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 as 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) to effectively execute the changed parts of the program and infect the program state. Experimental results on ** versions of four programs show that our approach requires about **% fewer runs (i.e., explored paths) on average to cause the execution of a changed region and **% fewer to cause program-state differences after its execution than exploration without guidance. In addition, our approach requires **% fewer runs to cover all the changed blocks by utilizing 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, 20, 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 as 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 [25] 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] (and its implementation called `eXpress`) 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 utilize the existing test suite (if available) by seeding the tests in the test suite to the program exploration. Our technique of seeding the exploration with existing test suite can be used to effectively augment an existing test suite

[1]An earlier version of this work [22] is described in a four-page paper that will appear 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 utilizing the existing test suite for effectively generating regression tests. Third, we automate our approach by developing a tool called `eXpress`. Fourth, we conduct more 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.

so that various parts of the program that were previously uncovered by the existing test suite are covered by the augmented test suite.

This paper makes the following major contributions:

Path Exploration for Regression Test Generation. We propose an approach 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 specifically for regression test generation.

Incremental Test Generation. We develop a technique for utilizing 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 existing test suite for automated regression test generation.

Implementation. We have implemented our approach in a tool `eXpress`, an extension for Pex [23], 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 [23]. The current Pex has been downloaded by thousands of times in industry. Some parts of Pex may be integrated to Microsoft Visual Studio 2010, benefiting an enormous number of developers in industry. `eXpress` efficiently generates regression tests given an original and new version of a software program. The generated tests on execution can detect behavioral differences (if any exist) between the two versions.

Evaluation. We have conducted experiments on ** versions of four programs. Experimental results show that our approach requires about **% fewer runs (i.e., explored paths) on average to cause the execution of a changed region and **% fewer runs on average to cause program-state differences after its execution than exploration without guidance. In addition, our approach requires **% fewer runs to cover all the changed program blocks by utilizing an existing test suite than exploration without using the test suite.

2. BACKGROUND

In this section, we present a background on DSE based tool Pex [23] that we use in our approach. Pex starts the program execution 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 each constraint at a branching statement as a branch condition. The conjunction of all the branch condition in the path followed by the 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, chooses one of the unexplored branch of the execution tree (dynamically discovered thus far). Pex flips the chosen branching node in the dynamic execution tree (discovered thus far) to generate a new input that follows a new execution path. Pex chooses a branching node from the execution tree of the program using various search search strategies attempting to achieve high statement coverage fast. Pex combines all search strategies it uses into a meta-strategy that performs a fair choice between the strategies.

3. EXAMPLE

In this section, we illustrate the `eXpress` approach with an example. `eXpress` takes as input two versions of a program and produces as output a regression test suite. The test suite on execution detects 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.

```
static  public  int  testMe(int  x,
int[] y){
1   int j=1;
2   if(x==90){
3    for(int i=0; i< y.Length; i++){
4      if(y[i] == 15)
5           x++;
6      if(y[i] == 16)
7           j=2;
8      if(y[i] == 25)
9          return x;
10     if(x == 110)
11         x = j+2;//x = 2*j+1
12     if(x>110)
13        return x;
14  }
15 }
16 return x;
}
```

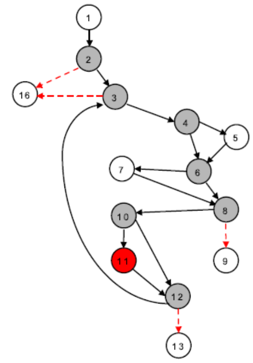

Figure 1: An example program

Consider the example in Figure 1. Suppose that the statement at Line 11 of `testMe` has been modified resulting in the one shown in the comment at Line 11. The `Difference Finder` component of `eXpress` 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 statement at Line 11 is changed in the new version. The `Graph Builder` component of `eXpress` then builds a Control-Flow Graph (CFG) of the new version of the program under test and marks the changed vertices in the graph. Figure 1 also shows the CFG of the example program. The labels of vertices in the CFG show the corresponding line numbers in Figure 1. The red (dark) vertex shows the changed statement at Line 11. The `Graph Traverser` component traverses the CFG to find all the branches[4] b in a program such that if b is taken, the program execution cannot reach the dark vertex at Line 11. On traversing the CFG in Figure 1, the `Graph Traversal` detects that taking the branches $< 2, 16 >$, $< 3, 16 >$, $< 8, 9 >$, and $< 12, 13 >$ (dotted/dark red edges in Figure **??**), the program execution cannot the vertex at Line 11. Since after taking these branches, the execution cannot reach the changed statement at Line 11, the execution of these branches cannot help in executing the statement at Line 11; behavioral differences between two versions of the program cannot be detected without executing the changed statement at Line 11. Hence, these branches are not explored by the `Dynamic Test Generator` component of `eXpress` while generating regression tests for the program under test.

