $P \in g : e_{ij} \in P$.
Category I. Let $C \in g$ be the set of nodes in all the changed regions. Let some $c_i \in C$ be executed i.e, at least one instance c_{ij} of c_i is present in the dynamic execution tree T. Consider that the program state is not infected after the execution of the changed region containing c_i. Our approach prunes all the branching nodes $B \in T$, such that for all $b_i \in B : \rho(c_{ij}, b_i) \neq \phi, b_i \notin C$, where $\rho(c_i, b_i)$ is the path between the vertices c_i and b_i (if exists) in T. As an effect of pruning B, our approach does not explore the subpaths after a changed region's execution that does not cause any state infection.

4.5.2 *Incremental Exploration*

A regression test suite achieving high code coverage may be available along with the original version of a program. This test suite may be manually written or generated by an automated test generation tool for the original version. However, the existing test suite might not be able to cover all the changed regions of the new version of the program. Our approach can reuses the existing test suite so that changed regions of the program can be executed efficiently due to which test generation is likely to find behavior differences earlier in path exploration. Our approach executes the existing test suite to build an execution tree for the tests in the test suite. Our approach then starts the program exploration using the dynamic execution tree instead of starting from an empty tree. Our approach of seeding test inputs can help efficiently cover the changed regions of the program with two major reasons:
Discovery of hard-to-discover branching nodes. By seeding the existing test suite for Pex to start exploration with, our approach executes the test suite to build an execution tree of the program. Some of the branching nodes in the built execution tree may take a large number of DSE runs (without seeding any tests) to get discovered. Flipping some of these discovered branching nodes nearer in CFG to the changed parts of the program has more likelihood of covering the changed regions of the program [3]. Although, our approach currently does not specifically prioritize flipping of branch-

[14] A dynamic execution tree is the tree formed from the paths executed in the previous executions. Multiple instances $c_{i1}, c_{i2}, ..., c_{in}$ of a node c_i in CFG can be present in a dynamic execution tree due to loops in a program.

ing nodes near the changed regions, our approach can help these branching nodes to get discovered (by executing the existing test suite), which might take large number of DSE runs as shown in the example in Section 3.

Priority of DSE to cover not-covered regions of the program. DSE techniques employ branch prioritization so that a high coverage can be achieved faster due to which DSE techniques choose a branch from the execution tree (built thus far) that have a high likelihood of covering changed regions (that are not covered by existing test suite for the original version). By seeding the existing test suite to program exploration, the DSE techniques do not waste time on covering the regions of the program already covered by the existing test suite. Instead, the DSE techniques give high priority to branching nodes that can cover not-covered regions of the program, which include the changed parts. Hence, the changed parts are likely to be covered earlier in path exploration.

5. EXPERIMENTS

We conducted experiments on four programs and their 72 versions (in total) collected from three different sources. In our experiments, we try to address the following research questions:

RQ1. How much fewer DSE runs does Pex require to execute the changed regions between the two versions of a program with the assistance of `eXpress`?

RQ2. How much fewer DSE runs does Pex require to infect the program states with the assistance of `eXpress`?

RQ3. How much more tests does Pex, with the assistance of `eXpress`, generate that execute changed regions between the two versions of a program?

RQ4. How much more tests does Pex, with the assistance of `eXpress`, generate that infect the program states?

RQ5. How much fewer DSE runs does Pex require to cover the changed regions when the program exploation is seeded with the existing test suite?

5.1 Subjects

To answer the research questions, we conducted experiments on four subjects. Table 1 shows the details about the subjects. Column 1 shows the subject name. Column 2 shows the number of classes in the subject. Column 3 shows the number of classes that are covered by tests generated in our experiments. Column 4 shows the number of versions (not including the original version) used in our experiments. Column 5 shows the number of lines of code in the subject.

