Search-Based Detection of Deviation Failures in the Migration of Legacy Spreadsheet Applications

M. Moein Almasi
University of Manitoba, Canada

Hadi Hemmati
University of Calgary, Canada

Gordon Fraser
University of Passau, Germany

Phil McMinn
University of Sheffield, UK

Janis Benefelds
SEB Life & Pension Holding AB, Latvia

ABSTRACT

Many legacy financial applications exist as a collection of formulas implemented in spreadsheets. Migration of these spreadsheets to a full-fledged system, written in a language such as Java, is an error-prone process. While small differences in the outputs of numerical calculations from the two systems are inevitable and tolerable, large discrepancies can have serious financial implications. Such discrepancies are likely due to faults in the migrated implementation, and are referred to as *deviation failures*. In this paper, we present a search-based technique that seeks to reveal deviation failures automatically. We evaluate different variants of this approach on two financial applications involving 40 formulas. These applications were produced by SEB Life & Pension Holding AB, who migrated their Microsoft Excel spreadsheets to a Java application. While traditional random and branch coverage-based test generation techniques were only able to detect approximately 25% and 32% of known faults in the migrated code respectively, our search-based approach detected up to 70% of faults with the same test generation budget. Without restriction of the search budget, up to 90% of known deviation failures were detected. In addition, three previously unknown faults were detected by this method that were confirmed by SEB experts.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Empirical software validation**; *Search-based software engineering*;

KEYWORDS

Search-based testing, spreadsheet, automated test generation, deviation failure, financial applications

ACM Reference Format:
M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Phil McMinn, and Janis Benefelds. 2018. Search-Based Detection of Deviation Failures in the Migration of Legacy Spreadsheet Applications. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3213846.3213861

1 INTRODUCTION

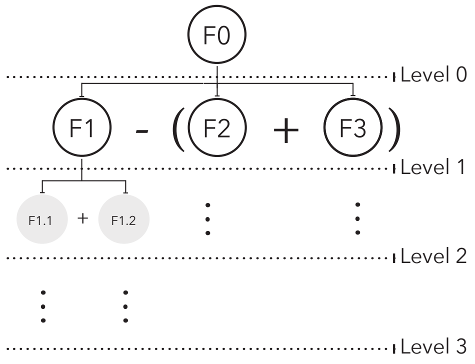
Spreadsheets are traditionally used in financial corporations to perform complex calculations. Demands on performance, precision, support for automation, conformance with other in-house tools and frameworks, changes to a corporation's IT's strategic plans, and many other reasons may force a corporation to migrate from spreadsheets to more generic solutions such as web-based services or applications. The reimplementation of a legacy spreadsheet application may, however, introduce faults. If such faults cause the outputs of the calculations performed by the reimplementation to deviate from the outputs of the calculations of the original spreadsheet — i.e., a *deviation failure* occurs — then serious financial implications may be at stake.

Testing is an important activity for identifying such issues [22], but finding specific inputs that reveal deviation failures is a challenging task. In particular, financial calculations may involve small and inconsequential deviations in the outputs due to the precision differences of arithmetic operations in different languages. Companies define formula-specific thresholds of tolerable deviations (i.e., error margins), and as such the challenge lies in finding inputs that produce deviations larger than the thresholds specified.

Automation is often used to support testers in challenging testing activities. Techniques such as random testing [9, 19, 20], search-based testing [1, 6, 12, 16], or those based on symbolic execution [3, 7, 8, 25] can effectively generate test inputs automatically. However, a general limitation of automated test generation is that it relies on the existence of a test oracle [2] that can decide whether the system is behaving correctly for an automatically generated test input. This is particularly challenging for complex applications such as those in the financial domain, where it is difficult to determine what the correct output should be for a specific input. However, in the case of the reimplementation of legacy spreadsheet programs, the legacy spreadsheets specify the expected behavior, and can thereby serve as test oracles, making automated test generation possible.

In this paper, we introduce a new automated search-based test generation approach that aims to find tests that maximize the deviation failures between a given spreadsheet and its reimplementation. Search-based approaches [18] use techniques such as genetic algorithms to transform software engineering problems into optimization problems, where the objective of the test generation process is encoded by a fitness function that guides the search. The main fitness function in our scenario is the difference between a root (top level) formula's return value in Excel, compared to its Java implementation. We refer to this approach as the Output-based Search Technique (OST). To improve fault finding, and to help localize faults, we also propose a novel spreadsheet-based technique,

Figure 1: The Excel formula hierarchy for *Invested Premium*



referred to as the Spreadsheet-based Search Technique (SST), which recursively focuses the search on lower-level sub-formulas.

We evaluate these techniques using a total of 40 formulas with known and previously unknown faults from two products developed by SEB Life & Pension Holding AB. The first product, a financial calculator engine known as *APP1*, is a newly implemented life insurance and pension products calculator engine written in approximately 80,000 lines of Java code. The second product, called *APP2*, is a pension funds calculator [13], also developed in Java, and consists of approximately 170,000 lines of code. Both were developed to replace legacy Excel spreadsheet calculators.

The original motivation for our research came from SEB reporting that a substantial amount of manual effort that had to be invested to identify deviation failures in the Java reimplementations. This was because deviation failures were typically missed by the developer-written unit test cases, and could only be detected by a series of manual acceptance tests that were expensive to construct and conducted by business analysts.