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. The instrumentation allows us to compare the internal behavior of running the same generated test on the two versions.

Figure 2 shows the code of `testMe`'s new version after instrumentation. The `Instrumenter` component inserts a statement (Line 12 in Figure 2) just after any changed statement (Line 11 in Figure 1). The instrumented statement allows us to store the current value of `x` in a particular run (i.e., an explored path) of DSE. In particular, this statement results in an assertion `PexAssert.IsTrue("uniqueName", x == currentX)`[5] in the generated test, where `currentX` is the value of x at Line 12 in the new version of the program. One such assertion is generated by Pex each time the statement is executed in the loop. Hence, if the loop containing the changed statement executes 20 times, 20 such assertions will

[4] A branch is an outgoing edge of a branching node

[5] `PexAssert` is an API class provided by Pex.

```
     public boolean testMe(int x, int[]  y){
          ...
10        if(x == 110){
11            x = 2*j+1;
12            PexStore.ValueForValidation("uniqueName", x);
13        }
          ...
  }
```

Figure 2: Instrumented example program after instrumentation

be added to the test generated by Pex. The generated test can be executed on the original version of `testMe` to compare program states at Line 12 after the execution of the changed statement with the ones captured in the execution of the new version.

If there are multiple changed statements in the program, our approach first finds multiple regions each of which contains nearby changed statements in the program. We refer to each of such regions as a changed region in the rest of the paper. Our approach finds all the variables and fields that are identified as defined in a changed region and inserts statements (such as the statement at Line 12 of Figure 2) to log the value of each defined variable or field in the changed region. If a defined variable is a non-primitive type, such a statement enables to compare the object graphs reachable from the logged values to compare program states. The `Dynamic Test Generator` component of `eXpress` then performs DSE on the instrumented new version of the program to generate regression tests.

`eXpress` performs DSE on the instrumented new version of the program. After each run of DSE, `eXpress` executes the generated test on the instrumented original version (in the same way as instrumentation of the new version) to check whether the program state is infected after the execution of a changed region. 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.

The `Dynamic Test Generator` component of `eXpress` uses Dynamic Symbolic Execution (DSE) to generate tests for the new version. DSE iteratively generates test inputs to cover various feasible paths in the program under test. In particular, DSE flips some branching node from a previous execution to generate a test input for covering a new path. The node to be flipped is decided by a search strategy (also called exploration starategy) such as depth-first search. `Dynamic Test Generator` implements a search strategy for Pex [23] to efficiently find behavioral differences between two versions of a program.

To cover the changed statement at Line 11, DSE needs inputs $x = 90$ and the array y of length greater than 20 where at least 20 elements of y have a value 15 and no element has a value 25. To generate the input, DSE needs to execute the loop at least 20 times. In each iteration DSE has the choice of flipping branching nodes at Lines 4, 6, 8, 10, 12, a search space of 2^{100} paths (considering a loop bound of 20).

To reduce the branch search space of DSE, the `Dynamic Test Generator` component adopts the PIE model [25] of error propagation described in Section 1. In particular, the `Dynamic Test Generator` prunes all the following branches from the search space

```
public int TestMe(char[] c)
  {
  1     int state = 0;
  2     for (int i = 0; i < c.Length; i++)
        {
  3             if (c[i] == '[' && state == 0)
                    state = 1;
  5             else if (c[i] == '{' && state == 1)
                    state = 2;
  7             else if (c[i] == '(' && state == 2)
                    state = 3;
  9             else if (c[i] == '\\' && state == 3)
                    state = 4;
 11             else if (c[i] == '*' && state == 4)
               {
                  state = 5;
 12               if (c.Length == 8)
                      break;
               }
       }
      return state;
  }
```

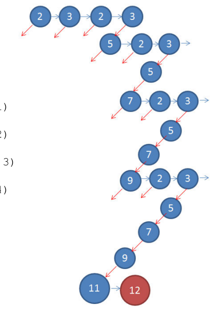

Figure 3: An example program on the left side with a part of its execution tree for c[] = {[, {, (, \, * } on the right

of DSE.