`replace` and `siena` are programs available from the Subject Infrastructure Repository (SIR) [7]. `replace` and `siena` are written in C and $Java$, respectively. `replace` is a text-processing program, while `siena` is an Internet-scale event notification program. We chose these two subjects (among the others available at the SIR) in our experiments we could convert these subjects into C# using the Java 2 CSharp Translator[15]. We could not convert other subjects available at the SIR with the exception of `tcas`. The experimental results on `tcas` are presented in a previous version of this work [25] and show similar conclusions as the results from the subjects used in the experiments here. We seeded all the 32 faults available for `replace` at the SIR one by one to generate 32 new versions of `replace`. For `siena`, SIR contains 8 different sequentially released versions of `siena` (versions 1.8 through 1.15). Each version provides enhanced functionalities or corrections with respect to the preceding version. We use these 8 versions in our experiments. In addition to these 8 versions, there are 9 seeded faults available at SIR. We seeded all the 9 faults available at SIR one by one to synthesize 9 new versions of `siena`. In total, we conduct experiments on these 17 versions of `siena`. For `replace`, we use the `main` method as a PUT for generating tests. For `siena`, we use the methods `encode` (for changes that are transitively reachable from `encode`) and `decode` (for changes that are transitively reachable from `decode`) in the class `SENP` as PUTs for generating tests. The method `encode` requires non-primitive arguments. Existing Pex cannot handle non-primitive argument types effectively but provides support for using factory methods for non-primitive types. Hence, we manually wrote factory methods for the non-primitive types in `SENP`. In particular, we wrote factory methods for classes `SENPPacket`, `Event` and `Filter`. Each factory method invokes a sequence (of length up to three) of the public state-modifying methods in the corresponding class. The parameters for these methods, and the length of the sequence (up to three) are passed as inputs to the factory methods. During exploraton, Pex generates concrete values for these inputs to cover various parts of the program under test.

STPG[16] is an open source program hosted by the codeplex website, Microsoft's open source project hosting website[17]. The codeplex website contains snapshots of check-ins in the code repositories for STPG. We collect three different versions of the subject STPG from the three most recent check-ins. We use the `Convert(string path)` method as the PUT for generating tests since `Convert` is the main conversion method that converts a string path data definition to a `PathGeometry` object.

`structorian`[18] is an open source binary data viewing and reverse engineering tool. `structorian` is hosted by Google's open source project hosting website[19]. The website also contains snapshots of check-ins in the code repositories for `structorian`. We collected all the versions of snapshots for the classes `StructLexer`, `BaseLexer`, and `StructParser`. We chose these classes in our experiments due to three factors. First, these classes have several revisions available in the repository. Second, these classes are of non-trivial size and complexity. Third, these classes have corresponding tests available in the repository. For the classes `StructLexer` and `StructParser` , we generalized one of the available concrete test methods by promoting primitive types to arguments of the test methods and removing the assertions. We used these generalized test methods as PUTs for our experiments. `structorian` contains a manually written test suite. We use this test suite for seeding the exploration for addressing the question RQ5.

To address questions RQ1-RQ4, we use all the four subjects, while to address question RQ5, we use `structorian` because of two major factors. First, `structorian` has a manually written test suite that can be used to seed the exploration. Second, revisions of `structorian` contains non-trivial changes that cannot be covered by the existing test suite. Hence, our technique of seeding the existing test suite in the program exploration is useful for covering these changes. `replace` contains changes to one statement due to which most of the changes can be covered by the existing test suite. `siena` and `STPG` do not have an existing test suite to use.

5.2 Experimental Setup

For `replace` and `siena`, we conduct regression test generation between the original version and each version $v2$ synthesized from

[15]`http://sourceforge.net/projects/j2cstranslator/`

[16]`http://stringtopathgeometry.codeplex.com/`
[17]`http://www.codeplex.com`
[18]`http://code.google.com/p/structorian/`
[19]`http://code.google.com`

Table 1: Experimental subjects

Project	Classes	Classes Covered	Versions	LOC
replace	1	1	32	625
STPG	1	1	2	684
siena	6	6	17	1529
structorian	70	8	21	6561

the available faults in the SIR. We use `eXpress` and the default search strategy in Pex [26, 30] to conduct regression test generation. In addition to the versions synthesized by seeding faults, we also conduct regression test generation between each successive versions of `siena` (versions 1.8 through 1.15) available in SIR, using `eXpress` and the default search strategy in Pex [26, 30]. For STPG and `structorian`, we conduct regression test generation between two successive pairs of versions that we collected.

