eXpress: Guided Path Exploration for Regression Test Generation

Kunal Taneja[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2], and Wolfram Schulte[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC
{ktaneja,txie}@ncsu.edu
[2] Microsoft Research, One Microsoft Way, Redmond, WA
{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. DSE explores all feasible paths in the program but exploration of all these paths can often be expensive. However, if our aim is to detect behavioral differences between two versions of a program, we do not need to explore all these paths in the program, since not all these paths are relevant for detecting behavioral differences. In this paper, we propose an approach on guided path exploration that avoids exploring irrelevant paths in terms of detecting behavioral differences. In addition, our approach guides path exploration towards infecting the program state by instrumenting the program with additional branches such that execution of these branches implies state infection. Hence, behavioral differences are more likely to be detected earlier in path exploration. Furthermore, our approach leverages the existing test suite (if available) for the original version to efficiently execute the changed regions of the program and infect program states. Experimental results on 67 versions (in total) of four programs show that our approach requires about 36% fewer amount of time (on average) to detect behavioral differences than exploration without using our approach. In addition, our approach detects 6 behavioral difference that could not be detected by exploration without using our approach (within a time bound). Furthermore, our approach requires 67% less amount of time to find behavioral differences by exploiting an existing test suite than exploration without using the test suite.

1 Introduction

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

Regression test generation can be automated by using Dynamic Symbolic Execution (DSE) [7, 13], a state-of-the-art test generation technique, to generate a test suite

achieving high structural coverage. DSE explores paths in a program to achieve high structural coverage, and exploration of all these paths can often be expensive. However, if our aim is to detect behavioral differences between two versions of a program, we do not need to explore all these paths in the program since not all these paths are relevant for detecting behavioral differences.

To formally investigate irrelevant paths for exposing behavioral differences, we adopt the Propagation, Infection, and Execution (PIE) model [17] 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 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[3] `eXpress` and its implementation 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, I, and P 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[4]). In particular, our approach guides DSE to avoid from flipping branching nodes[5], which on flipping execute some irrelevant branch. In addition, to effectively achieve condition I of the PIE model, our approach instruments a program under test with additional branches, such that if any of these branches are executed, program state is infected. Furthermore, our approach can exploit the existing test suite (if available) for the original version by seeding the tests in the test suite to the program exploration. Our technique of seeding the exploration with the existing test suite efficiently augments an existing test suite so that various changed parts of the program (that are not covered by the existing test suite) are covered by the augmented test suite.

This paper makes the following major contributions:

Path Exploration for Regression Test Generation. We propose an approach called `eXpress` that uses DSE for efficient generation of regression tests. `eXpress` prunes paths that cannot help in achieving conditions E, I, and P. In addition, `eXpress` instruments the program such that condition I can be achieved effectively after condition E is achieved.

Incremental Exploration. We develop a technique for exploiting an existing test suite, so that path exploration focuses on covering the changes rather than starting from scratch.

---

[3] An earlier version [15] 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 techniques for efficiently achieving state infection after a change is executed. Third, we automate our approach by developing a tool. Fourth, we conduct extensive experiments to evaluate our approach.

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

[5] 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.

Implementation. We have implemented our eXpress approach in a tool as an extension for Pex [16], 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 faults [16]. The current Pex has been downloaded tens of thousands of times in industry.
Evaluation. We have conducted experiments on 67 versions (in total) of four programs. Experimental results show that our approach requires about 36% fewer amount of time (on average) to detect behavioral differences than exploration without using our approach. In addition, our approach detects 6 behavioral difference that could not be detected by exploration without using our approach (within a time bound). Furthermore, our approach 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

In our approach, we use Pex [16] as an engine for Dynamic Symbolic Execution (DSE). Pex starts program exploration with some default inputs. Pex then collects constraints on program inputs from the predicates at the branching statements executed in the program. We refer to these constraints at branching statements as branch conditions. The conjunction of all branch conditions in the path followed during execution of an input is referred to as a path constraint. Pex keeps track of the previous executions to build a dynamic execution tree. Pex, in the next run[6], chooses one of the unexplored branches in the execution tree (dynamically built thus far). Pex flips the chosen branching node to generate a new input that follows a new execution path. Pex uses various heuristics [18] for choosing a branching node (to flip next) using various search strategies with an objective of achieving high code coverage fast. We next present definitions of some terms that we use in the rest of this paper.
Discovered node. We refer to all the branching nodes that are explored in the current DSE run but were not explored in previous runs as discovered branching nodes (in short as discovered nodes).
Instance of a branching node. A branching node c_i in a Control Flow Graph (CFG) of a program can be present multiple times in the dynamic execution tree (of the program) due to loops in the program. We refer to these multiple branching nodes in the tree as instances of c_i.

