Synthesizing Method Sequences for High-Coverage Testing *

Suresh Thummalapenta[1], Tao Xie[2], Nikolai Tillmann[3], Jonathan de Halleux[3], Zhendong Su[4]

[1]IBM Research, Bangalore, India
[2]Department of Computer Science, North Carolina State University, Raleigh, USA
[3]Microsoft Research, Redmond, USA
[4]Department of Computer Science, University of California, Davis, USA
surthumm@in.ibm.com, xie@csc.ncsu.edu, {nikolait, jhalleux}@microsoft.com, su@cs.ucdavis.edu

Abstract

High-coverage testing is challenging. Modern object-oriented programs present additional challenges for testing. One key difficulty is the generation of proper method sequences to construct desired objects as method parameters. In this paper, we cast the problem as an instance of program synthesis that automatically generates *candidate programs* to satisfy a user-specified *intent*. In our setting, candidate programs are method sequences, and desired object states specify an intent. Automatic generation of desired method sequences is difficult due to its large search space—sequences often involve methods from multiple classes and require specific primitive values. This paper introduces a novel approach, called *Seeker*, to intelligently navigate the large search space. Seeker synergistically combines static and dynamic analyses: (1) dynamic analysis generates method sequences to cover branches; (2) static analysis uses dynamic analysis information for *not-covered branches* to generate candidate sequences; and (3) dynamic analysis explores and eliminates statically generated sequences. For evaluation, we have implemented Seeker and demonstrate its effectiveness on four subject applications totalling 28K LOC. We show that Seeker achieves higher branch coverage and def-use coverage than existing state-of-the-art approaches. We also show that Seeker detects 34 new defects missed by existing tools.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques—*Object-oriented programming*; D.2.5 [Software Engineering]: Testing and Debugging—*Symbolic execution*; D.2.6 [Software Engineering]: Programming Environments—*Integrated environments*;

General Terms: Languages, Experimentation

Keywords: Object-oriented testing, Symbolic execution

1. Introduction

High-Coverage Testing. An important goal of software testing is to achieve full or at least high coverage (either structural coverage such as branch coverage or data flow coverage such as def-use coverage) of the code under test. Achieving high coverage of object-oriented code requires desired object states for the receiver or arguments of a method under test (MUT). These desired object states help cover `true` or `false` branches of the conditional statements (such as `if` statements) in the MUT. For example, consider the two classes from the C# QuickGraph [36] library shown in Figure 1. A desired object state for covering the `true` branch of Statement 24 (Branch B4) in Figure 1 is that the `graph` object should include at least one edge.

There exist two common approaches for producing desired object states: *sequence generation* [8, 15, 17, 32, 35, 43, 45] and *direct construction* [3]. With sequence generation, desired object states are produced via generating method sequences that create and mutate objects, while with direct construction, desired object states are produced via directly setting values to member fields. In this paper, we adopt the general sequence-generation approach since directly setting values to member fields such as private fields of a class can easily lead to invalid object states. For example, the following method sequence (S1) produces the preceding desired object state for the `graph` object, thereby covering B4.

```
00: AdjacencyGraph ag = new AdjacencyGraph();
01: Vertex v1 = new Vertex(0);
02: ag.AddVertex(v1);
03: ag.AddEdge(v1, v1);
```

In this sequence, `AddVertex` should precede `AddEdge` to satisfy the requirement that the vertices passed as arguments should already exist in the graph object (Statements 7 and 10).

* The majority of the research reported in this submission was conducted while the first author was associated with North Carolina State University.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

```
00:class AdjacencyGraph : IVEListGraph {
01:  private Collection edges;
02:  private ArrayList vertices;
03:  public void AddVertex(IVertex v){
04:     vertices.Add(v); // B1
05:  }
06:  public Edge AddEdge(IVertex v1, IVertex v2){
07:     if (!vertices.Contains(v1))
08:        throw new VNotFoundException("");
09:     // B2
10:     if (!vertices.Contains(v2))
11:        throw new VNotFoundException("");
12:     // B3
13:     // create edge
14:     Edge e = new Edge(v1, v2);
15:     edges.Add(e);
16:  } ...
17:}

//UDFS:UndirectedDepthFirstSearch
18:class UDFSAlgorithm {
19:  private IVEListGraph graph;
20:  private bool isComputed;
21:  public UDFSAlgorithm(IVEListGraph g){
22:       ... }
23:  public void Compute(IVertex s){ ...
24:     if(graph.GetEdges().Size() > 0){ // B4
25:        isComputed = true;
26:        foreach (Edge e in graph.GetEdges()){
27:           ... // B5
28:        }
29:     }
30:  } ...
31:}
```

Figure 1. Two classes from C# QuickGraph library [36].

Program Synthesis. In this paper, we cast the problem of generating method sequences as an instance of general *program synthesis* that automatically generates *candidate programs* to satisfy a user-specified *intent*. The intent can be expressed in various forms such as high-level specifications, natural language, or input-output examples. Based on the intent, synthesizers, unlike compilers that perform transformations, search for programs that satisfy the user-specified intent over a space of all possible candidate programs.

Automatic program synthesis has seen interesting advances in recent years due to the availability of more advanced computing resources and better reasoning techniques such as SMT solvers [47]. In contrast to previous work [12] that focuses on synthesizing algorithms such as sorting or bit-manipulation routines, this paper focuses on synthesizing object-oriented programs that involve method sequences. In particular, we accept a user-specified intent in the form of a desired object state and automatically synthesize a method sequence that produces the desired object state. For example, given the user-specified intent as a conditional branch

```
Client Code:
00: public static void foo(UDFSAlgorithm udfs) {
01:    ...
02:    if(udfs.GetIsComputed()) {
03:       ... // B6
04:    }
05:    // B7
06: }
```

Figure 2. User-specified intent expressed as a desired object state.

```
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph();
03: ag.AddVertex(s1);
04: ag.AddEdge((IVertex)s1, (IVertex)s1);
05: UDFSAlgorithm ud = new UDFSAlgorithm(ag);
06: ud.Compute((IVertex)null);
```

Figure 3. An example method sequence.

(such as Branch B6 in Statement 2, Figure 2) that describes a desired object state, the goal of our approach is to automatically synthesize a method sequence (such as the sequence shown in Figure 3) that produces the desired object state.

Challenges. Automatic synthesis of target sequences that produce a desired object state, is challenging due to three major factors. First, target sequences often include methods from multiple classes, resulting in a large search space of candidate sequences. Second, target sequences require specific primitive values that help exercise desired paths in the code under analysis. Third, object-oriented programming features such as encapsulation pose additional challenges, since values cannot be directly set to member fields.

Our Approach. To address these preceding challenges, we propose a novel systematic approach, called *Seeker*, that intelligently navigates the large search space via synergistically combining static and dynamic analyses. We next explain why either pure dynamic or static analysis alone cannot address this problem. A major issue with pure dynamic-analysis-based approach is that dynamic analysis does not have the knowledge of methods that are not yet explored. Furthermore, it is not possible to explore all methods especially in real-world applications, since real-world applications often include classes and methods from system libraries such as .NET framework libraries.

On the other hand, static analysis alone cannot generate target sequences due to imprecision of static analysis. For example, consider generating the desired object state produced by the preceding Sequence S1 using static analysis alone. Static analysis, being conservative, identifies three methods (`AddVertex`, `RemoveVertex`, and `ClearVertex` of the `AdjacencyGraph` class) that modify the field `vertices` as candidates for Statement 2. For simplicity, we did not show

methods `RemoveVertex` and `ClearVertex` in Figure 1. Similarly, static analysis identifies six candidates for Statement 3, resulting in a total of 18 (3 * 6) candidate sequences. Among these candidates, static analysis alone cannot identify the target sequence due to its imprecision. Although dynamic analysis can be used to identify the target sequence among these candidates, the number of such candidate sequences could be quite high in practice.

To address these aforementioned challenges, Seeker includes three steps that form a feedback loop between dynamic and static analyses. First, given a desired object state in the form of a conditional branch, dynamic analysis attempts to generate a target sequence. Second, if dynamic analysis fails to generate the target sequence, static analysis uses information for *not-covered branches* from dynamic analysis to generate candidate sequences. Third, dynamic analysis explores and eliminates statically generated sequences. For example, in Statement 2 of S1, Seeker statically identifies three methods (`AddVertex`, `RemoveVertex`, and `ClearVertex`) as candidates. Seeker next uses dynamic analysis to filter out `RemoveVertex` and `ClearVertex` methods that do not help produce the desired object state. This feedback loop between static and dynamic analyses is the key essence of Seeker and helps systematically explore a potentially large space of candidate sequences, thereby scaling Seeker to large real-world applications. Furthermore, Seeker stores the knowledge gained regarding individual method calls while generating a target sequence, and reuses the knowledge while generating other target sequences, thereby increasing efficiency.

To handle encapsulation in object-oriented programs, Seeker includes a novel technique based on method-call graphs. A method-call graph is a directed graph that includes caller-callee relations among methods. This technique helps synthesize sequences that generate desired values for member fields (both primitive and non-primitive) including private fields.

Evaluation. We developed a prototype based on our Seeker approach for object-oriented test generation. We compared our approach with two state-of-the-art test-generation approaches: Pex [41] and Randoop [35] that are representative of dynamic-symbolic-execution-based and random approaches, respectively. Our evaluation results show that Seeker achieves higher coverage (both structural and data-flow) than Pex and Randoop. Achieving such higher coverage compared to Pex and Randoop is significant, since the branches that are not covered by these approaches are generally quite hard to cover.