The contributions of this paper are therefore as follows:

(1) A new search-based test generation approach, the Output-based Search Technique (OST), which aims to detect deviation failures using a fitness function that compares the outputs of original spreadsheet formulas against their Java reimplementations.
(2) A modified fitness function that utilizes a spreadsheet's nested formula structure, implemented into an approach called the Spreadsheet-based Search Technique (SST). SST aims to improve the detection of deviation failures and assist in subsequent fault localization.
(3) The evaluation of the proposed OST and SST approaches using 40 complex formulas with real deviation failures (known and previously unknown) in two commercial financial applications. As part of this evaluation we compare the success rates and actual deviation values of our approaches with automatically generated test cases for branch coverage of the Java code, randomly generated test cases, as well as manual, developer-written test cases. In addition, all reported deviations were double checked and confirmed SEB experts.

Our empirical studies with *APP1* show that our top-level search-based approach (OST) correctly revealed 47% of the deviation failures, which represents a substantial 22% improvement over a baseline of random testing and 10% over test cases optimized for branch coverage. Using our advanced SST method, a further 23% of previously known deviation failures are detected. Finally, relaxing time constraints and increasing the search budget led to 90% of known deviation failures being detected. Applying this approach on *APP2*, we also managed to detect three new deviation failures that were not previously detected by the manual test cases.

```java
public Double investedPremium(AppObj obj, Double targetAmount,
                              String strategy) {
  premium = Util.round(paidInsurance(obj) - (adminFees(strategy) +
      riskFees(targetAmount)), 100);
  return premium;
}
```

Listing 1: Java reimplementation of the "Invested Premium" formula (F0)

```java
public Double paidInsurance(AppObj obj) {
  Double investmentReturnRate =
      Double.valueOf(rateProperties("lv.return.rate"));
  paidInsurance = futureMonthlyContrib(obj.getCurrentAmount(),
      obj.getMonthlyPayments(), investmentReturnRate,
      obj.getRetirementTime(), obj.getSavingTime(),
      obj.getFutureAssets()) +
      currentAssets(obj.getPensionAssets());
  return paidInsurance;
}
```

Listing 2: Java reimplementation of the "Paid Insurance" formula (F1)

```java
public Double futureMonthlyContrib(Double currentAmount,
        Double monthlyPayments, Double investmentReturnRate,
        Integer retirementTime, Integer savingTime,
        Double futureAssets) {
  monthlyContrib = (currenAmount + monthlyPayment) *
      (((Math.pow((1 + (investmentReturnRate / 12)),
      (retirementTime - savingTime))) - 1)) /
      (investmentReturnRate / 12) + futureAssets
      * Math.pow((1 +(investmentReturnRate / 12)),
      (retirementTime - savingTime));
  return monthlyContrib;
}
```

Listing 3: Java reimplementation of the "Monthly Contribution" formula (F1.1)

2 MOTIVATING EXAMPLE

To provide a better understanding of the specifics of spreadsheet-based financial applications and the challenges faced during their reimplementation, we present an anonymized example. Figure 1 shows the hierarchy of the a legacy Excel formula to calculate "Invested Premium" in our first case study ("*APP1*-f1" in our evaluation of Section 4).

In Formula F0, P is the "Invested Premium", I is the "Paid Insurance", A represents the "Administrative Fees", while R corresponds to the "Risk Fees". The formula rounds to the nearest multiple of 100, via the "*MROUND*" function. Listing 1 shows the reimplementation of the formula in Java.

$$P = MROUND(I - (A + R), 100) \tag{F0}$$

In the next formula, F1, I is the "Paid Insurance", M is the "Monthly Future Contribution", while S represents the "Current Assets". F1 computes M as a subformlua of F0. Listing 2 shows its reimplementation in Java.

$$I = M + S \tag{F1}$$

```
@Test(timeout = 4000)
public void test1()  throws Throwable  {
  Pension pension0 = new Pension();
  AppObj appObj0 = new AppObj();
  Double double0 = new Double(30.0);
  appObj0.setCurrentAmount(double0);
  appObj0.setPensionAssets(double0);
  Double double1 = pension0.investedPremium(appObj0, 0.0, "S1");
}
```

Listing 4: Sample test case generated with the goal of obtaining branch coverage

```
@Test(timeout = 4000)
public void test3()  throws Throwable  {
  Pension pension0 = new Pension();
  AppObj appObj0 = new AppObj();
  Double double0 = new Double(10.0);
  appObj0.setCurrentAmount(double0);
  Double double1 = new Double(25.0);
  appObj0.setMonthlyPayments(double1);
  Double double2 = new Double(11.0);
  appObj0.setSavingTime(double2);
  Double double3 = new Double(100.0);
  appObj0.setFutureAssets(double3);
  Double double4 = new Double(40.0);
  appObj0.setRetirementTime(double4);
  Double double5 = pension0.investedPremium(appObj0, 838.0, "S1");
}
```

Listing 5: Sample test case generated by our proposed search technique

Formula F1.1 calculates the value of an asset at specific future date based on its growth interest rate over a period of time.

$$M = FV(rate, nper, pmt, pv, type) \tag{F1.1}$$

Here, *FV* ("*Future Value*") represents monthly future contributions, *rate* is the interest rate per period, *nper* is the total number of payment periods, *pmt* is the payment made in each period, *pv* is the present value, and *type* indicates due payments. Listing 3 presents the reimplementation of this formula in Java. In the reimplemented code, *rate* is `investmentReturnRate/12`, which is the monthly investment return rate, *nper* is `retirementTime−savingTime`, which is the remaining months until retirement, *pmt* is `currentAmount + monthlyPayment`, which is the current contributed amount plus the upcoming monthly contribution and *pv* is `futureAssets`, which is based on the value of future assets. The Java reimplementation of this formula contains a fault — the reference to the variable `futureAssets` (shown in bold in Listing 3) is missing in the faulty Java reimplementation. This causes it to produce a different output compared to its Excel implementation. Thorough testing may reveal this problem. However, finding a specific input that reveals the failure is non-trivial. It is not enough to just execute the faulty code, since the effect of the fault may not propagate to the output and produce a failure.