To address RQ1, we compare the number of runs of DSE required by the default search strategy in Pex (in short as Pex) with the number of runs required by Pex enhanced with `eXpress` (in short as Pex+eXpress) to execute a changed region. To address RQ2, we compare the number of runs required by Pex with the number of runs required by Pex+eXpress to infect the program states. To address RQ3, we compare the number of tests that cover a changed region generated by Pex with the number of such tests generated by Pex+eXpress. If more tests are generated that cover a changed region, it is easier for developers (or testers) to debug the program under test (if the changes are faulty) and gives more confidence to developers that the changes they made do not introduce any unintended side effects. To address RQ4, we compare the number of tests that infect the program state after the execution of changed region generated by Pex with the number of such tests generated by Pex+eXpress. To address RQ5, we compare the number of DSE runs required by Pex (and Pex+eXpress) to cover all the blocks[20] in all the changed regions with and without seeding the program exploration (with the existing test suite).

Currently, we have not automated the steps to prune branches that cannot help in achieving I of the PIE model. To simulate the pruning of branches to achieve I, in our experiments, we manually instrument the new version to throw an exception immediately after the changed regions, if the program state is not infected after the execution of the changed region. If the changed region is located inside a loop, we throw the exception immediately after the loop. In future work, we plan to automate the pruning of branches that cannot help in satisfying I. The rest of the approach is fully automated and is implemented in a tool as an extension[21] to Pex [26]. We developed its components to statically find irrelevant branches as a .NET Reflector[22] AddIn.

5.3 Experimental Results

Table 2 shows the experimental results. Due to space limit, we provide only the total and average values for the subjects `replace`, `siena`, and `STPG`. The detailed results for experiments on all the versions of these subjects are available on our project web[23]. However, we provide detailed results for `structorian` in this paper.

Column S shows the name of the subject. For `structorian`, the column shows the class name. Column V shows the number of version pairs for which we conducted experiments for the subject. For `structorian`, the column shows the version numbers on which the experiments were conducted. These version numbers are the revision numbers in the google code repository of `structorian`. Column E_{Pex} shows the total number of DSE runs required Pex for satisfying E. Column $E_{eXpress}$ shows the total number of DSE runs required by Pex+eXpress for satisfying E. Column E_{Red} shows the percentage reduction in the number of DSE runs by Pex+eXpress for achieving E. Column Ne_{Pex} shows the total number of tests that execute a changed region, generated by Pex. Column $Ne_{eXpress}$ shows the total number of tests that execute a changed region, generated by Pex+eXpress. Column Ne_{Inc} shows the percentage increase in the number of generated tests that execute a changed region. Column I_{Pex} shows the total number of DSE runs required by Pex for satisfying I. Column $I_{eXpress}$ shows the total number of DSE runs required by Pex+eXpress for satisfying I. Column I_{Red} shows the percentage reduction in the number of DSE runs by Pex+eXpress for achieving I. Column Ni_{Pex} shows the total number of tests generated by Pex, that infect the program state. Column Ni_{eXp} shows the total number of tests generated by Pex+eXpress, that infect the program state. Column Ni_{Inc} shows the percentage increase in the number of generated tests that execute a changed region by Pex+eXpress.

