An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application

M. Moein Almasi*, Hadi Hemmati†, Gordon Fraser‡, Andrea Arcuri§, Jānis Benefelds††
*Department of Computer Science, University of Manitoba, Canada
†Department of Electrical and Computer Engineering, University of Calgary, Canada
‡Department of Computer Science, University of Sheffield, UK
§Westerdals ACT, Norway, and the University of Luxembourg
††SEB Life & Pension Holding AB Riga Branch
*almasi@cs.umanitoba.ca, †hadi.hemmati@ucalgary.ca, ‡gordon.fraser@sheffield.ac.uk, §arcand@westerdals.no, ††janis.benefelds@seb.lv

***Abstract*—Automated unit test generation has been extensively studied in the literature in recent years. Previous studies on open source systems have shown that test generation tools are quite effective at detecting faults, but how effective and applicable are they in an industrial application? In this paper, we investigate this question using a life insurance and pension products calculator engine owned by SEB Life & Pension Holding AB Riga Branch.**

To study fault-finding effectiveness, we extracted 25 real faults from the version history of this software project, and applied two up-to-date unit test generation tools for Java, EVOSUITE and RANDOOP, which implement search-based and feedback-directed random test generation, respectively. Automatically generated test suites detected up to 56.40% (EVOSUITE) and 38.00% (RANDOOP) of these faults. The analysis of our results demonstrates challenges that need to be addressed in order to improve fault detection in test generation tools. In particular, classification of the undetected faults shows that 97.62% of them depend on either "specific primitive values" (50.00%) or the construction of "complex state configuration of objects" (47.62%).

To study applicability, we surveyed the developers of the application under test on their experience and opinions about the test generation tools and the generated test cases. This leads to insights on requirements for academic prototypes for successful technology transfer from academic research to industrial practice, such as a need to integrate with popular build tools, and to improve the readability of the generated tests.

***Keywords*-Automated Tests Generation; Empirical Software Engineering; Search-based Testing; Random Testing;**

I. INTRODUCTION

Software testing is an essential part of software development processes to assure the quality of software systems. Developers typically underestimate the required testing effort, and therefore developer-written tests are generally not comprehensive [6]. To overcome the challenges of manual test generation, automatic techniques and tools based on different approaches have been introduced (e.g., [10], [16], [20]).

In order for developers to adopt these tools, it is important to provide an understanding of their capabilities and the quality of the tests they produce. A common way in the literature to evaluate generated test suites is by measuring their code coverage (e.g., [13]). However, code coverage as a measure of test effectiveness has been challenged by several recent studies (e.g., [17], [18]). Alternatives are to evaluate the ability of tools to detect real faults (e.g., [14], [23]) or artificial faults (e.g., mutation analysis [15]). Another shortcoming of empirical studies on automated test generation tools is that they only focus on open source projects. Evaluations on industrial systems (e.g., [13]) are still rare, possibly because they require a level of maturity of the underlying tools that is difficult to achieve for research prototypes. Consequently, there is a need to provide further evidence of the capabilities of automated unit test generation on industrial systems.

The aim of this paper is to evaluate automated test generation on a relatively complex, production ready financial application known as *LifeCalc*, owned and developed internally by *SEB Life & Pension Holding AB Riga Branch*. We selected a search-based test generation tool (EVOSUITE [10]) and a random test generation tool (RANDOOP [20]) for experimentation, and provide the following concrete contributions in this paper:

- We describe the results of an experiment using 25 real faults of the *LifeCalc* industrial application to assess the effectiveness of automatically generated test suites in terms of detecting real-world faults.
- We analyze the undetected faults in *LifeCalc* in order to guide future research on automated test generation tools.
- We surveyed the developers of *LifeCalc* to get their feedback about the generated unit tests.
- We provide general lessons learned from the application of the automated unit test generation to industrial software.

Our experiments with the *LifeCalc* application reveal that EVOSUITE detected 56.40% of the faults, and RANDOOP 38.00%. A closer investigation of the undetected faults shows that 97.62% of them depend on either "specific primitive values" (50.00%) or the construction of "complex state configuration of objects" (47.62%), and this can guide future research towards improved techniques for detecting such defects. Our interactions with the developers of *LifeCalc* reveal several aspects of the test generation research prototypes that inhibit a successful technology transfer from academia to industry, which can guide testing researchers to achieving impact in practice.

This paper is organized as follows. Section II presents background. Section III states the experiment setup. Section IV presents results. Section V discusses lessons learned. Section VI reviews related work. Section VII concludes the paper.

II. Automated Unit Test Generation

In object-oriented programming, a unit test is a small, executable piece of code that exercises a functionality of a class under test. While there is a wide range of techniques to automatically generate tests in general, the specific case of unit test generation is mainly addressed by approaches based on random generation of call sequences, search-based optimization of call sequences, and symbolic approaches.

A. Random Testing

Random testing [4] is perhaps the most basic and straight forward form of test generation [9], as it consists of invocation of functions with random inputs. Guided random testing is a refined approach that starts with random input data and then uses some form of extra knowledge to produce further input data. One of the main examples of this category is feedback-directed random testing [21]. Feedback-directed random test generation enhances random test generation by incorporating feedback collected from executing test inputs that is used to avoid generating duplicate and illegal input data. This technique takes a set of classes as input and creates method sequences incrementally. It iteratively executes new sequences and checks them against general contracts. Sequences that violate the contract are considered as failing tests, and sequences that pose normal behavior are treated as regression tests.

B. Search-based Testing

Search Based Software Engineering is an approach that transforms software engineering problems into optimization problems [19], where the objective of the test generation is implemented by a fitness function that guides the search. Among the many meta-heuristic search techniques used for test generation, Genetic Algorithms are perhaps the most common [19]. In a Genetic Algorithm, randomly selected candidate solutions are evolved by applying evolutionary operators, such as mutation and crossover, resulting in new offspring individuals, with better fitness values. An example objective function for unit test generation is the whole test suite's code coverage [12].

