Randoop Project Report

Zirui Fang

November 18, 2025

Table of Contents

1 Introduction and Motivation

The reliability of software systems has become increasingly critical as technology permeates every aspect of modern life. Software bugs can have severe consequences, ranging from minor inconveniences to catastrophic failures with significant financial and human costs. Studies estimate that software errors cost the global economy billions of dollars annually. Despite decades of software engineering research, creating correct and robust software remains a significant challenge.

Data structures form the fundamental building blocks of virtually all software systems. From operating systems to web applications, programs rely heavily on efficient and correct implementations of lists, trees and other foundational structures. However, many programming language standard libraries provide only partial implementations of these essential components, forcing developers to either implement their own versions or depend on third-party libraries of varying quality. This situation creates substantial risks, as even minor implementation errors in these fundamental components can propagate throughout an entire system.

Software testing represents the primary defense against such defects, yet manual test creation is notoriously time-consuming and often incomplete. Developers frequently focus on "happy paths" while neglecting edge cases and error conditions. Automated test generation tools like Randoop offer a promising solution by systematically exploring program behaviors that human testers might overlook. These tools can generate both regression tests to preserve existing functionality, and error-revealing tests to detect current bugs, e.g., through the violation of implicit properties, or explicitly provided invariants.

This project investigates an approach to verifying data structure reliability through automated testing. Our methodology involves: (1) selecting Go language data structure implementations from reputable open-source repositories, (2) carefully translating these implementations to Java while preserving their original semantics, (3) specifying intended behavior through representation invariants (repOK methods), and (4) applying Randoop for automated test generation and bug detection. By focusing on two representative data structures - a dynamic List implementation and a self-balancing AVL Tree - we aim to evaluate both the effectiveness of automated test generation and the quality of the translated implementations. The selection of Go as a target programming language for our study is not a coincidence; we have chosen Go because the language's standard library only contains implementations of very simple data structures, and thus more sophisticated data structures are, in the context of Go projects, used from reputed third party implementations.

The project contributes to software reliability research by demonstrating a practical methodology for verifying data structure implementations across programming languages. Our work also provides insights into the effectiveness of representation invariants as partial specifications for guiding automated test generation tools. Ultimately, we hope to show how such techniques can help developers create more reliable software foundations with reduced manual effort.

2 Preliminaries and Preparation

This section outlines the foundational work required for our study, including language translation, test generation setup, and specification of data structure invariants.

2.1 Go vs. Java

The project attempts to analyze the functional correctness of data structure implementations in Go. However, the proposed analysis mechanism is feedback-directed random test generation, a technique that currently does not offer tools to directly apply to Go programs. This project then will involve the translation from Go to Java, a task that is non-trivial and for which we will resort to large language models, as we describe below.

As mentioned before in this document, unlike Java's comprehensive collections framework, Go's standard library deliberately omits implementations of many fundamental data structures. This property makes Go a particularly interesting subject for data structure analysis, as third-party implementations vary widely in quality and correctness.

2.2 Testing and Test Generation

Software testing is a very demanding approach to analyze software correctness. This issue has been acknowledged by the software engineering community, and different tools to automate the process of generating tests have been proposed. In this project, we will use a specific tool and technique, called Randoop. Randoop implements a feedback-directed random testing algorithm, that incrementally generates tests, and profits from previous tests to guide in the search for tests evaluating newer software situations, and avoiding ill-formed test sequences in the process. The technique was put forward in a well-known paper at the International Conference on Software Engineering in 2007, and since then has been applied to a variety of software analyses. The authors constantly maintain a tool that implements this technique, mostly targeting Java as a programming language.

While the tool documentation is comprehensive, this project involved understanding how the algorithm works, what the algorithm requires, how to interpret its output, etc, besides, of course, executing all the technical steps for tool configuration, setup and execution (details are provided below, in "steps for using Randoop").

2.3 Representation Invariants (repOK)

Randoop bases test generation and bug finding directly on the code of a software implementation. Thus, in principle, it has no mechanism to detect whether a non-crashing generated test case (a scenario exercising the software in a particular way) corresponds to the desired behavior of the software, or on the other hand is the manifestation of a software bug (a divergence from the expected behavior). For crashing test cases the situation is different: if the crash is due to a runtime exception, for example a null pointer exception, then it clearly represents a manifestation of a bug, regardless of the expected behavior of the software.