This paper makes the following major contributions:

- A novel approach, called Seeker, that accepts a user-specified intent in the form of a desired object state and automatically synthesizes a method sequence that produces the desired object state.
- An application of our approach to automatically generate test inputs for object-oriented programs and a prototype implementation based on an existing test-generation approach [41].
- A technique based on method-call graphs to handle encapsulation. Our technique also effectively handles private member fields that are of non-primitive types.
- Evaluation results on four popular applications (totalling 28 KLOC) to show the effectiveness of Seeker approach. Our results show that Seeker achieves 12% (653 new branches) and 26% (1571 new branches) higher branch coverage than Pex and Randoop, respectively. Our results also show that Seeker achieves 15.7% (428 pairs) and 15.3% (416 pairs) higher def-use coverage than Pex and Randoop, respectively. Seeker also detects 34 new defects, including an infinite loop defect in QuickGraph [36].

The rest of the paper is organized as follows. Section 2 presents background on existing test-generation approaches. Section 3 explains our Seeker approach with illustrative examples. Section 4 presents formal definitions of terms used in the paper. Section 5 presents the key algorithms of Seeker. Section 6 describes implementation details of the prototype developed for our approach. Section 7 presents the evaluation results. Section 8 discusses the limitations of our approach. Section 9 presents the related work. Finally, Section 10 concludes.

2. Background

In this section, we present two state-of-the-art test generation approaches Pex [41] and Randoop [35] that are used as representative approaches for systematic and random approaches, respectively, in the rest of the paper.

Pex. Pex [41] is a systematic approach based on a test-generation technique, called Dynamic Symbolic Execution (DSE) [7, 11, 19, 22, 41]. DSE is a recent state-of-the-art test generation technique that explores an MUT and generates test inputs that can achieve high structural coverage of the MUT. Pex, developed based on DSE, explores an MUT with default inputs. During exploration, Pex collects constraints on inputs from the predicates in branch statements. Pex negates collected constraints and uses a constraint solver to generate new inputs that guide future program explorations along different paths. To generate method sequences, Pex uses a simple heuristic-based approach that generates fixed sequences based on static information of constructors and other methods (of classes under test) that set values to member fields, hopefully helping produce desired object states.

Randoop. Randoop [35] is a random approach that generates sequences incrementally by randomly selecting method calls. For each randomly selected method call, Randoop uses random values and previously generated sequences for primi-

tive and non-primitive arguments, respectively. For each generated test input, Randoop avoids reusing or extending previously generated sequences that throw uncaught exceptions.

3. Example

We next explain our approach using the same illustrative examples shown in Figure 1. The figure shows two classes under test `AdjacencyGraph` and `UDFSAlgorithm` from the QuickGraph library [36]. `AdjacencyGraph` represents a graph structure including vertices and edges, which are added using `AddVertex` and `AddEdge`, respectively. `UDFSAlgorithm` performs an undirected depth first search on the graph structure. We added an additional method `IsComputed` for illustrative purposes. Consider the `foo` method (Figure 2), where the user-specified intent is expressed as an `if` condition (Statement 2), describing the desired object state that `isComputed` should be `true`. Here, synthesizing a method sequence that produces the desired object state can be transformed as a testing problem of generating a test input that covers the `true` branch (B6) of Statement 2. A necessary requirement to achieve the desired object is that the `graph` object in `UDFSAlgorithm` should contain both vertices and edges.

We first present the branch coverage achieved by Pex and Randoop on classes shown in Figures 1 and 2 and next describe our Seeker approach. The test inputs generated by Randoop and Pex achieved branch coverage of 36.8% (21 out of 57) and 35.1% (20 out of 57), respectively. The reason for low coverage is that neither Randoop nor Pex could satisfy the requirement of `AddEdge` to successfully add an edge to the graph object (Branch B3 in Statement 12 of Figure 1). Therefore, neither Pex nor Randoop could generate a sequence that helps cover Branch B6 in Statement 2 (Figure 2). As shown through this example, it is quite challenging to achieve high branch coverage of these classes under test due to the requirement of complex sequences. Such requirement is often encountered when testing object-oriented code.

We next present how our Seeker approach achieves high branch coverage by synthesizing sequences using a combination of dynamic and static analyses. In particular, Seeker leverages DSE for dynamic analysis and applies DSE to generate a target sequence. If DSE cannot generate the target sequence, Seeker statically analyzes the branches that are not covered by DSE and synthesizes method sequences. Seeker next uses DSE with the assistance of statically synthesized sequences. In our approach, we use Pex, which is based on DSE, for dynamic analysis. Although we describe our approach in the context of Pex, our approach is independent of Pex and can be used to assist any other DSE-based approach [1]. Initially, Seeker applies DSE to explore Branch B6 in the `foo` method. DSE explores `foo` but fails to generate a target sequence that covers Branch B6.

Seeker statically analyzes B6 and suggests B4 (in the `Compute` method) as a pre-target branch that could help cover B6. In particular, the sequence that helps cover B4 can be leveraged to cover B6 as well. Due to imprecision of static analysis, Seeker may suggest more than one pre-target branches and not all those pre-target branches can help cover B6. To address this issue, Seeker applies DSE on pre-target branches and filters out irrelevant pre-target branches. Since DSE alone cannot cover B4, Seeker in turn uses static analysis to identify further pre-target branches for B4. This feedback loop eventually identifies the pre-target branches as follows: "B6 $\Leftarrow$ B4 $\Leftarrow$ B3 $\Leftarrow$ B2 $\Leftarrow$ B1". Here, the notation "B6 $\Leftarrow$ B4" indicates that B4 is a pre-target branch for B6.

Consider that Seeker successfully covered B3. In this scenario, Seeker generates the following sequence:

```
01: AdjacencyGraph ag = new AdjacencyGraph();
02: Vertex v1 = new Vertex(0);
03: ag.AddVertex(v1);
04: ag.AddEdge(v1, v1);
```

Seeker next uses this sequence to assist DSE for covering the next target branch B4 and the process continues. Figure 3 shows the final target sequence (generated by Seeker) that covers Branch B6. The sequence includes four classes and six method calls. Using this sequence, Seeker achieved 84.2% (48 out of 57) branch coverage. The remaining not-covered branches are related to the event handling mechanism, which is currently not handled by our implemented prototype. It is quite challenging to generate such sequences either randomly or using heuristics, since these three classes include 39 methods. However, the feedback loop between static analysis (that suggests candidate methods) and dynamic analysis (that identifies correct candidate method and generates data) generates target sequences, thereby achieving high structural coverage of the code under test.

4. Problem Formulation

This section formalizes the problem of method sequence generation and introduces the terminology that we use throughout the rest of the paper.

For a given application $\mathcal{A}$ under test, let $\mathcal{C}$ and $\mathcal{M}$ denote its sets of classes and methods, respectively. Let $PrimTy$ and $PrimVal$ represent the set of all primitive types, such as `int` or `bool`, and primitive values, respectively. Each method $M \in \mathcal{M}$ is represented by the method's type signature: $C \times T_1 \times \cdots \times T_n \rightarrow T$, where $C \in \mathcal{C}$ is the type of the receiver object, $T_i \in \mathcal{C} \cup PrimTy$ denotes the type of the i-th argument for $i \in [1..n]$, and $T \in \mathcal{C} \cup PrimTy \cup \{void\}$ denotes the type of the return value. Since $T_i \in \mathcal{C} \cup PrimTy$, M's arguments can be either primitive values or objects.

DEFINITION 4.1. Method Sequence (MCS). A *method sequence* is a sequence of method calls $(m_1, \ldots, m_r)$, such that for $i \in [1..r]$, we have

- $m_i = o.M_i(a_1, \ldots, a_n)$ where $\mathcal{M} \ni M_i : C \times T_1 \times \cdots \times T_n \rightarrow T$. In other words, m_i is *well-typed*: $o : C$[1] and $a_j : T_j$ for all $j \in [1..n]$;
- $o = ret(m_k)$ for some $k \in [1..i)$, and for all $j \in [1..n]$, $a_j \in PrimVal \vee a_j = null \vee a_j = ret(m_l)$ for some $l \in [1..i)$. In other words, the sequence is *well-formed* with the proper data dependence.

In the preceding definition, $ret(m_k)$ denotes the return value of the method m_k. Also note that for brevity of presentation, we do not explicitly model constructor calls and static methods in the preceding definition. To model them, one can simply drop the conditions on the receiver object o.

For each method call $m_i = o.M_i(a_1, \ldots, a_n)$ in an MCS, the receiver object o should be the return object $ret(m_k)$ of another method call m_k that precedes m_i within the sequence. Furthermore, each m_i in the MCS can have either primitive or non-primitive arguments. For primitive arguments, the preceding definition requires that the arguments should take on primitive values of the corresponding types, such as `true` for the `bool` type. For non-primitive arguments, they must be $null$ or return values of some preceding method calls within the sequence. For example, in the sample sequence shown in Figure 3, the non-primitive argument `s1` of `AddVertex` in Statement 3 is the return value of another preceding method call `new Vertex()` in Statement 1. Our definition ensures that method sequences are well-formed, and executable code can be generated directly from those sequences.

It is helpful to also define the notion of a *sequence skeleton*. Intuitively, a sequence skeleton is an MCS except that primitive arguments are not required to take on concrete values. The definition below provides a precise description.

DEFINITION 4.2. **Skeleton (SKT).** A *sequence skeleton* is a sequence of method calls $(m_1, \ldots, m_r)$, such that for $i \in [1..r]$, we have

- $m_i = o.M_i(a_1, \ldots, a_n)$ where $\mathcal{M} \ni M_i : C \times T_1 \times \cdots \times T_n \rightarrow T$.
- $o = ret(m_k)$ for some $k \in [1..i)$, and for all $j \in [1..n]$, $a_j : PrimTy \vee a_j \in PrimVal \vee a_j = null \vee a_j = ret(m_l)$ for some $l \in [1..i)$.