C. Symbolic Testing

Symbolic approaches represent execution paths through a program as constraints on the input values. A common approach is *Dynamic Symbolic Execution* (e.g., [16]), where the paths of a program are systematically explored by iteratively negating one branch condition in a path constraint at a time, and using a constraint solver to generate a new test input for that path. Most approaches of this kind target generation of specific input data and require manual construction of test drivers.

III. Experiment Setup

In this section, we describe our experiment methodology. The main objective of this study is to evaluate the effectiveness of existing mature test generation tools, in terms of revealing known real faults in an industrial application, and understanding the existing barriers for practitioners when adopting these tools.

A. Research Questions

To achieve the mentioned goal, we have designed an experiment to answer the following research questions:

RQ1*: How effective are automatically generated unit tests in terms of finding real faults?* This question intends to assess the capability of two test generation tools that are widely adopted in academia, in terms of revealing real faults.

RQ2*: What categories of faults are harder to detect using the current automated test generation tools?* This question aims to categorize the faults that have not been detected by any of the examined test generation tools.

RQ3*: What major barriers do developers see when adopting automatic test generation tools?* This question tries to identify some of the practitioner's requirements which are not currently supported by automatic test generation tools. (Due to resource limitations we only consider EvoSuite for *RQ3*)

B. Subject of Study

We have conducted our experiment on a life insurance and pension products calculator engine known as *LifeCalc*, which is a standalone software component owned by *SEB Life & Pension Holding AB Riga Branch*. *LifeCalc* is written using the Java technology stack. Its implementation started in early 2015 and it has been released to production in early 2016. *LifeCalc*'s development team consists of 5 developers, 2 testers and 1 business analyst/project manager. *LifeCalc* is a medium-sized application that consists of complex critical calculations with plenty of business rules that are implemented using complex conditions, which makes it challenging for test generation tools. *LifeCalc*'s stacks comprise of front-end (client) and back-end (core, services) modules with approximately 80,000 LOC.

LifeCalc is built in a nightly build using the Jenkins [2] Continuous Integration (CI) build management system, and all tests are executed during the nightly builds. Moreover, builds are also triggered on demand if there are any critical bug fixes that need to be deployed to production. In these cases, there is a restricted subset of important tests which will be executed to test the critical functionalities of the application. The company's developers provided us 25 faults from their issue tracking system. They selected these faults randomly, trying to include examples from different times throughout the life cycle of the project. We studied the commits in the version repository that contained the fix for the fault until we understood them well enough to know how to replicate them by writing manual unit tests. For each fault, we extracted the faulty and fixed program versions, such that they differ by a minimal change that demonstrates the isolated fault fix.

C. Fault Analysis

Based on the fault descriptions, we distinguish between the following two types of faults in our dataset: 1) Specification-based faults and 2) Exception-related faults. The first category requires knowledge about the expected logic (specification) vs. the existing implemented logic. This means that, in most cases, a JUnit assertion is required to detect the fault in the faulty version of the program. The second category consists of faults that can be detected without knowing the exact logic of the code. In other words, the unit test's failure is due to an unhandled exception thrown in the code under test, and not about a failing assertion in the test.

In the rest of this section we explain these categories with an anonymized code snippet example per category (the anonymization is due to our agreement with *SEB Life & Pension Holding AB Riga Branch* for publication).

1) Specification Faults: Among the collected 25 faults, we identified five as specification faults because the expected business logic was not implemented correctly. For instance, the code snippet from *LifeCalc-b23*, shown below, is one of the *Specification Faults*, in which the assigned developer was retrieving tariff values yearly, whereas tariff values in the properties file are defined on a monthly basis, and thus `param1` needs to be multiplied by 12. One way to detect this fault is to have an assertion in the generated tests on the fixed version to check the value of `param3`.

```
public double faultyMethod(int param1, int param2) {
  ...
  double param3 = 0.0;
  //Faulty Statement
  - param3 = Double.valueOf(PropertiesReader.getProperty("
     mt.m[" + param1 + "]")) * Math.pow((1 + param2), -1);
  //Fixed Statement
  + param3 = Double.valueOf(PropertiesReader.getProperty("
     mt.m[" + param1 * 12 + "]")) * Math.pow((1 + param2),
     -1);
  ...
  return param3;
}
```

Listing 1. Example of a specification fault in our study

2) Exception-related Faults: We identified 20 faults in this category in our pool of 25 faults. We encountered several types of common exceptions such as `NullPointerException` `(NPE)` (thrown when an application attempts to call methods on a `null` object instance), `ArithmeticException` (thrown when an exceptional arithmetic condition has occurred) and `NumberFormatException` (thrown to indicate that the application has attempted to convert a string with invalid format to one of the numeric types). However, most of the faults from this category were due to `NPE`s. The following code snippet from *LifeCalc-b18* shows one of the exception throwing faults: An exception is thrown due to an invalid parameter value (property key) of the getProperty method. Therefore, `PropertiesReader.getProperty(invalidKey)` returns a `null` value, which then causes the program to throw a `NPE` on conversion of the `String` to a `Double` object.

```
public void faultyMethod(ObjectX objx, String param1,
     String param2) {
  ...
  if(param1.equalsIgnoreCase(Enum1.P_012.getValue())){
    //Faulty Statement
    Double rate = Double.valueOf(PropertiesReader.
    getProperty("rate_" + param2 + "")) * objx.getObj().
     getPaymentFrequency();
  }
  ...
}
```

Listing 2. Example of an exception-related fault in our study

D. Automated Unit Test Generation Tools

As mentioned earlier, *LifeCalc* is written in Java, and therefore we had to consider test generation tools for Java. For the Java programming language, there are mature tools that can generate JUnit test cases using random testing (e.g, RANDOOP) and search-based testing (e.g., EVOSUITE). However, for symbolic approaches, usually research prototypes in Java only generate test data, and not JUnit test cases [8] (i.e., testers have to manually write test drivers for those symbolic tools for each single class). Therefore, we only selected tools from the categories of random unit test generation and search-based unit test generation. RANDOOP [20] and JCrasher [7] are instances of random unit test generation tools. We decided to use RANDOOP as it is one of the most used random test generation tools in academia, and it is still being actively maintained. EVOSUITE [10] and TestFul [5] are representative of evolutionary test generation tools which apply search techniques in order to optimize test suites based on various coverage criteria. EVOSUITE was chosen as it is actively being maintained and extended, and ranked first in recent SBST tool competitions (e.g., see [11]).