For instance, Listing 4 presents the test case generated automatically as part of a branch-coverage optimized test suite by the EvoSuite tool [5]. This test case executes (covers) the fault, but does not reveal a failure. The calculated *Invested Premium (P)* in both programs is 1000, and therefore, based on the generated test data, the difference between the output of the legacy Excel formula and its Java reimplementation of *Invested Premium (P)* is 0.

Listing 5 shows a test case generated by our approach. This test case results in a difference in the output of the *Invested Premium (P)* in the Excel and Java programs as presented in Table 1 ($\Delta = |1100 - 1200| = 100$). The difference results from the faulty reimplementation of *Invested Premium (P)* in one of the sub-formulas (i.e., the Monthly Contribution calculation, Formula F1.1) and is ($\Delta = |1204 - 1258| = 54$).

Table 1: Invested Premium calculation using generated tests case in Listing 5.

Variables	P	I	A	R	S	M
Excel	1100	1604.8	500	0	400	1204.9
Java	1200	1658.8	500	0	400	1258.8

In this paper, we aim to automatically generate test cases that uncover such differences. We propose a two search-based approaches. The first uses a fitness function that, in this example, would be the delta between the results of the the "root" calculation (*Invested Premium (P)*) in Excel and Java. The second uses a fitness function that focuses on the delta between lower level sub-formulas. The next section introduces the proposed approaches in detail.

3 SEARCH-BASED APPROACHES FOR DEVIATION FAILURE DETECTION

In this section, we describe two search-based approaches that aim to generate tests that maximize the deviation between an Excel formula and its Java reimplementation. Our techniques focus on one Excel formula and its Java implementation at a time, and the tests are generated as JUnit tests for the Java implementation.

3.1 Deviation Failures

Deviation failures are failures where two alternative implementations produce different outputs. In financial applications, like many other domains, not all deviations are immediately considered as a failure, that is, some degree of variation in the outputs is tolerable. These tolerable variations are specified in the form of threshold limits, or error margins, defined for each formula and its sub-formulas. These error margins are business-specific and are inputs for the automated techniques. For instance, in our study, we were provided with a list of error margins, from the company expert. Consequently, a deviation failure is deemed to have occurred when the output of two programs differ by a value equal to or more than its specified error margin (threshold limit):

Definition 1 (Deviation Failure). A deviation failure occurs when the numerical output o_1 from a program p_1 differs from the output o_2 for a similar program p_2 by a difference equal to or greater than a tolerable threshold (i.e., error margin) t. That is, $|o_1 - o_2| \geq t$.

Note that, in practice, the differences smaller than the threshold are ignored since they tend to be rooted in small precision differences in arithmetic operators in the two implementation languages, and are not due to defects. Their actual effect on the final output of the system tends to be negligible.

For ease of understanding, we normalize deviation failures in this paper, such that violations are represented by numbers ≥ 1.0, regardless of the actual formula-specific tolerable threshold values t (which we cannot report due to the confidentiality reasons):

Definition 2 (Normalized Deviation Failure). A normalized deviation failure occurs when the numerical output o_1 from a program p_1, and the output o_2 from a similar program p_2, divided by a tolerable threshold t, is equal to or exceeds 1.0; i.e., $\frac{|o_1 - o_2|}{t} \geq 1.0$.

3.2 The Output-based Search Technique (OST)

Search-based test generation techniques, in general, formulate the test objective as an optimization problem and apply meta-heuristic search algorithms, such as genetic algorithms, to find an optimal solution [18]. Typically, the fitness function drives the search to generate test cases that cover as much of the code as possible. In this study, however, we focus on the detection of deviation failures. We therefore first propose a search-based method that guides the search for test cases towards deviations between program versions that exceed defined tolerable thresholds.

We refer to this method as the Output-based Search Technique (OST). Assume the Excel formula under test (F0) is re-implemented by a method called M0, in Java. Let M0 and F0's input parameters be $x_0, \ldots, x_n$. Each test case may therefore be represented by the vector $< x_0, \ldots, x_n >$, and a fitness function can be defined as:

$$FF\_OST = |\mathrm{F0} - \mathrm{M0}|$$

To implement OST, we used the search-based test generation tool EvoSuite [5], and extended it with our own fitness function. OST's stopping criterion can be any typical criterion such as a timeout, maximum number of generations or fitness evaluations, etc.

The other search operators such as crossover, mutation, selection strategy, etc. are the same as default settings in EvoSuite (i.e., the crossover operator combines different parents (*P1* and *P2*) to generate new offspring (*O1* and *O2*) which is then mutated by adding, modifying or deleting statements. Once the reproduction is over, there will be new parents ready for selection).

We identified the corresponding Java method (M0) per Excel formula (F0) manually in this study. It can be potentially automated for instance using keyword matching, but it may become very imprecise. Therefore, we did not attempt to automate this step.

3.3 Spreadsheet-based Search Technique (SST)

OST explores the top level formula of a spreadsheet only. The Spreadsheet-based Search Technique (SST) delves into sub-formulas. Assume F0 from the previous section consists of two sub-formulas F1.1 and F1.2, which are implemented in M1.1 and M1.2, respectively. M1.1 and M1.2 are called in M0 and have the input parameters $< a_1, \ldots, a_i >$ and $< b_1, \ldots, b_j >$. Now, assume the defect that results in a deviation failure is in M1.2 and it will be detected only with a specific value sets for $< b_1, \ldots, b_j >$. Following OST, the search algorithm's inputs are $< x1, \ldots, x_n >$ thus there is no specific guidance towards values for $< b_1, \ldots, b_j >$ during the search. The *FF_OST* fitness function is detached from the sub-formula inputs, and only looks at the final output.