Table 3 shows the time taken for finding the irrelevant branches, time taken to generate tests, and the number of irrelevant branches found. Column S shows the subject. Column T_{static} shows the average time taken by `eXpress` to find irrelevant branches that cannot help in satisfying E of the PIE model. Column $T_{Pex}(s)$ shows the average time taken by Pex to generate tests. Column $T_{eXpress}$ shows the average time taken by Pex+eXpress to generate tests. Column B_{Irr} shows the average number of irrelevant branches that cannot help in satisfying E of the PIE model. In general, irrelevant branches are more if changes are towards the beginning of the PUT since there are likely to be more branches in the program that do not have a path to any changed regions. These branches also include the branches whose branching conditions are not dependent on the inputs of the program and therefore do not correspond to branching conditions during path exploration. Hence, pruning these branches is not helpful in making DSE efficient. Column B_{Tot} shows the total number of branches in the CFG.

Results of replace. For the `replace` subject, among the 32 pairs of versions, the changed regions cannot be executed for 4 of theses versions (Versions 14, 18, 27, and 31) by Pex or by Pex+eXpress in 1000 DSE runs. We do not include these versions while calculating the sum of DSE runs for satisfying I and E of the PIE model. For 3 of the versions (Versions 3, 22 and 32), the changed region was executed but the program state is not infected in 1000 DSE runs. We do not include these versions while calculating the sum of DSE runs for satisfying I of the PIE model. For 3 of the versions (Versions 12, 13, and 21), the changes are in the fields due to which there are no benefits of using Pex+eXpress. We exclude these three versions from the experimental results shown in Table 2, which includes the results of 32 versions.

Pex+eXpress takes around 5.7 seconds (on average) to find the irrelevant branches for each version of `replace`. We also observe that the time varies for different versions (between 0.3 to 21.4 seconds) since our optimizations (discussed in Section 4) depend on the location of a change. In total, Pex+eXpress took 51.6% fewer runs in executing the changes with a maximum of 77.6% for Versions 23 and 24. For these versions, Pex+eXpress takes 95 DSE runs in contrast to 425 runs taken by Pex to execute the changed locations. In addition, Pex+eXpress took 46% fewer runs, in infecting the program state, with a maximum of 73.8% for Version 6. For this version, eXpress takes 83 DSE runs in contrast to 317

[20]A block is a set of statements in a program having a single entry and a single exit (i.e a block cannot contain if, $while$, $switch$ and $return$ statements).

[21]`http://pexase.codeplex.com/`

[22]`http://www.red-gate.com/products/reflector/`

[23]`https://sites.google.com/site/asergrp/projects/express/`

Table 2: Experimental results

		Execution						Infection					
S	V	E_{Pex}	E_{eXpress}	$E_{Red}(\%)$	Ne_{Pex}	Ne_{eXpress}	$Ne_{Inc}(\%)$	I_{Pex}	I_{eXpress}	$I_{Red}(\%)$	Ni_{Pex}	Ni_{eXpress}	$Ni_{Inc}(\%)$
replace	32	1946	789	59.4	1183	2249	90	3203	1716	46.4	358	579	62
siena	17	286	166	42	549	1214	121.1	284	172	39.4	336	908	170.2
STPG	2	341	250	26.1	38	48	26.3	378	255	32.4	10	13	30
Total	51	2573	1205	53.1	1770	3511	98.4	3865	2143	44.6	704	1500	113.1
——————structorian——————													
SL	2-9	102	75	26.5	24	38	58.3	102	75	26.5	24	38	58.3
SL	9-139	102	75	26.5	24	38	58.3	152	107	29.6	8	11	37.5
SL	139-150	102	75	26.5	24	38	58.3	102	75	26.5	13	18	38.5
SL	150-169	53	46	13.2	20	25	25	53	46	13.2	20	25	25
SL	169-174	55	48	12.7	323	411	21.4	55	48	12.7	230	281	22.2
SL	174-175	102	75	26.5	24	38	58.3	-	-	-	-	-	-
SL	175-184	19	15	21.1	41	48	17.1	21	21	0	13	17	30.8
Total(SL)		535	396	26	480	636	24.5	485	372	23.3	308	390	26.6
BL	45-174	2	2	0	999	999	0	3	3	0	243	265	9.1
BL	174-175	2	2	0	999	999	0	3	3	0	243	265	9.1
SP	2-5	-	1866	-	-	-	-	-	2587	-	-	-	-
SP	5-6	-	2614	-	-	-	-	-	2614	-	-	-	-
SP	9-13	-	1866	-	-	-	-	-	1866	-	-	-	-
SP	37-39	6	6	0	128	165	28.9	-	851	-	-	5	-
SP	39-40	2	2	0	150	167	11.3	-	-	-	-	-	-
SP	50-62	6188	1053	82.9	-	-	-	6188	1053	82.9	-	-	
SP	45-47	2	2	0	43	53	23.3	2	2	0	43	53	23.3
SP	47-50	2	2	0	43	53	23.3	2	2	0	43	53	23.3
SP	62-124	2	2	0	43	53	23.3	2	2	0	43	53	23.3
SP	124-125	2	2	0	43	53	23.3	2	2	0	43	53	23.3
SP	125-166	-	7452	-	-	-	-	-	7452	-	-		-
SP	40-45	-	8214	-	-	-	-	-	8276	-	-	-	-

