MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code*

Suresh Thummalapenta[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2], Wolfram Schulte[2]
[1]Department of Computer Science, North Carolina State University, Raleigh
[2]Microsoft Research, One Microsoft Way, Redmond
[1]{sthumma, txie}@ncsu.edu, [2]{nikolait, jhalleux, schulte}@microsoft.com

ABSTRACT

An objective of unit testing is to achieve high structural coverage of the code under test. Achieving high structural coverage of object-oriented code requires desirable method-call sequences that create and mutate objects. These sequences help generate target object states such as argument or receiver object states (in short as target states) of a method under test. Automatic generation of sequences for achieving target states is often challenging due to a large search space of possible sequences. On the other hand, code bases that use object types (such as argument or receiver object types) include sequences that can be used to assist automatic test-generation approaches in achieving target states. In this paper, we propose a novel approach, called MSeqGen, that mines code bases and extracts sequences related to receiver or argument object types of a method under test. Our approach uses these extracted sequences to enhance two state-of-the-art test-generation approaches: random testing and dynamic symbolic execution. We conduct two evaluations to show the effectiveness of our approach. Using sequences extracted by our approach, we show that a random testing approach achieves 8.69% (with a maximum of 20% for one namespace) higher branch coverage and a dynamic-symbolic-execution-based approach achieves 17.4% (with a maximum of 22.45% for one namespace) higher branch coverage than without using our approach. Such an improvement is significant as the branches that are not covered by these state-of-the-art approaches are generally quite difficult to cover.

1. INTRODUCTION

A primary objective of unit testing is to achieve high structural coverage such as branch coverage. Achieving high structural coverage with passing tests gives high confidence in the quality of the code under test. To achieve high structural coverage of object-oriented code, unit testing requires desirable method-call sequences (in short as *sequences*) that create and mutate objects. These sequences help generate target object states (in short as *target states*) of the receiver or arguments of the method under test (MUT). To give a real example for a target state, consider the `Compute` MUT shown in Figure 1. The MUT performs a depth-first search on an undirected graph. A target state for reaching Statements 8, 14, or 22 requires that a graph instance used in test execution has a non-empty set of vertices and edges.

To generate target states, there exist three major categories of automatic sequence-generation approaches: bounded-exhaustive [13, 30], evolutionary [11, 27], and random [5, 12, 19] approaches. Bounded-exhaustive approaches generate method-call sequences exhaustively up to a small bound of sequence length. However, generating target states often requires longer sequences beyond the small bound that can be handled by the bounded-exhaustive approaches. On the other hand, evolutionary approaches accept an initial set of sequences and evolve those sequences to produce new sequences that can generate target states. These approaches use *fitness* [16], a metric computed toward reaching a target

```
//UndirectedDFS: Short form of UndirectedDepthFirstSearch
00:class UndirectedDFS {
01:  IVertexAndEdgeListGraph VisitedGraph; ...
02:  public UndirectedDFS(IVertexAndEdgeListGraph g) {
03:    ...
04:  }
05:  public void Compute(IVertex s) {
06:    //init vertices
07:    foreach(IVertex u in VisitedGraph.Vertices) {
08:      Colors[u]=GraphColor.White;
09:      if (InitializeVertex != null)
10:      InitializeVertex(this, new VertexEventArgs(u));
11:    }
12:    //init edges
13:    foreach(IEdge e in VisitedGraph.Edges) {
14:      EdgeColors[e]=GraphColor.White; }
15:    //use start vertex
16:    if (s != null) {
17:      if (StartVertex != null)
18:        StartVertex(this,new VertexEventArgs(s));
19:      Visit(s); }
20:    // visit vertices
21:    foreach(IVertex v in VisitedGraph.Vertices) {
22:      if (Colors[v] == GraphColor.White) {
23:        if (StartVertex != null)
24:          StartVertex(this,new VertexEventArgs(v));
25:        Visit(v); }
26:    }
27:  }
28:}
```

Figure 1: A method under test from the QuickGraph library [20].

*This work is supported in part by NSF grant CNS-0720641 and Army Research Office grant W911NF-07-1-0431.

state, as a guidance for producing new sequences. However, the generation is still a random process and shares the same characteristics as random approaches discussed next.

Random approaches use a random mechanism to combine individual method calls to generate method-call sequences. Although random approaches are shown as effective as systematic approaches theoretically [6], random approaches still face challenges in practice to generate sequences for achieving target states. The reason is that there is often a low probability of generating required sequences at random for achieving target states. To illustrate the challenges faced by random approaches, we applied a state-of-the-art random approach, called Randoop [19], on the MUT shown in Figure 1. Randoop achieved branch coverage of 31.82% (7 out of 22 branches). The reason for low coverage is that the random mechanism of Randoop is not able to generate a graph with vertices and edges.

Another state-of-the-art category of unit-test-generation approaches is dynamic symbolic execution (DSE) [3, 10, 14, 15]. These approaches first execute code under test with arbitrary inputs. During execution, these approaches collect symbolic constraints on inputs obtained from predicates in branching statements along the execution. These approaches next use a constraint solver to compute variations of the previous inputs to guide future program executions along different paths. A recent DSE-based approach, called Pex [25], generates method-call-sequence skeletons (in short as *skeletons*), which are basically sequences with symbolic values for primitive types. These skeletons can be considered as a general form of sequences, where symbolic values are used instead of constant values for primitive types. Pex computes concrete values for the symbolic values in skeletons based on constraints collected from branching statements in the code under test. Based on our experience of applying Pex on industrial code bases, we observed that many branches in the code under test are not covered due to lack of proper skeletons. For example, Pex achieved branch coverage of 45.45% (10 out of 22 branches) on the MUT shown in Figure 1. The reason for low coverage is that Pex is also not able to generate a graph with vertices and edges.