3 Example

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

Consider the example in Figure 1. The left side of the figure shows a program TestMe. Lines 10 and 11 of the program are added in a new version.
Finding Irrelavant Branches. The left side of Figure 2 shows the CFG of the program in Figure 1. The labels of vertices in the CFG denote the corresponding line numbers in

[6] A run is an exploration iteration.

```
   static public int TestMe(char[] c, int n){
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)
11          state = state + n;//Added in new version
        }
12      if(c[i]==' ')
13        return;
14      if(!(c[i] >= 'a' && c[i] <= 'z'){
15        state=-1; return;
        }
      }
    }
16  if(c[15] == '}')
17    return state;
18  return -1;
  }
```

```
  static public int fOld(int state,
    int length, int n){
1   return state;
  }

  static public int fNew(int state,
    int length, int n){
1   if(c.Length==15) state = state + n;
2   return state;
  }

  static public int TestMe(char[] c, int n){
    ...
10  int stateCopy = state;
11  state = fNew(state, c.Length, n);
12  stateCopy = fOld(stateCopy, c.Length, n);
13  if(state != stateCopy);   //Infection Condition
    ...
  }
```

Fig. 1. An example program on the left side and the instrumented program on the right.

Figure 1. The black vertex denotes the newly added statements at Lines 10 and 11. The gray vertices denote the branching nodes (for the branching statements in the program), while the white vertices denote the other statements in the program. From the CFG, eXpress finds the following categories of branches.

- **Category B_{E+I}.** On traversing the CFG in Figure 2, eXpress detects that taking the branches $<2, 3>$[7], $<4, 16>$, $<16, 17>$, $<16, 18>$, $<12, 13>$ and $<14, 15>$ (dotted edges in Figure 2), the program execution cannot reach any of the black vertices. Hence, the execution of these branches cannot help in executing the changed statements or infecting the program state.

- **Category B_P.** In addition, eXpress finds out that among the source vertices of the six branches in Category B_{I+E}, there is no path from any of the black vertices to vertex 3. Hence, a state infection after the execution of the black vertex cannot propagate through the branch $< 2, 3 >$.

Instrumentation for State Infection. A DSE-based test generation tool tries to execute all feasible branches in the program. However the execution of all branches is not sufficient for infecting the program state. For example, a test generation tool can cover all the branches of TestMe (left side of Figure 1) with $n = 0$ (a default test input used by some tools) but to infect the program state, the tool need to generate a nonzero value of n (used at Line 11). To guide the test generation tool to effectively achieve state infection, eXpress instruments the new program version. In particular, eXpress invokes the changed regions[8] from the two program versions (with same inputs) and compares the outputs of the two changed regions. eXpress inserts branches in the new program version that are executed only when the outputs of the executions of the changed regions differ, i.e., when program state is infected after the execution of the changed regions.

[7] $<x, y>$ represents a branch from source vertex x to destination vertex y

[8] A changed region (in a new or modified program version) is a minimal set of statements that contains all modified (in new or original program version), added (in new program version), or deleted (in original program version) statements in a method. The changed region for the new program version in TestMe on the left side of Figure 1 is shown in rectangular box, while the changed region of the original program version is empty.

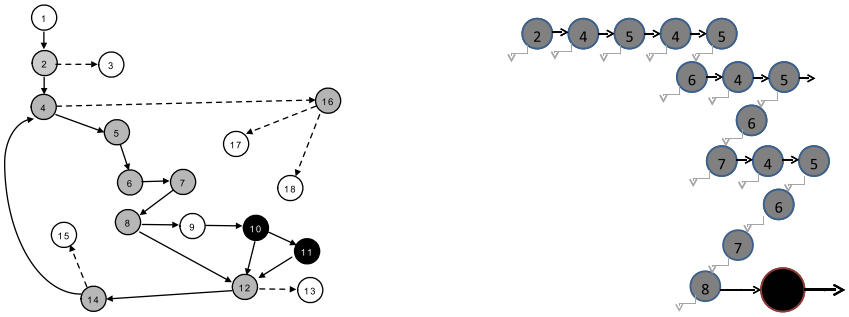

Fig. 2. The left side of the figure shows the part of execution tree of the program for c = {"[", "{", "<", "*" }, while the right side shows the Control-Flow Graph for the program in Figure 1.