An additional feature of Randoop is that it allows the user to specify the expected behavior of software via so-called *representation invariants*. A representation invariant is a *specification* of the intended behavior of software, that is specifically associated with data abstractions or class representations. Technically, given a class C, a representation invariant or class invariant for C is a state property P that can be evaluated in the state of objects of C, and:

- is established after every constructor of C, i.e., when an object is created, P holds for the object,
- is preserved by every public method of C, i.e., provided that an object o satisfies P, after executing a publich method $m()$ on o, P still holds.

Representation invariants, in the context of data structure implementations, characterize properties of data representation. For instance, in a linked list implementation, such property may state that the linked structure is acyclic (no node in the list is reached from itself by performing traversals through the "next" field), that a sentinel node has no associated data (the data field of the head of the list is null), etc.

A way of specifying a representation invariant as code, i.e., without having to do so via documentation or a formal logical language compromising executability of specifications, is with so called repOK routines. A repOK routine or method is a method of a class that implements its corresponding representation invariant. It is typically assumed to be parameterless (it only consults the internal fields of the object without the need for additional parameters), it returns a boolean value (true of the representation invariant holds, false otherwise), and does not have side effects (i.e., a query for the objects of the class). Further details on the representation invariants used in this project can be found below under "`repOK` method implementation".

2.4 LLM-Assisted Code Translation

This project needs to overcome a non-trivial task, the translation from Go to Java. The reason is that our target programs are Go data structure implementations, but our chosen tool for software analysis (test generation) is implemented for Java programs.

Although the translation from Go to Java may appear at first sight relatively simple, it is indeed rather complex. The reason is that both languages have very different programming idiosyncrasies. Some notable differences are the strong use of concurrency in Go, the use of a very simplistic hierarchy of

exceptions in Go, when compared to Java, and different techniques for defensive programming associated with both languages.

In this project, we opted to use large language models to implement the code translation. Large language models have proven very useful and rather accurate for code-to-code translation tasks. We used a general purpose large language model for this task, namely DeepSeek. For our experiments, we used simple and straightforward prompts, obtaining reasonable results. Generalizations of this work may require the use of various alternative LLMs, the combination of LLM results using techniques such as LLM-as-a-judge, and more sophisticated prompt designs, including their combination with other tools such as compilers. This is, however, beyond the scope of this project.

3 Subject Programs and Testing Methodology

This section describes the subject programs (the data structures under study) and explains why they were selected. It also outlines our choice of Randoop as the test generation tool and its testing approach.

3.1 Overview of Subject Programs

This study focuses on two fundamental data structures: a dynamic List and a self-balancing AVL Tree, both translated from Go to Java for evaluation. These structures were selected due to their widespread use in software development, their representation of distinct structural paradigms, and the unique challenges posed by their absence in the Go standard library. At the same time, their respective invariants are relatively well-known, and this facilitated our manual analysis of the code in relation to representation invariant design and implementation.

3.1.1 Dynamic List

The List implementation, as provided in List.java, is a generic, array-based data structure that supports a range of operations, including insertion (insert, append, push), deletion (remove, pop), and traversal (iterator, reverseIterator). It offers flexibility through dynamic resizing, with methods to expand or shrink the underlying array based on usage, as well as a fixed-size mode for constrained environments. Additional features include checking for emptiness (isEmpty), fullness (isFull), and element presence (has), alongside utilities for copying (copy) and clearing (clear) the list. This implementation is critical in applications requiring sequential data storage and manipulation, such as queues, stacks, or buffers.

3.1.2 AVL Tree

The AVL Tree, implemented in AvlTree.java, is a self-balancing binary search tree that maintains a height difference (balance factor) of at most one between the left and right subtrees of any node. It ensures efficient operations—search (get), insertion (put), and deletion (remove)—with a time complexity of $O(logn)$. The implementation includes methods for checking emptiness (isEmpty), size (size), and key presence (contains), as well as an in-order traversal for retrieving keys (keys). Balancing is achieved through different types of rotations, which are triggered when the balance factor exceeds the allowed range. This structure is vital in applications requiring ordered data, such as dictionaries in general, priority queues, etc.