1) RANDOOP: RANDOOP [20] is one of the most stable random test generation tools, with easy to follow instructions to get it up and running in short time. RANDOOP implements feedback-directed random test generation, by generating sequences of method invocations for all the classes under test. In other words, it builds test inputs incrementally, and then the newly created test inputs extend previous ones. As soon as these test inputs are created, they will be executed and the results collected from these executions will be used to guide the generation of new ones. RANDOOP can be used for both fault-detection and regression testing. For regression testing, the tests contain assertions that capture the current state. For fault detection it checks various predefined or custom contracts, and the violation of any of these contracts indicates an error in the tested classes. RANDOOP requires the user to provide a list of classes under test. For the experiments in this paper, we had to manually identify a list of classes that are the dependencies of the faulty class, for each of the 25 analyzed faults. We used all the default settings, except for the stopping criterion, for which we used two configurations: The default setting of 3 minutes, and an increased duration of 15 minutes.

2) EVOSUITE: EVOSUITE [10] is a search-based unit test generation tool for Java that uses a genetic algorithm to evolve a set of test cases with the intention of maximizing code coverage. EVOSUITE initially generates random sets of test cases and then uses evolutionary search operators such as selection, mutation, and crossover to improve the generated test cases. This evolution process is guided by a fitness function calculated based on various coverage criteria [22]. EVOSUITE then performs optimizations with respect to the defined coverage criteria on the test suite with the highest coverage. Ultimately, it enforces sanitization checks to ensure the generated tests are valid and executable. In our experiments, we executed EVOSUITE on the faulty classes with the branch coverage criterion as fitness function. We applied the same stopping criteria (i.e., 3 and 15 minutes) as we used for RANDOOP.

E. Test Generation Scenario

We generated the tests on the fixed versions so that the automatically generated JUnit assertions are based on the correct implementations. Generating tests on the fixed version is useful in the context of regression testing, and allows us to simulate whether the specification faults can be detected in such a testing scenario. For exception-related faults we could have also generated the tests directly on the faulty version, since throwing of exceptions can be used as an automated oracle. In other words, the current experiment design makes more sense in the context of regression testing, where one needs to create a regression test suite that passes in the current version, to be used to guard from future faulty changes. However, as the faulty and fixed versions differ only in terms of the fault fix and the interfaces are identical, we expect that results on exception-faults would be similar, if tests were generated directly on the faulty version.

F. Experiment Procedure

The overall experiment procedure is as follows:

- First, we extracted both fixed and faulty versions of *LifeCalc* based on the identified commit IDs provided by *SEB Life & Pension Holding AB Riga Branch*.
- Then, for each fault, we generated test suites using both, EVOSUITE and RANDOOP, on the fixed version.
- To determine whether a fault was found, we executed all generated test cases on the corresponding *LifeCalc* faulty version. These executions were done manually using the Eclipse IDE. A test case is considered to detect the fault if it fails on the faulty version. This failure can be due to an exception in the executed classes or a JUnit assertion failure in the tests.
- We used two different stopping criteria (search budget of 3 and 15 minutes), and repeated test generation 10 times for each fault and stopping criterion.
- We collected all the statistics from the execution logs and manually verified the validity of the failing test cases, in order to avoid possible false positives [23].

In order to accommodate for the randomness of the test generation tools, each tool was executed 10 times for each fault. *RQ1* uses two set ups for test generation budget (3 and 15 minutes), which will be discussed more in the next section.

The measure that we use to assess the effectiveness of the test suites is the percentages of the runs (how many out of 10) that detected the fault. We also aggregate these by averaging over several versions (faults).

G. Survey Procedure

Our participants in the conducted survey were the five developers of *LifeCalc* with different level of expertise, varying from 1 to 8 years of working with Java technology stack, and familiarity with the application code base. The participants were provided with a survey package containing sets of tasks to perform and their respective guidelines, as well as a questionnaire to be answered (The survey package is available online at https://github.com/moeinalmasi/sealab). The total duration of the survey was 2 hours per developer.

Based on the analysis of our fault effectiveness experiment, we gave our participants tasks such as executing the generated tests, and trying to debug and locate a fault (we gave each developer three faults of different level of difficulty, as determined in the RQ2 analysis; see Section IV-B). In addition, we asked them to write a manual test that covers the same faulty code as a generated test to better understand how generated tests relate to developer preferences. They were also provided with comprehensive guidelines including all the necessary commands to run and generate tests using EVOSUITE.

After performing these tasks, the developers answered a questionnaire containing seven demographic questions, four questions using Likert-scale to rate aspects of tools and generated tests, and six free-text questions. See the discussion in Section IV-C for details on the questions.

IV. EXPERIMENT RESULTS

In this section, we discuss the results of our experiments and answer the research questions presented in Section III-A.

A. Effectiveness of Automatically Generated Unit Tests

RQ1*: How effective are automatically generated unit tests in terms of finding real faults?*

The results of test suite executions are summarized in Table I. We first consider the 3 min. scenario: The first observation is that both tools can find some of the faults, and are unsuccessful at detecting the others. The average fault detection rate is not particularly high (50.80% in EVOSUITE vs. 36.80% in RANDOOP). However, the variation of effectiveness for different faults, which we can see in Table I, is a more interesting observation. There are cases (like *LifeCalc-b4*, *LifeCalc-b7*, etc.) where none of the tools can detect the fault even in ten executions. On the other hand, there are cases (like *LifeCalc-b6*, *LifeCalc-b15*, *LifeCalc-b18*, etc.) where every single test suite out of the 10 per tool can detect the fault.