runs taken by Pex to infect the program state after the execution of changed locations.

Results of siena. We observe that the changes in seven of the versions of `siena` are covered within ten runs by Pex and Pex+eXpress. For these changes, there is no reduction in the number of runs . However, the number of generated tests that cover a changed region increases by a significant amount while using Pex+eXpress as compared to Pex. The reason for the preceding phenomenon is that these changes are close to the entry vertex in the CFG. Hence, these changes can be covered in a relatively small number of runs. Moreover, for these types of changes, Pex+eXpress finds relatively large number of irrelevant branches because many of the branches in the CFG after these changes need not be explored to execute the changed region. As a result, test generation focuses on flipping significantly fewer branches (that are close to before the change) due to which the tests that cover a changed region increases significantly. In two of the versions, changed regions were not covered by either Pex+eXpress or Pex. An exception is thrown by the program before these changes could be executed. Pex and Pex+eXpress are unable to generate a test input to avoid the exception. Two of the changes are refactoring due to which the program state is never infected. In summary, Pex+eXpress executed the changed region in 42% fewer runs to execute the changes as compared to Pex and generates 121.1% more tests that execute the changed regions. In addition, Pex+eXpress infects the program state in 39.4% fewer runs and generates 170.2% more tests that infect the program state.

Results of structorian. The seven rows with SL in column S of Table 2 show the experimental results for changes in the class `StructLexer`. The next two rows (with BL in column S) show the experimental results for changes in the class `BaseLexer`, while the last 12 rows (with SP in column S) show the experimental results on versions of the class `StructParser`. For the versions of `StructLexer`, Pex+eXpress takes 26% fewer runs to execute a changed region than Pex. In addition, Pex+eXpress generates 24.5% more tests that cover a changed region than Pex. In addition, Pex+eXpress infects the program state in 23.3% fewer runs and generates 26.6% more tests that infect the program state. The

Table 3: Time and irrelevant branches

S	$T_{static}(s)$	T_{Pex}	T_{eXp}	B_{Irr}	B_{Tot}
replace	5.83	151.2s	139.5s	90	181
siena	4.11	78s	75s	34	185
STPG	35	176s	173s	16	272
structorian (SL)	0.47	135 s	131s	33	383
structorian (BL)	0.5	148 s	151s	9	75
structorian (SP)	703	1:27 hr	45 min	49	447

changes in `BaseLexer` were just after the CFG entry vertex due to which all the generated tests execute the changed region. Neither Pex+eXpress nor Pex were able to cover any changed region for five of the versions of class `StructParser` in 1000 DSE runs (a bound that we use in our experiments for all subjects). For these versions, we increased the bound to 10,000 runs. For only 1 of these 6 versions (Version 50-62), Pex was able to execute the changed region in 10,000 runs, while Pex+eXpress executes the changed regions for all the 6 versions and infect the program state for all of the 6 versions. Pex+eXpress takes a non-trivial time of 700 seconds to find irrelevant branches for the class `StructParser` due to a large number of method invocations. However, considering that most of the changes cannot be covered even in 10,000 runs by Pex (more than 2.5 hours of exploration) the time taken to find irrelevant branches is significantly less. Average exploration time of Pex+eXpress was 45 minutes as compared to 87 minutes for Pex. We stopped the exploration (for the versions we ran our experiments with a bound of 10,000 runs) as soon as we found a state infection. As a result, Pex had to explore more paths than Pex+eXpress due to which the average exploration time of Pex is significantly more.