The right side of Figure 1 shows the instrumented version of `TestMe` on the left side of Figure 1. `eXpress` identifies $state$, $length$, and n as the inputs to the two changed regions and $state$ as the output to the two changed regions. `eXpress` then extracts the changed regions in the original and new program versions as separate methods (`fOld` and `fNew`, respectively). The parameters of the two methods are same as the identified inputs, while the return values are same as the identified output. `eXpress` replaces the changed region of the new version of `TestMe` by the invocation of the methods `fOld` and `fNew` (Lines 11 and 12 on the right side of Figure 1). Note that the variable `stateCopy` passed to `fOld` is a copy of the variable $state$ passed to `fNew` (Line 10 on the right side of 1) because $state$ may be updated by `fNew` since $state$ is also an output of the changed regions. `eXpress` then inserts an `if` statement (Line 13 on the right side of Figure 1) in the program comparing the outputs of the two methods. Note that the outputs of the two methods are different if and only if the program state is infected. A DSE-based test generation tool determines that it needs to generate inputs satisfying `c.Length == 15 AND state != state + n` to cover the true branch of the added `if` statement. Hence, the tool generates a nonzero value of n to satisfy the constraint and infect the program state.

Test Generation. To cover the changed statements at Lines 10 and 11 (in the left part of Figure 1) and infect the program state, DSE needs at least 6 DSE runs (starting from an empty input array c). However, the number of runs depends on the choice of branches that DSE flips in each run. In each run, DSE has the choice of flipping 8 branches in the program. For `TestMe`, Pex takes 441 DSE runs to cover the true branch of statement at Line 10. The number of runs can be much more if the number of branching statements in the program increases. To reduce the search space of DSE, `eXpress` prunes branches in category B_{E+I} until the black vertex is executed and the program state is infected after its execution. Once the program state is infected, `eXpress` prunes out the branches in category B_P to efficiently propagate the state infection to an observable output.

Incremental Exploration. `eXpress` can reuse an existing test suite for the original version so that changed parts of the program can be explored efficiently due to which test generation is likely to find behavior differences earlier in path exploration. Assume that there is an existing test suite covering all the statements in the old version of the program in Figure 1. Suppose that the test suite has a test input I = {"[", "{", "<", "*" }. The input covers all the statements in the new version of `TestMe` except the newly

added statement at Line 11. If we start the program exploration from a 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 program exploration using the dynamic execution tree built by executing the existing test suite instead of starting from an empty tree. Some branches in the tree may take many runs for Pex to discover if starting from an empty tree. Figure 2 shows part of the execution tree for the input I. A gray edge in the tree indicates the false side of a branching node while a black (horizontal) edge indicates the true side. To generate an input for the next DSE runs, Pex flips a branching node b in the tree whose other side has not yet been explored and generates an input so that program execution takes the unexplored branch of b. Pex chooses such branching node for flipping using various heuristics for covering changed regions of the program. It is likely that Pex chooses the branching node 10 (colored black), which on execution covers the added statement at Line 11. When starting the program exploration from scratch, Pex takes 420 runs before it reaches (discovers) the black branching node in Figure 2. Using our approach of seeding the tests from the existing test suite, Pex takes 39 runs to flip the branching node and cover the statement at Line 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 (PUT); such a PUT serves as a test driver for path exploration. When an existing test suite is available for the original version, `eXpress` conducts incremental exploration that exploits the test suite for generating tests for the new version. We next discuss in detail the `eXpress` approach.

4.1 Finding Differences Between Versions

`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. Currently our approach considers as different methods the methods that have undergone Rename Method or Change Method Signature refactoring. A refactoring detection tool [3] 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 $I_{M_{i2}}$. We denote the set of all added, modified, or deleted instructions as Δ.

[9] 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.