A common characteristic among these previous approaches for generating sequences is that these approaches generate sequences either randomly or based on implementation informations of method calls used in the sequence. As these existing approaches are not effective in generating sequences, in this paper, we address this significant problem of sequence generation from a novel perspective of how these method calls are used together in practice. More specifically, using information of how the method calls are used in practice helps generate more sequences that can achieve target states. To gather such usage information of method calls, our approach mines code bases that are already using the object types such as receiver or argument object types of the MUT. For a MUT, these code bases include source code of the application that the MUT belongs to and test code for that application. In addition, code bases also include other applications using the receiver or argument object types of the MUT available in both proprietary and open source code (available on the web).

To the best of our knowledge, our approach, called MSeqGen, is the first one that addresses the significant problem of sequence generation by leveraging the information of how method calls are used in practice. Our approach mines code bases to extract sequences related to receiver or argument object types of a MUT. Our approach uses extracted sequences to assist both random and DSE-based approaches in achieving higher structural coverages. More specifically, our approach addresses three major issues in extracting sequences from code bases to assist test generation approaches. First, code bases are often large and complete analysis of these code bases can be prohibitively expensive. To address this issue, our approach searches for code portions relevant to receiver or argument object types of the MUT and analyzes those relevant code portions only. Second, constant values in sequences extracted from code bases can be different from values required to achieve target states. To address this issue, our approach converts extracted sequences into skeletons by replacing constant values for primitive types with symbolic values. We refer this process of converting sequences into skeletons as *sequence generalization*. Third, extracted sequences individually may not be useful in achieving target states. Our approach addresses this issue by combining extracted sequences randomly to generate new sequences that may produce target states.

In summary, this paper makes the following major contributions:

- The first approach that leverages the information of how method calls are used in practice to address the significant problem of sequence generation in object-oriented unit testing.

- A technique that mines large code bases by searching for code portions relevant to receiver or argument object types of a MUT. Our approach analyzes *only* those relevant code portions as analyzing complete code bases can be prohibitively expensive. Our approach uses extracted sequences to assist test generation approaches such as random and DSE-based test-generation approaches.

- A technique for generalizing (i.e., converting into skeletons) extracted sequences to assist DSE-based approaches. Generalization helps to address issues where constant values in extracted sequences are different from values required to achieve target states.

- An implementation of the approach and its evaluation upon two state-of-the-art industrial testing tools: Randoop and Pex. Both Pex and Randoop were shown to find serious defects in industrial code bases [19, 25]. Our approach represents a significant, successful step towards addressing complex testing problems in industrial practice, targeting at complex desirable sequences from multiple classes rather than sequences on single classes such as data structures heavily focused by previous approaches [11, 27] (where sequence generation is already a quite challenging problem).

- Empirical results from two evaluations show that our approach can effectively assist state-of-the-art random and DSE-based approaches in achieving higher branch coverage. Using our approach, we show that a random approach achieves 8.69% (with a maximum of 20% for one namespace) higher branch coverage and a DSE-based approach achieves 17.4% (with a maximum of 22.45% for one namespace) higher branch coverage

than without using our approach. Such an improvement is significant as the branches that are not covered by these state-of-the-art approaches are generally quite difficult to cover.

The rest of the paper is structured as follows: Section 2 presents background on a random and a DSE-based approaches. Section 3 explains our approach with an example. Section 4 describes key aspects of our approach. Section 5 presents our evaluation results. Section 6 discusses threats to validity. Section 7 discusses our limitations and future work. Section 8 presents related work. Finally, Section 9 concludes.

2. BACKGROUND

In this paper, we use two existing approaches Randoop [19] and Pex [25] as example state-of-the-art approaches for random and DSE-based approaches, respectively. We next briefly describe these two example approaches.

2.1 Randoop

Randoop [19] is a random testing approach that constructs test inputs (in the form of sequences) incrementally by randomly selecting method calls. For each such randomly selected method call, Randoop finds arguments from previously-constructed inputs or tries to generate new sequences for those arguments. These constructed sequences are considered as test inputs. Unlike a pure random approach, Randoop incorporates feedback obtained from previously constructed test inputs. As soon as a test input is constructed, Randoop executes the test input and verifies the output against a set of contracts and filters.

2.2 Pex

Pex [25] is an automatic unit-test-generation tool from Microsoft. Parts of Pex may be integrated into a future version of Microsoft Visual Studio 2010, benefiting an enormous number of developers in industry. Pex accepts parameterized unit tests (PUT) [26] as input; PUTs are a new advancement in unit testing and these PUTs accept parameters unlike conventional unit tests, which do not accept any parameters. Pex accepts PUTs as inputs and generates conventional unit tests that can achieve high structural coverage of the code under test. More specifically, Pex is a DSE-based approach [10] that initially executes the code under test with arbitrary inputs. During execution, Pex collects symbolic constraints on inputs obtained from predicates in branch statements along the execution. Pex uses a constraint solver to compute variations of the previous inputs to guide future program executions along different paths. As PUTs also accept non-primitive types as arguments, Pex uses a combination of static and dynamic analysis techniques for generating sequences for those non-primitive types. Using static analysis, Pex determines possible constructors and other methods of a class that set values to different fields of that class. Based on constraints collected during execution of the code under test, Pex determines which methods can set values for required fields and tries to cover target branches by combining constructors and method calls.

3. EXAMPLE

We next explain our approach with an illustrative example shown in Figure 1. The figure shows a MUT, called `Compute`, taken from the QuickGraph library [20]. The MUT requires two non-primitive instances: `IVertexAndEdgeListGraph` and `IVertex`. The MUT requires an instance of `IVertexAndEdgeListGraph` (which represents a graph) since the constructor of the receiver object of the MUT has the argument of type `IVertexAndEdgeListGraph`. The MUT accepts a vertex in the graph as argument and computes a depth-first search of the graph. To achieve high code coverage of the MUT, the minimal requirement is that the graph instance should include vertices and edges. We used both Randoop and Pex to generate unit tests for the MUT. Randoop achieved branch coverage of 31.82% (7 of 22). The reason for low branch coverage is that the random mechanism of Randoop is not able to generate a graph instance with vertices and edges.

