eXpress: Guided Path Exploration for Efficient Regression Test Generation

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

ABSTRACT

Software programs evolve throughout their lifetime undergoing various changes. While making these changes, software developers may introduce regression faults. It is desirable to detect these faults as quickly as possible to reduce the cost involved in fixing them. One existing solution is continuous testing, which runs an existing test suite to quickly find regression faults as soon as code changes are saved. However, the effectiveness of continuous testing depends on the capability of the existing test suite for finding behavioral differences across versions.

To address the issue, we propose an approach, called eXpress, that conducts efficient regression test generation based on a path-exploration-based test generation (PBTG) technique, such as dynamic symbolic execution. eXpress prunes various irrelevant paths with respect to detecting behavioral differences to optimize the search strategy of a PBTG technique. As a result, the PBTG technique focuses its efforts on regression test generation. In addition, eXpress leverages the existing test suite (if available) for the original version to efficiently execute the changed code regions of the program and infect program states. Experimental results on 67 versions (in total) of four programs (two from the subject infrastructure repository and two from real-world open source projects) show that, using eXpress, a state-of-the-art PBTG technique, called Pex, requires about 36% less amount of time (on average) to detect behavioral differences than without using eXpress. In addition, Pex using eXpress detects four behavioral differences that could not be detected without using eXpress (within a time bound). Furthermore, Pex requires 67% less amount of time to find behavioral differences by exploiting an existing test suite than exploration without using the test suite.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, Testing tools*

General Terms

Verification, Reliability

Keywords

Test Generation, Regression Testing

ISSTA '11, July 17 - 21, 2011, Toronto, ON, Canada

1. INTRODUCTION

Software programs continue to evolve throughout their lifetime undergoing various kinds of changes. While making changes to a program, software developers may introduce regression faults in the program. It is highly desirable to detect these regression faults as quickly as possible to reduce the cost of developers in fixing the introduced faults. Continuous testing [15] tests a program as soon as developers make changes to the program and these changes are compilable. To detect regression faults quickly, existing continuous testing techniques [15] execute an existing test suite as soon as the changes are saved in an editor. The tests that fail on the modified program version (and pass on the original version) expose behavioral differences[1] between the two versions. Developers can inspect these behavioral differences to determine whether they are intended or unintended (i.e., regression faults). However, the effectiveness of existing continuous testing techniques depends on the capability of the existing test suite in detecting behavioral differences between the original and the new program versions.

The existing test suite might not be able to detect behavioral differences as it is usually created (or generated) without taking into consideration the changes to be made in the future. Then the existing test suite can be augmented using existing test generation techniques to improve the capability of the test suite in terms of detecting behavioral differences. Existing test generation techniques such as path-exploration-based test generation (PBTG) [17, 8, 12, 2, 3, 23, 9] and search-based test generation [22, 10] focus their efforts on increasing structural coverage and do not specifically focus on detecting behavioral differences between two versions of a program. As a result, these techniques are ineffective and inefficient for regression test generation, even with increasing computing power thanks to multi-core architectures and cloud computing.

To address the issue, we propose an approach called eXpress[2] for efficient regression test generation with PBTG techniques. PBTG techniques (such as dynamic symbolic execution [21, 27] and concolic test generation [8, 17]) are gaining popularity due to their effectiveness in generating a test suite that achieves high structural coverage. To achieve high structural coverage, PBTG techniques try to explore all feasible paths in the program under test, and such exploration is typically quite expensive. However, if our aim is to detect behavioral differences between two versions of a program,

[1]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.

[2]An earlier version [20] of this work is described in a four-page paper that appears in the NIER track of ICSE 2009. This work significantly extends the previous work in the following major ways. First, we develop techniques for exploiting the existing test suite for efficiently generating regression tests. Second, we develop more prioritization techniques. Third, we automate our approach by developing a tool. Fourth, we conduct extensive experiments to evaluate our approach.

we do not need to explore all these paths in the program since some of these paths are irrelevant paths, i.e., paths whose executions can never detect any behavioral differences. These irrelevant paths need not be explored to make regression test generation efficient. eXpress prunes out these irrelevant paths from the exploration space of a PBTG technique. In general, our path pruning can be used to optimize both path-oriented (such as PBTG) as well as goal-oriented test generation techniques (such as eToc [22]) for regression test generation. Goal-oriented techniques generate tests to execute a goal (such as a branch). For regression test generation, the goal can be the changes made to a program. Our approach can optimize goal-oriented techniques by pruning irrelevant branches (i.e., branches whose executions can never detect any behavioral differences) from the list of branches that may lead to the changes (and propagate the change effects).

eXpress includes a novel practical application of a theoretical fault model: the Propagation, Infection, and Execution (PIE) model [24] 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. Our key insight is that execution of many paths in a program guarantees not to satisfy any of the conditions E, I, or P of the PIE model. These paths can be pruned out from the exploration space of a PBST technique, directing its efforts towards regression test generation. In particular, eXpress first determines a set of paths ($P_{\neg E}$) that cannot lead to any changed code region and a set of paths ($P_{\neg P}$) through which a state infection cannot propagate to any observable output. eXpress then prunes paths $P_{\neg E}$ and $P_{\neg P}$ (from the exploration space of a PBST technique). In addition, eXpress prunes other irrelevant paths ($P_{\neg I}$) that are determined during exploration with a PBST technique (see Section 4); these paths do not cause state infection immediately after the execution of any changed code region. Our technique for pruning paths $P_{\neg E}$ can be used in general to achieve new code coverage (or violate assertions) by treating not-covered locations (or assertions) as changed code regions, while the pruning of paths $P_{\neg P}$ and $P_{\neg I}$ is specific for regression test generation.

There are two technical challenges that our approach addresses. First, to find paths $P_{\neg E}$ and $P_{\neg P}$ (before path exploration is started), one needs to build an inter-procedural control-flow graph (CFG), and often the construction of an inter-procedural CFG is not scalable for real-world programs. To address the preceding challenge, we build a minimal inter-procedural CFG for which our purpose of finding sets $P_{\neg E}$ and $P_{\neg P}$ can still be served. Second, to determine $P_{\neg I}$ (during path exploration), we need to determine whether the program state is infected by a generated test. A PBST technique could be modified to simultaneously explore both the program versions to determine whether the program state is infected by a generated test, but realizing such simultaneous exploration can be challenging. To address the preceding challenge, eXpress explores only the new program version and executes a generated test (that executes a changed code region) on the original program version to determine whether the program state is infected by the generated test.