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

4.2 Finding Irrelavant Branches.

eXpress efficiently constructs the inter-procedural CFG $g <V, E>$ of the new version 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 traverses the graph g to find a set of branches B_{E+I} via which the execution cannot reach any of the branches in V and a set of branches B_P via which a state infection cannot propagate to any observable output. 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. The vertex v_i is referred to as a branching node. We next describe the sets B_{E+I} 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 $\forall v \in C, \exists \iota \in \Delta : v \leftrightarrow \iota$. $\rho(v_i, v_j, e_{ij})$ denotes a path between a source vertex v_i to a destination vertex v_j that takes the branch e_{ij} (if $\rho(v_i, v_j, e_{ij}) = \phi$, there is no such path from v_i to v_j), $\rho(v_i, v_j)$ denotes a path from a source vertex v_i to a destination vertex v_j(if $\rho(v_i, v_j) = \phi$, there is no such path from v_i to v_j).

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

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

4.3 Code Instrumentation

For each changed method pair $<M_{i1}, M_{i2}>$ for which $\Delta_i \neq \phi$, eXpress finds changed regions δ_{i1} and δ_{i2} (for original and new program version, respectively) containing all the changed instructions in the program. A changed region 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. Inputs $I = \{i_1, i_2..., i_n\}$ of a changed region δ are variables (or fields) that are used in the changed region δ but are declared outside the changed region, while outputs $O = \{o_1, o_2..., o_m\}$ are variables (or fields) that are defined in δ and used outside the changed region. As an example, for the changed region in the new version of TestMe in Figure 1, $I = \{c.Length, n, state\}$ and $O = \{state\}$. Let I_1 and O_1 be the inputs and outputs of $\delta 1$, while I_2 and O_2 be the inputs and outputs of $\delta 2$, respectively.

eXpress extracts the instructions in δ_{i1} and δ_{i2} as methods $M_{\delta_{i1}}$ and $M_{\delta_{i2}}$ (shown in Figure 3), respectively, both having arguments of type and name as $I_1 \bigcup I_2$ and returning outputs $O_1 \bigcup O_2$. We refer to the arguments of $M_{\delta_{i1}}$ as $A_1 = i_{11}, i_{21}, ..., i_{n1}$ and the arguments of $M_{\delta_{i2}}$ as $A_2 = i_{12}, i_{22}, ...i_{n2}$, such that the types of i_{l1} and i_{l2} are same as the type of $i_l \in I_1 \bigcup I_2$. We refer to the variables corresponding to $o_i \in O_1 \bigcup O_2$ in $M_{\delta_{i1}}$ as o_{i1} and in $M_{\delta_{i2}}$ as o_{i2}. eXpress synthesizes a class C_o (show in Figure 3), such that for all $o_i \in O_1 \bigcup O_2$, there is a corresponding public field in C_o with types and names same as o_i. The methods $M_{\delta_{i1}}$ and $M_{\delta_{i2}}$ return an object c_o of type C_o. The class enables the two methods $M_{\delta_{i1}}$ and $M_{\delta_{i2}}$ to return multiple outputs

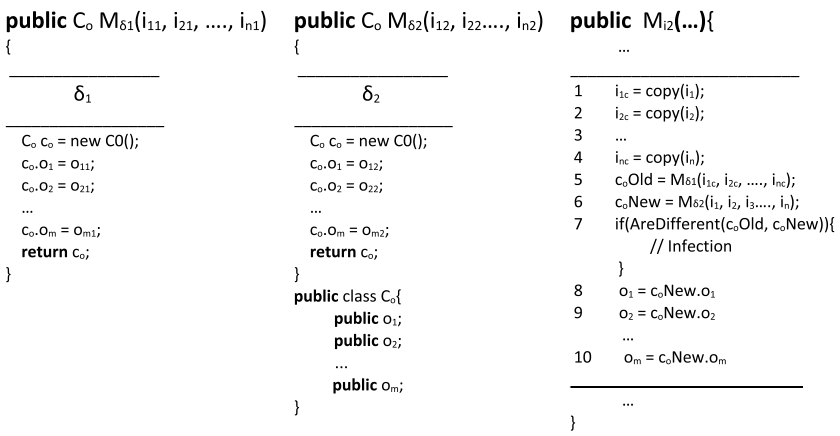

Fig. 3. Instrumented Code synthesized by eXpess

$O_1 \bigcup O_2$ as fields of C_o. As shown in Figure 3, the method $M_{\delta_{i1}}$ (and $M_{\delta_{i2}}$) first contains the instructions in δ_{i1} and later copies all the outputs o_{i1} in $M_{\delta_{i1}}$ (o_{i2} in $M_{\delta_{i2}}$) to the respective fields of c_o and returns the object c_o.