Branches not satisfying E. The branches $< 2, 16 >$, $< 3, 16 >$, $< 8, 9 >$, and $< 12, 13 >$ found to be irrelevant by the `Graph Traverser` component cannot help in executing the changed statement at Line 11. Hence, these branches are not explored by the `Dynamic Test Generator` component.

Branches not satisfying I. Suppose that we cover the changed statement at Line 11 in Figure 1 using inputs $x = 90$ and the array y of length 20 where each element of y has a value 15. The execution takes a path P executing the loop 20 times, assigning the variable x to 110 and eventually covering the changed statement at Line 11 . However, the program state after the execution of changed statement is not infected since after the first execution of the changed statement, the value of x is 3 in both versions. In such situations, the branching nodes in the execution path that are after the last instance of the changed node are not explored. For the example, the branch $< 12, 3 >$ is pruned from exploration.

3.1 Incremental Test Generation

Our approach can reuse the existing test suite so that changed parts of the program can be executed effectively due to which test generation is likely to find behavior differences earlier in path exploration. Figure 3 shows the new version of a program `testMe`. Suppose statements at Lines 12 and 13 have been added in the new version. Let there be an existing test suite covering all the blocks in the old version of the program. Suppose the test suite has a test input I = {"[", "{", "(", "\", "*" }. The input covers all the blocks in the new version of `TestMe` except the newly added block at Line 13. If we start the program exploration from scratch (i.e default input), Pex takes 441 runs to cover the block at Line 13. However, we can reuse the existing test suite for covering the block effectively. Our approach executes the 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 discovered by executing the existing test suite instead of starting from an empty tree. Some of the branches in the tree might take many runs for Pex to discover. The right side of Figure 3 shows the part of the execution tree for the input I. The red edges in the tree indicate false side of the source branching node while the blue edges indicate the true side. For generating an input for the next DSE run, Pex chooses a branching node b in the tree whose 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 using various heuristics for covering new parts of the program. Its likely that Pex chooses to flip the branching node 12 (colored red), which on execution covers the block at Line 12. When starting the program exploration from scratch, Pex takes 420 runs before it could

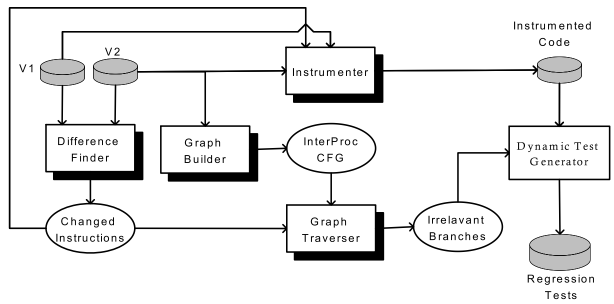

Figure 4: Overview of `eXpress`

discovers the red branching node in Figure 3. Using our approach of seeding the input, Pex takes 39 runs to flip the branching node and cover the block at Line 12.

4. APPROACH

Figure 4 shows the overview of `eXpress`. `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)[6] [24]; such a PUT serves as a test driver for path exploration.

The `Difference Finder` component of `eXpress` finds the set of differences between each of the corresponding method pair in the two versions of program. The `Graph Builder` component builds a partial inter-procedural graph G with the input PUT as the starting method. The `Graph Traversal` component traverses the graph to find all the branches B that need not be explored for executing the changed regions (found by the `Graph Traverser`). The `Dynamic Test Generator` then generates tests for the input PUT, pruning from its search strategy the branches B. We next discuss in detail the major components in `eXpress`.

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[7] of the method, Sig is the signature[8] 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 instruction 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 the rest of its 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 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.

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

Since a moderate-size software system 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 software systems. Hence, we build a minimal inter-procedural CFG for which our purpose of finding branches not reachable to 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. An intra-procedural CFG g is constructed for the method τ. For each method invocation vertex[9] (invoking Method c) in g, the algorithm `InterProceduralCFG` is invoked

[6]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.

[7]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.

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

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

recursively with the invoked method c as the argument (Line 22 of Algorithm 1), while 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:

- **c is in call stack.** If c is already in the call stack, `Inter ProceduralCFG` 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 inter-procedural graph stops after A is encountered the second time.