Seeding program exploration with existing tests. Table 4 shows the results obtained by using the existing test suite to seed the program exploration. Column C shows the class name. Column V shows the pair of version numbers. The next four columns show the number of runs taken by the four techniques: Pex, Pex with seeding, Pex+eXpress, and Pex+eXpress with seeding, respectively, for covering all blocks in all changed regions. In Table 4, if all the changed blocks are not covered, we take the number of runs as 10,000 (the maximum number of runs that we ran our experiments

Table 4: Results obtained by seeding existing test suite for structorian

C	V	N_{Pex}	Np_{seed}	$N_{eXpress}$	Ne_{seed}
SP	2-5	10000*	10000*	2381	181
SP	37-39	1355	60	851	47
SP	39-40	10000*	304	10000*	251
SP	45-47	10000*	10000*	10000*	10000*
SP	47-50	10000*	81	1341	64
SP	62-124	10000*	59	2067	41
SL	169-174	34	18	34	18
SL	150-169	53	37	46	29
SL	9-139	10000*	69	1089	52
Total		71452	20697	28898	10735

*If all the changed blocks are not covered, we take the number of runs as 10,000 (the maximum number of runs that we ran our experiments with)

with). For 9 of the version pairs of `structorian` (out of 21 that we used in our experiments), the existing test suite of `structorian` could not cover all blocks of the changed regions. Therefore, we consider these 9 version pairs for our experiments. Pex could not cover all the blocks in the changed regions for 6 of the 9 version pairs in 10,000 runs. Seeding the program exploration with the existing test suite helps Pex in covering all the blocks in under 100 runs for 4 of these version pairs under test. There is a considerable reduction in the number of runs in the other version pairs with the exception of versions 2-5 and 45-47 in which seeding cannot help Pex in covering all the blocks in changed regions. In summary, Pex requires around 71% of the original runs (required by Pex without test seeding) and Pex+eXpress requires around 63% of the original runs (required by Pex+eXpress without test seeding).

6. RELATED WORK

Previous approaches [9, 24] generate regression unit tests achieving high structural coverage on both versions of the class under test. However, these approaches explore all the irrelevant paths, which cannot help in achieving any of the conditions I or E in the PIE model [28]. In contrast, we have developed a new search strategy for DSE to avoid exploring these irrelevant paths.

Santelices et al. [2, 22] use data and control dependence information along with state information gathered through symbolic execution, and provide guidelines for testers to augment an existing regression test suite. Unlike our approach, their approach does not automatically generate tests but provides guidelines for testers to augment an existing test suite.

Some existing search strategies [3, 30] guide DSE to efficiently achieve high structural coverage in a program under test. However, these techniques do not specifically target covering a changed region. In contrast, our approach guides DSE to avoid exploring paths that cannot help in executing a changed region. In addition, our approach avoids exploring paths that cannot help in I of the PIE model [28].

Differential symbolic execution [19] determines behavioral differences between two versions of a method (or a program) by comparing their symbolic summaries [10]. Summaries can be computed only for methods amenable to symbolic execution. However, summaries cannot be computed for methods whose behavior is defined in external libraries not amenable to symbolic execution. Our approach still works in practice when these external library methods are present since our approach does not require summaries. In addition, both approaches can be combined using demand-driven-computed summaries [1], which we plan to investigate in future work.