To generate test inputs using Pex, we created a PUT that includes `UndirectedDFS` as a parameter. As the constructor of `UndirectedDFS` accepts an interface `IVertexAndEdgeListGraph` as argument, Pex can automatically generate a new class implementing the `IVertexAndEdgeListGraph` interface. However, such a new class may not support the (implicit) contracts associated with the interface implementation. Therefore, we provided minimal assistance for Pex by describing which implementing classes can to be used for interfaces. For example, we feed to Pex information that it can use the `AdjacencyGraph` class as an implementing class for the `IVertexAndEdgeListGraph` interface. Pex achieved branch coverage of 45.45% on the MUT. Although Pex achieved higher branch coverage than Randoop, the coverage is still low (only 45.45%). Similar to Randoop, Pex was not able to generate a graph instance with edges and vertices.

We next describe how MSeqGen can assist Randoop and Pex by extracting sequences from existing code bases. We need sequences for instances of three classes[1]: `UndirectedDFS`, `IVertexAndEdgeListGraph`, and `IVertex`. We need a sequence for an instance of the `UndirectedDFS` class to construct a desired receiver object state. We also need sequences for instances of classes implementing the `IVertexAndEdgeListGraph` and `IVertex` interfaces. We collected a set of applications (code bases of 3.85 MB of .NET assembly code) from an open source C# repository[2] that reuse classes of the QuickGraph library. MSeqGen analyzes these code bases and extracts sequences that produce instances of these classes.

MSeqGen extracted 5 sequences for the `AdjacencyGraph` class that implements `IVertexAndEdgeListGraph` interface, and 11 sequences for the `Vertex` class that implements the `IVertex` interface. Figures 2 and 3 show example sequences for creating instances of `AdjacencyGraph` and `Vertex` classes. The sequence for `AdjacencyGraph` satisfies our minimal requirement that the resulting graph should include vertices and edges. It is challenging to generate these sequences automatically, especially due to the large number of possible combinations. In contrast, MSeqGen can easily extract such sequences from existing code bases.

One issue with extracted sequences is that these sequences can include additional classes. For example, Statement 3 of Figure 2 shows that the sequence requires another instance of the class `VertexAndEdgeProvider`. MSeqGen automatically identifies such additional classes and gathers sequences that produce instances of these classes. MSeqGen extracted one sequence for the `VertexAndEdgeProvider` class from existing

[1]We use classes to collectively represent both classes and interfaces.

[2]http://www.codeplex.com/

```
01: VertexAndEdgeProvider vo; //requires as input
02: bool bVal; //requires as input
03: AdjacencyGraph agObj = new AdjacencyGraph(vo,bVal);
04: IVertex source = agObj.AddVertex();
05: IVertex target = agObj.AddVertex();
06: IVertex vertex3 = agObj.AddVertex();
07: IEdge edgObj1 = agObj.AddEdge(source,target);
08: IEdge edgObj2 = agObj.AddEdge(target,vertex3);
09: IEdge edgObj3 = agObj.AddEdge(source,vertex3);
```

Figure 2: A sequence for producing an `AdjacencyGraph` instance with vertices and edges.

```
AdjacencyGraph agObj; //requires as input
IVertex vObj = agObj.AddVertex();
```

Figure 3: A sequence for producing an `IVertex` instance.

code bases. Section 4 presents more details on how MSeqGen addresses challenges for extracting these sequences.

We next use Randoop with additional sequences extracted by MSeqGen. Randoop generated new test inputs incorporating sequences extracted by MSeqGen. The new test inputs achieved branch coverage of 86.36% (19 of 22) of the `Compute` method. When we use our extracted sequences to assist Pex, Pex also achieved the same branch coverage for the `Compute` method. The remaining not-covered branches are due to lack of event handlers that need to be registered with `UndirectedDFS`. This example describes our MSeqGen approach and highlights the significance of using sequences from existing code bases in achieving higher branch coverages with both random and DSE-based approaches.

4. APPROACH

Figure 4 shows a high-level overview of our MSeqGen approach. MSeqGen accepts an application under test and identifies classes and interfaces, declared or used by the application under test. These applications under test can also be frameworks or libraries. We refer extracted classes for which sequences need to be collected as target classes, denoted by $\{TC_1, TC_2, ..., TC_m\}$. MSeqGen also accepts a set of existing code bases, denoted by $\{CB_1, CB_2, ..., CB_n\}$, that already use these target classes. In our prototype implemented for the MSeqGen approach, these set of code bases are in the form of .NET assemblies. Initially, MSeqGen searches for relevant method bodies by using target classes as keywords. MSeqGen constructs control-flow graphs for these method bodies and extracts sequences that produce instances of target classes. MSeqGen extracts sequences by traversing these control-flow graphs. These extracted sequences are used to assist random and DSE-based approaches. For DSE-based approaches, MSeqGen converts extracted sequences into skeletons by replacing constant values for primitive types with symbolic values. We next explain each phase in detail using the illustrative example shown in Figure 1.