We have implemented eXpress as a search strategy for Dynamic Symbolic Execution (DSE) [8, 17], a state-of-the-art PBTG technique. In particular, our implementation guides DSE to avoid from flipping branching nodes[3], whose unexplored side is guaranteed to lead to an irrelavant path[4]). In addition, eXpress can exploit the existing test suite (if available) for the original version by seeding the tests in the test suite to further optimize exploration. Our seeding technique efficiently augments an existing test suite so that various changed code regions of the program (that are not covered by the existing test suite) are covered by the augmented test suite. As a result, behavioral differences are likely to be found earlier in path exploration.

This paper makes the following major contributions:

Path Exploration for Efficient Regression Test Generation. We propose an approach called eXpress for efficient generation of regression tests. To optimize the search strategy of a PBTG technique, eXpress prunes paths whose execution guarantees not to satisfy condition E, I, or P. As a result, behavioral differences are found efficiently by the PBTG technique with eXpress than without eXpress.

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 the scratch. As a result, behavioral differences are found more efficiently by the PBTG technique with eXpress based on an existing test suite than starting from the scratch.

Implementation. We have implemented our eXpress approach in a tool as an extension for Pex [21], an automated structural testing tool for .NET developed at Microsoft Research.

Evaluation. We have conducted experiments on 67 versions (in total) of four programs with two from the Subject Infrastructure Repository (SIR) [5] and two from real-world open source projects. Experimental results show that Pex using eXpress requires about 36% less amount of time (on average) to detect behavioral differences than without using eXpress. In addition, Pex using eXpress detects four behavioral difference that could not be detected without using eXpress (within a time bound). Furthermore, Pex requires 67% less amount of time to find behavioral differences by exploiting an existing test suite than exploration without using the test suite.

2. DYNAMIC SYMBOLIC EXECUTION

We implement our approach for Pex [21], an engine for Dynamic Symbolic Execution (DSE), a state-of-the-art PBST technique. Pex starts path exploration with some default inputs. Pex then collects constraints on program inputs from the predicates at the conditional statements executed in the program. We refer to these constraints at conditional 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 condition. Pex (and other PBST techniques) keeps track of the previous explored paths to build an execution tree. Each node in the tree is an instance of some conditional statement in the source code, each edge in the tree is an instance of some branch in the program source code (or its CFG), and different paths in the tree are (already explored) execution paths. The nodes of the tree are referred to as branching nodes. Pex, in the subsequent run[5], chooses one of the branching nodes in the execution tree (explored thus far), such that not all outgoing branches of the node have been explored yet. Pex flips the chosen branching node to generate a new input whose execution follows a new path. Intuitively, flip-

[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 to the false (or true) branch. Flipping a branching node for a switch statement is flipping the execution of the current branch to another unexplored branch.

[4] An irrelevant path here is a path whose execution guarantees not to satisfy any of the E, I, and P of the PIE model.

[5] A run is an exploration iteration.

ping a branching node in an old path is to construct a new path that shares the prefix (in the execution tree) to the node with an old path (containing the flipped node), but then deviates and takes a different branch of the node. Pex uses various heuristics [27] for choosing a branching node (to flip next) applying various search strategies with an objective of achieving high code coverage fast. Hence, the path exploration in PBST techniques (including DSE) is realized through flipping branching nodes.

We next present definitions of some terms that we use in the rest of this paper.

Instance of a Conditional Statement. Multiple instances of a conditional statement in the source code (of a program) can be present in the execution tree. The branching nodes in the execution tree corresponding to a conditional statement s in the source code are referred to as instances of s. Note that statements other than conditional statements are abstracted away from the execution tree.

Branch. A branch $<s_i, s_j>$ in the source code (or its CFG) is an edge connecting conditional statement s_i to statement s_j in the CFG. A conditional statement typically has two branches (or more than two branches for a `switch` statement): the true branch and the false branch.

Branch Instance. Let $<s_i, s_j>$ be a branch in the source code (or its CFG) from statement s_i to statement s_j such that s_i is a conditional statement. Let s_k be the first conditional statement that is encountered (in the CFG) after taking branch $<s_i, s_j>$. Note that if s_j is a conditional statement, $s_k = s_j$. A branch $br_{ik} = <b_i, b_k>$ in the execution tree[6] of a program is an instance of $<s_i, s_j>$ iff branching node b_i is an instance of conditional statement s_i and branching node b_k is an instance of conditional statement s_k.

Unexplored Branch Instance. An unexplored branch instance of a branching node b_i in the execution tree of a program is a branch instance $br_{ik} = <b_i, b_k>$ of b_i such that the branch instance br_{ik} is not taken yet.

Discovered Node. A discovered branching node (in short as a discovered node) is a branching node that is explored in the current DSE run but whose corresponding conditional statement in the source code was not executed in previous runs.

Path. A path P in the execution tree of a program is a list of branching nodes $P = <b_1, b_2, b_3, .., b_n>$ in the execution tree such that the branching nodes $b_1, b_2, b_3, ...b_n$ are executed in order.

Unexplored Branch Instances along a Path. Let B be the set of all branches of all the branching nodes in path $P = <b_1, b_2, b_3, .., b_n>$. Unexplored branch instances along path P are all those branch instances $B_u \in B$ such that $\forall br_{ik} = <b_i, b_k> \in B_u$, br_{ik} is unexplored.

Path Prefix. Two paths P_1 and P_2 have a common prefix up to a branching node b_i iff the the two lists P_1 and P_2 have a maximum prefix ending with branching node b_i.

Branch Pruning. Let b_i be a branching node (an instance of an `if` or a `while`/`for` statement) such that at least one of the branch instances $br_{ik} = <b_i, b_k>$ of b_i is unexplored. Pruning of branch instance br_{ik} is the removal of b_i from the exploration space of a PBST technique. As a result, b_i is prevented from being flipped by the PBST technique. The effect is pruning all such paths that share the prefix up to branching node b_i, and take an unexplored branch instance of branching node b_i.

Branch Pruning along a Path. Pruning a branch $<s_i, s_j>$ (in the source code) along path $P = <b_1, b_2, b_3, .., b_n>$ is the pruning of all the instances of branch $<s_i, s_j>$ from P.

[6] A branch in the execution tree is an edge in the execution tree. Every edge in the execution tree is an instance of a branch in the source code. In the rest of this paper, we refer to a branch in the execution tree as a branch instance.

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

Figure 1: An example program

Changed Code Region. A changed code region (in a new program version) is a minimal set of statements S that contains all modified (in the new or original program version), added (in the new program version), or deleted (in the original program version) statements in a method such that the nodes corresponding to S in the CFG of the new program version form a single-entry-single-exit subgraph[7].

3. EXAMPLE

In this section, we illustrate our 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 the program. Although eXpress analyzes assembly code of C# programs, in this section, we illustrate the eXpress approach using program source code.