Li [16] prioritizes source code portions for testing based on dominator analysis. In particular, her approach finds a minimal set of blocks in the program source code, which, if executed, would ensure the execution of all of the blocks in the program. Howritz [13] prioritizes portions of source code for testing based on control and flow dependencies. These two approaches focus on testing in general. In contrast, our approach focuses specifically on regression testing.

Ren et al. develop a change impact analysis tool called Chianti [20]. Chianti uses a test suite to produce an execution trace for two versions of a program, and then categorizes and decomposes the changes between two versions of a program into different atomic types. Chianti uses only an existing test suite and does not generate new tests for regression testing. In contrast, our approach focuses on regression test generation.

Some existing capture and replay techniques [8, 18, 21] capture the inputs and outputs of the unit under test during system-test execution. These techniques then replay the captured inputs for the unit as less expensive unit tests, and can also check the outputs of the unit against the captured outputs. However, the existing system tests do not necessarily exercise the changed behavior of the program under test. In contrast, our approach generates new tests for regression testing.

Joshi et al. [14] use the path conditions from the paths followed by the tests in an existing test suite to generate inputs that violate the assertions in the test suite. The generated test inputs follow the same paths already covered by the existing test suite and do not explore any new paths. In contrast, our approach exploits the existing test suite to explore new paths. Majumdar and Sen [17] propose the concept of hybrid concolic testing. Hybrid concolic testing seeds the program with random inputs so that the program exploration does not get stuck at a particular program location. In contrast, our approach exploits the existing test suite to seed the program exploration. Since the existing test suite is expected to achieve a higher structural coverage, the existing test suite is expected to discover more hard-to-discover branching nodes in comparison with random inputs. Godefroid et al. [12] propose a DSE based approach for fuzz testing of large applications. Their approach uses a single seed for program exploration. In contrast, our approach seeds multiple tests to program exploration. Seeding multiple tests can help program exploration in covering the changes more efficiently as discussed in Section 4.5.2.

7. DISCUSSION

In this section, we discuss some of issues of the current implementation of our approach and how they can be addressed.

Added/Deleted and Refactored Methods. If a method M (or a field F) is added or deleted from the original program version, `eXpress` does not detect M (or F) as a changed region. The change is detected if a method call site (or reference to F) is added or deleted from the original program version. If the added or deleted method (or field) is never invoked, the behavior of the two versions is the same unless M is an overriding method. We plan to incorporate support for handling such overriding methods that are added or deleted. Similarly, if a method M is refactored between the two versions, `eXpress` does not detect M as a changed region. However, when a method is refactored, its call sites are changed accordingly (unless the method undergoes `Pull Up` or `Push Down` refactoring). Hence, `eXpress` detects the method containing call sites of M as changed. In our experiments, we considered versions of `replace` in which a method signature was changed, and versions of `structorian` in which a method was renamed.

Granularity of Changed Region. In our current implementation, a changed region is the list of continuous instructions that include all the changed instructions in a method. One method can have only a single changed region. Hence, a changed region can be as big as a method and as small as a single instruction. The granularity of a changed region can be increased to a single method or reduced to single instruction. Changing the granularity to single method M can affect the efficiency of our approach in reducing DSE runs since some of the branches in M that should be considered irrelevant would not be considered irrelevant. In contrast, reducing the granularity to a single instruction makes our approach more efficient in reducing DSE runs. However, the overhead cost of our approach is increased due to state checking at multiple points in the program. In future work, we plan to enhance `eXpress` to allow users to choose from different levels of granularity.

Original/New Program Version. In our current implementation, we perform DSE on the new version of a program. We then execute a test (generated after each run) on the original version. We can also perform DSE on the original version instead of the new version. One approach may be efficient than the other depending on the types of changes made to the program. In future work, we plan to conduct experiments to compare the efficiency of the two approaches with respect to the types of changes.