The definition for SKT is essentially the same as that for MCS, except that some values of primitive-type arguments are not required: $a_j \in PrimVal$ for MCS versus $a_j : PrimTy \vee a_j \in PrimVal$ for SKT.

DEFINITION 4.3. **Target Branch (TB).** A *target branch* is a `true` or `false` branch of a conditional statement[2].

In our setting of program synthesis, we use a not-covered target branch to denote the user intent.

[1] The notation $o : C$ indicates that the receiver object o is of type C.

[2] We model a `switch` statement as a series of `if-then-else` statements.

DEFINITION 4.4. **Method Sequence Synthesis.** Given a method under test $M \in \mathcal{M}$ and a target branch tb within M, synthesize a method sequence $(m_1, \ldots, m_r)$ that constructs the receiver object and arguments of M and drives M to successfully cover tb.

5. Seeker Algorithm

Algorithms 1 and 2 show the two key algorithms `DynAnalyzer` (dynamic analysis) and `StatAnalyzer` (static analysis) of our Seeker approach. Seeker leverages DSE for dynamic analysis to synthesize sequences that can cover a given target branch tb. In particular, given a target branch, Seeker first applies DSE and checks whether DSE can generate a sequence (referred to as target sequence) that covers the target branch. In case, DSE cannot generate the target sequence, Seeker uses static analysis to synthesize skeletons and again applies DSE to generate data, forming a *feedback* loop. Here, DSE assists static analysis in two major ways. First, DSE helps generate data for skeletons synthesized by static analysis. Second, DSE eliminates candidate methods (identified by static analysis) that do not help cover the target branch. The novelty of Seeker is that this feedback loop helps overcome individual limitations of static and dynamic analyses, thereby effectively synthesizing sequences[3]. We next explain each algorithm in detail using illustrative examples shown in Figures 1 and 2. Consider that the `DynAnalyzer` is invoked with the target branch tb as B6 in Figure 2 and $inpseq$ as `null`. Here, Branch B6 represents the `true` branch of Statement 2 (Figure 2).

5.1 DynAnalyzer Algorithm

`DynAnalyzer` accepts a target branch tb and an input sequence $inpseq$ as inputs, and generates a target sequence that covers tb. Initially, `DynAnalyzer` identifies the method m that includes tb (using `GetMethod`). `DynAnalyzer` next appends the method m to $inpseq$ using `AppendMethod` (Lines 1 and 2) and generates the skeleton $tmpskt$. Since `AppendMethod` does not know the parameter values for m, `AppendMethod` uses symbolic values as parameters for m in the skeleton $tmpskt$. If the method m is a non-static method and there exists no constructor in $inpseq$, `AppendMethod` automatically adds relevant constructors to $tmpskt$. For the tb B6, when `DynAnalyzer(B6, null)` is invoked, `AppendMethod` returns the skeleton `foo(<sym>)`, where `<sym>` represents a symbolic variable. Here, no constructor is added to $tmpskt$, since `foo` is a static method.

`DynAnalyzer` next applies DSE (referred to with a function call `DSE` in Line 3 of Algorithm 1) to explore `tmpskt` for generating a target sequence that covers tb. DSE accepts

[3] An astute reader can identify that, in a few scenarios, the recursion between our two algorithms can result in an infinite loop. Seeker includes techniques for detecting and avoiding such infinite loops. For brevity, we ignore such details while presenting our algorithms.

two arguments of types SKT and TB as inputs. DSE outputs three values described as follows:

- `targetseq` of type MCS: MCS that covers the given tb of type TB or `null`
- `CovB`: Set of covered branches
- `NotCovB`: Set of not covered branches

During exploration of DSE, if DSE happens to generate a target sequence, then DSE returns the target sequence; otherwise, it returns `null`. Apart from the target sequence, DSE also returns covered (`CovB`) and not-covered branches (`NotCovB`) in the method m. For example, when DSE is invoked with the skeleton `foo(<sym>)`, DSE generates a sequence that helps cover Branch B7, but not Branch B6. The reason is that DSE could not generate a target sequence that can help cover B6. Therefore, DSE returns `null`, {B7}, and {B6} for `targetseq`, `CovB`, and `NotCovB`, respectively. Note that the set `CovB` $\cup$ `NotCovB` does not represent the entire set of branches in the method m. The primary reason is that DSE, being a pure dynamic analysis technique, does not have the knowledge of those branches where both the branch and its alternative branch[4] are not explored by DSE.

After exploration using DSE, there can be three possible scenarios for the target branch.

- Scenario 1: The target branch tb is covered. In this scenario, `DynAnalyzer` returns `targetseq`.
- Scenario 2: The target branch tb is not covered and $tb \in NotCovB$. This scenario happens when DSE successfully covers the alternative branch of tb and could not cover tb. In this scenario, `DynAnalyzer` invokes `StatAnalyzer` to generate a sequence that can help cover tb.
- Scenario 3: The target branch tb is not covered and $tb \notin NotCovB$. This scenario happens when DSE could not cover all the dominant branches of tb in the method m. In this scenario, `DynAnalyzer` invokes `ComputeDominants` to identify dominant branches. In particular, `Compute Dominants` first identifies the dominant branch, referred to as *prime dominant*, whose alternative branch is covered by DSE. `ComputeDominants` next identifies all other dominant branches of tb between the prime dominant and tb. `DynAnalyzer` next recursively invokes itself for each such dominant branch starting from the prime dominant branch. `DynAnalyzer` returns a method sequence if all dominant branches are covered along with tb; otherwise, it returns `null`.

5.2 StatAnalyzer Algorithm

`StatAnalyzer` analyzes a target branch tb and identifies other branches (referred to as *pre-target* branches) that can help cover tb. We first explain the two major functions `DetectField` (Line 1) and `SuggestTargets` (Line 2) used by `StatAnalyzer`. Since examples shown in Figures 1 and 2 are complex, we use a simple example shown in Figure 4 to explain `DetectField` and `SuggestTargets`. Consider that `StatAnalyzer` is invoked with tb as Branch B8 and $inpseq$ as `null`.

[4] Given a branch b (such as the `true` branch) of a conditional statement, we use *alternative branch* to refer to the other branch (such as the `false` branch) of that conditional statement.

Algorithm 1 $DynAnalyzer(tb, inpseq)$

Require: tb of type TB
Require: $inpseq$ of type MCS
Ensure: $targetseq$ of type MCS covering tb or `null`

```
1:  Method m = GetMethod(tb)
2:  SKT tmpskt = AppendMethod(inpseq, m)
3:  DSE(tmpskt, tb, out targetseq, out CovB, out
    NotCovB)
4:  //Scenario 1
5:  if tb ∈ CovB then
6:      return targetseq
7:  end if
8:
9:  //Scenario 2
10: if tb ∈ NotCovB then
11:     return StatAnalyzer(tb, inpseq)
12: end if
13:
14: //Scenario 3
15: if tb ∉ NotCovB then
16:     List<TB> tblist = ComputeDominants(tb)
17:     for all TB domtb ∈ tblist do
18:         inpseq = DynAnalyzer(domtb, inpseq)
19:         if inpseq == null then
20:             Break
21:         end if
22:     end for
23:     if inpseq ≠ null then
24:         return DynAnalyzer(tb, inpseq)
25:     end if
26: end if
27: return null
```

DetectField. Given a tb, the function `DetectField` precisely identifies the target member field `tfield` that needs to be modified to produce a desired object state for covering tb. It is trivial to identify `tfield` for branches such as `if(stack.size == 10)`, where `tfield` (such as `size`) is directly included in the branch. However, in object-oriented code, branches often involve method calls such as `if(!vertices.Contains(v1))` in Statement 7 (Figure 1) rather than fields. It is challenging to identify target fields in the presence of method calls, since the return statements in these method calls may in turn can include further method calls, where the actual member field is returned.

Algorithm 2 $StatAnalyzer(tb, inpseq)$

Require: A target branch tb
Require: A sequence $inpseq$
Ensure: A sequence $targetseq$ covering tb

```
1: Field tfield = DetectField(tb)
2: List<TB> tblist = SuggestTargets(tfield)
3: for all TB pretb ∈ tblist do
4:     MCS targetseq = DynAnalyzer(pretb, inpseq)
5:     if targetseq ≠ null then
6:         targetseq = DynAnalyzer(tb, targetseq)
7:         if targetseq ≠ null then
8:             return targetseq
9:         end if
10:    end if
11:    //Try other alternative target branches
12: end for
13: return null
```

To address this issue, the function `DetectField` uses an inter-procedural execution trace (hereby referred to as *trace*), gathered during the runtime exploration with DSE. This trace includes the statements executed in each method. `DetectField` performs backward analysis of the trace starting from the method call involved in tb. We use $retvar$ to refer to the variable or value associated with the return statement in a method call. `DetectField` uses the following five steps with respect to $retvar$ to identify $tfield$.

1. If $retvar$ is a member field, `DetectField` identifies $retvar$ as $tfield$. This scenario can happen with methods such as `getter` methods.
2. If $retvar$ is data-dependent on a member field, the function `DetectField` identifies that member field as $tfield$.
3. If $retvar$ is data-dependent on the return of a nested method call, the function `DetectField` repeats these five steps with the nested method call to identify $tfield$.
4. If $retvar$ is control-dependent on a member field, `Detect Field` identifies that member field as $tfield$. This scenario can happen when DSE failed to generate other object states for that member field.
5. If $retvar$ is control-dependent on the return of a nested method call, `DetectField` repeats these five steps with that nested method call to identify $tfield$. The method `HasElements` (Lines 6-9 in Figure 4) shows an example of this scenario, where $retvar$ is control-dependent on the return of another nested method call `stack.size()`. In this scenario, `DetectField` repeats the preceding five scenarios with that method call `stack.size()`.