To make a PBST technique efficient, eXpress prunes paths from the exploration space of the PBST technique so that the PBST technique focuses its efforts on regression test generation. To prune various paths from the exploration space of the PBST technique, eXpress determines certain branches (referred to as irrelevant branches) before the path exploration by the PBST technique. eXpress then uses these irrelevant branches to prune irrelevant paths (during path exploration) from the exploration space of the PBST technique. To explain the path pruning by eXpress, we use DSE as a representative PBST technique.

Consider an example program `TestMe` in Figure 1. Lines 10 and 11 of the program are added in a new version.

Detection of Irrelevant Branches. The left hand 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 black vertices denote the newly added statements at Lines 10 and 11. The gray vertices denote the conditional nodes (for the conditional statements in the program), while the white vertices denote the other statements in the program. From the CFG, eXpress first determines two categories of branches B_E and B_P. eXpress then uses B_E and B_P to prune irrelevant paths during path exploration.

- **Category B_E.** On statically traversing the CFG in Figure 2, eXpress detects that taking the branches $<2, 3>$, $<4, 16>$,

[7] The requirement of single-entry-single-exit subgraph ensures that instrumented code (see Section 4.3), inserted just after a changed code region δ, post-dominates δ.

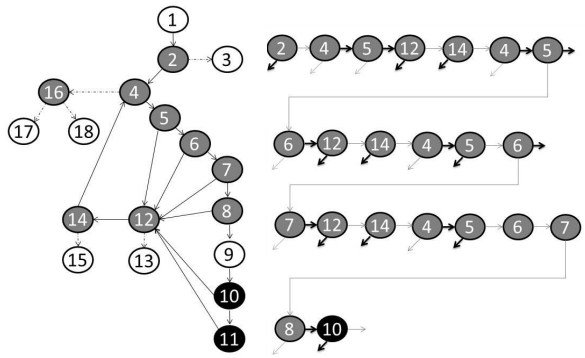

Figure 2: The left side shows the CFG for the program in Figure 1, while the right side shows a part of the execution tree of the program for test I = {"[", "{", "<", "*" }.

$<16, 17>$, $<16, 18>$, $<12, 13>$, and $<14, 15>$ (dotted edges in Figure 2), the program execution cannot reach any of the black vertices. Hence, the execution of these branches guarantees not to execute the changed statements.

- **Category B_P.** In addition, eXpress statically detects that among the source vertices of the six branches in Category B_E, there is no path from any of any black vertices to vertex 3. Hence, a state infection after the execution of any black vertex cannot propagate through branch $< 2, 3 >$.

Path Pruning. To cover the changed statements at Lines 10 and 11 (in Figure 1) and detect behavioral differences, DSE needs at least 6 DSE runs (starting from an empty input array c). However, the number of runs depends on the choice of the branching node that DSE flips in each run. In each run, DSE has the choice of flipping 8 new branching nodes (apart from the branching nodes that accumulate in previous runs) in the program. For `TestMe`, Pex takes 441 DSE runs to cover the true branch of the statement at Line 10. The number of runs can be much more if the number of branching statements in the program increases. Since path exploration is realized in DSE through flipping of branching nodes, eXpress dynamically prunes certain instances of branches in the execution tree. As an effect, eXpress prunes paths containing the instances of the branches.

- **Category $P_{\neg P}$.** eXpress prunes all instances of branches B_P from the exploration space of DSE since executing these branches guarantees not to execute a changed code region or propagate a state infection to an observable output. As an effect, all the paths containing instance of any node in B_P are pruned from the exploration space.
- **Category $P_{\neg E}$.** If along an already explored path no black vertex is executed, all the instances of branches in $B_E - B_P$ along the path are pruned. Note that category $P_{\neg P}$ already includes the paths containing B_P. Hence, to make the three categories exclusive, we prune $B_E - B_P$ instead of B_E. For example, if a test I = {"["} is generated that follows a path P_I =<..., 5, 12, 14, 4, 16, 18> and does not execute any black vertex, the unexplored branch instances $<12, 13>$, $<14, 15>$, and $<16, 17>$ (along the path P_I) are pruned since exploring these branch instances guarantees not to execute the black vertices along the path prefix shared with P_I.
- **Category $P_{\neg I}$.** If along an already explored path, some black vertex is executed but the program state is not infected after the execution of the black vertex, all the instances of branches in B_E after the execution of the last black vertex in the path are pruned. For example, if a test I = {"[", "{", "<", "*" } is generated to follow a path P_I =<..., 10, 12, 14, 4, 16, 18> is explored, the unexplored branch instances $<12, 13>$, $<14, 15>$, and $<16, 17>$ (along P_I) are pruned since the program state is not infected and exploring these branch instances guarantees not to execute the black vertices along the path prefix shared with P_I.

Incremental Exploration. eXpress can reuse an existing test suite for the original version so that changed code regions of the program can be explored efficiently due to which test generation is likely to find behavioral differences earlier in path exploration. Assume that there is an existing test suite covering all the statements in the original version of the program in Figure 1. Suppose that the test suite has a test I = {"[", "{", "<", "*" }. The input covers all the statements in the new version of `TestMe` except the newly added statement at Line 11. If we start the path exploration from scratch (i.e., with default inputs), Pex takes 441 runs to cover the statement at Line 11. 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 path exploration using the dynamic execution tree built by executing the existing test suite instead of starting from an empty tree. Some branching nodes in the tree may take many runs for Pex to discover if starting from an empty tree. The right side of Figure 2 shows a part of the execution tree for the input I. A gray edge in the tree indicates the false branch instance of a branching node while a black edge indicates the true branch instance. To generate an input for the next DSE runs, Pex flips a branching node b whose other side has not yet been explored and generates an input so that program execution takes the unexplored branch instance of b. Pex chooses such branching node for flipping using various heuristics for covering changed code regions of the program. It is likely that Pex chooses branching node 10 (colored black), which on execution covers the added statement at Line 11. Using our approach of seeding the tests from the existing test suite, Pex takes 39 runs (in contrast to 441) to flip the branching node and cover the statement at Line 11.

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[8](PUT). 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 eXpress approach.

4.1 Code-Difference Identification

eXpress analyzes the two versions $v1$ and $v2$ 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[9] of the method, Sig is the signature[10] 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. For each pair

[8]A parametarized unit test [21] is a test method with parameters. Such a method serves as a driver for path exploration.

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

[10]The signature of a method m is the combination of parameter types of m and the return type of m.