To overcome this problem we need to isolate the underlying deviated sub-formula at each level, starting from root formula all the way to its leaves. The Spreadsheet-based Search Technique (SST), implements this idea by starting from root-level deviations but narrowing it down to the defective method over time. At a high level, our proposed SST algorithm is a recursive version of OST, as follows:

(1) It applies an OST on a level i formula F$i.x$ (where i is the level starting from 0 — the root level — and x is the sequential ID for each sub-formula in the given level) (see Figure 1).
(2) The OST searches through the input space of M$i.x$ parameters. M$i.x$ is a Java method that corresponds to the F$i.x$ formula.
(3) The search continues until a deviation between the outputs of F$i.x$ and M$i.x$ is detected.
(4) After finding a deviation, the search continues for t seconds (default is $t = 60$), while still in level i, to potentially increase the already detected deviation.
(5) If the algorithm manages to increase the deviation in t seconds, step 4 is repeated.
(6) The algorithm stops exploring level i and moves to level $i+1$, when the best deviation could not be improved in t seconds.
(7) To move to level $j = i + 1$, the F$i.x$'s sub-formulas (Fj.1, ..., F$j.y$) are analyzed to find out which one is the "most contributing" sub-formula. The "most contributing" sub-formula is the one with the highest local deviation, where a local deviation is the delta between the sub-formula's output and its corresponding Java method's output, in level j.
(8) The algorithm repeats steps 1–7, until it reaches a global stopping criterion (e.g., it times out).

3.3.1 Preparation Step. There are at least three preparation steps that are required before running any of our SST techniques, as follows:

- **Identifying the hierarchy of sub-formulas per Excel formula**. We have automatically collected this information and recorded it in a matrix using a custom Excel macro.
- **Mapping Excel formulas to Java methods**. As mentioned before, we performed this task manually. The first co-author of the paper mapped all identified Java and Excels calculations by hand. Note that the thresholds per sub-formula was already available to us as "error margins" in the applications.
- **Dealing with external dependencies**. In order to avoid external dependency issues, such as access to properties files, we have replaced all the variables retrieved from property files with the actual values, in the Java code.

4 EMPIRICAL STUDY

In order to evaluate our proposed search-based test generation techniques for detecting deviation failures, we conducted an empirical study on two real financial applications. This section describes the details and results of this study.

4.1 Research Questions

The objective of the study has been broken into four research questions as follows:

RQ1 *How effective are the output-based (OST) and spreadsheet-based search techniques (SST) compared to baseline test generation at detecting deviation failures?* The aim of this research question is to evaluate how a focused search-based approach for deviation detection compares to common existing test generation techniques at defecting defects. We also evaluate the improvement of the SST over OST approach, given the same search budget.

RQ2 *How often do tests generated for deviation faults violate implicit preconditions?* Automated test generation techniques, in general, may create invalid inputs. That is, in our study, there may be cases where a generated test exceeds the tolerable threshold, but violates domain and business related implicit preconditions that the test generator is not aware of. The aim of this question is to compare different techniques with respect to their invalid test input generation.

RQ3 *What is the deviation failure finding potential of the Spreadsheet-based Search Technique?* The aim of this research question is to assess the overall fault finding ability of SST, without tight constraints on the search budget.

RQ4 *Can the Spreadsheet-based Search Technique detect new, previously unknown deviation failures?* The aim of this research question is to evaluate whether results on SST generalize and can detect previously unknown deviation failures.

4.2 Subjects of Study

Our experiments are based on two real financial applications. The first application is a life insurance and pension products calculator engine known as *APP1*. It is a medium-sized standalone software component with approximately 80,000 lines of code, and consists of complex critical pension products calculations with many business rules. Its implementation started in early 2015, and was been released to production in early 2016. The second application is *APP2*, a pension funds calculator engine [13] with approximately 170,000 lines of code. Both applications are developed and owned by SEB.

Initially, *APP1* and *APP2* were implemented using Excel sheets and have been used by the company for several years. For many internal strategic reasons, including better automation support and technology compatibility with their other products, these products have been reimplemented in Java. Throughout the implementation, the original Excel sheets have been used as the specification for the implementation of the new applications.

For RQ1–3, the company's developers provided us with 20 spreadsheet formulas from *APP1*. For each of these formulas they further provided us with a deviation failure report of the Java-based reimplementation, which had been resolved in their issue tracking system. They selected these formulas and failures arbitrarily, trying to include examples from different times throughout the lifecycle of the project. For each spreadsheet formula, we extracted the corresponding Java program version along side with any manual tests that had been used in order to detect and fix the particular fault.

For RQ4 (detection of unknown faults) we could not use the same formulas, since we already know that each of them contains contains a fault. Therefore, we further randomly selected 20 spreadsheet formulas from *APP1* and *APP2* (ten each), without knowing whether they contain any deviation failures. We did not have access to any manually written test cases for these formulas.

Table 2 reports characteristics of the studied 40 formulas in terms of some Excel and Java metrics. The "Levels" (number of nested levels in the hierarchy of the formula starting from 0 for the root formula) and "Sub-formulas" (total number of sub-formulas in all levels) are extracted from the Excel formula directly. However, the "Variables" (the total number of unique variables in corresponding Java methods for all sub-formulas in all levels) and "LOC" (the total number of lines of Java code for all sub-formula of the root) are extracted from the Java code[1]. These statistics show that the studied formulas cover a wide range of complexity, with "Variables" ranging from 3 to 36, and "Sub-formulas" ranging from 1 to 16.