4.1 Code Searching

We use code searching in our approach since code bases are often large and analyzing complete code bases can be prohibitively expensive. To avoid analysis of complete code bases, we use a keyword search to identify relevant method bodies including target classes. In particular, we use a text-based search, where the text is derived by decompiling .NET assemblies taken as inputs. We consider that a method body is relevant to a TC_j target class, if the method body includes the name of the TC_j target class. We use intra-procedural

```
00:public void Sort(VertexAndEdgeProvider vo) {
01:  AdjacencyGraph g = new AdjacencyGraph(vo, true);
02:  Hashtable iv = new Hashtable();
03:  int i = 0;  //adding vertices
04:  IVertex a = g.AddVertex();
05:  iv.Add(a);
06:  IVertex b = g.AddVertex();
07:  iv.Add(b);
08:  IVertex c = g.AddVertex();
09:  iv.Add(c);
10:  g.AddEdge(a,b); //adding edges
11:  g.AddEdge(a,c);
12:  g.AddEdge(b,c);
13:  //TSAlgorithm: TopologicalSortAlgorithm
14:  TSAlgorithm topo = new TSAlgorithm(g);
15:  topo.Compute(); ... }
```

Figure 5: A relevant method body for classes `AdjacencyGraph`, `VertexAndEdgeProvider`, `Hashtable`, `TopologicalSortAlgorithm`.

analysis to analyze only such relevant method bodies. We use intra-procedural analysis since intra-procedural analysis is more scalable than inter-procedural analysis. Although intra-procedural analysis is less precise than inter-procedural analysis, we address this precision issue by using an iterative strategy explained in subsequent sections. For example, we use `AdjacencyGraph` as a keyword and search for method bodies including that keyword. Figure 5 shows an example method body including the `AdjacencyGraph` keyword. As our code search is primarily a text-based search, code searching also returns irrelevant method bodies such as method bodies that include `AdjacencyGraph` as a variable name or a word in comments. We filter out such irrelevant method bodies in subsequent phases.

4.2 Code Analysis

We next analyze each relevant method body statically and construct a control-flow graph (CFG). Our CFG includes four types of statements: method calls, object creations, typecasts, and field accesses. The rationale behind choosing these statements is that these statements result in generating instances of target classes. While constructing CFG, we identify the nodes (in constructed CFG) that produce the target classes such as `AdjacencyGraph` and mark them as *nodes of interest*. For example, the node corresponding to Statement 1 in Figure 5 is marked as a node of interest for the target class `AdjacencyGraph`. We also filter out irrelevant method bodies identified during the code searching phase if their related CFGs do not contain any nodes of interest.

We next extract sequences from CFG using nodes of interest. For each node of interest related to a TC_j class, we gather a path from the node of interest to the end of the CFG. When loops are encountered, we annotate the nodes to store the additional information that these method calls exist inside loops. Often, an extracted sequence can include a few method calls that are unrelated to the target class in the gathered path. We use data-dependency analysis to filter out such unrelated method calls. Each gathered path results in a sequence that creates and mutates an instance of a target class. For example, Figure 2 shows a sequence gathered from the code example in Figure 5. MSeqGen extracts several such sequences for different classes from the same code example. For example, if the set of target classes also includes classes `Hashtable` and `TSAlgorithm`, MSeqGen automatically extracts one sequence for each of these classes as shown below from the code example.

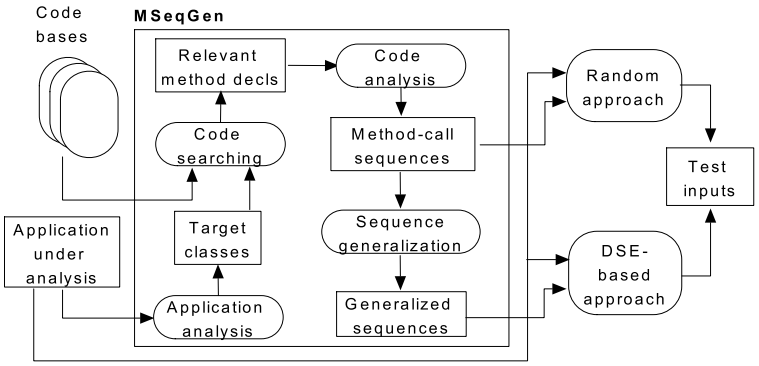

Figure 4: Overview of our MSeqGen approach.

Sequence for Hashtable:
```
IVertex a,b,c; //requires as input
Hashtable iv = new Hashtable();
iv.Add(a);
iv.Add(b);
iv.Add(c);
```

Sequence for TSAlgorithm:
```
AdjacencyGraph g; //requires as input
TSAlgorithm tsObj = new TSAlgorithm(g);
tsObj.compute();
```

One issue with extracted sequences is that these sequences can include additional non-primitive types. For example, the sequence for `AdjacencyGraph` (shown in Figure 2) requires non-primitive type `VertexAndEdgeProvider`. To achieve target states, we need new sequences for generating these additional non-primitive types. Usually call sites in code bases including sequences for a TC_j target class also include sequences for generating related additional non-primitive types. However, in practice, often these call sites do not include sequences for these additional non-primitive types due to two factors. (1) A sequence for an additional non-primitive type is available in another method body and is not found by our approach as it uses intra-procedural analysis for extracting sequences. (2) A sequence for an additional non-primitive type does not exist in the current code base CB_i (such as a framework or a library) and expects a reusing application to provide a necessary sequence.

We address this issue by extracting new sequences for additional non-primitive types by using an iterative strategy. More specifically, we first extract sequences for the initial set of target classes and collect all additional classes for which new sequences need to be extracted. We next extract sequences for these additional classes and collect more new additional classes. We repeat this process either till no new additional classes are collected or we reach a fixed number of iterations accepted as a configuration parameter, denoted by *NUM_ITERATIONS*. A high value for *NUM_ITERATIONS* can help collect more sequences, however, a high value can require more time for collecting those sequences. In our approach, we use the value *NUM_ITERATIONS* as five, which is set based on our initial empirical experience.

4.3 Method-Call Sequence Generalization

We generalize sequences to address an issue that constant values in extracted sequences can be different from values required to achieve target states. We refer to the process of converting sequences into skeletons (which are sequences with symbolic values instead of concrete values for primitive types) as sequence generalization. For example, consider a simple MUT and an example sequence (shown as MCS) shown in Figure 6. The sequence cannot directly achieve the `true` branch of the MUT as the value of `testMe` is set to 10. To address this issue, we generalize extracted sequences. More specifically, we replace constant values of primitive types in extracted sequences with symbolic values. Figure 6 also shows the skeleton, where a symbolic variable `symvar` of type `int` is taken as input for the sequence. This `symvar` variable replaces the constant value 10 in the MCS. When this skeleton is used along with a DSE-based approach, the DSE-based approach initially generates a concrete random value for the `symvar` symbolic variable and gathers the constraint (> 100) in the MUT through dynamic execution. The DSE-based approach next solves the constraint to generate another concrete value for `symvar` such as 200 that satisfies the gathered constraint.