To illustrate these five steps, consider Branch B8 as tb. Given this tb, `DetectField` applies the preceding steps and detects `_size` (in `ArrayList`) as $tfield$. Here, `Stack` includes a member field `list` of type `ArrayList`. Initially,

```
00: public class IntStack {
01:    private Stack stack;
02:    public IntStack() {
03:       this.stack = new Stack; }
04:    public void Push(int item) {
05:       stack.Push(item); }
06:    public bool HasElements() {
07:       if(stack.size() > 0) { return true; }
08:       else { return false; }
09:    }
10: }
11: public class MyCls {
12:    private IntStack ints;
13:    public MyCls(IntStack ints) {
14:       this.ints = ints; }
15:    public void MyFoo() {
16:       if(ints.HasElements()) {
17:         ...// B8
18:       }
19:    }
20: }
```

Figure 4. An integer stack class.

`DetectField` analyzes the method `IntStack.HasElements`. Since the executed return statement (Statement 8) is control-dependent on a nested method call `Stack.size`, `DetectField` analyzes the `Stack.size` method. Eventually, `DetectField` reaches the getter method that returns `_size` member field of `ArrayList`, and thereby identifies `_size` as $tfield$. Note that, in a few scenarios, there can be multiple $tfields$ for covering tb. However, currently we handle only those tb that can be covered by achieving desired value for a single $tfield$. We plan to handle multiple $tfields$ in our future work.

Along with identifying $tfield$, `DetectField` also captures two other pieces of information. First, `DetectField` identifies the condition on $tfield$ that is not satisfied. For example, `DetectField` identifies "`_size > 0`" (Statement 7) as the condition that should be satisfied to cover tb. `DetectField` applies a constraint solver on the preceding condition to get a desired value for $tfield$. Second, `DetectField` also captures the hierarchy of fields, referred to as field hierarchy, that includes all objects starting from the object enclosing tb to $tfield$. For Branch B8 as tb, the identified field hierarchy is as follows: "FH: **MyCls root** $\Rightarrow$ **IntStack is** $\Rightarrow$ **Stack stack** $\Rightarrow$ **ArrayList list** $\Rightarrow$ **int _size**". This field hierarchy describes that `_size` of type `int` is contained in the object `list` of type `ArrayList`, which is in turn contained in the object `stack` of type `Stack` and so on. Here, `root` represents the object of type `MyCls`. This field hierarchy is used by `SuggestTargets` discussed next. In our setting, inheritance is not an issue, since `DetectField` analyzes the execution trace produced by dynamic analysis, where actual objects are described.

SuggestTargets. Given a target field $tfield$ such as `_size`, its current and desired values, and field hierarchy, the function `SuggestTargets` identifies pre-target branches that need to be covered to cover the original target branch tb. Initially, `SuggestTargets` traverses the field hierarchy and identifies the object $tobject$ (in the field hierarchy) that can be modified to achieve a desired value for $tfield$. The objective of this traversal is to identify the $tobject$ that is nearest to $tfield$ and can be modified by either assigning a value directly or by invoking its public methods. The reason is that the shorter the distance between $tobject$ and $tfield$, the smaller the amount of code that needs to be explored to achieve a desired value for $tfield$. For example, consider the preceding field hierarchy FH. Here, the object **ArrayList list** is near `_size` ($tfield$) compared to **IntStack stack**. However, the `list` object cannot be $tobject$, since `list`, a private member field, cannot be modified directly or by invoking its public methods.

To identify $tobject$, `SuggestTargets` traverses the field hierarchy from `root` and considers the object whose next object cannot be modified either directly or through public methods as $tobject$. For example, `root` is not considered as $tobject$, since `ints` can be modified as `ints` is set through the constructor. For this field hierarchy, `SuggestTargets` identifies `ints` as $tobject$, since `stack` cannot be modified outside the `ints` object.

After identifying $tobject$, `SuggestTargets` identifies methods (and pre-target branches within those methods) that help produce a desired value for $tfield$. Identifying the methods of $tobject$ that modify $tfield$ is non-trivial, since there can be intermediate objects between $tobject$ and $tfield$, as identified by the field hierarchy. To address this issue, `SuggestTargets` uses a novel technique based on *method-call graphs*.

DEFINITION 5.1. **Method-Call Graph.** A *method-call graph* is a stratified-directed graph $G = (V, E)$, where V and E represent the set of all nodes and edges, respectively, such that

- $V = \{root\} \cup V_1 \cup \ldots \cup V_n$. Here, $root$ represents the node corresponding to *tfield*, and for all $i \in [1..n]$, $V_i = \{M_1, \ldots, M_r\}$ represents all nodes at level i. For all $j \in [1..r]$, $M_j \in \mathcal{M}$ and all methods M_j in V_i belongs to the same class or its parent classes;
- $E = \{(root, M_a) \mid M_a \in V_1\} \cup \bigcup_{i \in [1 \ldots n-1]} \{(M_b, M_c) \mid M_b \in V_i \ \wedge M_c \in V_{i+1}\}$. In essence, there can be two kinds of edges. The first kind of edges represent the edges from $root$ to nodes at level 1. An edge between $root$ and $M_a \in V_1$ (nodes at level 1) represent that M_a modifies $tfield$. The second type of edges represent the edges between the nodes in any two successive levels. An edge between $M_b \in V_i$ and $M_c \in V_{i+1}$ represent that M_c invokes M_b.

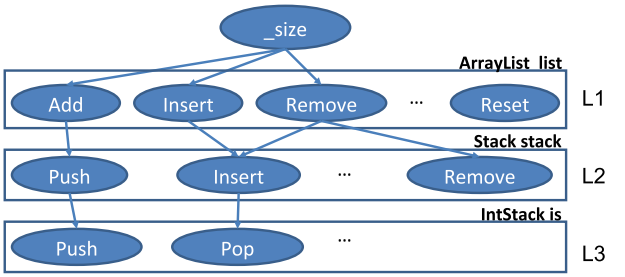

Figure 5. A sample method-call graph.

`SuggestTargets` constructs the method-call graph on demand based on the field hierarchy identified by `DetectField`. Figure 5 shows a sample method-call graph constructed for the field hierarchy FH. The root node of the graph includes `tfield`. The first level of the graph includes the methods (in the declaring class) that modifies $tfield$. Initially, `SuggestTargets` statically analyzes all public methods of the declaring class of $tfield$ to identify the target methods that modify $tfield$. In particular, `SuggestTargets` identifies assignment statements, where $tfield$ is on the left hand side. For example, `SuggestTargets` identifies the methods such as `Add`, `Insert`, and `Reset` of `ArrayList` as target methods for the $tfield$ `_size`. From the second level, the graph includes the methods from the declaring classes of fields in the field hierarchy. The graph includes an edge from a method M_b in one level to a method M_c in the next level, if M_b is called by M_c. For example, `Stack.Push` invokes the `List.Add` method and the corresponding edge is shown from Levels L1 to L2.

`SuggestTargets` next traverses constructed graph from the top to bottom to identify methods that can be invoked on $tobject$ to achieve a desired value for $tfield$. Furthermore, `SuggestTargets` identifies pre-target branches within each method that need to be covered to invoke the method call of the preceding level. For example, `SuggestTargets` identifies that the `IntStack.Push` method can help achieve a desired value for $tfield$. Furthermore, `SuggestTargets` identifies the pre-target branch in `IntStack.Push` that helps invoke `Stack.Push` method. This pre-target branch is considered as a new target branch that needs to be covered, so as to cover the original target branch tb. `SuggestTargets` returns several candidate pre-target branches.

StatAnalyzer. We next describe the `StatAnalyzer` algorithm. Given a target branch tb, `StatAnalyzer` first identifies other pre-target branches that need to be covered using `DetectField` and `SuggestTargets`. Due to imprecision of static analysis, not every pre-target branch identified by `SuggestTargets` can help cover tb. For example, along with `IntStack.Push`, `SuggestTargets` can also identify that `IntStack.Pop` can help cover tb. To address this issue, `StatAnalyzer` invokes `DynAnalyzer` to generate a sequence that covers these pre-target branches. If `DynAnalyzer` successfully covers any pre-target branch, `StatAnalyzer`

```
07: DynAnalyzer(B1, null)
06: StatAnalyzer(B2, null)
05: DynAnalyzer(B2, null)
04: DynAnalyzer(B3, null)
03: StatAnalyzer(B4, null)
02: DynAnalyzer(B4, null)
01: StatAnalyzer(B6, null)
00: DynAnalyzer(B6, null)
```

Figure 6. A snapshot of the execution stack.

Subject	Version	# Classes	# Methods	KLOC	# Downloads
QuickGraph	1.0	88	634	5.1	62,734
Dsa	0.6	27	308	3.3	6,724
XUnit	1.6.1	151	1267	11.9	86,702
NUnit	2.5.7	225	2344	8.1	193,563
TOTAL		**491**	**4553**	**28.4**	**349,723**

Table 1. Subjects and their characteristics.

uses the generated sequence to cover the original target branch tb.