Table 2: Characteristics of 40 formulas in terms of #Unique variables (V), #Nested levels in the formula hierarchy (L), #Total sub-formulas within all levels (S) and total #Lines of Java code (C).

(a) 20 known faulty formulas

Formula	V	L	S	C
***APP1*-f1**	8	2	3	41
***APP1*-f2**	15	3	9	165
***APP1*-f3**	12	4	7	109
***APP1*-f4**	6	2	3	56
***APP1*-f5**	24	3	12	270
***APP1*-f6**	19	6	10	307
***APP1*-f7**	26	2	8	120
***APP1*-f8**	21	3	11	235
***APP1*-f9**	12	2	8	154
***APP1*-f10**	5	1	4	89
***APP1*-f11**	31	6	16	534
***APP1*-f12**	18	3	9	192
***APP1*-f13**	36	5	13	389
***APP1*-f14**	7	1	5	104
***APP1*-f15**	3	1	1	13
***APP1*-f16**	10	1	2	29
***APP1*-f17**	8	4	5	95
***APP1*-f18**	7	2	3	45
***APP1*-f19**	27	5	10	216
***APP1*-f20**	15	6	9	182

(b) 20 NEW formulas

Formula	V	L	S	C
***APP1*-f21**	23	4	9	233
***APP1*-f22**	13	3	7	112
***APP1*-f23**	11	3	5	77
***APP1*-f24**	9	3	6	117
***APP1*-f25**	14	4	9	203
***APP1*-f26**	33	7	14	309
***APP1*-f27**	25	5	13	230
***APP1*-f28**	17	2	4	78
***APP1*-f29**	11	4	7	145
***APP1*-f30**	18	3	6	97
***APP2*-f1**	29	3	7	142
***APP2*-f2**	15	3	7	134
***APP2*-f3**	11	4	9	234
***APP2*-f4**	7	1	3	80
***APP2*-f5**	13	2	5	71
***APP2*-f6**	19	4	8	173
***APP2*-f7**	5	2	4	101
***APP2*-f8**	8	1	3	42
***APP2*-f9**	22	4	10	217
***APP2*-f10**	11	3	7	88

Note that in all the cases, the deviation failure has been revealed using the manual test cases written by the developers *after* the bug had been reported by business analysts. In other words, the initial test cases written by the developers were not able to detect these faults. The business analysts' manual acceptance testing is the last resort before releasing the product. Therefore, late detection of deviation failures is expensive and risky. Hence, our goal is to provide an automated test generation approach that can detect such faults with smaller budget and earlier in the development phase.

4.3 Experiment Design

In this section, we discuss the design of our experiment to answer each research question.

4.3.1 Normalization. For each execution of a technique in our experiments, we record the maximum deviation detected in the root-level. Remember that deviations are considered a failure only if they are above certain tolerable threshold associated with the formula. Therefore, to help with readability of the tables, the deviations are normalized by dividing them by the actual thresholds as per Definition 2. As per the definition, zero means no deviation, values in (0, 1) are tolerable deviations, and any value greater than or equal to 1.0 is considered a detected deviation failure.

Our main goal is to detect any deviation equal to or greater than 1.0. However, the higher the deviation the better, assuming that higher values of deviations may correspond to more serious effects. For example, suppose the system's threshold for an annual pension

[1]http://metrics.sourceforge.net

value is \$0.001, and so a \$0.002 deviation is a non-tolerable failure but may not affect the customer satisfaction by large if not detected and fixed right away. However, a deviation of \$10 or \$100 will create a much bigger impact and dissatisfaction among clients. Note that we follow the assumption of the company on the thresholds and do not report less than threshold deviations as failures since they would be simply ignored by the practitioners.

4.3.2 *Invalid Test Cases.*

Automated test generation techniques can create invalid inputs. For example, in *APP1*-f16 an input value must not be more than 100,000.0 and a generated test with a value 184,414.16 is not acceptable based on business assumptions. These business assumptions can only be automatically derived and preserved if they explicitly exist in the form of constraints in the code. Otherwise, an automated test data generator like ours can generate invalid test inputs. Therefore, the only reliable way for us to validate the results is to ask domain experts. For all results in our experiments we therefore validated the solutions (test cases) with SEB's domain experts.

4.3.3 *Metrics.*

The metrics used in our results are as follows:

- **Median Root Deviation:** The median of normalized deviations in the root-level, over 30 runs, per technique per failure.
- **Median Sub-Formula Deviation:** The median of normalized deviations in a specific sub-formula, over 30 runs, per technique per failure. Note that this metric will only be used for SST techniques in RQ3.
- **Success Rate:** The ratio of detecting a deviation ≥ 1.0, over all 30 runs per formula per technique.
- **Validated Success Rate:** The ratio of detecting a "valid" deviation ≥ 1.0, over all 30 runs per formula per technique (validated by company's experts).
- **Invalid Test Case Ratio:** The ratio of reported deviations that are due to "invalid" test data over total reported deviations, in 30 runs per formula (validated by company's experts).

4.3.4 *Statistical Tests.*

Whenever we compare deviation values directly, we not only look at the median values, but also run a non-parametric statistical significance test (using the Mann-Whitney U test) to make sure the differences between two techniques are not due to chance. We also report the median Success Rates and median Validated Success Rates, over all formulas under study.

4.3.5 *RQ1 Methodology.*

This research question compares OST and SST against two baseline test generation approaches:

- **Random testing:** Random search is used as a sanity check. If OST is not better than random testing, then there would be no need for a search-based approach, because the search problem is either very simple, or too challenging. We used EvoSuite's random test generation to do a random search with the same search budget as OST (with the minimization option disabled).
- **Branch coverage testing:** Code coverage criteria such as statement and branch coverage are quite common in industry, and they are commonly targeted by search-based test generation. We compared our approach with the coverage-based test generation implemented in EvoSuite. Since coverage based test generation is also search-based, this allows us to focus only on the different fitness functions (from maximizing code coverage to deviation values).