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

- **c is in `ReachableToChangedRegion`.** The set `Reachable ToChangedRegion` is populated whenever a changed method[10] is encountered. In particular, if a changed method is encountered, the methods currently in the call stack are added to the set `ReachableToChangedRegion` (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).

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 instruction, the preceding three techniques help in achieving the aim while reducing the cost of building the inter-procedural CFG. In addition, the size of the inter-procedural CFG is also 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` component and a set V of changed vertices in the CFG g. `Graph Traverser` traverses the graph to find a set of branches B, being those via which the execution cannot reach any of the branch 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 a 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.

For each Vertex v in g such that $degree(v) > 1$, the `Graph Traverser` performs a depth first search (DFS) from Vertex v (Line 6) to finds a path between v and some vertex $c \in V$ (the pseudo code of DFS is shown in Algorithm 3). 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 discovered (Lines 16-27 in Algorithm 2) by the DFS, there may still be a branch $b_i =< v, v_i >$ that is not reachable to any vertex $c \in V$. To find such branch, we remove the edge $e_j =< v, v_j >: e_j \in \rho$ 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.

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

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** ρ 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:     **for all** Vertex $n \in v.OutVertices$ **do**
17:         $R \leftarrow R\cup < v, v_j >: v_j \in \rho$
18:         $g \leftarrow g.RemoveEdge(e)$
19:         $\rho \leftarrow FindPathUsingDFS(v, V, Reachable)$
20:         **if** ρ is empty **then**
21:             **for all** Vertex $n \in v.OutVertices$ **do**
22:                 $B \leftarrow B\cup < v, n >$
23:             **end for**
24:             $g \leftarrow g.AddEdges(R)$
25:             goto Line 2
26:         **end if**
27:     **end for**
28:     $g \leftarrow g.AddEdges(R)$
29: **end for**
30: **return** B

Algorithm 3 $FindPathUsingDFS(g, v, R)$

Input: A Graph g, a vertex v in the graph g, a set of vertices R.
Output: A path ρ in Graph g from Vertex v to Vertex $c \in V$.

1: $v.Visited \leftarrow true$;
2: ρ.Append(v)
3: **if** $v \in R$ **then**
4:     **return** ρ
5: **end if**
6: **for all** Vertex n in v.OutVertices **do**
7:     **if** $n.Visited \neq true$ **then**
8:         goto Line 6
9:     **end if**
10:     ρ.Append(n);
11:     **if** $n \in R$ **then**
12:         **return** ρ
13:     **end if**
14:     $\varrho \leftarrow FindPathUsingDFS(g, n, R)$
15:     **if** ϱ is not empty **then**
16:         **return** ϱ
17:     **end if**
18:     $\rho.Remove(n)$
19: **end for**
20: ρ.Remove(v)
21: **return** ϕ

If no path is found, the branches from v containing the remaining vertices 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, the DFS is stopped; returning the current path (Lines 3-4 of Algorithm 3).

4.4 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, our approach instruments both versions of the program under test. The instrumentation allows us to compare the internal behavior of running the same generated test on the two versions.

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 at Line 12 in the new version of the example program in Figure 1. The `Dynamic Test Generator` component generates tests for the new version $v2$. Once a test is generated that executes a changed region, the test is executed on Version $v1$ to compare program states after the execution of the changed region with the ones captured in the execution of Version $v2$.

4.5 Dynamic Test Generator

The `Dynamic Test Generator` component performs Dynamic Symbolic Execution (DSE) [5, 13, 11, 20, 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 from a previous execution 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 Traversal` component. We next describe the two categories of paths that our approach avoids exploring, and then describe their corresponding branches.

4.5.1 *Paths being Pruned*

Our approach avoids exploring the following categories of paths:

- **Rationale E: Paths not leading to any changed region.** Paths that cannot reach any changed region (denoted as δ) need not be explored. For example, consider the `testMe` program in Figure 1. The changed statement is at Line 11 (δ). While searching for a path to cover δ, we do not need explore paths containing the `true` branch of the condition at Line 8.