Figure 3 shows the instrumented part of the method M_{i2}. The methods $M_{\delta_{i1}}$ and $M_{\delta_{i2}}$ are invoked (Lines 5 and 6 of the Method M_{i2} in Figure 3) from the method M_{i2} instead of the changed region. For the method $M_{\delta_{i2}}$, $i_1, ...i_n$ are passed as arguments, while for the method $M_{\delta_{i1}}$ copied values $i_{1c}, ...i_{nc}$ of $i_1, ...i_n$ are passed as arguments (Lines 1 to 4). These copies are deep copies for non primitive type variables. The outputs of $M_{\delta_{i1}}$ and $M_{\delta_{i2}}$ are stored in objects c_oOld and c_oNew, respectively. An `if` statement is inserted (Line 7) such that its `true` branch is executed when the outputs c_oOld and c_oNew are different. To compare the two outputs, `eXpress` changes modifiers of all fields transitively reachable from objects of the two given classes to `public` and compares these fields directly to detect difference in the outputs. Finally, the fields of c_oNew are assigned to the corresponding output variables (Lines 8 to 10).

4.4 Test Generation

`eXpress` performs Dynamic Symbolic Execution (DSE) [7, 13] to generate regression tests for the two given versions of a program. DSE iteratively generates test inputs to cover various feasible paths in the program under test (the new version in our approach). In particular, DSE flips some branching node discovered in previous executions to generate a test input for covering a new path. To make path exploration efficient in finding behavioral differences, `eXpress` avoids from exploring the following categories of paths:

Category E+I. `eXpress` does not explore these paths until a test is generated that infects the program state. Our approach avoids exploring all the branches in B_{E+I}. In other words, if a branching node v_i is in the built dynamic execution tree[11], such that the branch $e_{ij} \in B_{E+I}$ is not explored yet, our approach does not flip v_i to make program execution take branch e_{ij}.

Category P. `eXpress` does not explore these paths after a test is generated that infects the program state. Our approach avoids exploring all the branches $e_{ij} \in B_p$. In other

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

words, if a branching node v_i is in the built dynamic execution tree such that one of its branch e_{ij} is not explored yet, our approach does not flip v_i to make program execution take branch e_{ij}.

4.5 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 regions of the new version of the program. Our approach can reuses the existing test suite so that changed regions of the program can be executed efficiently due to which test generation is likely to find behavior differences earlier in path exploration. Our approach executes the existing test suite to build an execution tree for the tests in the test suite. Our approach then starts the program exploration using the dynamic execution tree instead of starting from an empty tree. Our approach of seeding test inputs can help efficiently cover the changed regions of the program with two major reasons:
Discovery of hard-to-discover branching nodes. By seeding the existing test suite for Pex to start exploration with, our approach executes the test suite to build an execution tree of the program. Some of the branching nodes in the built execution tree may take a large number of DSE runs (without seeding any tests) to get discovered. Flipping some of these discovered branching nodes nearer in CFG to the changed parts of the program has more likelihood of covering the changed regions of the program [2]. Although, our approach currently does not specifically prioritize flipping of branching nodes near the changed regions, our approach can help these branching nodes to get discovered (by executing the existing test suite), which might take large number of DSE runs as shown in the example in Section 3.
Priority of DSE to cover not-covered regions of the program. DSE techniques employ branch prioritization so that a high coverage can be achieved faster due to which DSE techniques choose a branch from the execution tree (built thus far) that have a high likelihood of covering changed regions (that are not covered by existing test suite for the original version). By seeding the existing test suite to program exploration, the DSE techniques do not waste time on covering the regions of the program already covered by the existing test suite. Instead, the DSE techniques give high priority to branching nodes that can cover not-covered regions of the program, which include the changed parts. Hence, the changed parts are likely to be covered earlier in path exploration.

5 Experiments

We conducted experiments on four programs and their 67 versions (in total) collected from three different sources. In our experiments, we try to address the following research questions:
RQ1. How many fewer DSE runs and how much fewer amount of time does Pex require to find behavioral differences with the assistance of `eXpress`?
RQ2. How many fewer DSE runs does `eXpress` require to infect the program state using our code instrumentation technique than without using the technique?
RQ3. How many fewer DSE runs and how much fewer amount of time does Pex require to find behavior differences when the program exploration is seeded with the existing test suite?