As all studied test generation approaches, in this paper, are implemented in EvoSuite, the comparisons have fewer confounding factors related to the implementation and optimization.

In this RQ, we use the 20 known faulty formulas of *APP1* (explained above) as our targets. Each technique (R: Random Technique, BR: Branch Coverage Technique, OST: Output-based Search Technique, and SST Spreadsheet-based Search Technique) was executed 30 times per faulty formula, resulting in a total of 600 executions. For SST, we used a timeout of $t = 60$ seconds for each level. As each formula is implemented in a single Java class, each execution consisted of applying EvoSuite on the corresponding class with either random test generation, branch coverage optimization, or the OST fitness function. We set a global stopping criterion for all four algorithms as five minutes.

4.3.6 *RQ2 Methodology.*

This RQ investigates how often the solutions provided by any of the discussed approaches violates company's implicit business rules. To measure this we asked the company's experts to manually investigate our solutions. Note that as discussed, such invalid test cases are unavoidable when the rules are not explicit.

4.3.7 *RQ3 Methodology.*

In order to explore the maximum power of the SST-based approach regardless of the underlying cost, this RQ uses the same time interval as SST ($t = 60$ seconds) at each level, but it does not impose a global timeout (e.g., the 5 minutes used in RQ1). Instead, the algorithm searches until there are no sub-formulas left. To distinguish this configuration from the configuration used in RQ1 and RQ2, we call this version Extended SST (ESST). Obviously, this algorithm is going to be more expensive than RQ1 techniques, which is why RQ3 is instead focused on analyzing the ultimate effectiveness of an SST approach.

4.3.8 *RQ4 Methodology.*

For RQ4, we compared ESST with manual testing, as a baseline. We used 20 new spreadsheet formulas, which were different from those used in RQ1–3. These formulas were not given to us as formulas known to be faulty; we selected them randomly from two applications *APP1* and *APP2* (10 formulas each). This means that at the time of experimentation 1) we did not know which formulas contained faults, if any, 2) for the detected deviations we did not know if manual testing had uncovered a deviation there as well or not, and 3) we did not know if ESST had missed any deviation. Therefore, this dataset was perfect for analyzing the effectiveness of ESST in detecting unknown deviation failures. After generating tests, we validated all deviations found with the developers.

4.4 RQ1: How effective are OST and SST compared to baseline test generation at detecting deviation failures?

Table 3 summarizes the Median Root Deviations for R, BR, OST, and SST for the 20 known failures of *APP1*. All Median Root Deviations

Table 3: Comparing the Median Root Deviations over 30 executions per 20 formulas of *APP1* using R (Random), BR (Branch Coverage), OST (Output-based Search), and SST (Spreadsheet-based Search) Techniques. Those highlighted are detected deviations above threshold per technique.

Formula	R	BR	R	SST
***APP1*-f1**	1.18	1.25	2.62	1.53
***APP1*-f2**	0.64	0.88	0.99	1.72
***APP1*-f3**	0.40	0.54	0.82	1.10
***APP1*-f4**	0.78	1.15	1.63	1.87
***APP1*-f5**	0.84	1.14	0.99	1.82
***APP1*-f6**	0.46	0.64	0.96	1.50
***APP1*-f7**	0.86	0.90	1.36	1.88
***APP1*-f8**	0.73	0.57	0.99	1.25
***APP1*-f9**	0.42	0.43	0.57	0.83
***APP1*-f10**	3.24	4.26	5.74	3.88
***APP1*-f11**	0.61	0.98	0.97	1.39
***APP1*-f12**	0.65	0.55	0.88	1.83
***APP1*-f13**	0.69	0.68	0.84	1.13
***APP1*-f14**	1.06	1.09	1.50	1.75
***APP1*-f15**	2.01	1.38	2.30	1.80
***APP1*-f16**	1.97	1.68	2.11	1.53
***APP1*-f17**	0.55	0.60	0.79	1.37
***APP1*-f18**	0.94	0.82	0.88	1.96
***APP1*-f19**	0.61	0.70	0.96	1.33
***APP1*-f20**	0.61	0.66	1.03	1.81
Median	0.71	0.85	0.99	1.63
Mean	0.96	1.05	1.45	1.67

≥ 1.0 are highlighted as well. Overall, in 19 out 20 cases SST's Median Deviations are above thresholds. In contrast, OST, BR, and R only exceed the threshold in only 8, 7, and 5 cases out of 20, respectively.

Looking at the actual Median Root Deviation values, in most cases, BR is better than R (14 out of 20 cases) and OST is better than both (18 out of 20). Finally, SST is better than OST in 16 cases out of 20. Since the raw comparison of medians may be misleading, we also ran a statistical significance Mann-Whitney U test ($\alpha = 0.05$), which shows that OST's results are significantly different (with higher medians) in 14 cases out of 20 compared to Random and 13 cases out of 20 compared to Branch.

Comparing SST and OST, with respect to the Mann-Whitney U Test in 4 out of 20 cases (*APP1*-f1, f10, f15, and f16) the differences are not significant. Therefore, overall, in 16 out 20 cases SST approaches outperforms OST.

Calculating the root-level deviations of SST requires manual propagation of SST to root-level, which is very labor intensive. In the propagation process, developer inputs were used where inputs were not generated for a lower level formula that could be propagated up through the formula hierarchy. In our case, 50 hours of manual work was required to apply it on 600 solutions of SST (20 formulas each 30 runs). Note that the manual propagation is not part of the SST algorithm. We apply this just to calculate the extra evaluation metric.