- **Rationale I: Paths not causing any state infection.** Suppose that we cover δ at Line 11 in Figure 1 using inputs `x=90` and the array `y` of length 20 where each element of `y` has a value 15. The execution takes a path P executing the loop 20 times, assigning the variable `x` to 110 and eventually covering δ at Line 11 . However, the program state after the execution of δ is not infected since after the first execution of δ, the value of `x` is 3 in both versions. We need not explore the subpaths after the execution of a changed region that does not cause any state infection if these subpaths do not lead to any other changed region.

4.5.2 *Branching Nodes being Pruned*

In DSE, path exploration is realized by flipping branching nodes. We next describe two categories of branching nodes that we avoid flipping corresponding to the preceding two categories of paths that we intend to avoid exploring.

- **Category E.** This category contains all the branching nodes whose the other unexplored branch cannot lead to any changed region. These branches are obtained from the `Graph Traversal` component, which traverses the inter-procedural CFG constructed by the `Graph Builder` component.

- **Category I.** If a changed region is executed but the program state is not infected after the execution of the changed region, all the branching nodes after the changed region δ in the current execution path are included in this category. These branches are obtained by inspecting the path P followed in the previous DSE run. Let $P =< b_1, b_2, .., b_c...b_n >$, where b_i are the branching nodes in the Path P, while b_c is the last instance of branching node containing δ. We do not flip the branching nodes from b_c to b_n in P if the program state is not infected.

4.6 Incremental Test Generation

Often a regression test suite achieving a high code coverage is available along with a software system. This test suite may be manually written or generated by an automated test generation tool. However, the existing test suite might not be able to cover all the changed parts of the program. Our approach can reuse the existing test suite so that changed parts of the program can be executed effectively 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 effectively cover the changed program parts because of two major reasons:

Discovery of hard to discover branching nodes. By seeding the existing test suite to Pex, our approach executes the tests to build an execution tree of the program (discovered by executing the existing test suite). Some of the branches in the discovered execution tree may take a large number of DSE runs (without seeding any tests) to get discovered. Flipping some of these discovered branching nodes have more likelihood of covering the changed regions of the program as these might be nearer in CFG to the changed parts of the program [3].

Priority of DSE to cover uncovered parts 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 (discovered thus far) that have a high likelihood of covering new program parts. By seeding the existing test suite to program exploration, the DSE techniques do not waste time on covering the parts of the program already covered by the existing test suite. Instead, the DSE techniques give priority to branches that can cover new parts of the program, which include the changed parts. Hence, the changed parts are likely to be covered earlier in path exploration.

5. EVALUATION

We conducted experiments on four programs and their 68 versions (in total) collected from three different sources to assess the effectiveness of `eXpress`. In our evaluation, we try to answer the following research questions:

RQ1. Can `eXpress` more efficiently execute the changed regions between the two versions of a program than without using `eXpress`?

RQ2. Can `eXpress` more efficiently infect the program states after the execution of changed regions than without using `eXpress`?

RQ3. Can `eXpress` effectively help generate tests that execute changed regions between the two versions of a program than without using `eXpress`?

RQ4. Can `eXpress` effectively effectively help generate tests that infect the program states after the execution of changed regions than without using `eXpress`?

RQ5. Can our approach of seeding the exploration with existing unit-tests effectively help covering the changed regions and infect program states?

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 4 column 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 software system. We chose these two subjects (among the others available at the SIR) in our experiments as we could convert these subjects into C# using Java 2 CSharp Translator[11]. We could not convert other subjects available at the SIR with the exception of `tcas`. The experimental results on `tcas` are presented in the previous version of this work [22]. 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 eight 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 eight versions in our experiments. In addition to these eight versions, there are nine seeded faults available at SIR. We seeded all the nine faults available at SIR one by one to synthesize nine 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 types effectively but provides support for writing 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 state-modifying public 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[12] is an open source program hosted by the codeplex website, Microsoft's open source project hosting website[13]. We converted the `replace` program to C# (since the original `replace` is written in C). 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 main method in `replace` as a PUT [24] for generating tests. For STPG, we use the `Convert(string path)` method as the PUT for generating tests. The method `Convert` is the main conversion method that converts a string path data definition to a `PathGeometry` object.

`structorian`[14] is an open source binary data viewing and reverse engineering tool. `structorian` is hosted by Google's open source project hosting website[15]. 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 reasons. 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 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.

For addressing questions RQ1-RQ4 we use all the four subjects, while for addressing the question RQ5 we use `structorian` because of two major reasons: 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 existing test suite. Hence, our approach of seeding 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 for us to use.