5.3 Example

We next describe how Seeker generates the sequence shown in Figure 3 for the tb B6 in Figure 2. Initially, Seeker invokes `DynAnalyzer(B6, null)`. Since B6 $\in$ $NotCovB$ after exploration with DSE, `DynAnalyzer(B6, null)` invokes `StatAnalyzer(B6, null)`. `StatAnalyzer` analyzes B6 and identifies B4 in the `Compute` method as a pre-target branch. Therefore, `StatAnalyzer(B6, null)` invokes `DynAnalyzer(B4, null)`. This process continues and eventually reaches a stage where `StatAnalyzer(B2, null)` invokes `DynAnalyzer(B1, null)`. Figure 6 shows the contents of the execution stack when `DynAnalyzer(B1, null)` is invoked. At this point, `DynAnalyzer(B1, null)` covers Branch B1, and returns the following sequence, say S2:

```
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph();
03: ag.AddVertex(s1);
```

`StatAnalyzer(B2, null)` next invokes `DynAnalyzer(B2, S2)`, which successfully generates a sequence that covers B2, resulting in the following sequence, say S3:

```
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph();
03: ag.AddVertex(s1);
04: ag.AddEdge(s1, null);
```

Seeker continues further and eventually covers the original target branch B6, thereby synthesizing the method sequence shown in Figure 3.

6. Implementation

We implemented a prototype of Seeker using Pex 0.92 [41] for dynamic analysis. Pex provides extensible Application Programming Interfaces (APIs) that can be overridden to add new functionalities. Given a tb, Seeker launches Pex multiple times to synthesize target sequences. Seeker uses a file-based repository to cache information across multiple launches of Pex. After each launch, Pex returns branches that are not yet covered due to lack of desired method sequences. Seeker analyzes those not-yet-covered branches statically and generates skeletons.

Seeker persists these generated skeletons in the repository and reuses them during the next launch of Pex. Seeker relaunches Pex with new skeletons to generate primitive data for the method calls in the skeletons. Due to these multiple launches of Pex and using a file-based repository, our current prototype has high runtime overhead. We expect that the runtime performance of our prototype can be significantly improved by implementing Seeker within Pex or by adopting a memory-based repository, which is left as our immediate future work.

Our `DynAnalyzer` algorithm assumes that DSE (shown in Step 3) could not cover the target branch tb due to the lack of proper sequence. However, in practice, there can be other issues such as environment dependency due to which DSE could not cover tb. Seeker identifies such other issues and returns from Step 3 of `DynAnalyzer`. Our current open-source prototype can be downloaded from `http://pexase.codeplex.com/releases/view/50822`.

7. Evaluation

To show the effectiveness of our Seeker approach, we applied Seeker to automatically generate test inputs for object-oriented programs. We also compared our approach with two categories of approaches: *random* and *DSE-based approaches*. We used two state-of-the-art tools Randoop [35] and Pex [41] as representative tools for random and DSE-based approaches, respectively. We applied all three approaches on four popular real-world applications. Along with these two approaches, we compared our results with the results of existing manually written tests available with the subject applications. We also compared Seeker with our previous approach, called MSeqGen [40]. MSeqGen, unlike Seeker, generates method sequences based on how method calls are used in practice. Since Seeker and MSeqGen complement each other, this comparison helps show the benefits and limitations of Seeker, MSeqGen, and their combined approach. More details of the subjects and results of our evaluations are available at `http://research.csc.ncsu.edu/ase/projects/seeker/`. All evaluations were conducted on a machine with 3.33GHz Intel Core 2 Duo processor with 4 GB RAM.

Subject	Namespace	# Branches	Randoop			Pex			Seeker			Manual	
			# Tests	Cov	Time	# Tests	Cov	Time	# Tests	Cov	Time	# Tests	Cov
QuickGraph	OVERALL	1119	10140	51.2	0.2	334	31.6	4.4	1923	**68.2**	3.2	21	26
	Algorithms	572	-	38.1	-	-	24.8	-	-	52.1	-	-	24.8
	Collections	269	-	87.7	-	-	17.8	-	-	94.0	-	-	11.2
	... (5 more)												
Dsa	OVERALL	665	10493	14.9	1.0	552	83.8	3.7	961	**90**	0.9	298	93.2
	Algorithms	198	-	41.9	-	-	100	-	-	100	-	-	88.3
	DataStructures	433	-	0	-	-	76.7	-	-	86.4	-	-	90.8
	... (2 more)												
xUnit	OVERALL	2379	10148	24.9	6.1	1265	38.6	4.5	1360	41.1	2.0	282	62.7
	Gui	432	-	34.3	-	-	40.8	-	-	**46.1**	-	-	17.8
	Sdk	706	-	25.1	-	-	35.6	-	-	**40.2**	-	-	86.3
	... (6 more)												
NUnit	Util	1810	10129	16.1	1.7	816	35.3	7.5	1804	**43.5**	3.7	319	63.9
TOTAL		**5973**	**40910**	**26**	**9.0**	**2967**	**41.3**	**20.1**	**6048**	**52.3**	**9.8**	**920**	**59.2**

Table 2. Branch coverage achieved by Randoop, Pex, Seeker, and manually written tests.

7.1 Research Questions

In our evaluation, we addressed the following research questions.

RQ1: How much higher percentage of branch coverage is achieved by Seeker compared to Randoop and Pex, respectively? This research question helps show that Seeker performs better than Randoop and Pex in achieving high structural coverage such as branch coverage of the code under test.

RQ2: How much higher percentage of def-use coverage is achieved by Seeker compared to Randoop and Pex, respectively? This research question helps show that Seeker performs better than Randoop and Pex in achieving high dataflow coverage [10] such as def-use coverage of the code under test.

RQ3: How many new defects are detected by Seeker compared to Randoop and Pex, respectively? This research question helps address whether Seeker has higher defect-detection capabilities compared to Randoop and Pex, respectively.

RQ4: How high percentage of branch coverage is achieved by Seeker, MSeqGen [40], and their combination?

7.2 Subjects

We used four popular applications as subjects in our evaluations. Table 1 shows their various characteristics, such as the number of classes and methods, their site, and number of downloads. QuickGraph [36] is a popular C# graph library that provides various graph data structures and algorithms such as depth-first search. Data structures and algorithms (Dsa)[5] provides various data structures, complementing those from the .NET framework. xUnit[6] and NUnit[7] are widely used open source unit testing frameworks for all .NET languages. For NUnit, we focused on applying all three approaches on its core component, the `util` namespace (including 8.1 KLOC). We used these applications as subject applications, since these applications are popularly used (as shown by their total downloads count as of *March 2011* in Column "Downloads") and also by previous work [40]. The subjects include a total of 28 KLOC.

7.3 Evaluation Setup

We next describe our evaluation setup for addressing the preceding four research questions. Seeker and Pex accept Parameterized Unit Tests (PUTs) [42] as input and generate conventional unit tests. Unlike conventional unit tests, PUTs accept parameters. Since PUTs are not available with our subjects, we automatically generated PUTs for each public method by using the *PexWizard* tool, which is provided with Pex. A PUT generated for the `Compute` method in Figure 1 using PexWizard is shown below.

```
00:[PexMethod]
01:public void Compute01(
02: [PexAssumeUnderTest]UDFAlgorithm target,
03: [PexAssumeUnderTest]IVertex s) {
04:   target.Compute(s);
05:  Assert.Inconclusive("this test needs review");
06:}
```

We first applied Seeker on PUTs generated for each subject application. We measured four metrics for generated test inputs: the branch coverage, def-use coverage, number of distinct defects detected, and the time taken. For *branch coverage*, we used a coverage measurement tool,

[5] `http://dsa.codeplex.com/`

[6] `http://xunit.codeplex.com/`

[7] `http://www.nunit.org/`

called NCover[8], to measure the branch coverage achieved by generated test inputs.

For *def-use coverage*, we developed a tool for C#, called DUCover, based on the techniques described in previous work [31, 34]. The reason for developing this new tool is that, to the best of our knowledge, there exist no def-use coverage measurement tool for C#. In object-oriented code, definitions and uses for member fields can occur in different member methods of classes under analysis. DUCover automatically measures coverage of such def-use pairs based on method sequences among generated test inputs. For *defects*, we measured distinct defects, since multiple failing test inputs could detect the same defect. Section 7.6 presents more details on how we identify defects from failing test inputs.

As mentioned in Section 6, our implementation has runtime performance overhead, since we launch Pex multiple times. To ensure that our results are not biased by the limitations of our implementation, we used customized settings for Pex and Randoop rather than using their default settings, respectively. These customized settings allow Pex and Randoop to run for the same or higher amount of time compared to Seeker. In essence, our settings favor Pex and Randoop compared to Seeker. We used the following customized settings for Pex and measured the three metrics for the test inputs generated by Pex.

```
Timeout = 500 sec. (default:120)
MaxConstraintSolverTime = 10 sec. (default:2)
MaxRunsWithoutNewTests = 2147483647 (default:100)
MaxRuns = 2147483647 (default:100)
```

The values in brackets represent the default values. For example, the default value of the timeout parameter is 120 seconds. Instead, we used 500 seconds for the timeout parameter. For Randoop, the default timeout value is 120 seconds. Since, Randoop is a random approach, we ran Randoop multiple times (each time with the timeout parameter as 120 seconds) for each subject so that the total time is equal or higher than the amount of time taken by Seeker for that subject. However, we observed that Randoop may generate thousands of test inputs that are too many to be compiled within Visual Studio for measuring metric values. Therefore, we limited the number of generated test inputs to $10,000$. For one subject under analysis, although we tried compiling remaining test inputs into several other Visual Studio projects and measured coverage, we found that there was no increase in the branch coverage.

To compare Seeker with MSeqGen, we first applied MSeqGen alone on QuickGraph, which is the only subject previously used for evaluating MSeqGen (integrated with Pex), and measured the three metrics[9]. For the combined approach, we used sequences extracted by MSeqGen as input to Seeker. In this setting, Seeker enhances the sequences extracted by MSeqGen to generate more sequences that could help produce desired object states.