Given the results, one follow up question is “Can we improve the effectiveness of the tools by allocating more search budget to them?”. To answer this question, we also ran the experiment of *RQ1* with the longer stopping criterion of 15 minutes. Note that this experiment is not meant to be a thorough study on the correlation between testing budget and effectiveness. The goal is just to have an idea on what can one expect by adding extra resources to these tools. A more thorough study would be an interesting future work.

In total, 19 out of 25 faults were found. However, as the tools are both based on randomized algorithms, these faults were not found in all of the 10 runs. On average, RANDOOP can find faults in 36.8% of the runs, whereas EVOSUITE in 50.8% of the runs. While increasing the search budget is useful

TABLE I
FAULT DETECTION RATE OF EVOSUITE AND RANDOOP FOR EACH FAULT OVER 10 EXECUTIONS PER FAULT

Fault	**EVOSUITE** (%)		**RANDOOP** (%)	
	3 min	15 min	3 min	15 min
LifeCalc-b1	40	+0	0	+0
LifeCalc-b2	30	+10	10	+0
LifeCalc-b3	60	+0	30	-10
LifeCalc-b4	0	+0	0	+0
LifeCalc-b5	100	+0	90	+10
LifeCalc-b6	100	+0	100	+0
LifeCalc-b7	0	+0	0	+0
LifeCalc-b8	20	+10	0	+0
LifeCalc-b9	80	+0	40	+10
LifeCalc-b10	0	+0	0	+0
LifeCalc-b11	60	+0	10	-10
LifeCalc-b12	0	+0	0	+0
LifeCalc-b13	70	+30	90	+10
LifeCalc-b14	50	+10	0	+0
LifeCalc-b15	100	+0	100	+0
LifeCalc-b16	30	+20	0	+0
LifeCalc-b17	80	+20	90	+10
LifeCalc-b18	100	+0	100	+0
LifeCalc-b19	60	+20	70	+10
LifeCalc-b20	60	+0	20	-10
LifeCalc-b21	0	+0	0	+0
LifeCalc-b22	0	+0	0	+0
LifeCalc-b23	100	+0	70	+10
LifeCalc-b24	100	+0	100	+0
LifeCalc-b25	30	+20	0	+0
Average	50.80%	+5.60%	36.80%	+1.20%

for EVOSUITE (+5.6%), it had only a moderate effect on RANDOOP (+1.2%).

RQ1: Existing tools can potentially detect most of the faults (19 out of 25 were detected at least once). But there are also some faults (6 out of 25) that are never found within the explored search budgets.

Given the variations in the results per fault, it would be interesting to see what kind of faults are easier to detect and on which faults the tools have difficulties. Therefore, in *RQ2* we explore this question.

B. Analysis of Faults Not Detected

RQ2*: What categories of faults are harder to detect using the current automated test generation tools?*

As discussed in detail in the fault analysis section (Section III-C), we categorized the faults into two types: 1) Specification faults and 2) Exception-related faults. Table II shows the percentages of faults that are detected in each category. The results show that, as expected, the detection rate is higher with 15 minutes search budget. However, looking into the *Exception-related Faults* we can see that there is quite a variation between the effectiveness of test suites on different faults (Detailed summary of all execution shows the fault category for each faulty version of *LifeCalc* which is available online at the following address: https://github.com/moeinalmasi/sealab). Therefore, we next try to characterize the faults in the exception-related faults category.

TABLE II
PERCENTAGE OF DETECTED FAULTS PER CATEGORY OVER ALL EXECUTIONS USING EVOSUITE AND RANDOOP, + INDICATES THE FAULT DETECTION RATE IS INCREASED

Fault Category	**EVOSUITE** (%)		**RANDOOP** (%)	
	3 min	15 min	3 min	15 min
Specification Faults	46	+4.00	34.00	+2.00
Exception-related Faults	52	+6.00	37.50	+1.00

TABLE III
FAULT DETECTION (%) VARIATION OVER 3 AND 15 MINUTES SEARCH BUDGET IN DIFFERENT FAULT CATEGORIES, + INDICATES THAT FAULT DETECTION RATE IS INCREASED WHILE - INDICATES THE FAULT DETECTION RATE IS DECREASED

Category	**EVOSUITE** (%)		**RANDOOP** (%)		Faults(#)
	3 min	15 min	3 min	15 min	
Easy	87.14	+10.00	91.43	+5.71	7
Hard	47.78	+5.55	12.22	-2.22	9
Challenging	0.00	0.00	0.00	0.00	4

We look at the faults in this category in three subclasses: 1) Easy Faults (that are detected by both tools in at least 80% of times, with 3 and/or 15 minutes stopping criterion), 2) Hard Faults (that are detected at least once by one tool and are not "easy faults"), and 3) Challenging Faults (that are never detected by either tools). The next subsections describe these faults with anonymized code examples from the industrial system.

1) Easy Faults: In the *Easy Fault* category, the faulty statement does not require satisfying a complex condition prior to its execution. In other words, those are faults that do not require specific input data and they do not depend on complex conditions. For instance, the following code snippet represents a simple `NullPointerException (NPE)` in *LifeCalc-b5*:

```
 ...
 if(!StringUtils.isEmpty(objx.getObj().getLocale)){
   //Faulty Statement
   Double interest = Double.valueOf(PropertiesReader.
    getProperty("interest.rate.A_" + objx.getObj().
    getLocale() + ""));
 }
 ...
}
```

Detecting this fault requires the invocation of `faultyMethod` with an `ObjectX` instance that has a non-empty locale attribute `String` with invalid value. As the invalid property key `interest.rate.A_InvalidKey` would be missing in the property file, there will be a `NPE` when converting a `null String` to `Double` object.

Table III shows the same results of Table I (for the exception-related faults only) grouped by the fault difficulty categories. Of the Easy Faults 87.14% were detected by EVOSUITE and 91.43% by RANDOOP.