Although DSE-based approaches are effective in practice, it is challenging for these approaches to generate concrete values for variables that require complex values such as doubles, IP addresses, or URLs. In such cases, constant values in extracted sequences are useful in quickly covering those

Class Definition:
```
00:class MyClass {
01:  private int testMe;
02:  private String ipAddr;
03:}
```

MUT:
```
00:public void Mut1(MyClass mc, String IPAddress) {
01:  if(mc.getTestMe() > 100) {
02:    if(IsAValidIPAddress(IPAddress)) { ... }
03:  }
04:}
```

Method-call sequence (MCS):
```
00:MyClass mcObj = new MyClass();
01:mcObj.SetTestMe(10);
02:mcObj.SetIpAddr("127.0.0.1");
```

Skeleton:
```
00:int symvar = *, string ipaddr = *;
01:MyClass mcObj = new MyClass();
02:mcObj.SetTestMe(symvar);
03:mcObj.SetIpAddr(ipaddr);
```

Figure 6: An illustrative example for method-call sequence generalization.

related branches such as the `true` branch in Statement 2 of the MUT. To address this issue, we preserve constant values in sequences along with newly introduced symbolic values.

4.4 Generation of New Sequences

MSeqGen extracts sequences from code bases and uses these sequences to assist random and DSE-based approaches. However, in some cases, these extracted sequences individually are not sufficient to achieve target states. MSeqGen tries to address this issue by generating new sequences from extracted sequences by combining extracted sequences randomly. For example, consider two target classes T_i and T_j. Consider that MSeqGen identified two method bodies, denoted by MD_1 and MD_2, in code bases relevant to both T_i and T_j. Consider that MSeqGen extracted sequences S_i^1 and S_j^1 for target classes T_i and T_j from MD_1, respectively. Similarly, MSeqGen extracted sequences S_i^2 and S_j^2 for target classes T_i and T_j from MD_2, respectively. The target class T_i has sequences S_i^1 and S_i^2, and the target class T_j has sequences S_j^1 and S_j^2. Given these sequences, MSeqGen can generate some or all of four different combinations of these sequences. These new sequences may further help achieve target states.

5. EVALUATION

We conducted three different evaluations to show the effectiveness of our MSeqGen approach. In our evaluations, we used two popular .NET applications: QuickGraph [20] and Facebook [9]. Our empirical results show that MSeqGen handles large code bases and extracts sequences that can help achieve target states. Our empirical results also show that our approach can effectively assist random and DSE-based approaches in achieving higher branch coverages. The details of subjects and results of our evaluation are available at `http://ase.csc.ncsu.edu/projects/mseqgen`. All experiments were conducted on a machine with 1.6GHz Xeon processor and 1GB RAM. We next present research questions addressed in our evaluations.

5.1 Research Questions

In our evaluations, we address the following research questions.

- RQ1: Can our approach handle large code bases in generating sequences for target classes of subject applications?
- RQ2: Can our approach assist a random approach in achieving higher code coverage of the code under test than without the assistance of our approach?
- RQ3: Can our approach assist a DSE-based approach in achieving higher code coverage of the code under test than without the assistance of our approach?

5.2 Subject Applications

We used two popular .NET applications for evaluating our MSeqGen approach: QuickGraph [20] and Facebook [9]. QuickGraph is a C# graph library that provides various directed/undirected graph data structures. QuickGraph also provides algorithms such as depth-first search, breadth-first search, and A* search [4]. QuickGraph includes 165 classes and interfaces and 5 KLOC. Facebook is a popular social network website that connects people with friends and others whom they work, study, and live around. In our evaluation, we use a Facebook developer toolkit that provides APIs necessary for developing Facebook applications. The Facebook developer toolkit includes 285 classes and interfaces and 40 KLOC.

5.3 RQ1: Generating Sequences

We next address the first research question on whether our approach can handle large code bases in gathering sequences for target classes of the QuickGraph and Facebook applications. For QuickGraph and Facebook, we use code bases including 3.85 MB and 5 MB of .NET assembly code, respectively. Our approach extracted 167 sequences for QuickGraph with a maximum length of 12 method calls for the `AdjacencyGraph` class. Our approach took 5.16 minutes for analyzing code bases used as input for QuickGraph. For Facebook, our approach extracted 355 sequences with a maximum length of 51 method calls for the `Hashtable` class. Although the sequence extracted for `Hashtable` is long, this sequence includes method calls such as `Add` for multiple times. Our approach took 4.5 minutes for analyzing code bases related to Facebook and to gather these sequences. Our results show that our approach can mine large code bases for extracting sequences to help achieve target states.

5.4 RQ2: Assisting Random Approach

We next address the second research question on whether our approach helps increase branch coverage achieved by a state-of-the-art random approach, called Randoop [19]. To address this research question, we first run Randoop on QuickGraph and Facebook applications, and generate test inputs. Randoop generates test inputs in the form of sequences of method calls. We execute generated test inputs and measure branch coverage using a coverage measurement tool, called NCover[3]. This measured coverage forms a baseline for comparing Randoop with and without the assistance from our approach. In our evaluation, we use default configurations provided by the Randoop developers.

To assist Randoop with our extracted sequences, we create static method bodies that include our sequences and return instances of target classes of our subject applications. For example, if a target class TC_j has four sequences, we create four static method bodies where each method body returns an instance of TC_j by executing an extracted sequence for TC_j. If a sequence for TC_j requires other instances of non-primitive or primitive types (whose values are not known in extracted sequences due to static analysis), we add those non-primitive and primitive types as arguments for the method bodies. For primitive types, Randoop randomly generates some values. For non-primitive types, Randoop randomly generates a new sequence or selects some other method body (suggested by our approach) that produces that non-primitive type. We gather newly generated test inputs that include the method bodies created by MSeqGen and add these new test inputs to existing tests to measure the increase in the branch coverage.