Table 4: Invalid Test Case Ratio (ITR) and Median Validated Success Rates (SR) for Random (R), Branch Coverage (BR), Output-based (OST), and Spreadsheet-based Search (SST) Techniques for 20 known faults in the *APP1*. The undetected deviations are represented by (-).

Formula	R (%)	BR (%)	OST (%)	SST (%)
***APP1*-f1**	40.00	22.73	19.05	17.39
***APP1*-f2**	14.29	21.43	6.67	4.76
***APP1*-f3**	-	-	20.00	4.55
***APP1*-f4**	22.22	31.58	15.79	17.39
***APP1*-f5**	18.18	11.11	20.00	17.86
***APP1*-f6**	27.27	22.23	0.00	10.00
***APP1*-f7**	33.33	14.29	10.00	16.67
***APP1*-f8**	40.00	-	0.00	14.81
***APP1*-f9**	-	-	0.00	0.00
***APP1*-f10**	46.15	40.00	33.33	26.67
***APP1*-f11**	0.00	26.67	0.00	3.85
***APP1*-f12**	0.00	0.00	0.00	0.00
***APP1*-f13**	14.29	25.00	10.00	18.52
***APP1*-f14**	6.67	27.78	19.05	5.00
***APP1*-f15**	43.48	10.53	19.23	8.70
***APP1*-f16**	25.00	10.00	14.29	12.5
***APP1*-f17**	-	-	0.00	6.25
***APP1*-f18**	25.00	20.00	23.08	16.67
***APP1*-f19**	-	-	7.14	8.00
***APP1*-f20**	0.00	-	13.33	22.22
Median ITR (%)	16.23	12.69	11.67	11.25
Median Validated SR (%)	25.00	31.66	46.67	70.00

It is also important to note that SST not only finds higher root-level deviations and detects more failures, but it also localizes the fault by identifying an exact low-level sub-formula with high deviation. The more test budget, the better localization one can achieve. We explore this in more details in RQ3.

Overall, our results suggest that neither test generation based on random search nor on branch coverage are sufficient to detect deviation failures. However, the basic search-based approach seems interesting and may have some potential. But the best results are for the spreadsheet-based approach. The results also conform with the motivational example given in Section 2.

In 16/20 cases, SST provides the highest Median Root Deviation values compared to R, BR, and OST.

4.5 RQ2: How often do tests generated for deviation faults violate implicit preconditions?

Deviations may in practice be created by invalid test inputs. While this is unavoidable if preconditions are implicit, there is the potential concern at to whether optimizing deviations with a search-based approach leads to more violated implicit preconditions. Therefore, we asked the business experts to manually validate the created test cases. Table 4 summarizes the Invalid Test cases Ratio per each technique. Looking at R, BR, OST, and SST, in Table 4, we can see that Invalid Test Ratios, in general, are not very high (~11% for SST

Table 5: Sub-formula Deviation (D), Method ID (M) and Level (L) of detected failures using SST and ESST in the 20 known faults in *APP1*.

Technique	SST			ESST		
Formula	L	M	D	L	M	D
***APP1*-f1**	2	1	1.81	2	1	3.28
***APP1*-f2**	2	2	1.25	2	2	2.40
***APP1*-f3**	3	3	1.26	4	4	2.53
***APP1*-f4**	2	5	1.61	2	5	3.83
***APP1*-f5**	2	6	2.04	2	6	4.24
***APP1*-f6**	4	7	1.95	4	7	6.86
***APP1*-f7**	2	9	1.82	2	9	4.70
***APP1*-f8**	2	10	1.97	2	10	2.36
***APP1*-f9**	2	11	0.94	2	11	1.47
***APP1*-f10**	1	12	4.68	1	12	10.67
***APP1*-f11**	4	13	1.45	6	15	2.64
***APP1*-f12**	2	16	1.16	2	16	1.91
***APP1*-f13**	4	17	1.44	5	19	3.05
***APP1*-f14**	1	20	1.15	1	20	2.93
***APP1*-f15**	1	21	1.53	1	21	7.37
***APP1*-f16**	1	22	1.22	1	22	6.64
***APP1*-f17**	2	23	1.17	3	24	2.68
***APP1*-f18**	2	25	1.80	2	25	2.92
***APP1*-f19**	4	26	1.54	5	28	3.02
***APP1*-f20**	4	29	2.10	6	31	3.81
Median Validated SR (%)	70.00			90.00		

and OST). Therefore, the developers' time that is spent to inspect the reported deviations are in most case well paid off. We also observe that the number of Invalid Test cases produced by SST and OST are in the same range as random search and branch coverage techniques. In other words, the Invalid Test cases problem is shared among all automated test generation techniques, but seems to have limited negative effect, overall. Thus we can confirm that SST does not lead to more invalid test cases.

Table 4 also summarizes the effect of Invalid Test cases on success rates as Median Validated Success Rates. It shows that SST's validated success rate is at 70% whereas OST's is at 46.67%, and R and BR are far behind at 25% and 31.66%.

11% to 16% of deviations detected are due to invalid test data. However, SST does not increase the number of invalid test data, and provides a very high (70%) Validated Success Rate.

4.6 RQ3: What is the deviation failure finding potential of the Spreadsheet-based Search (i.e., ESST)?

Since RQ1 considered SST with limited search budget, there remains the question whether more faults could have been found with higher search budget, or if the experiments have already shown the full potential of SST. Table 5 summarizes the "Validated Success Rates" for SST and ESST.

As the table shows, ESST manages to find deviation failures above the threshold for **all** formulas, whereas SST misses one case. After considering invalid test data, ESST outperforms SST by 20% (70% vs. 90%), over the 20 formulas under study from *APP1* (same setup as RQ1).