Overall, in this category, EVOSUITE found the fault in 61 out of 70 cases, while RANDOOP performed slightly better by detecting faults in 64 out 70 times. The following is an excerpt of a test case generated by EVOSUITE for an easy fault:

```java
1 public void test17()  throws Throwable  {
2   ...
3   FaultyClass faultyClass0 = new FaultyClass();
4   ObjectX objx = new ObjectX();
5   Object obj0 = new Object();
6   obj0.setProdCode("012_200");
7   obj0.setCoverSumLife(12);
8   obj0.setLocale("012_200");
9   objx.setObj(obj0);
10  faultyClass0.faultyMethod(objx);
11  ...
12 }
```

In this specific example, the cases where the fault had not been detected was due to the fact that a *locale* object was not initialized. Since there is a simple `null` check condition prior to the faulty statement, the faulty statement was not covered. RANDOOP generated, on average, around 14,500 test methods on each run for this particular fault (*LifeCalc-b5*) while EVOSUITE generated 18 test methods. Although most of the methods generated by RANDOOP do not cover this fault, there is at least one method that covers this fault. The following is a sample test generated by RANDOOP that detected the fault:

```java
1 public void test4() throws Throwable {
2   ...
3   FaultyClass var0 = new FaultyClass();
4   Objectx var1 = new Objectx();
5   Object var2 = new Object();
6   var2.setLocale("hi");
7   var1.setObj(var2);
8   var0.faultyMethod(var1);
9   ...
10 }
```

2) Hard Faults: This category describes faults that are either surrounded by conditions which require specific primitive values, or the faulty statement itself requires specific input data. The mentioned primitive values are not only explicit inputs of the tests, but also attributes of objects passed to the tests. The following code snippet shows a fault from the *Hard Fault* category.

```java
8   ...
9   List<Double> list = new ArrayList<Double>();
10  if (param3.equalsIgnoreCase(Enum1.POSITIVE.getValue())) {
11    for (int i = 1; i <= param1 * 12; i++) {
12      if (param4.equalsIgnoreCase(Enum2.LOW.getValue())) {
13        // Faulty statement
14        list.add(i,Math.pow((1+Double.valueOf(
        PropertyReader.getProperty("inv_min"))),(Double.valueOf
        (1)/Double.valueOf(12)))-1);
15      }
16    }
17  }
18  ...
19 }
```

The faulty method contains an `NPE` in *LifeCalc-b8*. To detect the fault using the generated tests, it needs to satisfy two conditions (lines #4 and #6) that require a specific string input. Note that the two nested if conditions require test inputs that can be extracted from `enum` values. Line #8 is the faulty statement in which the property value of `inv_min` is an invalid key in the properties file. This type of faults may occur due to two reasons: 1. the program tries to get a value of an invalid key in the properties file (*the code is faulty*). 2. The valid property key is missing in the properties file (*fault in the properties file*). The second case happens because the business analysts are able to change the property file values directly, and previously there was no sanitization check in place to validate the properties files. Therefore, due to human error, there could be properties with no or wrong values.

The *Hard Faults* category is where search-based approaches have the most advantages (47.78% for EVOSUITE vs. 12.23% of RANDOOP), because they can focus on generating corner cases. EVOSUITE found *Hard Faults* in 43 out of 90 executions while RANDOOP only detected such faults in 11 out of 90 executions. The following is an example of a test method generated by EVOSUITE for *LifeCalc-b8* which is a *Hard Fault*. The generated input data is able to satisfy both conditions and detects the fault. In the process of test data generation, EVOSUITE has extracted the required input data to satisfy both conditions in line #4 and #6 in the above listing from the values existed in the `enum`.

```java
1 public void test2()  throws Throwable  {
2   ...
3   FaultyClass faultyClass0 = new FaultyClass();
4   faultyClass0.faultyMethod(10,4,"positive","low");
5   ...
6 }
```

On the other hand, RANDOOP generated approximately 12,500 test methods on average per execution and none of the generated test methods were able to satisfy the conditions, mainly because they did not provide the required test data. The following is an example of a test generated by RANDOOP:

```java
1 public void test352() throws Throwable {
2   ...
3   FaultyClass var0 = new FaultyClass();
4   java.util.List var1 = var0.faultyMethod(1,2147483,"
      20160419_7232","c$$u#");
5   ...
6 }
```

3) Challenging Faults: The third category of faults we call *Challenging Faults*. In this type of fault, the faulty statement is usually surrounded by complex conditions that requires constructing complex objects that are populated with specific values. For instance, in the following example, line #11 is the faulty statement in which `getPaymentFrequency()` can have the potential value of 0 which causes an arithmetic fault of a division by zero. The root-cause of this fault has to do with a new scheme for payment frequency that was denoted by value zero.

```java
20  ...
21  List<DateTime> list = new ArrayList<DateTime>();
22  Double a = 0.0;
23  list.add(0, new DateTime());
24  list.add(1, a);
25  for (int i = 2; i <= Months.monthsBetween(a,b).getMonths
      (); i++) {
26    if(objectx.getCalcCoverList().get("CoverCode").
     getPromilUWDate().isAfter(b) && objectx.getProdCode().
     equalsIgnoreCase("ProductCode") && i==2){
27      list.add(i, list.get(i - 1).plusMonths(1));
28      // Faulty statement
29      a = (Days.daysBetween(a, list.get(i)).getDays() + 1)
     / objectx.getPaymentFrequency();
30    }
31  }
32  ...
33 }
```

Basically, the execution of the faulty statement requires satisfying a complex condition (line #8), which in turn requires specific data. It is considered complex since a randomly initialized `map` is unlikely to contain such specific keys and values. Once the outer condition has been satisfied, the faulty statement also demands a specific data which in this case, `getPaymentFrequency()`, is zero.