3.2 Reasons for Selection

The selection of the List and AVL Tree is driven by several factors:

1. **Absence in Go Standard Library**: The Go standard library does not provide built-in implementations for many fundamental data structures, including dynamic lists and self-balancing trees like AVL Trees. This gap compels developers to rely on third-party libraries or custom implementations, which may introduce errors. Testing such implementations is crucial to ensure their reliability in production environments.
2. **Representational Diversity**: The List and AVL Tree represent two distinct categories of data structures—linear and tree-based, respectively. The List is a simple, sequential structure, while the AVL Tree is a complex, hierarchical structure with self-balancing properties. This diversity allows the study to evaluate the effectiveness of automated testing across a spectrum of complexity.

3. **Ubiquity and Importance**: Both data structures are foundational to software systems. Lists are used in countless applications for managing sequences of data, while AVL Trees are critical for maintaining ordered data with guaranteed performance. Ensuring their correctness is essential for building reliable software.

3.3 Testing Tool: Randoop

To verify the reliability of the translated List and AVL Tree implementations, we employ Randoop, an automated test generation tool for Java. Randoop is a unit test generator for Java. It automatically creates unit tests for your classes, in JUnit format. Randoop generated tests are test sequences, composed of sequences of method invokations of the software under analysis, together with automatically generated regression test assertions, based on the execution and observation of generated tests. Randoop can be used for two purposes: to find bugs in programs, and to create regression tests to warn when program behavior changes overtime (e.g., to guarantee behavior preservation after a code refactor or efficiency improvement). Randoop uses a feedback-directed random test generation approach to create unit tests that systematically explore the behavior of the code under test. Its methodology can be broken down into the following steps:

1. **Sequence Generation**: It pseudo-randomly generates method/constructor invocation sequences for the classes under test. That is, it generates sequences of method calls, starting with constructors or static methods to create objects, followed by calls to other methods on those objects. For example, a sequence for the List might include creating a new list, appending elements, and checking its size.
2. **Execution and Observation**: Randoop executes each of these sequences and observes their outcomes, including return values and exceptions. It uses this feedback to refine subsequent sequences, focusing on those that reveal new behaviors or potential errors.
3. **Test Filtering**: Generated sequences are filtered to retain valid tests, such as those that expose defects (e.g., null pointer exceptions, index out-of-bounds errors) or cover new code paths. Invalid or redundant sequences are discarded.
4. **Test Output**: Randoop produces JUnit-compatible test cases, categorized as regression tests (to preserve existing functionality) or error-revealing tests (to identify current bugs). These tests can be integrated into a broader testing suite for ongoing validation.

Randoop's strength lies in its ability to explore a wide range of program behaviors, including edge cases and unexpected inputs, which are often overlooked in manual testing. For instance, it might generate tests that attempt to insert null elements into the List or perform insertions of varying keys into an AVL Tree, potentially uncovering defects in boundary conditions related to maintaining balance.

3.4 Error-revealing Tests and Regression Tests

When Randoop includes an error-revealing test in its output, (e.g., in a file ErrorTest.java), such test shows that the code violates its specification or contract (it makes false the representation invariant), or violates an implicit property, e.g., throws a null pointer exception. The tests that do not violate implicit or explicit properties are stored separately as regressions behaviors (e.g., in a RegressionTest.java file). Such tests assert the current behavior of the code under test: values returned or (excpeted) exceptions thrown. After changes in the code that do not affect the API exercised by the generated test suites, one may run these regression tests, which will alert the developer if the code changes affect the observable behavior of the classes.

4 Experimental Setup and Results

This section details the experimental setup, and results of applying Randoop to our subject programs (List and AVL Tree).

4.1 Experimental Setup

Our experimental setup involved the use of various tools, including an LLM to facilitate code translation, a specific version of Randoop, and the use of an IDE where minor code fixes were performed, the repOK

implementations were developed, the coverage analysis performed, etc. In our experiments, we used the DeepSeek large language model, directly from the web browser interface of the model. We used Randoop for Java, version 4.3.3. As an integrated development environment, we used the latest version of IntelliJ IDEA.