Branch Prioritization. `eXpress` currently prunes branches that cannot help in achieving E or I conditions in the PIE model. However, some branches in the program code can be more promising in achieving these conditions than others. Branching nodes can be prioritized based on the distance of a branching node to a changed region in the CFG. The distance $d(n1, n2)$ between any two nodes $n1$ and $n2$ in a CFG g is the number of nodes with degree > 1 between $n1$ and $n2$ in the shortest path between $n1$ and $n2$. Hence, the distance between a node b and a changed region Δ is the number of nodes with degree of more than one between b and the node δ representing the first instruction in Δ. The intuition behind this prioritization is that shorter the distance between the branching node and δ the easier it is to generate inputs to cause the execution of the changed region δ. This kind of branch prioritization is used by Burnim and Sen [3] for achieving high structural coverage. We can also prioritize branching nodes based on the probability to cause infection and to propagate the infection to an observable output. Moreover, we can prioritize branches based on data dependence from a changed region.

Pruning of Branches for Propagation. Currently, `eXpress` prunes branches that cannot help satisfy E or I of the PIE model for change propagation. In future work, we plan to prune more categories of branches that cannot help in Propagation (P). Consider that a changed region is executed and the program state is infected after the execution of the changed region; however, the infection is not propagated to any observable output. Let χ be the last location in the execution path such that the program state is infected before the execution of χ but not infected after its execution. χ can be determined by comparing the value spectra [29] obtained by executing the test on both versions of the program. This category contains all the branching nodes after the execution of χ. These branches can be obtained by inspecting the path P followed in the previous DSE run. Let $P =< b_1, b_2, .., b_\chi ... b_n >$, where bi is the a branching node in Path P, while b_χ is the last branching node containing χ. We flip the branching nodes starting from b_χ to b_n in P until a branching node where the infected program state is propagated.

8. CONCLUSION

Regression testing aims at generating tests that detect behavioral differences between two versions of a program. To expose behavioral differences, a test execution needs to satisfy the conditions: Execution (E), Infection (I), and Propagation (P), as stated in the PIE model [28]. Dynamic symbolic execution (DSE) can be used to generate tests for satisfying these conditions. DSE explores paths in the program to achieve high structural coverage, and exploration of all these paths can often be expensive. However, many of these paths in the program cannot help in satisfying the three conditions in any way. In this paper, we presented an approach and its implementation called `eXpress` for regression test generation using DSE. `eXpress` prunes paths or branches that cannot help in detecting the E or I condition such that these conditions are more likely to be satisfied earlier in path exploration. In addition, our approach can exploit the existing test suite for the original version to efficiently execute the changed regions (if not already covered by the test suite). Experimental results on various versions of programs showed that our approach can efficiently satisfy E or I conditions than without using our approach.

9. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. TACAS*, pages 367–381, 2008.

[2] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. TAICPART*, pages 137–146, 2006.

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

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proc. CCS*, pages 322–335, 2006.

[5] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *Proc. ECOOP*, pages 404–428, 2006.

[7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, pages 405–435, 2005.

[8] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *TSE*, 35(1):29–45, 2009.

[9] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. FSE*, pages 549–552, 2007.

[10] P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.

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

[12] P. Godefroid, M. Y. Levin, and D. A. Molnar. In *Proc. NDSS*, pages 151–166, 2008.

[13] S. Horwitz. Tool support for improving test coverage. In *Proc. ESOP*, pages 162–177, 2002.

[14] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: Amplifying the effectiveness of software testing. In *Proc. ESEC-FSE*, pages 561–564, 2007.

[15] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[16] J. J. Li. Prioritize code for testing to improve code coverage of complex software. In *Proc. ISSRE*, pages 75–84, 2005.

[17] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proc. ICSE*, pages 416–426, 2007.

[18] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc.WODA*, pages 1–7, 2005.

[19] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.

[20] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: A change impact analysis tool for java programs. In *Proc. ICSE*, pages 664–665, 2005.

[21] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, 2005.

[22] R. A. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. ASE*, pages 218–227, 2008.

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

[24] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.

[25] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *Proc. ICSE, NIER*, May 2009.

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

[27] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[28] J. Voas. PIE: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.

[29] T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *TSE*, 31(10):869–883, 2005.

[30] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, 2009.