Based on our definition, the *Challenging Faults* are difficult for both tools, and were not detected at all. The following is a sample test case generated by EvoSuite for *LifeCalc-b4*. It constructed an empty object and set some of the basic attributes (i.e., `setProdCode()`) properly, but it failed to initialize and set a relatively complex `List`, `getCalcCoverList()`, which is a `map` containing object `Cover`:

```java
public void test10()  throws Throwable  {
  ...
  FaultyClass faultyClass0 = new FaultyClass();
  LocalTime localTime0 = LocalTime.now();
  DateTime dateTime0 = localTime0.toDateTimeToday();
  LocalTime localTime1 = localTime0.plusHours(27);
  DateTime dateTime1 = localTime1.toDateTimeToday();
  ObjectX objx = new ObjectX();
  objx.setProdCode("017_200");
  faultyClass0.faultyMethod(objx, dateTime0, dateTime1);
  ...
}
```

Randoop also partially constructed the input data, but because `var1` creates a new `ObjectX`, both `var1.getCoverEndDate()` and `var1.getBirthDateInsured()` return `null`. Given that `var5` and `var6` are *null*, the fault is not covered:

```java
public void test474() throws Throwable {
  FaultyClass var0 = new FaultyClass();
  ObjectX var1 = new ObjectX();
  DateTime var5 = var1.getCoverEndDate();
  DateTime var6 = var1.getBirthDateInsured();
  var1.setDiscount((java.lang.Double)10.0d);
  var0.faultyMethod(var1, var5, var6);
  ...
}
```

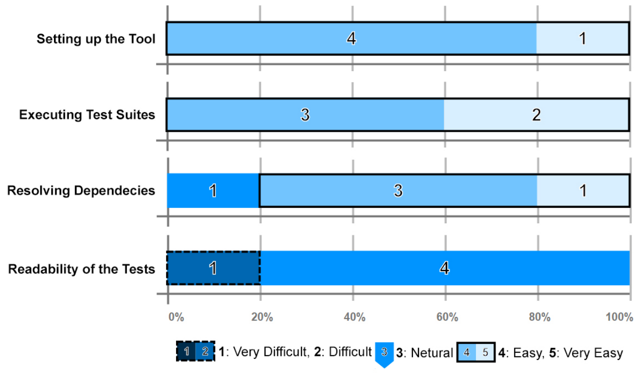
In order to detect faults from the mentioned categories, tools are required to improve their coverage and propagation, which means not only the faulty code needs to be covered, but also it has to be executed with a set of specific values in order to fail. We will discuss this in more detail in Section V-B.

Finally, we look at the results per category when we add extra test generation budget to the tools. Table III shows the new results in +/- percentages. Increasing the search budget did not facilitate the detection of *Challenging Faults*. On the other hand, detection of *Easy Faults* increased by at least 5.71%, which can be due to the fact that *Easy Faults* are detected if the faulty statement is covered, even without requiring any specific input data or construction of complex object.

RQ2: Faults whose triggering requires generating input object data with complex states are hard to detect.

Up to 56% of the faults have been detected by the test generation tools, but those are mostly *Easy Faults*. We would like to understand the developer's point of view about the generated tests, specifically about the tests which failed to detect any fault. Therefore, to answer *RQ3* we conducted a survey with the developers of *LifeCalc*.

Fig. 1. Likert chart for the answers provided by the developers for questions in *RQ3* that demonstrates the difficulty of the tasks from their point of view. (The survey package is available online at https://github.com/moeinalmasi/sealab)



C. Understanding Pitfalls of Test Generation Tools

RQ3*: What major barriers do developers see when adopting automatic test generation tools?*

The main goal of surveying *LifeCalc*'s developers is to assess the applicability of test generation tools and automatically generated unit tests. We gave our participants certain tasks and asked them to answer the following questions (see Section III-G for more details):

1) *How difficult was it for you to set up* EvoSuite*, execute the test suite, resolve dependencies and read the generated tests?* They rated all of above tasks relatively easy except for the readability of the generated tests. In terms of building and resolving dependencies, EvoSuite uses Maven [1] which they found really useful by saying *"... maven is definitely a plus point"*. Figure 1 reveals some concerns about the readability of the generated tests. As one of the developers said *"... it is hard to follow some of the generated tests"*.

The developers next had to read the bug report and re-execute the unit tests again, and try to debug and locate the bug using the given set of unit tests. Then, we asked them to answer the following question:

2) *How can the generated tests be improved?* In general, the developers did not like the generated assertions, as they said *"... poor assertions, sometimes there is an assertion and sometimes there is not? The assertions are mostly checking for simple stuff like list size and so on."*. In addition, they also did not like the generated input data. They described the generated data as *" ... set of extreme value test data with a correct datatype. They are not meaningful but complying with the method signature datatype"*. Some of the generated tests are readable but they are covering easy faults based on our fault categorization. On scale of 1 to 5, 1 being not helpful at all and 5 being very helpful, the developers rated the generated tests as sightly (4 developers) to moderately (1 developer) helpful.

We then asked them to manually write unit tests that cover the same code as the given generated tests (the rationale is that the following responses would not be purely subjective, and also this would help us to have a proper understanding of the tests) and answer the following questions:

3) *Describe what you like better about manually written tests than generated tests?* Developers prefer the test data (e.g., primitive values) used in the manually written tests. In addition, in manually written tests, the assertions are meaningful and useful unlike the generated ones.

4) *Would you keep the generated unit tests?* They mostly answered no to this question. They specified that generated tests are not as good as manually written tests in terms of test data. In addition, either there is no assertion or if there is, it is not validating useful data. However, it can easily be modified to become a useful assertion. Finally, we wanted to find out their overall opinion about the automatic test generation and identify the advantages and the pitfalls from their point of view by answering the below questions:

5) *Given your current infrastructure setup, how would you like to have automated unit test generation framework integrated?* They emphasized that it is important to integrate these tools into their development and continuous integration (CI) environments. They said *" ... supporting Jenkins is a must"*. In this case, EvoSuite provides plugins for both CI tools (Jenkins) and development tools (IntelliJ and Eclipse).