4.2 Data Structure Implementations

The data structure implementations, both in their original Go language and Java translation, and together with their accompanying representation invariants, are publicly available. The original source code and its corresponding translation can be found in:

`https://github.com/Ray221f/Randoop-List-and-AVL-Tree-Java-Source-Code`

The repOK implementations of both data structures can be found in:

`https://github.com/Ray221f/Randoop-Experimental-repOK-Source-code-and-Analysis`

The complete experiment source code can be found in:

`https://github.com/Ray221f/Randoop-Go-Bug-Detection-Automated-Testing-of-Go-Data-Structures-via-Java-Translation`

4.3 Test Generation Outcomes

4.3.1 List Implementation

- Generated test classes: `RegressionTest`, `RegressionTest0` through `RegressionTest11` (12 files in total)
- Error-revealing tests generated: 0
- Flaky methods detected: 0
- Invalid tests generated: 0

4.3.2 AVL Tree Implementation

- Generated test classes: `RegressionTest`, `RegressionTest0` through `RegressionTest13` (14 files in total)
- Error-revealing tests generated: 0
- Flaky methods detected: 0
- Invalid tests generated: 0

These results indicate that:

- Both implementations demonstrated robust behavior under Randoop's automated test generation.
- The absence of error-revealing tests suggests the core functionality was correctly implemented.
- Zero flaky methods indicate consistent test behavior across multiple runs.
- No invalid tests were generated, confirming proper method usage in test generation.

This is of course the result of our analysis on the final translations. It required multiple iterations over the translated code, performing minor edits due to problems in the translation process, among other things. Similar iterations were performed over the repOK implementations. The initial versions failed, but the fails (obtained failing test cases) were not due to bugs in the software, but due to incorrect interpretations of the implementations, which we had to update through modifications of the corresponding repOK methods.

4.4 Coverage Analysis

The automated test generation process is quantitatively evaluated through systematic code coverage analysis, which serves as an objective measure of test suite thoroughness. This examination reveals how

thoroughly the Randoop-generated tests exercise the List and AVL Tree implementations, providing insights into both the strengths and limitations of the automated testing approach.

Coverage metrics are presented across four critical dimensions that collectively assess testing completeness. Method coverage indicates the proportion of methods invoked during testing, while line coverage measures the percentage of executable statements reached. Branch coverage evaluates decision point validation, and condition coverage analyzes boolean sub-expression evaluation. These metrics enable developers to identify well-tested code regions and detect potential testing gaps that may require additional verification efforts.

The subsequent analysis compares coverage results between the two data structure implementations, highlighting patterns in test effectiveness and revealing opportunities for improving the test generation process. Particular attention is given to the relationship between structural complexity and achieved coverage levels, as evidenced by the differential results between the List and AVL Tree components.

The following coverage metrics were collected from executing the generated tests, and measuring coverage using plugins in our employed IDE:

4.4.1 List Implementation

Element	**Method**	**Line**	**Branch**	**Condition**
All	83% (15/18)	99% (5641/5648)	96% (214211/221550)	50% (12837/25642)
List	100% (3/3)	94% (32/34)	89% (104/116)	68% (81/118)
RegressionTest0-11	100% (1/1)	100% (varies)	93-97%	50%

Table 1: Coverage results for List implementation testing

4.4.2 AVL Tree Implementation

Element	**Class**	**Method**	**Line**	**Branch**
All	94% (16/17)	99% (6890/6896)	94% (126747/133691)	50% (21054/42106)
AvlTree	100% (2/2)	72% (16/22)	61% (51/100)	51% (34/66)
RegressionTest0-13	100% (1/1)	100% (varies)	93-95%	50%