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) [4]. `replace` and `siena` are written in C and $Java$, respectively. `replace` is a text-processing program, while `siena` is an Internet-scale event notification program. We chose these two subjects (among the others available at the SIR) in our experiments we could convert these subjects into C# using the Java 2 CSharp Translator[12]. We could not convert other subjects available at the 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 [15] and show similar conclusions as the results from the subjects used in the experiments here. We seeded all the 32 faults available for `replace` at the SIR one by one to generate 32 new versions of `replace`. For `siena`, SIR contains 8 different sequentially released versions of `siena` (versions 1.8 through 1.15). Each version provides enhanced functionalities or corrections with respect to the preceding version. We use these 8 versions in our experiments. In addition to these 8 versions, there are 9 seeded faults available at SIR. We seeded all the 9 faults available at SIR one by one to synthesize 9 new versions of `siena`. In total, we conduct experiments on these 17 versions of `siena`. For `replace`, we use the `main` method as a PUT for generating tests. We capture the concrete value of the string sub at the end of the PUT using `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.

STPG[13] and is an open source program hosted by the codeplex website. The codeplex website contains snapshots of check-ins in the code repositories for STPG. We collect three different versions of the subject STPG from the three most recent check-ins. We use the `Convert(string path)` method as the PUT for generating tests since `Convert` is the main conversion method that converts a string path data definition to a `PathGeometry` object. We capture the return value of `Convert` using the `PexStore` statement in the PUTs.

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

[12] `http://sourceforge.net/projects/j2cstranslator/`
[13] `http://stringtopathgeometry.codeplex.com/`
[14] `http://code.google.com/p/structorian/`

also contains snapshots of check-ins in the code repositories for `structorian`. We collected 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 the classes `StructLexer` and `StructParser` , we generalized one 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. Foe example if an assertion `Assert.IsEqual(v, 0)` exist 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 RQ2.

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 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. Hence, our Incremental Exploration technique is not beneficial for the version pairs under test. `siena` and `STPG` do not have an existing test suite to use.

Table 1. Experimental subjects

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

Experimental Setup. For `replace` and `siena`, we conduct regression test generation 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 [16, 18] to conduct regression test generation. In addition to the versions synthesized by seeding faults, we also conduct regression test generation between each successive versions of `siena` (versions 1.8 through 1.15) available in SIR, using `eXpress` and the default search strategy in Pex [16, 18]. For STPG and `structorian`, we conduct regression test generation between two successive version pairs that we collected.

To address RQ1, we compare the number of runs and the amount of time required by Pex with the number of runs required by Pex+eXpress to find behavioral differences between two versions of a program under test. To address RQ2, we compare the number of runs and the amount of time required by `eXpress` with the number of runs required by `eXpress` without our code instrumentation technique to infect the program state. To address RQ3, we compare the number of DSE runs and the amount of time required by Pex (and Pex+eXpress) to find behavioral differences with and without seeding the program exploration (with the existing test suite).

Currently, we have not automated our code-instrumentation technique. In future, we plan to automate the technique. The rest of the approach is fully automated and is

implemented in a tool as an extension[15] to Pex [16]. We developed its components to statically find irrelevant branches as a .NET Reflector[16] AddIn.

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

Table 2. Experimental results

S	V	P_{Pex}	$P_{Red}(\%)$	M_p	Tp_{Pex}	$Ts+ T_d$	$Tp_{Red}(\%)$	I_i	I_{wi}
replace	32	10312	75	49	711	235	67	2511	2235
siena	17	7301	42	15	1011	628	38		
STPG	2	378	32	32	353	255	28		
Total	51	17613	62		1722	863	50		
Total(SL)	6	4526	62	52	146.6	89.1	39		
Total(SP)	10	49889	68	77	5hr	3.25hr	35		

Experimental Results. Table 2 shows the experimental results. Due to space limit, we provide only the total, average, and median values. The detailed results for experiments on all the versions of these subjects are available on our project web[17].

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

We observe that, for replace, Pex+eXpress took 75% fewer runs (median 49%) and 67% fewer amount of time in finding behavioral differences. For siena, Pex+eXpress found behavioral differences in 42% fewer runs(median 15%) and 38% fewer amount of time than Pex. In addition, Pex+eXpress detected behavioral differences for two changes that were not detected by Pex within a bound of 5 minutes.