$<M_{i1}, M_{i2}>$ of corresponding methods, eXpress finds a set of differences Δ_{i1} (for the original version) and Δ_{i2} (for the new version) between the list of instructions $I_{M_{i1}}$ and $I_{M_{i2}}$ in the body of Methods M_{i1} and M_{i2}, respectively. Δ_{i1} (Δ_{i2}) includes all those instructions such that each instruction ι in Δ_{i1} (Δ_{i2}) is an instruction in $I_{M_{i1}}$ ($I_{M_{i2}}$ for Δ_{i2}), and ι is added (deleted for Δ_{i2}) or modified from list $I_{M_{i1}}$ to form $I_{M_{i2}}$. We denote the set of all added, modified, or deleted instructions in v_1 as Δ_1 and in v_2 as Δ_2. Currently our approach considers as different methods the methods that have undergone Rename Method or Change Method Signature refactoring. A refactoring detection tool [4] can be used to find such corresponding methods.

4.2 Graph Building

eXpress efficiently constructs the inter-procedural CFG $g <V, E>$ of the program under test such that each vertex $v \in V$ corresponds to an instruction $\iota \in M$ (denoted as $v \leftrightarrow \iota$), where M is some method in v_2. eXpress 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 eXpress to find branches (in the graph) via which the execution cannot reach any vertex containing a changed instruction in the graph.

Algorithm 1 Pseudo code of Construction of Inter-Procedural Control Flow Graph

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

```
InterProceduralCFG(τ)
1: g ← GenerateIntraProceduralCFG(τ)
2: MethodCallStack ← ∅, CanReachChangedRegion ← ∅, Visited ← ∅
3: ChangedMethods ← FindChangedMethods()
4: AChangedMethod ← m ∈ ChangedMethods
5: acg ← GenerateIntraProceduralCFG(AChangedMethod)
6: for all Vertex v ∈ g.Vertices do
7:     if v.Instruction = MethodInvocation then
8:         c ← getMethod(v.Instruction)
9:         if c ∈ MethodCallStack then
10:            goto Line 6 //To handle loops or recursions
11:        end if
12:        if c ∈ CanReachChangedRegion then
13:            g ← GraphUnion(acg, g, v)
14:            goto Line 6
15:        end if
16:        if c ∈ Visited then
17:            goto Line 6
18:        end if
19:        if c ∈ ChangedMethods then
20:            for all Method m ∈ MethodCallStack do
21:                CanReachChangedRegion.Add(m)
22:            end for
23:        end if
24:        MethodCallStack.Add(c)
25:        cg ← InterProceduralCFG(c)
26:        MethodCallStack.Remove(c)
27:        Visited.Add(c)
28:        g ← GraphUnion(cg, g, v)
29:    end if
30: end for
31: return g
```

Since a moderate-size program can contain a large number of method invocations (including those in its dependent libraries), often the construction of its inter-procedural CFG is not scalable to real-world situations. Hence, we build a minimal inter-procedural CFG for which our purpose of finding branches whose execution cannot later reach some changed code 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[11] (invoking method c) in g, the algorithm `InterProceduralCFG` is invoked recursively with the invoked method c as the argument (Line 25 of Algorithm 1), after adding c to the call stack (Line 24). After the control returns from the recursive call, the method c is removed from the call stack (Line 26) and added to the set of visited methods (Line 27). The inter-procedural graph `cg` (with c as an entry method) resulting from the recursive call at Line 25 is merged with the graph g (Line 28). The algorithm `InterProceduralCFG` is not invoked recursively with c as the argument in the following situations:

c **is in call stack.** If c is already in the call stack, `InterProceduralCFG` is not recursively invoked with c as the argument (Lines 9-10). 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 method A, then the construction of the inter-procedural graph stops after method A is encountered the second time.

c **is already visited.** If c is already visited, `InterProceduralCFG` is not recursively invoked with c as the argument (Lines 16-17). This technique ensures that we do not build the same subgraph again.

c **is in `CanReachChangedRegion`.** The set `CanReachChangedRegion` is populated whenever a changed method[12] is encountered. In particular, if a changed method is encountered, the methods currently in the call stack are added to the set `CanReachChangedRegion` (Lines 19-23). If c is in `CanReachChangedRegion`, `InterProceduralCFG` is not recursively invoked with c as the argument, while merging CFG of some changed method with g (Lines 12-15).

Note that if method `m` can invoke (directly or indirectly) a changed method `cm`, not all the branches in this method `m` may be able to reach[13] the changed region in `cm` (e.g., due to $return$ statements). Branches that cannot reach any changed code region (i.e., irrelevant branches) are found by eXpress. If a node b can (or cannot) reach a changed code region in inter-procedural CFG g built without using the preceding optimization, the node b can (or cannot) reach a changed code region (maybe a different one) in the graph built using the preceding optimizations. Since our aim of building the inter-procedural CFG is to find irrelevant branches, i.e., those in the graph via which the execution cannot reach any changed code 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 Code Instrumentation

The code instrumentation helps eXpress in determining whether the program state is infected by a generated test. For each changed method pair $<M_{i1}, M_{i2}>$ (i.e., $\Delta_{i1} \neq \emptyset$ or $\Delta_{i2} \neq \emptyset$) , eXpress finds changed code regions δ_{i1} and δ_{i2} (for the original and new program versions, respectively) containing all the changed instructions in the program. At the end of each changed code region δ_{i1} and δ_{i2}, eXpress inserts instructions to save the program state. In particular, eXpress finds the set of variables v_{di} that can potentially be defined in $\delta_i 1$ (and $\delta_i 2$). eXpress then inserts instructions to capture the value of each variable in v_{di} as an assertion (if the variable is of primitive type). If the variable is of a non-primitive type, eXpress captures the object state of the variable and linearizes the state

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

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

[13] A branch can reach a changed region if the edge (in the CFG) corresponding to the branch can reach the nodes corresponding to the changed region.

to a string. These observed values (or object states) are stored and inserted in assertions in a generated test to compare with the observed program state. A PBST technique is used to generate tests for the new version v_2. During path exploration, whenever a test is generated (for v_2) by a PBST technique to execute a changed code region, eXpress executes the generated test on the original program version (v_1) to determine whether the program state is infected immediately after the execution of the changed code region. If any of the captured values (or object states) is different across the two versions, an assertion fails for indicating program state infection. The instrumentation enables to perform only one instance of path exploration on the new version instead of performing two instances of path exploration: one on the original and the other on the new program version. Performing two instances of DSE can be technically challenging since the two DSE instances need to be performed 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.

4.4 Irrelavant-Branch Identification

There can be an infinite number of paths in the CFG of the program and many of them are infeasible. Hence, it is often not feasible to enumerate all irrelevant paths in the program thus far. Moreover, path exploration is realized through flipping branching nodes by PBST techniques. Hence, eXpress first finds branches whose corresponding branching nodes need not be flipped (under certain situations), and uses these branches for path pruning. In particular, eXpress traverses g to find a set of branches B_E (in the CFG) via which the execution cannot reach any of the instructions in Δ_2, and a set of branches B_P via which no state infection can propagate to any observable output. eXpress then uses these branches (and the already explored paths) to prune various paths (during path exploration) that need not be explored to find behavioral differences. A branch b in CFG g is an edge $e =<v_i, v_j>$: $e \in E$, $v_i \in V$ with an outgoing degree of more that one. We next describe the sets B_E and B_P.