Table 2: Coverage results for AVL Tree implementation testing

4.5 Analysis of Coverage Results

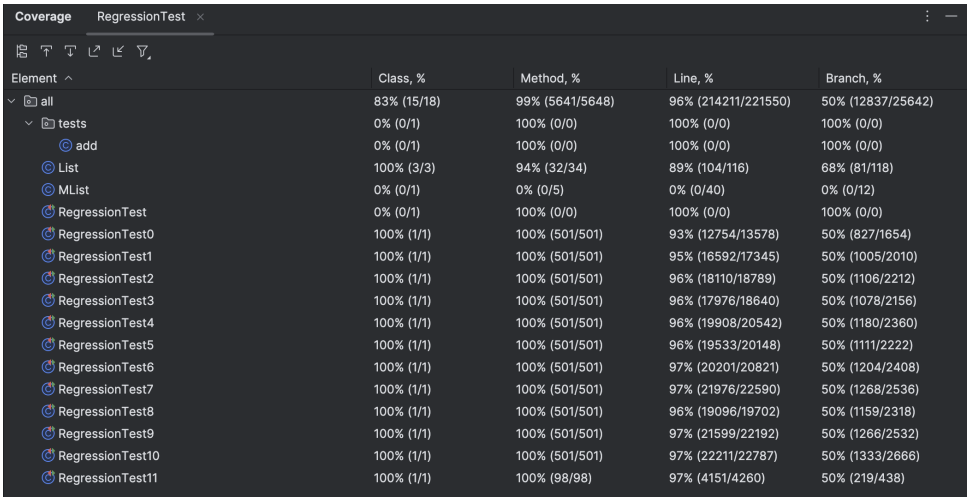
- Randoop generated tests achieved reasonable line coverage in both implementations (61-94%).
- Method coverage was high for both implementations (72% for AVL Tree, 100% for List).
- Branch and condition coverage achieved lower scores, especially for AVL Tree (51%-61%), indicating that other testing methodologies may be useful as a complement to the generated tests.

4.6 Screenshots of Coverage Analysis

Figures 1 and 2 show some screenshots of the results of performing coverage analysis, indicating more detailed results, e.g., coverage per test suite file.

5 Conclusion

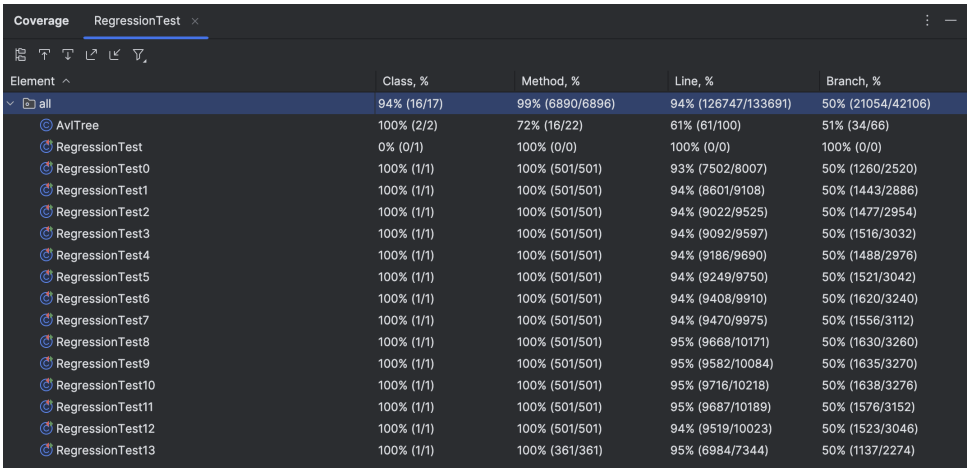
This project demonstrates the efficacy of using Randoop for automated test generation to verify the reliability of data structure implementations, specifically a dynamic List and a self-balancing AVL Tree translated from Go to Java. The absence of error-revealing tests and flaky methods, coupled with relatively high coverage metrics, especially for line coverage, indicates robust implementations and effective test generation. The use of `repOK` methods as representation invariants significantly enhanced Randoop's ability to detect invalid states, showing strengths of the implementations not only according to implicit properties (no null dereferences, etc), but also with respect to specific semantic properties of