5.2 Experimental Setup

For `replace` and `siena`, we find behavioral differences between the original version and each version $v2$ synthesized from the available faults in the SIR. We use `eXpress` and the default search strategy in Pex [23, 30] to find behavioral differences. In addition to the versions synthesized by seeding faults, we also find behavioral

[11] http://sourceforge.net/projects/j2cstranslator/

[12] `http://stringtopathgeometry.codeplex.com/`

[13] `http://www.codeplex.com`

[14] `http://code.google.com/p/structorian/`

[15] `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	18	6561

differences between each successive versions of `siena` (versions 1.8 through 1.15) available in SIR, using `eXpress` and the default search strategy in Pex [23, 30]. For STPG and `structorian`, we find behavioral differences 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 with the number of runs required by `eXpress` to execute a changed region. To address RQ2, we compare the number of runs required by the default search strategy in Pex with the number of runs required by `eXpress` to infect the program states after the execution of a changed region. To address RQ3, we compare the number of tests that cover a changed region generated by `eXpress` with the number of such tests generated by default search strategy in Pex. If more number of 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 unwanted side effects. To address RQ4, we compare the number of tests that infect the program state after the execution of changed region generated by `eXpress` with the number of such tests generated by default search strategy in Pex. To address RQ5, we compare the number of DSE runs required by the default search strategy in Pex (and eXpress) to cover all the blocks 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 called `eXpress`. We developed `eXpress` as an extension[16] to Pex [23]. We developed its components to statically find irrelevant branches as a .NET Reflector[17] AddIn.

5.3 Experimental Results

Table 2 shows the experimental results. Due to space constraints, we only provide 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[18]. 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 experiment was onducted. 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 by the default search strategy in Pex for satisfying E. Column $E_{eXpress}$ shows the total number of DSE runs required by `eXpress` for satisfying E. Column E_{Red} shows the percentage reduction in the number of DSE runs by `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 `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 the default search strategy in Pex for satisfying I. Column $I_{eXpress}$ shows the total number of DSE runs required by `eXpress` for satisfying I. Column I_{Red} shows the percentage reduction in the number of DSE runs by `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 `eXpress`, that infect the program state. Column Ni_{Inc} shows the percentage increase in the number of generated tests that execute a changed region.

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 time taken by `eXpress` to find irrelevant branches that cannot help in satisfying E of the PIE model. Column $T_{Pex}(s)$ shows the time taken by Pex to generate tests. Column $T_{eXpress}$ shows the time taken by `eXpress` to generate tests. Column B_{Irr} shows the number of irrelevant branches. Column B_{Tot} shows the number of irrelevant branches found by `eXpress` 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 condition is not dependent on the inputs of the program and therefore do not lead to branching conditions during path exploration. Hence, pruning these branches is not helpful in making the DSE efficient.

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 the default strategy in Pex or by `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 `eXpress`. We exclude these three versions from the experimental results shown in Table 2.

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

[16] `http://pex.codeplex.com/`

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

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

Table 2: Experimental Results

		Execution						Infection					
S	V	$E_{\texttt{Pex}}$	$E_{\texttt{eXpress}}$	$E_{Red}(\%)$	$Ne_{\texttt{Pex}}$	$Ne_{\texttt{eXpress}}$	$Ne_{Inc}(\%)$	$I_{\texttt{Pex}}$	$I_{\texttt{eXpress}}$	$I_{Red}(\%)$	$Ni_{\texttt{Pex}}$	$Ni_{\texttt{eXpress}}$	$Ni_{Inc}(\%)$
replace	32	1630	789	51.6	X	X	X	3203	1716	46.4	X	X	X
siena	17	286	166	42	549	1214	121.1	284	172	39.4	336	908	170.2
STPG	2	341	250	26.1	X	X	X	378	255	32.4	X	X	X
Total	51	2257	1205	46.6	X	X	X	3865	2143	44.6	X	X	X
——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	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
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	NR	1866	-	-	-	-	-	2587	-	-	-	-
SP	5-6	NR	2587	-	-	-	-	-	2587	-	-	-	-
SP	9-13	NR	1866	0	-	-	-	-	-	-	-	1866	-
SP	39-40	X	X	0	X	X	X	X	X	X	X	X	X
SP	50-62	6188	1053		-	-	-						
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	NR	7452	-	-	-	-	-	7452	-	-		-
SP	40-45	NR	8214	-	-	-	-	-	8276	-	-	-	-