Let $V = \{v_1, v_2, .., v_l\}$ be the set of all vertices in CFG $g<V, E>$ such that $v_i \in V$ and $v_i.degree > 1$. Let $E_i = \{e_{i1}, e_{i2}, ..., e_{im}\}$ be the set of outgoing edges (branches) from v_i. Let C be the set of vertices in the CFG g such that $\forall v \in C, \exists \iota \in \Delta_1 \bigcup \Delta_2 : v \leftrightarrow \iota$. $\rho(v_i, v_j, e_{ij})$ denotes the set of paths from a source vertex v_i to a destination vertex v_j such that these paths take the branch e_{ij} (if $\rho(v_i, v_j, e_{ij}) = \emptyset$, there is no such path from v_i to v_j), and $\rho(v_i, v_j)$ denotes the set of paths from a source vertex v_i to a destination vertex v_j (if $\rho(v_i, v_j) = \emptyset$, there is no such path from v_i to v_j).

Branches B_E. $B_E \subseteq E$ is the set of branches such that $\forall e_{ij} =< v_i, v_j >\in B_E \wedge \forall c_k \in C : \rho(v_i, c_k, e_{ij}) = \emptyset$. Since $\rho(v_i, c_k, e_{ij}) = \emptyset$, after taking a branch $e_{ij} \in B_E$, the program execution cannot reach a changed code region. Hence, the program state cannot be infected.

Branches B_P. $B_P \subseteq E$ is the set of branches such that $\forall e_{ij} =< v_i, v_j >\in B_P \wedge \forall c_k \in C : \rho(v_i, c_k, e_{ij}) = \emptyset \wedge \rho(c_k, v_i) = \emptyset$. Since $\rho(c_k, v_i) = \emptyset$, a state infection after a changed vertex c_k cannot reach v_i. Hence, the state infection cannot propagate through e_{ij}. Note that $B_P \subseteq B_E$.

4.5 Path Pruning

eXpress prunes various paths for a PBST technique to make path exploration efficient for regression test generation. During path exploration, eXpress uses the the set of branches B_E and B_P to determine paths that can be pruned from the exploration space of a PBST technique. These paths are guaranteed not to be able to detect behavioral differences between v_1 and v_2. We next describe the three categories of paths that eXpress prunes from the exploration space of a PBTG technique:

Category $P_{\neg P}$. Along all the paths already explored by a PBST technique, eXpress prunes all the instances of branches $b \in B_p$. As an effect, $P_{\neg P}$ contains all paths (in the program) that include an instance of some branch $b \in B_P$. Since all branches in B_P cannot reach a changed code region and no changed code region has a path to any branch in B_P, each path in $P_{\neg P}$ guarantees not to execute any of the changed code regions or propagate a state infection to an observable output. Hence, along these paths, no behavioral differences could be found. Note that the branches in $B_P - B_E$ may be able to propagate a state infection along some path in which a changed code region is executed. Hence, it is not safe to prune all paths including these branches.

Category $P_{\neg E}$. Let P_{ne} be the set of all paths (in the explored execution tree) that do not execute any changed code region, eXpress prunes all the instances of branches in $B_E - B_P$. Since along the paths P_{ne}, no changed code region is executed, the program state cannot be infected along these paths. Hence, it is safe to prune branches $B_E - B_P$ along these paths since these branches cannot reach a changed code region. As an effect, eXpress prunes all the paths that have a common prefix with some path in P_{ne} up to a branching node b_1 such that $b =<b1, b2>$ is an instance of branch br and $br \in B_E - B_P$, and b is not explored yet.

Category $P_{\neg I}$. Let P_{ni} be the set of all paths (in the explored execution tree) that execute some changed code region. However, the program sate is not infected after the execution of any changed code region. Along each path in P_{ne}, eXpress prunes the instances of branches B_E that are explored after the execution of the last changed code region. Since the state is not infected along any path in P_{ne} and the branches in B_E cannot reach any changed code region (again), it is safe to prune these branches.

4.6 Incremental Exploration

A regression test suite achieving high code coverage may be available along with the original version of a program. However, the existing test suite might not be able to cover all the changed code regions of the new version of the program. Our approach can reuse the existing test suite so that changed code regions of the program can be executed efficiently due to which test generation is likely to find behavioral 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 path exploration using the execution tree instead of starting from an empty tree. Our approach of seeding tests can help efficiently cover the changed code regions of the program with two major reasons:

Discovery of hard-to-discover branching nodes. By seeding the existing test suite for DSE 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 whose corresponding branches are closer in the CFG to the changed parts of the program has more likelihood of covering the changed code regions of the program [1]. Although our approach currently does not specifically first flip branching nodes whose corresponding branches are near the changed code regions, our approach can help these branching nodes to get discovered (by executing the existing test suite), which might take a 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 typically prioritize branching nodes for flipping so that high coverage can be achieved faster. Thus, DSE techniques choose a branching node from the execution tree (built thus far) such that flipping it has a high likelihood of covering changed code regions (that are not covered by the existing test suite for the original version). By seeding the existing test suite to path 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 the program's not-covered regions, which include the changed code regions. Hence, it is likely to cover the changed code regions earlier in path exploration.

5. EXPERIMENTS

We conduct experiments on four programs and their 67 versions (in total) collected from three different sources. In our experiments, we address the following research questions:

RQ1. How high percentage of paths explored by Pex belong to the three categories of irrelevant paths ($P_{\neg P}$, $P_{\neg E}$, and $P_{\neg I}$ as described in Section 4) being pruned by eXpress?

RQ2. How many fewer DSE runs and how much less amount of time does Pex using eXpress require to find behavioral differences than Pex without using eXpress?

RQ3. How many fewer DSE runs and how much less amount of time does Pex require to find behavioral differences when the path exploration is seeded with an existing test suite?

5.1 Subjects

To address 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) [5]. `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 choose these two subjects (among the others available at SIR) in our experiments as we could convert these subjects into C# using the Java 2 CSharp Translator[14]. We could not convert other subjects available at SIR (with the exception of `tcas`) because of extensive use of C or Java library APIs in these subjects. The experimental results on `tcas` are presented in a previous version of this work [20] and show similar conclusions as the results from the subjects used in the experiments here. We seed all the 32 faults available for `replace` at SIR one by one separately on the original version to synthesize 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 seed all the 9 faults available at SIR one by one separately on the original version 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. We capture the concrete value of the string `sub` at the end of the `PUT` using the `PexStore.ValueForValidation("v", v)` statement. This statement captures the current value of v in a particular run (i.e., an explored path) of DSE. In particular, this statement results in an assertion `Assert.AreEqual(v, cv)` in a generated test, where cv is the concrete value of v in the test during the time of exploration. This assertion is used to find behavioral differences when the tests generated for a new version are executed on the original version. 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. We capture the return values of these methods using the `PexStore` statement in the PUTs.