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

[9] We could not apply MSeqGen on other subject applications due to lack of usage information currently with us for these subject applications. In future, we plan to collect this usage information and compare Seeker with MSeqGen on the remaining subjects as well.

Subject	**Randoop**			**Pex**			**Seeker**		
	Avg.	SD	Max	Avg.	SD	Max	Avg.	SD	Max
QuickGraph	21.6	21.6	191	4.4	3.0	14	5.6	2.9	17
Dsa	3.0	2.5	20	2.7	2.0	12	3.2	1.9	12
xUnit	6.1	5.8	65	3.3	4.9	58	2.4	2.0	37
NUnit	4.7	5.0	121	4.1	3.0	20	4.3	2.9	19

Table 3. Statistics of generated sequences.

7.4 RQ1: Coverage

We next address the first research question. Table 2 shows our results for all subject applications. For each subject, due to space constraint, we show results for a few selected namespaces (Column "Namespace") that help provide insights described in subsequent sections, instead of all namespaces. Column "Branches" shows the number of branches in each application. Among the remaining columns, subcolumns "# Tests", "Cov", and "Time" show the number of test inputs generated by each approach (Randoop, Pex, Seeker, and manually written tests), branch coverage achieved, and time taken in hours, respectively. Table 3 shows further details regarding the sequences generated by each approach. Columns "Avg.", "SD", and "Max" show the average lengths, standard deviation, and maximum lengths of sequences generated by each approach, respectively. We next summarize our results.

Randoop. Our results show that Randoop achieved the lowest coverage among all approaches for all applications, except for Quickgraph. For QuickGraph, Randoop achieved higher coverage than Pex. Randoop could not achieve any coverage for the `DataStructures` namespace of Dsa, since Randoop cannot handle generics. Furthermore, Table 3 shows that sequences generated by Randoop are often longer than the sequences generated by other approaches. In summary, our results show that target sequences cannot be generated by combining method calls randomly to form longer sequences.

Pex and Seeker. Our results show that Pex, which is a DSE-based approach, can effectively handle generation of primitive data, but cannot generate target sequences. For example, Pex achieved 100% coverage for the `algorithms` namespace of Dsa. This namespace does not require sequences and includes implementations of various sorting algorithms such as mergesort. On the other hand, Pex achieved only 31.6% for QuickGraph, which requires complex sequences for achieving high coverage. In our evaluations, we used customized settings for Pex instead of default values, thereby favoring Pex compared to Seeker. For example, our settings help Pex run for 20 hours (for all subjects) compared to

Subject	# Def-Use pairs	Randoop		Pex		Seeker		Manual	
		# Covered	%	# Covered	%	# Covered	%	# Covered	%
QuickGraph	892	402	45.1	198	22.2	447	50.1	152	17.0
Dsa	583	0	0	96	16.5	222	38.1	185	31.7
xUnit	1256	196	15.6	316	25.2	357	28.4	24	1.9
TOTAL	**2731**	**598**	**21.9**	**610**	**22.3**	**1026**	**37.6**	**361**	**13.2**

Table 4. Def-Use coverage achieved by Randoop, Pex, Seeker, and manually written tests.

9.8 hours for Seeker. Still, Seeker achieved 12% (653 new branches) higher branch coverage than Pex. Indeed, allowing Seeker to run for longer time could help achieve more coverage. Therefore, our results show that it is difficult to achieve higher coverage by letting Pex run for longer time, showing the significance of our Seeker approach.

Although Seeker achieved higher coverage than Randoop and Pex, the coverage achieved is still not close to 100%. Moreover, coverage achieved by Seeker is lower than manually written tests for all subjects, except for QuickGraph and `Gui` namespace of xUnit. Section 8 discusses limitations on why Seeker could not achieve coverage close to 100%.

7.5 RQ2: Def-Use Coverage

We next address the second research question on whether Seeker achieves higher def-use coverage compared to Pex and Randoop. Table 4 shows the def-use coverage achieved by Randoop, Pex, Seeker, and manually written tests, respectively. We could not apply our DUCover tool on test inputs generated for NUnit, due to a technical limitation of executing NUnit tests using NUnit. Along with def-use coverage, we also measure all-defs coverage to provide more insights. All-defs criteria describe that for each definition in the code under test, some use of this definition is being exercised by a test input. Table 5 shows the all-defs coverage achieved by all approaches for each subject.

Our results show that Seeker achieved higher def-use and all-defs coverage compared to both Pex and Randoop, respectively, for all subjects. The results also show that Seeker achieved higher def-use coverage than manually written tests. A primary reason could be that programmers may not write tests to achieve high def-use coverage. Although Seeker achieved higher def-use coverage than Pex and Randoop, the coverage achieved by Seeker is not close to 100%. There are two major reasons. First, some of the def-use pairs are infeasible. For example, consider the `Deque` class shown in Figure 7. In this class, the `m_deque` field is defined in Statement 7 in the `Clear` method. On the other hand, the `m_deque` field is accessed in Statement 12 in the `DequeFront` method, forming a def-use pair. However, this def-use pair is an infeasible pair, since the `Clear` method sets the value of the `Count` field to zero (Statement 8) and the `DequeFront` method includes an additional condition check (in Statement 11) that throws an exception if the value of the `Count` field is zero. In future work, we plan to identify such infeasible pairs by constructing inter-procedural control-flow graphs and by using constraint solving to detect infeasible paths. Detecting such inter-procedural infeasible paths helps detect infeasible def-use pairs. Second, Seeker, which is developed around Pex, is primarily intended for achieving higher branch coverage rather than def-use coverage. In future work, we plan to develop a new search strategy for Seeker that guides Pex to achieve higher def-use coverage along with higher branch coverage.

```
00:public class Deque<T> {
01:   private DoublyLinkedList<T> m_deque;
02:   ...
03:   public override void Add(T item) {
04:      EnqueueBack(item);
05:   }
06:   public override void Clear() {
07:      m_deque.Clear();
08:      Count = 0;
09:   }
10:   public override T DequeFront() {
11:      Guard.InvalidOperation(Count == 0,
               Resources.DequeDequeueEmpty);
12:      T item = m_deque.Head.Value;
13:      m_deque.RemoveFirst();
14:      Count--;
15:      return item;
16:   }
17:   ...
18:}
```

Figure 7. The `Deque` class from Dsa.

7.6 RQ3: Defects

We next address the third research question regarding comparing defect-detection capabilities of Randoop, Pex, and Seeker. Table 6 shows our results. Subcolumns “AT”, “FT”, and “D” show the total number of generated test inputs, number of failing test inputs, and number of distinct defects detected, respectively, by each approach. For Randoop, due to the large number of failing test inputs, we regenerated test inputs with its default parameters, instead of analyzing all test inputs generated with the setting described in Section 7.4. Furthermore, all our test inputs are automatically generated and do not include test oracles. Therefore, we used uncaught exceptions as test oracles with focus on robustness issues.

Subject	# All Defs	Randoop		Pex		Seeker		Manual	
		# Covered	%	# Covered	%	# Covered	%	# Covered	%
QuickGraph	136	97	71.3	65	47.8	109	80.1	31	22.8
Dsa	112	0	0	34	30.3	59	52.7	59	52.7
xUnit	922	97	10.5	144	15.6	156	16.9	13	1.4
TOTAL	**1170**	**194**	**16.6**	**243**	**20.8**	**324**	**27.7**	**103**	**8.8**

Table 5. All defs coverage achieved by Randoop, Pex, Seeker, and manually written tests.

Subject	Randoop			Pex			Seeker		
	AT	FT	D	AT	FT	D	AT	FT	D
QuickGraph	6956	456	10	334	14	11	1923	117	34
Dsa	687	17	3	552	34	15	961	61	20
xUnit	112	0	0	1265	12	5	1360	12	5
NUnit	528	76	3	816	10	7	1804	16	13
Total	**8283**	**549**	**11**	**2967**	**70**	**38**	**6048**	**206**	**72**

AT: All Tests, FT: Failing Tests, D: Defects

Table 6. Defects detected by all approaches.

In particular, we considered the test inputs that throw exceptions as failing test inputs. However, we considered the failing test inputs that throw *expected* exceptions as passing test inputs. Furthermore, we ignored the defects related to `NullReferenceExceptions` that are thrown by passing `null` values to arguments of public methods. The primary reason is that often open source applications do not check `null` values for the arguments of public methods, and can also be fixed by automatically adding a `null` check on arguments of all public methods. To classify a failing test as a defect or expected exception, we inspected the source code of subjects under analysis and its associated Javadocs and comments. Since manually written tests of these subjects do not include any failing tests, we consider all defects detected by Randoop, Pex, and Seeker as new defects.

Our results show that Randoop, Pex, and Seeker detected 11, 38, and 72 distinct defects, respectively. We reported detected defects on hosting websites of our subject applications. In all subjects, defects detected by Randoop are related to `NullReferenceExceptions`. Similarly, except for Dsa, all defects detected by Pex are also related to `NullReferenceExceptions`. In Dsa, Pex detected two and five defects related to `OverflowException` and `IndexOutOfRange` exceptions, respectively. Seeker detected all defects detected by Randoop and Pex, and also detected new defects related to `InvalidOperationException` in QuickGraph. This exception is thrown when an attempt to modify a collection is made after an enumerator is created on that collection. It requires specific method sequences to cause this exception. Furthermore, Seeker detected a defect related to an infinite loop in QuickGraph. Figure 8 shows the test input that detected the infinite loop. The test input includes five classes and six method calls. Along with the

```
00: BidirectionalGraph bidGraph;
01: Random random;
02: VertexAndEdgeProvider s0 =
        new VertexAndEdgeProvider();
03: Vertex s1 = new Vertex();
04: bidGraph = new BidirectionalGraph
        ((IVEProvider)s0, PexSafeHelpers.
        ByteToBoolean((byte)16));
05: bidirectionalGraph.AddVertex((IVertex)s1);
06: random = new Random();
07: RandomGraph.Graph((IEdgeMutableGraph)bidGraph,
        0, 1, random, false);
```

Figure 8. A test input (generated by Seeker) that detected an infinite loop in QuickGraph.