For two versions of `StructLexer`, neither Pex nor Pex+eXpress were able to find behavioral differences. For others, Pex+eXpress takes 62% fewer runs (median 52%) and 39% fewer amount of time to find behavioral differences.

Neither Pex+eXpress nor Pex was able to find behavioral differences between some version pairs of class `StructParser` in 5 minutes (a bound that we use in our experiments for all subjects). For these version pairs, we increased the bound to 1 hour (or 10000 runs). Pex was not able to find behavioral differences for 4 version pairs even in 1 hour, while Pex+eXpress found behavioral differences for all these version pairs. If Pex was unable to detect behavioral differences within the bound of 1 hour, we put the time in the column T_{pex} as 1 hour and the number of runs as 10000 (the bound on the number of runs) to calculate the total in the last row of Table 2.

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

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

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

Changes between two version pairs (40-45 and 40-47) could not be covered by either Pex nor $Pex + eXpress$. One of the changes (between version pairs 47-50) was a refactoring. For this version pair, program state was infected but no behavioral differences were detected by either Pex or $Pex + eXpress$.

In summary, for structorian, $Pex + eXpress$ was able to detect behavioral differences for four of the version pairs that could not be detected by Pex. On average, Pex was able to find behavioral differences in 68% fewer runs (median 77%) and 35% fewer amount of time. The reduction in number of runs is substantially larger than reduction in amount of time due to non-trivial time taken by $eXpress$ in finding irrelevant branches.

Code Instrumentation Technique. Column I_i in Table 2 shows the DSE runs taken to achieve state infection by `eXpress`, while Column I_{wi} shows the DSE runs taken to achieve state infection by `eXpress` without the code-instrumentation technique (referred to as $\texttt{eXpress}_{wi}$). For $replace$, `eXpress` infects the program state in 11% fewer DSE runs. In addition, the added branches in the code, help in infecting the program state for 3 versions of $replace$ for which $\texttt{eXpress}_{wi}$ did not infect the program state in a bound of 5 minutes. For 3 versions, $\texttt{eXpress}_{wi}$ takes more DSE runs to infect the program state. We suspect the preceding phenomenon is due to extra branches (in the changed region of original version) added to the source code.

Seeding program exploration with existing tests. Table 3 shows the results obtained by using the existing test suite to seed the program exploration. Column C shows the class name. Column V shows the pair of version numbers. The next four columns show the number of runs 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 includes the seeded test runs.

In Table 3, if all the changed blocks are not covered, we take the number of runs as 10,000 (the maximum number of runs that we ran our experiments with). For 9 of the version pairs of `structorian` (out of 16 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. Pex could not find behavioral differences for 5 of the 9 version pairs in 10,000 runs. Seeding the program exploration with the existing test suite helps Pex in finding behavioral differences for 3 of these version pairs under test. Pex+eXpress could not find behavioral differences for 3 of the 9 version pairs in 10,000 runs. Seeding the program exploration with the existing test suite helps Pex+eXpress in finding behavioral differences for 2 of these version pairs under test.

In summary, Pex requires around 67.5% of the original runs and 67% less time (required by Pex without test seeding) and Pex+eXpress requires around 74% of the original runs and 70% less 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 finding irrelevant branches.

6 Related Work

Previous approaches [5, 14, 9] generate regression unit tests achieving high structural coverage on both versions of the class under test. However, these approaches explore all the irrelevant paths, which cannot help in achieving any of the conditions I or E in

Table 3. Results obtained by seeding existing test suite for structorian.

C	V	N_{Pex}/T	N_{Pseed}/T	$N_{eXpress}/T$	Ne_{seed}/T
SP	2-5	$10000/1hr^*$	$10000/1hr^*$	2381/35min	181/17min
SP	37-39	3699/26m	60/1m	851/22m	47/11m
SP	39-40	$10000/1hr^*$	304/2m	$10000/1hr^*$	251/12m
SP	45-47	$10000/1hr^*$	$10000/1hr^*$	$10000/1hr^*$	$10000/1hr^*$
SP	47-50	$10000/1hr^*$	81/1m	$10000/1hr^*$	64/10m
SP	62-124	$10000/1hr^*$	59/1m	7228/58m	41/10m
SL	169-174	478/1m	324/1m	34/1m	18/1m
SL	150-169	299/1m	37/1m	52/1m	29/1m
SL	9-139	2988/2m	69/1m	1002/1m	52/1m
Total		64476/6.5hr	20934/2hr8m	41568/5hr9m	10683/2hr3m

*If behavior differences are not detected, we take the number of runs as 10,000 (the maximum number of runs that we ran our experiments with)

the PIE model [17]. In contrast, we have developed a new search strategy for DSE to avoid exploring these irrelevant paths.