The method `encode` requires non-primitive arguments. Pex currently cannot handle non-primitive argument types effectively but provides support for using factory methods for non-primitive types. Hence, we manually write factory methods for the non-primitive types in `SENP`. In particular, we write 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 exploration, Pex generates concrete values for these inputs to cover various parts of the program under test.

STPG[15] is an open source program hosted by the codeplex website, which 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. We capture the return value of `Convert` using the `PexStore` statement in the PUT.

`structorian`[16] is an open source tool for binary-data viewing and reverse engineering. `structorian` is hosted by Google's open source project hosting website. The website also contains snapshots of check-ins in the code repositories for `structorian`. We collect all the versions of snapshots for the classes `StructLexer` 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 classes `StructLexer` and `StructParser`, we generalized some of the available concrete test methods by promoting primitive types to arguments of the test methods. Furthermore, we convert the assertions in the concrete test methods to `PexStore` statements. For example, if an assertion `Assert.IsEqual(v, 0)` exists in a concrete test, we convert the assertion to `PexStore.ValueForValidation("v", v)`. We use 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 RQ3.

To address questions RQ1-RQ2, we use all the four subjects, while to address question RQ3, 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` contain non-trivial changes that cannot be covered by the existing test suite. Hence, our technique of seeding the existing test suite in the path 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. Sim-

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

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

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

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	16	6561

ilarly, the changes in `siena` are covered by the existing test suite. Hence, our incremental exploration technique is not beneficial for the version pairs of `replace` or `siena` under test. `STPG` does 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 the available faults and released versions (if any) in SIR. We use eXpress and the default search strategy in Pex [21, 27] to conduct regression test generation. In addition to the versions synthesized by seeding faults, we also conduct regression test generation between each pair of successive versions of `siena` (versions 1.8 through 1.15) available in SIR, using eXpress and the default search strategy in Pex [21, 27]. For `STPG` and `structorian`, we conduct regression test generation between each pair of two successive versions that we collect.

To address RQ1, we categorize all the irrelevant paths explored by Pex (without using eXpress) as one of the three categories described in Section 4 and measure the percentage of paths in each category. To address RQ2, we compare the number of runs and the amount of time required by Pex with the number of runs required by Pex using eXpress (referred to as Pex+eXpress in the rest of this paper) to find behavioral differences between two versions of a program under test. To address RQ3, we compare the number of DSE runs and the amount of time required by Pex (and Pex+eXpress) to identify behavioral differences with and without seeding the path exploration (with the existing test suite).

Currently, we have not automated our code-instrumentation technique. We instrument each version manually for our experiments. In future work, we plan to automate the technique. The rest of the approach is fully automated and is implemented in a tool as an extension[17] to Pex [21]. We have developed its components to statically find irrelevant branches as a .NET Reflector[18] AddIn.

To find behavioral differences between two versions, we execute on the original version the tests generated for a new version. Behavioral differences are detected by a test if an assertion in the test fails.

5.3 Experimental Results

In this section, we present the experimental results to address the Research Questions RQ1, RQ2, and RQ3.

5.3.1 RQ1: Path Categorization

We next address RQ1 regarding the categorization of different irrelevant paths (described in Section 4) explored by Pex. Table 2 shows the categorization of paths explored by Pex. Column $Subject$ shows the subject name. Column #B_P shows the average number of branches in the set B_P. Column #B_E shows the average number of branches in the set B_E. Column #T shows the average number of branches in the CFG. Column P_1 shows the percentage of irrelevant paths (on average) in Category $P_{\neg P}$ among all the paths explored by Pex. Column P_2 shows the percentage of irrelevant paths (on average) in Category $P_{\neg E}$ among all the paths explored by Pex. Column P_3 shows the percentage of irrelevant paths (on average) in Category $P_{\neg I}$ among all the paths explored by Pex. Column $T(Irr)$ shows the percentage of irrelevant paths (on average) among all the paths explored by Pex. Note that the branch set $B_P \subseteq B_E$, while the path sets $P_{\neg P}$, $P_{\neg E}$, and $P_{\neg I}$ are disjoint.

Table 2: Categorization of the paths explored by Pex.

Subject	#B_P	#B_E	# T	P_1	P_2	P_3	$T(Irr)$
replace	4	84	206	23%	25%	8%	55%
siena	9	29	157	6%	18%	5%	29%
STPG	2	13	145	0%	12%	1%	13%
structorian (SL)	1	13	69	0%	11%	13%	24%
structorian (SP)	21	49	447	13%	28%	0%	41%

Table 3: Results of path pruning.

S	V	P_{Pex}	P_{Red}	M_p	T_{pPex}	Ts+ T_d	Tp_{Red}
replace	32	9812	63%	37%	711	305	57%
siena	17	6914	33%	11%	1011	718	29%
STPG	2	378	23%	23%	353	286	19%
SL	6	4326	37%	26%	144	98	31%
SP	10	49889	68%	77%	5hr	3.25hr	35%

The number of branches in B_P is substantially less than the number of branches in B_E. We observe that the number of branches in B_P is higher when a change is deeper inside the CFG, i.e., at a larger distance from the start node of the CFG. For example, the versions 5, 6, 8, 15, 16, and 24-27 of `replace` have a higher number of branches in B_P. As a result, the number of paths in Category $P_{\neg P}$ (P_1) is substantially higher (than the average) in these versions. In contrast, there are hardly any branches in B_P for the versions of STPG. Hence, there is no path in $P_{\neg P}$ (P_1) for these versions. The number of paths in $P_{\neg E}$ (P_2) is higher (on average) than the number of paths in $P_{\neg P}$ (P_1) as there are more branches in B_E. The number of paths in $P_{\neg I}$ (P_3) is smaller than in $P_{\neg P}$ (P_1) and $P_{\neg E}$ (P_2) as not many generated tests execute a changed code region such that the program state is not infected. We also observe that the number of irrelevant paths increases with the increase in the number of irrelevant branches. In total, 46% of the total paths explored by Pex are irrelavant paths. This percentage indicates the benefit that our path pruning techniques can potentially achieve in optimizing a PBST technique for regression test generation.