6) *What are the major barriers from your point of view in adopting automatic test generation tools?* Aside from assertions and test data that we mentioned earlier on, they also reported several other issues, like inability of test generation tools to support well-known, widely-used Java framework (e.g., Spring) and core components such as dependency injection. This highlights the importance of supporting major tools and frameworks.

RQ3: Assertions and readability of generated tests need to be improved. To be embraced by developers, test generation tools need to support the major development frameworks.

V. Lessons Learned

Throughout this section, we discuss the challenges we faced in terms of tools setup and interaction with the developers, and suggest technical improvements that can be addressed by the test generation tools.

A. Tool Set up Challenges

One of the major challenges we faced during this experiment was setting up each *LifeCalc* version with its required dependencies and ensuring its successful compilation. We had to make sure all environmental dependencies, some of which contained sensitive information, are mocked and available to *LifeCalc*.

As for experiments we used the command line versions of Randoop and EvoSuite, we had to derive the right classpaths of the system under test. Given that one application might have dependencies to many third party libraries, the manual resolving of dependencies was an error prone task. Basically we had to check the tools' logs to determine which library is still missing. Then we would add those libraries to the classpath to get a proper execution. For instance *LifeCalc-b3* was dependent on *Apache Commons Lang*, and when this library was missing on classpath, EvoSuite would silently generate empty test suites. After disabling minimization (*-Dminimize=false*) and enabling the debug (*-Dlog.level=debug*) mode in EvoSuite, we managed to identify the missing libraries and re-executed all the experiments for this particular fault.

However, in retrospective, as the target application was built with Maven, we could have used some of its EvoSuite's plugins to derive the right classpaths and properly setup all the needed dependencies. Note that EvoSuite does have a plugin for Maven, but it is not suitable for the type of experiments we ran in this paper. Generally, the documentation of these test data generation tools could be extended to explain how to use the command line versions on existing projects compiled with build tools like Maven, Ant and Gradle.

Insight 1: The use of unit test generation tools on the command line requires detailed understanding of the build infrastructure, and tool documentations are currently not helpful in achieving a correct setup.

B. Suggested Improvements

Based on the results of our experiment, the following are potential improvements for the test generation tools:

Construction of complex objects: Since for most of the *Challenging Faults* the generated test cases failed to satisfy the outer condition, due to incapability in constructing and populating complex objects, one priority for tool builders should be to improve the construction and population of complex objects.

Insight 2: 100% of the challenging faults remained undetected as none of the tools were able to construct and populate objects with complex structure. More research on how to solve this problem is required.

Generating specific input: In most of the *Hard Faults* the faulty statements are not covered as there is a conditional statement prior to them that requires specific primitive data. There are some cases where the faulty statement was covered, but only a very specific set of primitive data would trigger the failure. For example, in the code snippet highlighted in the Listing IV-B3 line #11, the faulty statement should not only be covered, but `getPaymentFrequency()` also needs to be 0 in order to throw a `java.lang.ArithmeticException`. One good example of improving value generation is implemented within EvoSuite by extracting `enum` values as a set of required input data.

Insight 3: Only 47.78% (EvoSuite) and 12.22% (Randoop) of hard faults which require specific primitive values have been detected, even if the faulty statements are executed. Covering code is not enough: further criteria to optimize should be designed to help these tools in generating this kind of input values.

Extension of assertions: We encountered cases such as *LifeCalc-b21*, specified below, where the fault could have been detected by the generated test case with a better assertion.

```
...
for (int i = 2; i <= Months.monthsBetween(param1, param2).
    getMonths(); i++) {
  // Faulty Statement
  list.add(i, list.get(i - 1));
  // Fixed Statement
  list.add(i, list.get(i - 1).plusMonths(1));
}
...
```

To detect this fault, tests need to check the content of the list. However, generated assertions tend to only consider direct observer methods of the objects in the test (e.g., `isEmpty()`, `size()`), and thus only check for the list size.

Insight 4: At least 50% (EvoSuite) and 64% (Randoop) of the specification faults could have been detected with more appropriate assertions. More research in effective assertion generation would hence be useful.

C. Developer Feedback

We demonstrated the tools and the results of our study to the *LifeCalc* developers. They were interested about the possibility of integrating automated test generation tools with continuous integration tools such as Travis and Jenkins. This is currently in its early implementation stage, as tools like EvoSuite have provided beta versions of a Jenkins plugin.

Insight 5: Developers in industry expect automated test generation tools to integrate with standard continuous integration tools. For an effective technology transfer from academic research to industrial practice, building plugins for these tools would be useful.

The other comments of the developers were related to the readability of the generated tests, and difficulties in navigating through the generated test suites.

Insight 6: Developers in industry are concerned about the readability of generated unit tests, the generated input data, and the generated assertions. These are topics that would warrant further research.

Given that test readability is a concern for developers, smaller automatically generated test suites may be more preferred to read and analyze. Table IV reports the size of test suites based on the total number of generated test methods. As expected from a random testing-based tool, Randoop has generated up to approximately 53,000 test methods (*LifeCalc-b1*) while EvoSuite's test suite size did not exceed 32 test methods. EvoSuite performs a minimization to ensure that redundant tests are excluded. It would be more practical if test generation tools filter all redundant tests throughout the process, and it would be beneficial to prioritize the generated tests in a way to detect faults earlier, specially in cases where the generated test suite is huge. This way, the test execution could have stopped sooner, for cases where the fault is detected, and the test execution resources would be optimized. This point may be more important for Randoop that tends to generated larger test suites. Moreover, recent extensions of EvoSuite and Randoop to support popular build management frameworks, such as Maven and Gradle, suggest that tool development is heading in the right direction.