Namespace	# Branches	Pex	M	S	M+S
Algorithms	572	24.8	27.4	**52.1**	44.2
Collections	269	17.8	63.2	**94.0**	95.6
Concepts	51	39.2	74.5	74.5	74.5
Exceptions	5	80.0	80.0	80.0	80.0
Predicates	58	93.1	93.1	100	98.3
Providers	5	60.0	80.0	80.0	80.0
Representations	159	52.2	64.8	67.9	67.3
TOTAL	**1119**	**31.6**	**47.3**	**68.2**	**64.3**

Table 7. Branch coverage achieved by MSeqGen (M) and Seeker (S) for QuickGraph.

skeleton generated by Seeker, the values "0" and "1" generated by Pex in Statement 7 helped trigger the infinite loop in the `RandomGraph.Graph` method. In summary, our results show that Seeker has higher defect-detection capabilities compared to Randoop and Pex.

7.7 RQ4: MSeqGen Comparison

We next address the fourth research question regarding comparing branch coverage achieved by Seeker with MSeqGen. MSeqGen took 1.3 hours to generate test inputs for QuickGraph. Table 7 shows our results. Columns "Pex", "M", and "S" show branch coverage achieved by Pex, MSeqGen, and Seeker, respectively. Column "M + S" shows branch coverage achieved by combining MSeqGen and Seeker. In particular, we used the sequences extracted by MSeqGen as beginning sequences for Seeker rather than starting Seeker

from the scratch. Although MSeqGen achieved higher coverage than Pex, our results show that Seeker achieved much higher coverage than MSeqGen, especially for complex namespaces such as `Algorithms` and `Collections`. There are two major reasons for the lower coverage of MSeqGen compared to Seeker. First, sequences extracted by MSeqGen from the existing code bases do not include sequences for many classes under test. For example, although we used 3.85MB of .NET assembly code for extracting sequences, none of these code bases include sequences for the `EdgeDoubleDictionary` or `EdgeStringDictionary` classes. Therefore, MSeqGen could not achieve any coverage for these classes. On the other hand, Seeker achieved 100% coverage for these two classes. Second, MSeqGen-extracted sequences are different from desired sequences required for producing desired object states.

In contrast to our original expectation, "M + S" achieved lower coverage than Seeker alone, except for the namespace `Collections`. Through our inspection, we found that "M + S" often resulted in more sequences, thereby increasing the exploration space for Pex. Although we can address this issue by using customized settings for Pex (similar to those used for RQ1), the limitations of the current Seeker prototype prevents from using such customized settings. In future work, we plan to combine both these approaches by improving the performance of Seeker. In summary, Seeker achieved higher branch coverage than MSeqGen, and unlike MSeqGen, Seeker does not require any additional information such as usage information.

8. Discussion and Future Work

In our evaluation, we used code coverage as a criterion for showing the effectiveness of Seeker compared to other approaches. Our criterion is based on a recent case study [33], which showed that field defects reduce with increased test coverage. This case study helps show that achieving high coverage can help improve the quality of code under test. Furthermore, our evaluation showed results for def-use coverage, which is a stronger criterion compared to branch coverage. The reason is that achieving higher coverage with respect to a stronger coverage criterion such as def-use coverage further helps to show the effectiveness of Seeker compared to other existing approaches.

We next summarize major limitations due to which Seeker could not achieve branch coverage close to 100%. These limitations can be broadly classified into two categories: general limitations of DSE and limitations specific to our Seeker approach. These general limitations of DSE also affect Seeker, since Seeker inherently uses DSE for dynamic analysis.

8.1 General limitations of DSE

We next describe general limitations of DSE that also apply for our Seeker approach.

Path explosion. Although Seeker suggests shorter skeletons (as shown in Table 3), we identify that skeletons suggested by Seeker increase the number of paths to be explored by Pex. Note that Seeker helps reduce the number of candidate sequences by using a combination of static and dynamic analyses, but do not reduce the number of paths to be explored within a suggested candidate sequence. For example, for the `algorithms` namespace of QuickGraph, Seeker achieved 52.1%. Although Seeker suggested desired skeletons to Pex, Pex could not generate test inputs using those skeletons for this namespace. The primary reason is that Pex, by default, attempts to cover all feasible paths among method calls within the suggested sequences. In future work, we plan to address this issue by developing a search strategy that can guide Pex. The insight for our future work is that not all paths in the method calls of suggested skeletons need to be explored for producing desired object states.

Environment dependency. A primary reason for the low coverage achieved by Seeker for xUnit and NUnit is their dependency on environments; dealing with such dependencies is currently beyond the scope of Seeker. For example, in xUnit, majority of the classes requires assembly files that include tests or project files in XML formats. However, Seeker achieved 28.3% (121 new branches) higher coverage than manually written tests for the `Gui` namespace, which includes some classes that require sequences and do not depend on the environment. In future work, we plan to address this issue by combining Seeker with other approaches [30] that mock environments, thereby isolating the environment dependency.

8.2 Specific limitations of Seeker

We next describe two major limitations of our Seeker approach.

Loop-based Sequences. Seeker is effective in generating sequences that involve multiple methods (that can be from different classes as well). Figure 8 shows an example sequence that includes six method calls from four different classes. However, Seeker faces challenges in generating sequences that require method calls to be repeated multiple times to produce the desired object state. For example, consider the `IntStack` class shown in Figure 4. Seeker can easily handle target branches with conditions such as `if(stack.count > 0)`, which requires the `Push` method to be invoked only once. Instead, consider the following target branch B9.

```
00: public static void foo1(IntStack ints) {
01:     if(ints.size() > 3) {
02:         ... // B9
03:     }
04: }
```

To cover the target branch B9, the target sequence should invoke the `Push` method at least four times. However, our Al-

gorithms 1 and 2 cannot handle this scenario. In particular, when `StatAnalyzer(B9, null)` is invoked, `StatAnalyzer` identifies the pre-target as `Push`. After `DynAnalyzer` successfully covers this pre-target (Line 4 of Algorithm 2), `StatAnalyzer` invokes `DynAnalyzer(B9, IntStack.Push)`. However, `DynAnalyzer` still cannot cover B9, since `Push` is invoked only once in the suggested sequence. To address this issue, Seeker includes the a heuristic-based technique described next.

Along with suggesting a method (and a pre-target in that method), Seeker also observes how the suggested method such as `Push` modifies the target field $tfield$. We refer to this information as side-effect information. Using the desired value for $tfield$ and the side-effect information, Seeker computes the number of times the suggested method has to be invoked to produce a desired value for $tfield$. For example, the desired value for the $tfield$ `_size` is four for covering the target branch B9. Therefore, Seeker identifies that `Push` method has to be invoked for four times in the suggested sequence, since each invocation of `Push` increases the value of the $tfield$ `_size` by one. Our technique can handle only a limited set of scenarios and cannot handle all scenarios that require method calls to be repeated multiple times. For example, our technique cannot handle the scenario where more than one method has to be repeated multiple times. In future work, we plan to address this issue by developing a fitness-based approach, where a fitness function incrementally guides the number of times suggested methods have to be invoked to achieve a desired value for $tfield$.

Abstract classes, interfaces, and callback methods. All our subjects are libraries or frameworks that include elements such as abstract classes or interfaces, whose implementations are often not available within those libraries or frameworks. These libraries or frameworks expect client applications to provide such implementations. For example, Dsa provides three abstract classes such as `CommonBinaryTree`. Without these abstract classes, Seeker achieved 94.3% coverage (higher than manually written tests) for the namespace `DataStructures` of Dsa. Similarly, xUnit includes methods (such as `ExecutorCallback.Wrap`) that require a callback method. We identify that manually written tests achieved higher coverage than Seeker, since those tests include necessary implementations. In future work, we plan to address this issue by developing a technique similar to mocking environments.

9. Related Work

Our Seeker approach is related to three major research areas to be discussed next.

9.1 Object-oriented Test Generation

Existing approaches for object-oriented test generation can be broadly classified into two major categories: *implementation-based* and *usage-based* approaches.

Implementation-based approaches. These approaches use the implementation information of classes under test for generating test inputs. These approaches can further be classified into two sub-categories: *direct construction* [3] and *sequence generation* [8, 15, 17, 32, 35, 43, 45].

The direct construction approaches such as Korat [3] construct desired object states by directly assigning values to member fields of classes under test. However, these approaches require specifications such as class invariants [26], which are rarely documented by developers. In contrast, Seeker is a sequence-generation approach and does not require class invariants.

Among sequence-generation approaches, Buy et al. [5] proposed an approach that generates sequences for exercising the def-use pairs associated with member fields of classes under test. Their approach can be used for testing classes in isolation to achieve def-use coverage. *Bounded-exhaustive* approaches [45] generate sequences exhaustively up to a small bound of sequence length. However, target sequences involving classes from real-world applications often require longer sequences beyond the small bound handled by bounded-exhaustive approaches.