5.3.2 RQ2: Path Pruning

Table 3 shows the experimental results of applying our path pruning techniques. Due to space limit, we provide only the total, average, and median metric values of the versions for which behavioral differences were found by both Pex and Pex+eXpress. The detailed results for experiments on all the versions of these subjects are available on our project web[19].

Column S shows the name of the subject. The class `StructLexer` is denoted by SL and the class `StructParser` is denoted by SP. Column V shows the number of version pairs. Column P_{Pex} shows the total number of DSE runs required by Pex for satisfying P for all version pairs. Column P_{Red} shows the average percentage reduction in the number of DSE runs by Pex+eXpress for satisfying P (i.e., finding behavioral differences). Column M_p shows the median percentage reduction in the number of DSE runs by Pex+eXpress for satisfying P. Column T_{pPex} shows the time (in seconds) taken by Pex for satisfying P. Column $T_s + T_d$ shows the time (in seconds) taken by Pex+eXpress for satisfying P. This time includes the time taken to statically identify irrelevant branches. Column T_{PRed} shows the average percentage reduction in amount of time taken by Pex+eXpress for satisfying P.

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

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

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

Results of replace. For the `replace` subject, among the 32 pairs of versions, the changed code regions cannot be executed for 4 of theses version pairs (version pairs 0-14, 0-18, 0-27, and 0-31, where 0 is the original version) by Pex or by Pex+eXpress in 1000 DSE runs. We do not include these version pairs while calculating the sum of DSE runs for satisfying I and E of the PIE model. For 3 of the version pairs (version pairs 0-12, 0-13, and 0-21), the changes are in the fields due to which there are no benefits of using Pex+eXpress. We exclude these 3 version pairs from the experimental results shown in Table 3. For 3 of the version pairs (version pairs 0-3, 0-22, and 0-32), a changed code region is executed but the program state is not infected (by Pex or Pex+eXpress) in the time bound of 5 minutes. In addition, for 3 of the version pairs, the state infection is not propagated to an observable output within the bound of 1000 DSE runs. We do not include these version pairs while calculating the sum of DSE runs for finding behavioral differences. We observe that, for `replace`, Pex+eXpress takes 63% fewer runs (median 37%) and 57% less amount of time in finding behavioral differences.

Results of siena. We observe that the behavioral differences between 7 of the version pairs of `siena` are found within 20 runs by Pex and Pex+eXpress. For these version pairs, there is no reduction in the number of runs. The reason for the preceding phenomenon is that changes in these version pairs are close to the start node in the CFG. Hence, these changes can be covered within a relatively small number of runs. In 2 of the version pairs, changed code regions are 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 to avoid the exception. Changes between 2 of the version pairs are refactorings due to which the program state is never infected. We observe that, for `siena`, Pex+eXpress finds behavioral differences in 33% fewer runs (median 11%) and 29% less amount of time than Pex.

Results of STPG. We observe that for the 2 version pairs of STPG, Pex+eXpress finds behavioral differences in 23% fewer runs (median 23%) and 19% less amount of time than Pex.

Results of structorian. For two versions of `StructLexer`, neither Pex nor Pex+eXpress is able to find behavioral differences. For the others, Pex+eXpress takes 37% fewer runs (median 26%) and 31% less amount of time to find behavioral differences. Neither Pex+eXpress nor Pex is able to find behavioral differences between all version pairs of class `StructParser` within 5 minutes (a bound that we use in our experiments for all subjects). For these version pairs, we increase the bound to 1 hour (or 10000 runs). Pex is not able to find behavioral differences for 8 version pairs even within 1 hour, while Pex+eXpress finds behavioral differences for 4 of these 8 version pairs. If Pex is unable to detect behavioral differences, for a version pair, within the bound of 1 hour, we use 1 hour (for the version pair) to calculate the total in column T_{pPex}. In addition, we use the number of runs as 10000 (the bound on the number of runs) to calculate the total in column P_{pex}. Changes between two version pairs (40-45 and 40-47) could not be covered by either Pex or Pex+eXpress. One of the changes (between version pairs 47-50) is a refactoring. For this version pair, the program state is infected but no behavioral differences are detected by either Pex or Pex+eXpress. In summary, for `structorian`, Pex+eXpress is able to detect behavioral differences for 4 of the version pairs that could not be detected by Pex. On average, Pex is able to find behavioral differences in 68% fewer runs (median 77%) and 35% less amount of time. The reduction in the number of runs is substantially larger than reduction in the amount of time due to non-trivial time taken by eXpress in identifying irrelevant branches.

Table 4: Results of seeding the existing test suite.

C	V	N_{Pex}/T	Np_{sd}/T	N_{eXp}/T	Ne_{sd}/T
SP	2-5	**10000/60***	**10000/60***	2381/35	181/17
SP	37-39	3699/26	60/1	851/22	47/11
SP	39-40	**10000/60***	304/2	**10000/60***	251/12
SP	45-47	**10000/60***	**10000/60***	**10000/60***	**10000/60***
SP	47-50	**10000/60***	81/1	**10000/60***	64/10
SP	62-124	**10000/60***	59/1	7228/58	41/10
SL	169-174	478/1	324/1	34/1	18/1
SL	150-169	299/1	37/1	52/1	29/1
SL	9-139	2988/2	69/1	1002/1	52/1
Tot		64476/330	20934/128	41568/309	10683/123

*If behavioral differences are not detected, we take the number of runs as 10000 (the maximum number of runs that we run our experiments with).

Overall for all the subjects, Pex is able to find behavioral differences in 62% fewer runs and 36% less amount of time.

5.3.3 RQ3: Incremental Exploration

Table 4 shows the results of using the existing test suite to seed the path exploration. Column C shows the class name. Column V shows the pair of version numbers. The next four columns show the number of runs and time taken by the four techniques: Pex, Pex with seeding, Pex+eXpress, and Pex+eXpress with seeding, respectively, for finding behavioral differences. Note that DSE runs required by our incremental exploration also include the seeded test runs. In Table 4, if none of the changed blocks is covered, we take the number of runs as 10000 (the maximum number of runs that we run our experiments with). For 9 of the 16 version pairs of `structorian` that we used in our experiments, the existing test suite of `structorian` could not find behavioral differences. Therefore, we consider these 9 version pairs for our experiments for RQ3. Pex could not find behavioral differences for 5 of the 9 version pairs in 10000 runs. Seeding the path exploration with the existing test suite helps Pex in finding behavioral differences for 3 of the 5 version pairs under test. Pex+eXpress could not find behavioral differences for 3 of the 9 version pairs in 10000 runs. Seeding the path exploration with the existing test suite helps Pex+eXpress in finding behavioral differences for 2 of these 3 version pairs under test.