Table 1 shows the results of our evaluation with both subject applications. The table shows the results for all namespaces of the subject applications. As we include test code available with subject applications in code bases used for extracting sequences, we show branch coverages achieved by

[3] `http://www.ncover.com/`

Application	# of classes	Test Code T	Random R	Random + MSeqGen R + M	% Increase in Branch coverage
QuickGraph.Algorithms	104	18.37	63.27	63.27	-
QuickGraph.Algorithms.Search	11	40.26	33.33	47.62	**14.29**
QuickGraph.Algorithms.ShortestPath	4	0	29.25	30.19	0.94
QuickGraph.Algorithms.Visitors	11	0	86.36	86.36	-
QuickGraph.Collections	19	11.15	73.98	83.27	9.29
QuickGraph.Exceptions	3	40	100	100	-
QuickGraph.Predicates	9	8.62	43.1	48.28	5.18
QuickGraph.Providers	1	100	80	100	**20**
QuickGraph.Representations	3	43.05	35.1	49.01	**13.91**
facebook	25	48.91	14	23.27	9.27
facebook.Components	3	0	30.65	30.65	-
facebook.desktop	14	0	18.48	21.01	2.53
facebook.Forms	4	0	11.11	11.11	-
facebook.Properties	1	31.25	37.5	37.5	-
facebook.Schema	216	6.06	20.83	24.86	4.03
facebook.Types	1	0	100	100	-
facebook.Utility	8	49.06	22.64	37.74	**15.1**
facebook.web	12	0	3.28	4.51	1.23
AVERAGE					8.69

Table 1: Evaluation results showing higher branch coverages achieved by Randoop with the assistance of MSeqGen. `T: Test code, R: Randoop, M: MSeqGen`

```
00:class BidirectionalGraph { ...
01:  public IEdge AddEdge(IVertex src, IVertex tg) {
02:    // look for the vertex in the list
03:    if (!VertexInEdges.ContainsKey(src))
04:      throw new VertexNotFoundException
         ("Could not find source");
05:    if (!VertexInEdges.ContainsKey(tg))
06:      throw new VertexNotFoundException
         ("Could not find target");
07:    // create edge
08:    IEdge e = base.AddEdge(src, tg);
09:    VertexInEdges[target].Add(e);
10:    return e;
11:  }
12:}
```

Figure 7: A MUT `AddEdge` in the `BidirectionalGraph` class of QuickGraph.

the test code alone in Column "T". Column "R" shows branch coverages achieved by Randoop. Column "R + M" shows branch coverages achieved by Randoop with the assistance of our MSeqGen approach. Column "Increase in Branch Coverage" shows additional branch coverages achieved with the assistance from our MSeqGen approach. As shown in our results, "R + M" achieved higher coverage than Randoop and test code (except for namespaces `facebook` and `facebook.Utility`). There are two primary reasons for lower coverage of "R + M" for these two namespaces: random mechanism of Randoop and our current implementation limitations. Due to the random mechanism used by Randoop, various method calls used in test code that contributed to higher coverage achieved by the test code are not used by Randoop in generating test inputs. Section 7 presents our current implementation limitations on why "R + M" achieved lower coverage than existing test code for namespaces `facebook` and `facebook.Utility`. Our results show that there is a considerable increase of 8.69% on average[4] (with a maximum of 20%) in branch coverages achieved by Randoop with assistance from our approach.

We next provide examples to describe scenarios where our approach can assist random approaches. We also describe scenarios where our approach cannot assist random approaches. We use a MUT, called `AddEdge`, in the `BidirectionalGraph` class of the `QuickGraph.Representations` namespace (shown in Figure 7). Although Randoop generated three test inputs (in the form of sequences) for the `AddEdge` MUT, Randoop achieved low branch coverage of 40% (2 out of 5 branches). The reason for not achieving high coverage for the `AddEdge` MUT is that the `AddEdge` MUT requires a specific receiver object state. To reach Statement 8 of the MUT, the `VertexInEdges` field should include the new vertices represented by `src` and `tg` that are passed as arguments. With the sequences extracted by our approach, Randoop achieved a branch coverage of 80% (4 out of 5 branches). As our sequences are extracted from code bases that include usage scenarios on how these method calls are used in real practice, our sequences helped achieve high coverage for the `AddEdge` MUT.

Although Randoop achieved higher branch coverage with the assistance from our approach, the test inputs generated by Randoop did not cover the `true` branch of Statement 5 to reach Statement 6. The reason is that our sequences do not include a usage scenario where the `AddEdge` MUT is invoked with one vertex in `VertexInEdges` and the other vertex not in `VertexInEdges`. Such usage scenarios rarely exist in code bases that are used for extracting sequences as these usage scenarios are related to testing the MUT for negative cases rather than reusing the MUT in real practice. However, a more systematic approach such as a DSE-based approach can cover such not-covered branches with the assistance from our approach.