Coverage RegressionTest

Element	Class, %	Method, %	Line, %	Branch, %
all	83% (15/18)	99% (5641/5648)	96% (214211/221550)	50% (12837/25642)
tests	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
add	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
List	100% (3/3)	94% (32/34)	89% (104/116)	68% (81/118)
MList	0% (0/1)	0% (0/5)	0% (0/40)	0% (0/12)
RegressionTest	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
RegressionTest0	100% (1/1)	100% (501/501)	93% (12754/13578)	50% (827/1654)
RegressionTest1	100% (1/1)	100% (501/501)	95% (16592/17345)	50% (1005/2010)
RegressionTest2	100% (1/1)	100% (501/501)	96% (18110/18789)	50% (1106/2212)
RegressionTest3	100% (1/1)	100% (501/501)	96% (17976/18640)	50% (1078/2156)
RegressionTest4	100% (1/1)	100% (501/501)	96% (19908/20542)	50% (1180/2360)
RegressionTest5	100% (1/1)	100% (501/501)	96% (19533/20148)	50% (1111/2222)
RegressionTest6	100% (1/1)	100% (501/501)	97% (20201/20821)	50% (1204/2408)
RegressionTest7	100% (1/1)	100% (501/501)	97% (21976/22590)	50% (1268/2536)
RegressionTest8	100% (1/1)	100% (501/501)	96% (19096/19702)	50% (1159/2318)
RegressionTest9	100% (1/1)	100% (501/501)	97% (21599/22192)	50% (1266/2532)
RegressionTest10	100% (1/1)	100% (501/501)	97% (22211/22787)	50% (1333/2666)
RegressionTest11	100% (1/1)	100% (98/98)	97% (4151/4260)	50% (219/438)

Figure 1: Code coverage results for List implementation



Coverage RegressionTest

Element	Class, %	Method, %	Line, %	Branch, %
all	94% (16/17)	99% (6890/6896)	94% (126747/133691)	50% (21054/42106)
AvlTree	100% (2/2)	72% (16/22)	61% (61/100)	51% (34/66)
RegressionTest	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
RegressionTest0	100% (1/1)	100% (501/501)	93% (7502/8007)	50% (1260/2520)
RegressionTest1	100% (1/1)	100% (501/501)	94% (8601/9108)	50% (1443/2886)
RegressionTest2	100% (1/1)	100% (501/501)	94% (9022/9525)	50% (1477/2954)
RegressionTest3	100% (1/1)	100% (501/501)	94% (9092/9597)	50% (1516/3032)
RegressionTest4	100% (1/1)	100% (501/501)	94% (9186/9690)	50% (1488/2976)
RegressionTest5	100% (1/1)	100% (501/501)	94% (9249/9750)	50% (1521/3042)
RegressionTest6	100% (1/1)	100% (501/501)	94% (9408/9910)	50% (1620/3240)
RegressionTest7	100% (1/1)	100% (501/501)	94% (9470/9975)	50% (1556/3112)
RegressionTest8	100% (1/1)	100% (501/501)	95% (9668/10171)	50% (1630/3260)
RegressionTest9	100% (1/1)	100% (501/501)	95% (9582/10084)	50% (1635/3270)
RegressionTest10	100% (1/1)	100% (501/501)	95% (9716/10218)	50% (1638/3276)
RegressionTest11	100% (1/1)	100% (501/501)	95% (9687/10189)	50% (1576/3152)
RegressionTest12	100% (1/1)	100% (501/501)	94% (9519/10023)	50% (1523/3046)
RegressionTest13	100% (1/1)	100% (361/361)	95% (6984/7344)	50% (1137/2274)

Figure 2: Code coverage analysis for AVL Tree implementation

the implementations. This project also involved the use of LLMs for code translation, illustrating a very interesting application of this emerging technology, to enable cross-language software analysis tool execution. This work also offers interesting lines for future development. On one hand, we may work on systematizing the translation, attempting to fully remove the need for manual validation, e.g., by introducing auxiliary mechanisms such as compilation of results and metamorphic property checking on the translated code. Secondly, the limitations in branch and condition coverage may ask for improvement on the generated test suites, and may identify program situations that are difficult to cover for random testing. Finally, extending our analysis to other data structures and Go data structure libraries may allow us to find bugs in real data structure implementations, improving the quality of the corresponding libraries.

6 Appendix: Guide for Randoop `repOK` Debugging Coverage

6.1 Steps for using Randoop

This section provides a comprehensive, step-by-step guide to implementing Randoop for automated test generation in Java projects. The workflow has been optimized through practical application. We outline the complete process from environment configuration through test analysis, with specific instructions for IntelliJ IDEA users.