In summary, Pex requires around 68% of the original runs and 67% less time (than time required by Pex without test seeding) and Pex+eXpress requires around 74% of the original runs and 70% less time (than time required by Pex+eXpress without test seeding). In terms of time, Pex with seeding marginally wins over Pex+eXpress with seeding due to time taken by Pex+eXpress in identifying irrelevant branches.

5.3.4 Summary

In summary, this section addresses research questions RQ1, RQ2, and RQ3.

RQ1. Among the total paths explored by Pex, 46% (on average) are irrelavant. 14%, 26%, and 6% of all the paths explored by Pex belong to $P_{\neg P}$, $P_{\neg E}$, and $P_{\neg I}$, respectively.

RQ2. Pex+eXpress requires 36% less amount of time (on average) to detect behavioral differences than without using eXpress.

RQ3. Pex with test seeding requires 67% less amount of time to find behavioral differences than Pex without test seeding.

6. 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, eX-

press does not identify M (or F) as a changed code region. The change is identified 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 (or accessed), 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 identify M as a changed code 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 identifies the method containing call sites of M as changed. In our experiments, we consider versions of `replace` in which a method signature is changed, and versions of `structorian` in which a method is renamed.

Granularity of Changed Code Region. In our current implementation, a changed code region is the list of continuous instructions that include all the changed instructions in a method. One method can have only a single changed code region. Hence, a changed code region can be as big as a method and as small as a single instruction. The granularity of a changed code 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.

Pruning of Branches for Propagation. In future work, we plan to prune more categories of branches whose execution guarantees not to satisfy Propagation (P). Consider that a changed code region is executed and the program state is infected after the execution of the changed code 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 [26] obtained by executing the test on both versions of the program. All the branching nodes after the execution of χ can be removed from the exploration space of a PBST technique.

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.

Factors Affecting Test Seeding. The effectiveness of our incremental exploration technique is dependent on the characteristics of the existing test suite. In future work, we plan to conduct more extensive experiments with test suites of different characteristics, as done by Xu et al. [29, 28].

Incremental Call Graph Analysis. In our current implementation, the static analysis to construct the inter-procedural CFG and identify irrelevant branches is done from the scratch. However, the static analysis can be applied incrementally [18] to amortize the cost of static analysis across versions to further reduce the cost of incremental exploration.

7. RELATED WORK

Previous approaches [6, 19, 11] generate regression unit tests achieving high structural coverage on both versions of the class under test. However, these approaches explore all the irrelevant paths, whose execution guarantees not to satisfy any of the conditions E, I, or P in the PIE model [24]. In contrast, we have developed a new search strategy for DSE to avoid exploring these irrelevant paths.

Some existing search strategies [1, 27] guide DSE to efficiently achieve high structural coverage in a program under test. However, these techniques do not specifically target covering a changed code region. In contrast, our approach guides DSE to avoid exploring paths whose execution guarantees not to satisfy any of the conditions E, I, or P of the PIE model.

Santelices et al. [16] 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. Differential symbolic execution [13] determines behavioral differences between two versions of a method (or a program) by comparing their symbolic summaries [7]. 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. Qi et al. [14] propose an approach for guided test generation for evolving programs. The approach guides path exploration towards executing a change and propagating state infection to an observable output. However, their approach cannot deal with multiple interacting changes in the program in contrast to our approach. In addition, our approach can prune some paths (belonging to $P_{\neg E}$ and $P_{\neg I}$) that are explored by their approach.

Our previous Orstra approach [25] 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 tests for which the behavior of a new version differs from the original version.

Xu and Rothermel [30] propose a directed test generation technique that uses the existing test suite to cover parts of the program not covered by the existing test suite. In particular, the approach first collects the set of branches B not covered by the existing test suite. To cover a branch $<s_i, s_j> \in B$, the approach selects all the tests T that cover statement s_i. For each test $t_i \in T$, the approach collects the path condition p_i of the path followed by t_i until the first instance of s_i, negates the predicate at the first instance of s_i from p_i to get path condition p'_i. The approach then generates a test that covers the branch $<s_i, s_j>$ by solving the path condition p'_i . However, if none of the preceding path conditions p'_i derived from the paths followed by the tests T is solvable, the approach cannot generate a test to cover the branch $<s_i, s_j>$, which can furthermore compromise the coverage of additional branches. In

contrast, our incremental exploration technique can still generate a test to cover such branches. In addition, Xu et al. [29, 28] propose a search-based test augmentation technique that seeds the existing test suite for test generation. All these techniques focus on satisfying condition E of the PIE model. In contrast, our approach helps in satisfying E, I, and P of the PIE model.

Our previous approach [19] instruments a program to add branches such that behavioral differences can be found effectively. However, a test generation tool needs to explore branches in both the original and new versions of the program to detect behavioral differences. In contrast, eXpress prunes irrelevant branches to find behavioral differences efficiently. The two approaches are complementary and can be combined for effective and efficient regression test generation.

8. CONCLUSION

Regression testing aims at generating tests that detect behavioral differences between two versions of a program. To expose behavioral differences, test execution needs to satisfy three conditions: Execution (E), Infection (I), and Propagation (P), as stated in the PIE model [24]. Path-exploration-base test generation (PBTG) techniques can be used to generate tests for satisfying these conditions. PBTG techniques explore paths in the program to achieve high structural coverage, and exploration of all these paths can often be expensive. However, the execution of many of these paths in the program guarantees not to satisfy any of the three conditions in any way. In this paper, we have presented an approach and its implementation called eXpress for efficient regression test generation. eXpress prunes paths or branches whose execution guarantees not to satisfy the E, I, or P 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 code regions (if not already covered by the test suite). Experimental results on various versions of programs have shown that our approach can efficiently find behavioral differences than without using our approach.

Acknowledgments

This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, an NCSU CACC grant, and ARO grant W911NF-08-1-0443.

9. REFERENCES

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

[2] 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.

[3] L. Clarke. A system to generate test data and symbolically execute programs. *TSE*, 2(3):215–222, 1976.

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

[5] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

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

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

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

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, pages 151–166, 2008.

[10] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

[11] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proc. ICST*, pages 137–146, 2010.

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

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

[14] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *Proc. ASE*, pages 397–406, 2010.

[15] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Proc. ISSRE*, pages 281–292, 2003.

[16] 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.

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

[18] A. L. Souter and L. L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proc. ICSM*, pages 682–691, 2001.

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

[20] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *Proc. ICSE, NIER*, pages 311–314, 2009.

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

[22] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[23] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java Pathfinder. In *Proc. ISSTA*, pages 97–107, 2004.

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

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

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

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

[28] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proc. GECCO*, pages 1365–1372, 2010.

[29] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proc. FSE*, pages 257–266, 2010.

[30] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Proc. APSEC*, pages 406–413, 2009.