Another category of approaches, called evolutionary approaches [2, 14, 15, 24, 43], accept an initial set of sequences and evolve those sequences to produce new sequences that can generate desired object states. Two of these approaches [15, 43] can be used to test individual classes only and cannot generate target sequences that involve methods from multiple classes (as shown in their evaluations). Testful [2] addressed some of the issues faced by these two approaches and proposed a semi-automated approach, where the user has to provide data to augment the efficiency. Harman and McMinn [14] further presented a theoretical and empirical analysis of a global search technique used in evolutionary approaches. Based on their empirical results, they proposed a hybrid global-local search (a memetic) algorithm. Although a direct comparison of Seeker with these approaches helps show the benefits of Seeker, we could not perform such comparison due to language restrictions. In particular, prototypes developed for these evolutionary approaches target C or Java programs, whereas Seeker targets .NET (C#) programs. Nevertheless, Lakhotia et al. [24] conducted an empirical study that compares a search-based test generation approach, called AUSTIN [23], with a DSE-based approach, called CUTE [22]. Their study on testing C code shows that both the approaches achieved similar branch coverages. Their study also shows that neither of the approaches achieved more than 50% branch coverage. A major issue identified by their study for the DSE-based approach is related to the path exploration strategy used by the approach. In particular, CUTE could not achieve high coverage due to unbounded depth-first search strategy that often cannot handle programs with loops effectively. Recent approaches such as Fitnex [46] integrated within Pex [41] can

help address those issues. Based on this empirical study, we expect that our Seeker approach can perform better than evolutionary approaches too, since our evaluation results show that Seeker performs better than Pex.

Randoop [35] is a random approach that generates sequences by randomly combining method calls. Zheng et al. [48] proposed a heuristic approach that assists a random approach with sequences that mutate member fields accessed by a method under test. However, due to the large search space of possible sequences, there is often a low probability for randomly generating target sequences. In contrast to these approaches, ours is a systematic approach that generates sequences incrementally based on the branches that are not yet covered, thereby significantly reducing the number of candidate target sequences.

Korel [20, 21] proposed a chaining approach that identifies alternate target branches that need to be covered to cover a given target branch. Seeker also uses a similar technique. However, their approach can handle only procedural code such as C code and cannot handle object-oriented code that includes additional challenges such as inheritance and nested classes. McMinn and Holcombe [32] proposed an extended chaining approach that identifies a sequence of methods that need to be executed to cover a target branch. Our approach significantly differs from their approach in two major aspects. First, similar to Korel's approach, their approach can handle only procedural code. Second, their approach requires users to provide a bound on the length of the desired sequence and method calls that can be included in that sequence. In contrast, our approach does not require any manual effort and automatically synthesizes sequence that produces desired object states.

A recent approach, called Covana [44], precisely identifies the problems that prevent tools from achieving high structural coverage. Covana focuses on two major problems: (1) external-method-call problem; (2) object- creation problem. Covana reports these problems to developers, so that developers can provide guidance to tools in achieving high structural coverage. Similar to Covana, Seeker focuses on object-creation problem. However, in contrast to Covana that reports problems to reduce effort of developers in guiding tools, Seeker automatically synthesizes sequences (that assist tools such as Pex) and does not require any manual effort.

Usage-based approaches. In our previous work, we proposed a mining-based approach, called MSeqGen [40], which statically mines method-call sequences based on their usages from existing code bases. MSeqGen uses mined sequences to assist random and DSE-based approaches. A major issue with MSeqGen is that it is not effective in the scenarios where no code bases that use classes required for target sequences are available or code bases include sequences that are different from target sequences. For example, if a class c is newly introduced, it is not possible to find code bases using class c. Furthermore, mined sequences may not include all necessary method calls required for producing desired object states.

Jaygarl et al. proposed OCAT [16] that captures object states dynamically during program executions and reuses captured object states to assist a random approach. Similarly, another approach, called DyGen [38], mines dynamic traces recorded during program executions and generates regression test inputs from mined sequences. A major issue with OCAT and DyGen is that these approaches require system test inputs for capturing object states and sequences, respectively. Furthermore, captured object states or sequences can be different from desired ones. Although OCAT includes a mutation technique, the mutation technique requires class invariants to effectively mutate private member fields. Seeker complements these approaches and does not require any additional information. Furthermore, Seeker can also effectively handle private member fields through method-call graphs.

9.2 Program Synthesis

Earlier research in program synthesis focused on programming by demonstration [9, 25], where programs are synthesized automatically by observing the manual actions performed by the user. Further efforts in end-user programming attempted to bridge the gap between natural languages and programming languages by developing structured editors [18] or providing semantics to natural-language interfaces [28]. In contrast to these approaches, Seeker targets at reducing efforts of programmers rather than end users.

Little and Miller [27] proposed an approach that allows end users to leverage scripting interfaces provided by applications such as Microsoft Word. In particular, their approach allows end users to specify a task, such as formatting a document, using keywords. Their approach attempts to map those keywords to APIs of particular system. In contrast to their approach that focuses on identifying APIs, Seeker focuses on generating method sequences that produce a desired object state. However, in future work, we plan to adopt a similar strategy of accepting desired object states in the form of keywords and automatically generate method sequences.

Gulwani [12] proposed a framework that describes three dimensions of program synthesis: a user-specified intent, the space of candidate programs, and the search technique. Our Seeker approach can be formulated based on this framework. Gulwani et al. [13] also proposed another approach for synthesizing loop-free programs. In contrast to these approaches that focus on synthesizing algorithms such as sorting or bit-manipulation routines, Seeker focuses on synthesizing object-oriented programs that involve method sequences.

There exist two other approaches [29, 39] accept queries of the form "*Source* $\Rightarrow$ *Destination*". These approaches generate method sequences that accept an object of type *Source* as input and produce an object of type *Destination*. In contrast to these approaches that generate some object of the

Destination type, Seeker focuses on generating a desired object state of the *Destination* type.

9.3 Static and Dynamic Analyses

Seeker uses a combination of static and dynamic analyses to intelligently navigate through a large search space. Similar to Seeker, there exist other dynamic-analysis approaches [4, 6, 37] that also leverage static analysis. However, static analysis used in Seeker differs significantly from the static analysis used in these approaches. In particular, these existing approaches analyze control-flow, data-flow, or program-dependence graphs to assist dynamic analysis. In contrast to these approaches, Seeker uses method-call graphs. Furthermore, these approaches handle procedural code such as C, whereas Seeker handles object-oriented code.

10. Conclusion

Over the past decade, program synthesis has gained focus due to the recent advances in computing and reasoning techniques. In this paper, we proposed an approach, called Seeker, that accepts a user-specified intent as a desired object state and synthesizes programs in the form of method sequences that produce the desired object state. We have shown the effectiveness of Seeker by applying it to the problem of object-oriented test generation. In our evaluation, we have shown that Seeker achieved higher coverage (both structural and data-flow coverage) than existing state-of-the-art DSE-based and random approaches on four subject applications (totalling 28KLOC). We have also shown that Seeker detected 34 new defects. In future work, we plan to extend Seeker to accept a user-specified intent in the form of natural language. In particular, we plan to transform the intent into a series of desired object states and leverage Seeker to automatically synthesize programs.

Acknowledgments

This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CCF-0915400, CCF-0546844, CCF-0702622, CCF-1117603, CNS-0958235, CNS-0627749, CNS-0917392, ARO grant W911NF-08-1-0443, and US Air Force grant FA9550-07-1-0532.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proc. TACAS*, pages 134–138, 2007.

[2] L. Baresi and M. Miraz. Testful: automatic unit-test generation for Java classes. In *Proc. ICSE*, pages 281–284, 2010.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.

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

[5] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proc. ISSTA*, pages 39–48, 2000.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, 2008.

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

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

[9] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: Programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[10] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, 1988.

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

[12] S. Gulwani. Dimensions in program synthesis. In *Proc. PPDP*, pages 13–24, 2010.

[13] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. PLDI (to appear)*, 2011.

[14] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36:226–247, March 2010.

[15] 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.

[16] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: object capture-based automated testing. In *Proc. ISSTA*, pages 159–170, 2010.

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

[18] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37:83–137, June 2005.

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

[20] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(3):203–213, 1992.

[21] B. Korel. Automated test generation for programs with procedures. In *Proc. ISSTA*, pages 209–215, 1996.

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

[23] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *Proc. GECCO*, pages 1759–1766, 2008.

[24] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Syst. Softw.*, 83:2379–2391, December 2010.

[25] H. Lieberman, editor. *Your wish is my command: Programming by example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[26] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[27] G. Little and R. C. Miller. Translating keyword commands into executable code. In *Proc. UIST*, pages 135–144, 2006.

[28] H. Liu and H. Lieberman. Programmatic semantics for natural language interfaces. In *Proc. CHI*, pages 1597–1600, 2005.

[29] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. PLDI*, pages 48–61, 2005.

[30] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proc. AST*, pages 149–153, 2009.

[31] V. Martena, A. Orso, and M. Pezzè. Interclass testing of object oriented software. In *Proc. ICECCE*, pages 145–154, 2002.

[32] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Proc. GECCO*, pages 1013–1020, 2005.

[33] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proc. ESEM*, pages 291–301, 2009.

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

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

[36] QuickGraph: A 100% C# graph library with Graphviz Support, Version 1.0, 2008. `http://www.codeproject.com/KB/miscctrl/quickgraph.aspx`.

[37] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, N. James, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proc. ICISS*, pages 1–25, 2008.

[38] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proc. TAP*, pages 77–93, 2010.

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

[40] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, 2009.

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

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

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

[44] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proc. ICSE*, pages 611–620, 2011.

[45] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, pages 365–381, 2005.

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

[47] Z3: An efficient theorem prover, 2005. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[48] W. Zheng, Q. Zhang, M. Lyu, and T. Xie. Random unit-test generation with MUT-aware sequence recommendation. In *Proc. ASE*, pages 293–296, 2010.