TABLE IV
Total generated test methods for all faults per execution setup (3 and 15 minutes) for both EvoSuite and Randoop

Fault	**EvoSuite**		**Randoop**	
	3 min	15 min	3 min	15 min
LifeCalc-b1	2	3	31371	52636
LifeCalc-b2	4	4	1294	3579
LifeCalc-b3	4	4	2977	3965
LifeCalc-b4	15	15	20019	27150
LifeCalc-b5	18	32	14502	17899
LifeCalc-b6	4	4	13893	17643
LifeCalc-b7	6	6	3122	5992
LifeCalc-b8	7	14	12630	15767
LifeCalc-b9	4	4	6893	9092
LifeCalc-b10	2	2	18488	23119
LifeCalc-b11	8	9	2121	6655
LifeCalc-b12	12	12	3022	6987
LifeCalc-b13	21	22	9876	12876
LifeCalc-b14	3	3	5433	7652
LifeCalc-b15	8	10	1232	3989
LifeCalc-b16	2	2	2457	4998
LifeCalc-b17	5	7	11542	13432
LifeCalc-b18	10	17	15432	17878
LifeCalc-b19	3	3	5679	8553
LifeCalc-b20	14	14	4390	7658
LifeCalc-b21	8	14	8992	11234
LifeCalc-b22	7	10	11675	21245
LifeCalc-b23	16	24	1959	3009
LifeCalc-b24	13	19	2832	6721
LifeCalc-b25	6	11	9772	14289

D. Threats to Validity

In our experiment, we only considered two major test generation tools representing search-based and random testing approaches. However, tools based on dynamic symbolic execution or any other approaches might be more suitable in the case where there is a complex condition need to be satisfied. For each of the selected tools, EvoSuite and Randoop, we have used their latest version with default settings. The tools might perform better if some of the settings are fine tuned. Since we had a limited number of known faults, in order to mitigate internal validity threat, we managed to analyze all the fault detection results to ensure that they are failed with the same reason as the manually written test cases, by going through the produced error logs. However, given the limited number of faults in the experiments, we might have some external validity threats, which we tried to mitigate by asking developers to provide the faults rather than choosing the faults ourselves.

The categorization of the undetected defects was manual, by taking into account the execution results with certain level of subjectivity in the process. We reduced that by having multiple people with different level of expertise going through the categorization independently to mitigate this risk.

We had only five participants in the survey but all of them are professional developers with certain level of familiarity with system under test. In addition, our participants didn't have prior knowledge with automated unit test generation but they were provided with a guideline on how to setup, execute and generate tests for EvoSuite.

VI. Related Work

Recently, researchers have shown an increased interest in automated software testing. There are a number of studies in which tools and techniques are being evaluated in terms of coverage. For example, Fraser and Arcuri [13] evaluated EvoSuite on 110 open-source projects. Besides decent levels of achieved coverage, they reported challenges they had due to practical limitations such as environmental dependencies, which Arcuri *et. al.* [3] later addressed. This work was relevant in our experiments as we had a set of faulty classes with environmental dependencies, such as the file system. Xiao *et. al.* [24] identified complex domain object creation as one of the main challenges in unit test generation, which we also encountered throughout our experiment (see discussion on "challenging faults" in Section IV-B3).

The work of Shamshiri *et. al.* [23] is perhaps the most related one to what is presented in this paper. Shamshiri et al. evaluated the effectiveness of automated test generation tools (e.g., Randoop and EvoSuite), but in contrast to our work they used open source projects (the Defect4J benchmark). Quantitatively, the number of faults detected on these open source software is comparable with what is reported in this paper for our industrial case study: Shamshiri *et. al.* report a fault detection rate of 55.7%, which is comparable to our conducted experiment, in which the test generation tools found up to 56.4% of the real faults. In addition, our paper also has an additional qualitative study (the survey).

VII. Conclusions

In this paper, we performed a systematic study to determine the effectiveness of automatically generated test suites in terms of revealing real industrial faults. We used two of the most common test generation tools in academia, EvoSuite and Randoop. We evaluated them on a life insurance and pension products calculator engine developed by *SEB Life & Pension Holding AB Riga Branch*. Our experiment results demonstrate that test generation tools detected up to 56.40% (EvoSuite) and 38.00% (Randoop) of faults in all executions.

Our fault categorization shows that at least 41% of the undetected faults are on *Hard Faults* (they remained undetected due to the tests not being able to satisfy specific primitive values required by the faulty methods) and *Challenging Faults* (test generation tools were not able to detect the faults due to the incapability of constructing complex objects as input data). Increasing the search budget had minimal impact on fault detection rate in case of *Hard Faults*, but it increased the fault detection rate by at least 5.71% for *Easy Faults*.

We have investigated the challenges that need to be addressed by test generation tools in order to be adapted by practitioners by conducting a survey with developers. Based on the survey result and our analysis, the tools are not yet there to be used by industry but certainly they are on the right track. We analyzed the undetected faults in order to find out the areas which test generation tools can be improved and hopefully our concrete insights will lead the future research to address these challenges.

Acknowledguments

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada, the National Research Fund, Luxembourg (FNR/P10/03), and EPSRC project (EP/N023978/1).

References

[1] Apache maven. https://maven.apache.org. Accessed: 2017-02-04.

[2] Jenkins. https://jenkins.io. Accessed: 2017-02-04.

[3] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 79–90. ACM, 2014.

[4] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[5] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for Java. In *Software testing, verification and validation (ICST), 2010 third international conference on*, pages 185–194. IEEE, 2010.

[6] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[7] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[8] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[9] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, (4):438–444, 1984.

[10] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[11] G. Fraser and A. Arcuri. Evosuite at the second unit testing tool competition. In *Future Internet Testing*, pages 95–100. Springer, 2013.

[12] G. Fraser and A. Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, 2013.

[13] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[14] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[15] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Trans. on*, 38(2):278–292, 2012.

[16] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223, 2005.

[17] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82. ACM, 2014.

[18] H. Hemmati. How effective are code coverage criteria? In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 151–156. IEEE, 2015.

[19] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[20] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 75–84. IEEE, 2007.

[22] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Search-Based Software Engineering*, pages 93–108. Springer, 2015.

[23] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Automated Software Engineering, 30th IEEE/ACM International Conference on*, pages 201–211, 2015.

[24] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux. Precise identification of problems for structural test generation. In *Proceedings of 33rd International Conference on Software Engineering*, pages 611–620, 2011.