Results of siena. We observe that the changes in seven of the versions of `siena` are covered within ten runs by the default search strategy in Pex and `eXpress`. For these changes, there is no reduction in the number of runs . However, the number of generated tests that cover a changed region increase by a significant amount while using `eXpress` as compared to the default search strategy in Pex. The reason for the preceding phenomenon is that these changes are nearer to the entry vertex in the control flow graph. Hence, these changes can be covered in a relatively small number of runs. Moreover, for these types of changes, `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 near to the change) due to which the tests that cover a changed region increase significantly. In two of the versions, changed regions were not covered by either `eXpress` or the default search strategy in Pex. An exception is thrown by the program before these changes could be executed. Pex and `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, `eXpress`, executed the changed region in 42% less runs to execute the changes as compared to the default search strategy in Pex and generates 121.1% more tests that execute the changed regions. In addition, `eXpress` infects the program state in 39.4% less runs and generates 170.2% more tests that infect the program state.

Results of structorian. The first six rows show the experimental results for changes in the class `StructLexer`. Rows 7 and 8 show the experiments for changes in the class `BaseLexer`, while the last 11 rows show the experimental results on versions of the class `StructParser`. For the versions of `StructLexer`, `eXpress` takes 24.6% less runs to execute a changed region than the default search strategy in Pex. In addition `eXpress` generates 43.3% more tests that cover a changed region than the default search strategy in Pex. In addition, `eXpress` infects the program state in 24.7% less runs and generates 39.7% more tests that infect the program state. The changes in `BaseLexer` were just after the CFG entry vertex

Table 3: Time and Irrelevant Branches for structorian

	Time and Irrelevant Branches for replace, siena, and STPG				
S	$T_{static}(s)$	$T_{Pex}(s)$	$T_{eXp}(s)$	B_{Irr}	B_{Tot}
replace	5.83	X	X	2527	5068
siena	4.11	X	X	576	3146
STPG	0.7	X	X	32	548
structorian (SL)	0.47	X	X	X	548
structorian (BL)	0.5	X	X	X	X
structorian (SP)	703	X	X	X	X

due to which all the generated tests execute the changed region. Both `eXpress` and default search strategy were not able to cover any changed region for six 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 five of these versions (4 and 5), default search strategy was not able to execute the changed region even in 10,000 runs, while `eXpress` executes the changed 4 regions and infect the program state for all of these versions. `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 hours of exploration) the time taken to find irrelevant branches is comparatively less.

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. Column N_{Pex} shows the number of DSE runs required by Pex to cover all the blocks in all the changed regions. Column Np_{seed} shows the number of DSE runs required by Pex to cover all the blocks in all the changed regions by using our approach of seeding the exploration with the existing test suite. Column N_e shows the number of DSE runs required by Pex to cover all the blocks in all the changed regions. Column Ne_{seed} shows the number of DSE runs required by `eXpress` to

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

C	V	N_{Pex}	Np_{seed}	$N_{eXpress}$	Ne_{seed}
SP	2-5	-	-	X	X
SP	37-39	1355	60	X	X
SP	39-40	-	304	X	X
SP	45-47	-	-	X	X
SP	47-50	-	81	X	X
SP	62-124	-	59	X	X
SL	169-174	34	18	X	X
SL	150-169	57	41	X	X
SL	9-139	-	69	X	X

cover all the blocks in all the changed regions by using our approach of seeding the exploration with the existing test suite. For nine of the version pairs of `structorian` (out of 19) that we used in our experiments, the existing test suite of `structorian` could not cover all the blocks of changed regions. We consider these nine version pairs for our experiments. Pex could not cover all the branches in changed regions for six of these version pairs in 10,000 runs. Seeding the program execution with the existing test suite, helps Pex in covering all the branches in under 100 runs for four of these version pairs under test. There is a considerable reduction of runs in the other version pairs with the exception of versions $2-5$ in which seeding cannot help Pex in covering all the blocks in changed regions.