5.5 RQ3: Assisting DSE-based Approaches

We next address the third research question on whether our approach can help increase branch coverages achieved by a DSE-based approach. To address this research question, we use a state-of-the-art DSE-based approach called Pex [25]. Pex accepts PUTs as input and generates conventional unit tests from these PUTs using DSE. As PUTs are not available with our subject applications, we generated

[4]We compute average from those namespaces that has a non-zero increase in the branch coverage

Application	# C	P	P + M	Increase
QuickGraph.Algorithms	104	8.16	30.61	**22.45**
QuickGraph.Algorithms.Search	11	0	13.85	**13.85**
QuickGraph.Algorithms.ShortestPath	4	1.89	1.89	-
QuickGraph.Algorithms.Visitors	11	50	50	-
QuickGraph.Collections	19	14.87	29	**14.13**
QuickGraph.Exceptions	3	60	60	-
QuickGraph.Predicates	9	31.03	31.03	-
QuickGraph.Representations	1	2.65	21.85	**19.2**
AVERAGE				**17.4**

Table 2: Evaluation results showing higher branch coverages achieved by Pex with the assistance of MSeqGen. `# C: number of classes, P: Pex, M: MSeqGen`

PUTs for each public method in our subject applications using the *PexWizard* tool. PexWizard is a tool provided with Pex and this tool automatically generates PUTs for each public method in the application given as input. A PUT generated for the `Compute` MUT (Figure 1) is shown below.

```
00:[PexMethod]
01:public void Compute01(
02:     [PexAssumeUnderTest]UndirectedDFS target,
03:     [PexAssumeUnderTest]Vertex s) {
04:  target.Compute(s);
05:  Assert.Inconclusive("this test has to be reviewed");
06:}
```

The receiver object and argument objects required for the `Compute` MUT are accepted as arguments for the PUT. Pex generates skeletons for the non-primitive arguments by using a heuristic-based approach (Section 2.2). For this evaluation, we used only the QuickGraph application. The reason is that Pex does not terminate in generating unit tests for the Facebook application. In future work, we plan to investigate the issues with Pex and apply Pex on the Facebook application. To provide a baseline for showing the effectiveness of our approach, we first applied Pex on PUTs generated for the QuickGraph application. We executed generated unit tests and measured branch coverages achieved by these unit tests for different namespaces in our subject applications. In our evaluation, we use default configurations of the Pex tool.

We next used our extracted sequences to assist Pex. Pex provides a feature called *factory* methods, which allows programmers to provide assistance to Pex in generating non-primitive object types. We used this feature by converting our extracted sequences into factory methods. One issue with factory methods is that the current Pex allows only one factory method for a non-primitive object type. As our approach can extract multiple sequences for creating an instance of a non-primitive type, we combine all sequences related to a non-primitive type into one factory method by using a `switch` statement. We next apply Pex on the subject application with new factory methods created based on our extracted sequences. We again generate unit tests using Pex and measure new branch coverages.

Table 2 shows our results by applying Pex with and without our sequences on the QuickGraph application. On average, our approach helped increase the branch coverage by 17.4% (with a maximum increase of 22.45% for one namespace). Although there is a considerable increase in branch coverages with the assistance from our approach, overall Pex achieved low branch coverages. This result is due to a limitation with the current Pex release that cannot automatically identify implementing classes for interfaces and use their related factory methods. Often, factory methods generated by our approach accept interfaces as arguments. Therefore, Pex is not able to identify relevant factory methods for interfaces, although factory methods for their implementing classes are generated by our approach. In future work, we plan to to address this limitation and we expect that our results can be much better after addressing this limitation of Pex.

We next present example scenarios where our approach is quite useful in achieving higher branch coverages with Pex. We use the `TopologicalSortAlgorithm` class in the `QuickGraph.Algorithms` namespace as an illustrative example. Without assistance from our approach, Pex did not achieve any coverage of the `TopologicalSortAlgorithm` class as Pex was not able to generate any sequences for creating instances of the `TopologicalSortAlgorithm` class. Using the factory methods generated by our approach, Pex achieved a branch coverage of 57.89% (11 out of 19 branches). Our results show that our approach can assist DSE-based approaches in achieving higher code coverages than without using our approach.

6. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject applications used in our evaluation are representative of true practice. We used two real non-trivial subjects in our evaluation: a medium-scale application QuickGraph [20] and a large-scale application Facebook [9]. These threats could be reduced by using more subjects in our evaluation. The threats to internal validity are instrumentation effects that can bias our results. Faults in our MSeqGen prototype might cause such effects. Furthermore, faults in random and DSE-based approaches used in our evaluations might also cause such effects. To reduce these threats, we inspected a significant sample set of generated test results.

7. DISCUSSION AND FUTURE WORK

Although random and DSE-based approaches show considerable increase in branch coverages with the assistance from our approach, overall coverages achieved by these approaches are still not close to 100% coverage. The reason is that often code under test includes complex branches that are quite difficult to cover. We next give an example of a difficult branch that is not covered by any of the approaches used in our evaluation. We use the code example shown in Figure 8 as an illustrative example. This difficult branch is in the `Visit` method of the `BreadthFirstSearchAlgorithm` class. The receiver-object state to reach Statement 8 requires that the `VisitedGraph` instance has a non-empty set of vertices and edges. Reaching Statement 8 also requires a specific object state for the argument `s`. In particular, the vertex represented by the argument `s` should already exist in the `VisitedGraph` instance and should have outgoing edges. Although our extracted sequences include a sequence for achieving a desired receiver-object state, our sequences do not include a necessary sequence for achieving a desired argument-object state. In future work, we plan to further address these issues by generating new sequences using evolutionary approaches [11, 27]. There, we can use our extracted sequences as an initial set for these evolutionary approaches to evolve.

```
00:public void Visit(IVertex s) {
01:  ...
02:  m_Q.Push(s);
03:  while (m_Q.Count != 0) {
04:    IVertex u = (IVertex)m_Q.Peek();
05:    m_Q.Pop();
06:    ...
07:    foreach(IEdge e in VisitedGraph.OutEdges(u)) {
08:      ... //Difficult branch
09:    }
10:  }
11:}
```

Figure 8: An example difficult branch not reached by all approaches used in our evaluation.

In our evaluations, for the `facebook` and `facebook.Utility` namespaces, branch coverages achieved by Randoop (with the assistance of our approach) are lower than the branch coverages achieved by the test code (commonly written by the application developers). There are two primary reasons for lower coverage of these namespaces: limitations of the random mechanism of Randoop and our current implementation. Our current implementation does not handle several features such as inheritance or C# generics. Therefore, our implementation could not capture some sequences due to their use of these features. In future work, we plan to extend our implementation to support these features.