Santelices et al. [12] 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 [10] determines behavioral differences between two versions of a method (or a program) by comparing their symbolic summaries [6]. Summaries can be computed only for methods amenable to symbolic execution. However, summaries cannot be computed for methods whose behavior is defined in external libraries not amenable to symbolic execution. Our approach still works in practice when these external library methods are present since our approach does not require summaries. In addition, both approaches can be combined using demand-driven-computed summaries [1], which we plan to investigate in future work. Qi et al. [11] propose an approach for guided test generation for evolving programs. The approach guides path exploration towards executing a change and propagating a state infection to observable output. However, the approach cannot deal with multiple interacting changes in the program in contrast to our approach. In addition, our approach provides techniques for effectively infecting program state.

Godefroid et al. [8] propose a DSE based approach for fuzz testing of large applications. Their approach uses a single seed for program exploration. In contrast, our approach seeds multiple tests to program exploration. Seeding multiple tests can help program exploration in covering the changes more efficiently as discussed in Section 4.5. Xu and Rothermel [19] propose a directed test generation technique that uses the existing test suite to cover parts of the program that are not covered by the existing test suite. In particular, the approach first collects the set of branches B that are not covered by the existing test suite. To cover a branch $b_i =< v_i, v_j >\in B$, the approach selects all the tests T that cover the vertex v_i. For each test $t_i \in T$, the approach collects the path constraints p_i of path followed by t_i until v_i, negates the predicate at v_i from p_i to get path condition p'_i. The approach then generates a test that covers the branch b_i by solving the path conditions p_i . However if all the path conditions of paths followed by the tests T are not solvable, the approach cannot generate a test to cover the branch b_i, 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, the approach focuses only on satisfying condition E of the PIE model, while our approach helps in satisfying E, I, and P of the PIE model to find behavioral differences.

Our previous approach [14], called DiffGen, instruments the program to add branches such that behavioral differences can be found effectively. However, a test gen-

eration tool needs to explore branches in both the original and new version of the program to detect behavioral differences. In contrast, our approach adds branches for causing state infection effectively. As a result, a test generation tool needs to explore only the changed regions of the original program version (in addition of the new program version). In addition, `eXpress` prunes irrelavant branches to find behavioral differences efficiently.

7 Conclusion

Regression testing aims at generating tests that detect behavioral differences between two versions of a program. To expose behavioral differences, a test execution needs to satisfy the conditions: Execution (E), Infection (I), and Propagation (P), as stated in the PIE model [17]. Dynamic symbolic execution (DSE) can be used to generate tests for satisfying these conditions. DSE explores paths in the program to achieve high structural coverage, and exploration of all these paths can often be expensive. However, many of these paths in the program cannot help in satisfying the three conditions in any way. In this paper, we presented an approach and its implementation called `eXpress` for regression test generation using DSE. `eXpress` prunes paths or branches that cannot help in detecting the E, I, or P condition such that these conditions are more likely to be satisfied earlier in path exploration. In addition, `eXpress` instruments the source code to achieve condition I. Furthermore, our approach can exploit the existing test suite for the original version to efficiently execute the changed regions (if not already covered by the test suite). Experimental results on various versions of programs showed that our approach can efficiently find behavioral differences than without using our approach.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. TACAS*, pages 367–381, 2008.
2. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446, 2008.
3. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *Proc. ECOOP*, pages 404–428, 2006.
4. 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.
5. R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. FSE*, pages 549–552, 2007.
6. P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.
7. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *Proc. PLDI*, pages 213–223, 2005.
8. P. Godefroid, M. Y. Levin, and D. A. Molnar. In *Proc. NDSS*, pages 151–166, 2008.
9. W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proc. ICST*, 2010.
10. S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.
11. D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *Proc. ASE*, pages 397–406, 2010.
12. 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.
13. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. FSE*, pages 263–272, 2005.
14. K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.
15. 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.
16. N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.
17. J. Voas. PIE: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.
18. T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, 2009.
19. Z. Xu and G. Rothermel. Directed test suite augmentation. In *APSEC*, pages 406–413, 2009.