Step 1: Environment Setup and Dependency Configuration (IntelliJ IDEA)

1. Install Randoop package: `https://github.com/randoop/randoop/releases/latest`
2. Configure project dependencies:
   - Navigate to `Project Structure → Modules → Dependencies`
   - Add `randoop-all-4.3.3.jar`
   - Select "`Compile`" scope
   - Apply changes
3. Import `randoop-all-4.3.3.jar` into the project's `src` directory

Step 2: Code Preparation with `repOK` Method Implementation

1. Convert code to proper Java class format
2. Implement `repOK` method with:
   - Proper import: `import randoop.CheckRep`
   - Annotation: `@CheckRep`

Step 3: Test Generation via Command-line Execution Execute the following commands:

```
javac -cp .:randoop-all-4.3.3.jar [class_name].java
java -classpath .:randoop-all-4.3.3.jar randoop.main.Main gentests \
  --testclass=[class_name] \
  --junit-output-dir=tests
```

Step 4: Analysis of Generated Tests and Coverage Metrics

- Measure code coverage for successful tests
- For error cases:
  - Conduct root cause analysis
  - Develop solutions with supporting rationale

Step 5: Reporting and Documentation Procedures

- Implement code fixes
- Document findings including:
  - Reproduction steps

- Proposed solutions
- Test evidence

6.2 `repOK` Method Implementation

The `repOK` method is a fundamental component in Randoop's automated test generation framework, serving as a representation invariant validator for Java classes. This section provides a technical specification for implementing effective `repOK` methods that enable Randoop to automatically verify class invariants during test case generation, detect invalid object states through systematic runtime checks and Generate meaningful assertions for regression testing.

The following provides some technical points when writing `repOK` and some implementation examples.

6.2.1 Method Signature Requirements and Return Type Conventions

`repOK` should be "public", "no arguments" and either:

1. Boolean-returning version:
   - Signature: `public boolean methodName()`
   - Returns `true` if invariants are satisfied
   - Returns `false` or throws exception if invariants are violated
2. Void-returning version:
   - Signature: `public void methodName()`
   - Normal return (no exception) indicates invariants are satisfied
   - Thrown exception indicates invariant violation

6.2.2 Invariant Verification

A comprehensive `repOK` should verify:

- Field value constraints (ranges, formats)
- Inter-field relationships
- Non-null constraints
- Collection/map invariants
- General consistency rules

6.2.3 Good Practices for Maintainable Checks

- Name the method clearly (`repOK`, `checkInvariants`, etc.)
- Document all invariants in comments
- Make checks independent for better debugging

6.2.4 Implementation Examples

```
@CheckRep
public boolean isBalanceValid() {
    return balance >= 0;  // Returns true if balance is non-negative
}
```

Listing 1: Boolean-returning example

```
@CheckRep
public void checkConsistency() {
    if (width > length) {
        throw new IllegalStateException("Width cannot exceed length");
    }
```

```
}
```

Listing 2: Void-returning example

6.3 Debugging Generated Tests

Effective debugging of automatically generated tests requires a systematic approach to identify, analyze, and resolve issues in both the test cases and the production code. Here presents a comprehensive methodology for debugging Randoop-generated tests, covering: test failure analysis, steps for debugging generated tests, strategies to optimize the debugging process.

6.3.1 Test Failure Analysis

1. Examine `ErrorTest` files for:
    - Exception types
    - Triggering input values
    - Stack traces

6.3.2 Steps for Debugging Generated Tests

1. Set breakpoints at test method entry points
2. Execute in debug mode
3. Step through execution using:
    - Step Over
    - Step Into
    - Resume
4. Inspect variable states

6.4 Code Coverage Analysis

Code coverage analysis provides quantitative insights into the effectiveness of your test suite by measuring how much of the source code is exercised during test execution. This section details the methodology for evaluating Randoop-generated tests using coverage metrics, which serve as key quality indicators for automated test generation.

6.4.1 Execution Steps (IntelliJ IDEA)

1. Right-click test file
2. Select "`Run 'TestClass' with Coverage`"
3. Analyze coverage metrics

6.4.2 Coverage Descriptions

1. Class Coverage: Percentage of classes exercised
2. Method Coverage: Percentage of methods invoked
3. Line Coverage: Percentage of lines executed Branch Coverage: Percentage of branches taken

6.4.3 Coverage Indicators

- Green: Fully covered
- Yellow: Partially covered
- Red: Not covered

References

1. Randoop Homepage: `https://randoop.github.io/randoop/`
2. Randoop Manual: `https://randoop.github.io/randoop/manual/index.html`
3. Randoop Release: `https://github.com/randoop/randoop/releases/latest`
4. Data Structures Repository: `https://github.com/timtadh/data-structures`