In summary, our evaluation of `eXpress` answers the following questions that we mentioned at the beginning of this section:

RQ1. On average, `eXpress` requires 51.6% fewer runs (i.e., explored paths) on average than the existing search strategy in Pex to execute the changed regions of the 51 versions (in total) of our three subjects. For the fourth subject, `eXpress` was able to execute the changed regions of five versions that cannot be executed by default search strategy in `Pex`

RQ2. On average, `eXpress` requires 45% fewer runs on average than the existing search strategy in Pex to infect the program states after the execution of changed regions of the 51 versions (in total) of our three subjects. For the fourth subject, `eXpress` was able to infect the program state for five versions for which the program state could not be infected by thed efault search strategy in `Pex` .

RQ3. `eXpress` generates 121.1% more tests that execute the changed regions than the default search strategy in Pex.

RQ4. `eXpress` generates 170.2% more tests that execute the changed regions than the default search strategy in Pex.

RQ5. Seeding the program exploration with the existing suite helps reduce the DSE runs to cover all the blocks in all the changed regions by **

6. RELATED WORK

Previous approaches [9, 21] 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 P, I, or E in the PIE model. In contrast, we have developed a new search strategy for DSE to avoid exploring these irrelevant paths.

Santelices et al. [2, 19] 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 effectively achieve high structural coverage in a software system under test. However, these techniques do not specifically target to cover 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 P or I of the PIE model [25].

Differential symbolic execution [16] 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 as 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.

Our previous Orstra approach [27] automatically augments an automatically generated test suite with extra assertions for guarding against regression faults. Orstra first runs the given test suite and collects the return values and receiver-object states after the execution of the methods under test. Based on the collected information, Orstra synthesizes and inserts new assertions in the test suite for asserting against the collected method-return values and receiver object states. However, this approach observes the behavior of the original version to insert assertions in the test suite generated for only the original version. Therefore, the test suite might not include test inputs for which the behavior of a new version differs from the original version.

Li [14] 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 [12] 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 [17]. Chianti uses a test suite to produce an execution trace for two versions of a software system, 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 aim to exercise behavioral differences between the two versions of the software system under test. In contrast, the goal of our approach is specifically to expose behavioral differences across versions.

Some existing capture and replay techniques [8, 15, 18] 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 to specifically find behavioral differences.

7. DISCUSSION

Added/Deleted and Refactored Methods. If a method M is added or deleted from the original program version, `eXpress` does not detect M as a changed region. The change is detected if a method call site is added or deleted from the original program version. If the added or deleted method is never invoked, the behavior of the two versions is the same unless M is an overriding method. We plan to incorporate support for 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 method signature was changed along with other changes.

Granularity of Changed Region. In our current approach, 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 choosse from different levels of granularity.

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

Branch Prioritization. `eXpress` currently prunes branches that cannot help in detecting behavioral differences between the two versions. However, some branches in the program code can be more promising in detecting behavioral differences than others. Branching nodes can be prioritized based on the distance of the 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 representing the first instruction in Δ. The intuition behind this prioritization is that shorter the distance between the branching node and δ, it is likely to be easier to generate inputs to cause the execution of the changed region δ. This kind of branch prioritization is used by Burnim and Sen [3]. 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 [28, 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 b_i are the branching nodes in the Path P, while b_χ is the last instance of branching node containing χ. We do not flip the branching nodes from b_χ to b_n in P until if the program state is not propagated after the execution of χ.

Changes in Fields. Currently, `eXpress` does not detect changes in program code that is outside method bodies. For example, if the declaration of a field f is modified, `eXpress` cannot help in reducing DSE runs to detect behavioral differences that may be introduced in the program due to the change. In such situations, the source code can be searched to find the references of f. The corresponding instructions for all these statements referring to f can be considered as changed. If a field is added or deleted, `eXpress` can still be helpful in reducing DSE runs as in the case of added or deleted methods as discussed earlier in this section.

8. CONCLUSION

Regression testing aims at generating tests that detect behavioral differences between two versions of a software program. Dynamic symbolic execution (DSE) can be used to generate such difference exposing tests. 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 detecting behavioral differences 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 behavioral differences such that behavioral differences are more likely to be detected earlier in path exploration. Experimental results on various versions of programs showed that our approach can efficiently detect behavioral differences than without using our approach.

Acknowledgments

This work is supported in part by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443.

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

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

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

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

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

[17] 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.

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

[19] 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.

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

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

[22] 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.

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

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

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

[26] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.

[27] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.

[28] T. Xie and D. Notkin. Checking inside the black box: Regression testing based on value spectra differences. In *Proc. ICSM*, pages 28–37, 2004.

[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.