Our approach extracts sequences from code bases using the receiver or argument object types of a MUT and generates method bodies to assist test generation approaches. Sometimes, these sequences may include object types specific to the code bases. For example, these object types can be classes that implement the interfaces provided by a subject framework. In such scenarios, the method bodies generated by our approach are not compilable. Currently, we fix those compilation errors manually. In future work, we plan to automatically compile and verify extracted sequences to reduce this manual effort. We currently gather *only* one path from a CFG to generate sequences. However, there can be different sequences across multiple paths in the CFG. In future work, we plan to collect sequences from multiple paths in the CFG. We also plan to develop clustering heuristics to cluster similar sequences.

8. RELATED WORK

Test-generation approaches [5, 12, 18, 30] were developed for object-oriented testing and these approaches accept a class under test (CUT) and generate random sequences of method calls belonging to the CUT with random values for method arguments. Another set of approaches [11] replaces random values for method arguments with symbolic values and compute concrete values for these arguments by solving constraints inside the method under test. Tonella [27] proposed an approach that exploits genetic algorithms to generate new sequences evolved from an initial set of sequences. Tonella's approach requires the users to provide this initial set of sequences. Inkumsah and Xie [11] extended Tonella's approach by integrating evolutionary testing with symbolic execution. However, all these approaches cannot handle multiple classes and their methods due to a large search space of possible sequences.

Randoop [19] executes constructed sequences at each stage and computes feedback in order to guide the search process to generate valid sequences. However, Randoop still relies on random techniques and cannot effectively generate sequences for achieving target states as shown in our evaluation. Our approach extracts sequences from code bases and use those sequences to assist other approaches such as Randoop in achieving higher structural coverages.

Our approach is also related to other category of approaches based on mining source code [1, 8, 23, 24, 29]. These approaches mine code bases statically and extracts frequent patterns as implicit programming rules. These approaches use mining algorithms such as frequent itemset mining [28] or association rule mining [2] for extracting frequent patterns. These mined programming rules are used for assisting programmers in writing code or detecting violations in an application under analysis. Our approach also uses static analysis for extracting patterns as sequences that can generate instances of receiver or arguments types of a MUT. Unlike these existing approaches, our approach uses extracted sequences in a novel way for assisting test generation approaches in achieving high structural coverage.

Another category of existing work [7,17,21] uses a capture-and-replay approach for generating unit tests. During the capture phase, their approach monitors the interaction of a unit such as the class under test with the rest of the system (to which the class belongs to). Their approach generates unit tests for the class under test based on monitored interactions. During the replay phase, their approach executes generated unit tests. Our previous approach, called UnitPlus [22], captures sequences in existing test code and suggests those sequences to developers in reducing the effort of writing new unit tests. Our new MSeqGen approach captures sequences from existing code bases but uses those sequences for assisting test generation approaches. Unlike existing approaches that replay exactly the same captured behavior, our MSeqGen approach replays beyond the captured behavior using techniques such as sequence generalization or generating new sequences by combining extracted sequences.

9. CONCLUSION

Generation of desirable method-call sequences for achieving high structural coverage of the code under test is a known challenging problem in unit testing of object-oriented code. Existing work [5, 13, 30] in addressing this problem is based primarily on the implementation information of the class under test. In this paper, we proposed the first approach that addresses this problem from a novel perspective of incorporating other sources of information such as how method calls are used in practice. Our approach gathers the information of how method calls are used in practice by mining code bases that use receiver or argument object types of a method under test. Our approach extracts sequences related to these object types and uses extracted sequences to enhance two state-of-the-art test-generation approaches: random testing and dynamic symbolic execution. We have demonstrated the effectiveness of our approach with two evaluations. Using sequences extracted by our approach, we showed that a random testing approach achieved 8.69% (with a maximum of 20% for one namespace) higher branch coverage and a DSE-based approach achieved 17.4% (with a maximum of 22.45% for one namespace) higher branch coverage than without using our approach. Such an improvement is significant as the branches that are not covered by these state-of-the-art approaches are generally quite difficult to

cover. Our approach represents a step towards a new direction of leveraging research in the field of mining software engineering data to assist test generation, serving as a synergy between these two major research areas.

Acknowledgments

We thank Shuvendu Lahiri and Thomas Ball for sharing Randoop application.

10. REFERENCES

[1] M. Acharya, T. Xie, and J. Xu. Mining Interface Specifications for Generating Checkable Robustness Properties. In *Proc. ISSRE*, pages 311–320, 2006.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.

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

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[6] J. Duran and M. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, 1984.

[7] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. FSE*, pages 253–264, 2006.

[8] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc. SOSP*, pages 57–72, 2001.

[9] Facebook developer toolkit, 2008. `http://www.codeplex.com/FacebookToolkit`.

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

[11] 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.

[12] Parasoft. Jtest manuals version 5.1. Online manual, 2006. `http://www.parasoft.com`.

[13] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pages 553–568, 2003.

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

[15] S. Koushik, M. Darko, and A. Gul. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[16] X. Liu, H. Liu, B. Wang, P. Chen, and X. Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *Proc. ASE*, pages 337–341, 2005.

[17] A. Orso and B. Kennedy. Selective capture and replay of program executions. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[18] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. ECOOP*, pages 504–527, 2005.

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[20] QuickGraph: A 100% c# graph library with graphviz support, version 2.0, 2008. `http://www.codeproject.com/KB/miscctrl/quickgraph.aspx`.

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

[22] Y. Song, S. Thummalapenta, and T. Xie. UnitPlus: Assisting Developer Testing in Eclipse. In *Proc. ETX*, 2007.

[23] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. ASE*, pages 204–213, 2007.

[24] S. Thummalapenta and T. Xie. Mining Exception-Handling Rules as Conditional Association Rules. In *Proc. ICSE*, 2009.

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

[26] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

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

[28] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. ICDE*, pages 79 – 88, 2004.

[29] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.

[30] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.