Table 6: Average execution time used (minutes) over 30 executions for each failure, using ESST.

Formula	Avg. Time	Formula	Avg. Time
***APP1*-f1**	8	***APP1*-f11**	27
***APP1*-f2**	12	***APP1*-f12**	15
***APP1*-f3**	14	***APP1*-f13**	20
***APP1*-f4**	8	***APP1*-f14**	7
***APP1*-f5**	22	***APP1*-f15**	5
***APP1*-f6**	23	***APP1*-f16**	5
***APP1*-f7**	15	***APP1*-f17**	11
***APP1*-f8**	16	***APP1*-f18**	7
***APP1*-f9**	13	***APP1*-f19**	17
***APP1*-f10**	6	***APP1*-f20**	19

This substantial improvement of ESST over SST is due to the increased search budget. Table 6 summarizes the average cost of ESST, over 30 runs, for each of the 20 formulas. The costs are presented as minutes spent on average to finish one run of ESST. They range from 5 (e.g., *APP1*-f15) to 27 minutes (*APP1*-f11). Since different SSTs may go into different levels and find deviations in different sub-formulas, direct comparison of deviation values is not possible. However, it is interesting to see whether ESST has managed to localize the faults better than SST or not. We can measure this by counting the cases where ESST identifies a faulty sub-formula in a deeper level than SST.

Table 5 also lists the "Median Sub-Formula Deviations" for SST and ESST techniques together with their identified methods ID and level. As we can see in 5 out of 20 cases ESST has better localized the fault (deeper level faulty method is identified). In the other 15 cases out of 20, the two techniques identified the same method as faulty but ESST detected a higher deviation (all p-values are also less than 0.05). Therefore, we can conclude that the deviation failure finding potential of the Spreadsheet-based Search, as it is reflected in ESST, is much higher than the approach's basic setup (SST).

Although the maximum (27 minutes) is almost 5.5 times that of the SST and OST budget, it still looks quite reasonable compared to the time that is needed for manual testing: The faults that were identified as "detected" by manual testing, were only reported when in a separate sprint(s) business analysts went through a thorough and expensive set of manual acceptance testing. The analysts would use the domain knowledge and datasets that typical developers won't know about. Hence, in general, creating a failure deviation detecting test case by a developer required a round of unit level testing by the developer followed by at least one round of manual acceptance testing by the analysts to report the symptoms of the bug, which finally would result in a unit test case by developers to be added into regression test suite. It also worth mentioning that the required budget in ESST varies across formulas and is highly correlated with the size and complexity metrics reported in Table 2.

Given enough time, the Spreadsheet-based approach (ESST) found deviation failures for all formulas, and 90% of the deviations were validated by the developers.

Table 7: Invalid Test Case Ratio (ITR) and Sub-Formula Deviation (D) using ESST for the 20 new formulas.
(- / ✓/ ✗) represents no failure/detected/not detected using Manual techniques.

Technique	ESST		Manual
Formula	**ITR**	**D**	**Detected?**
***APP1*-f21**	-	-	-
***APP1*-f22**	6.66	93.34	✓
***APP1*-f23**	-	-	-
***APP1*-f24**	13.33	86.64	✓
***APP1*-f25**	6.66	93.34	✓
***APP1*-f26**	-	-	-
***APP1*-f27**	23.33	76.64	✓
***APP1*-f28**	-	-	-
***APP1*-f29**	-	-	-
***APP1*-f30**	-	-	-
***APP2*-f1**	16.66	83.34	✗
***APP2*-f2**	20.00	80.00	✗
***APP2*-f3**	-	-	-
***APP2*-f4**	-	-	-
***APP2*-f5**	-	-	-
***APP2*-f6**	-	-	-
***APP2*-f7**	-	-	-
***APP2*-f8**	-	-	-
***APP2*-f9**	10.00	90.00	✗
***APP2*-f10**	-	-	-
Median Validated SR (%)	86.67		57.14

4.7 RQ4: Can ESST detect new and unknown deviation failures?

In total, deviation failures were found for 7 out of the 20 new formulas at least once. Among the 7 faulty formulas, three of *APP2* had not been detected by Manual testing but were detected by ESST. After validating the results, the SEB confirmed that these faults are new and are due to their limited manual acceptance testing for the new product. Four faults were detected in *APP1*, and these were confirmed as known faults. Note that *APP1* is an older application with a lot of time already spent on manual acceptance testing by the business analysts. No known faults were missed by ESST.

Table 7 presents the Ratio of Invalid Test cases for the new 20 formulas of the two applications under study (*APP1* and *APP2*), for ESST. It also shows which formulas were identified as faulty by manual testing. Note that the information of which formulas contain known faults was not available at the time of test generation; we obtained this information *after* test generation, when validating our tests with the domain experts.

To summarize, Table 7 also shows the "Validated SR" for ESST (86.67%) and Manual testing (57.14%) in the second 20 formulas, which were randomly selected. Manual testing misses all three deviations that ESST detected in *APP2*, which indicates that more manual acceptance testing is required for that application.

All failure detecting manual unit test cases were written after the faults were reported by the analysts. This means that the original manual unit test cases' "Validated Success Rate" was zero. It also means that getting the reported success rates by manual testing requires the business analysts involvement in manual acceptance testing and thus is very expensive, compared to the automated ESST's approach which can be integrated with developers unit testing framework.

ESST detected 86.67% of deviation failures (including three unknown failures), whereas the expensive manual testing detected 57.14%, in the second set of 20 formulas.

4.8 Threats to Validity

The main threat to the validity of this study is the generalizability issues. Given that the evaluation is done based on a case study in one company, we can not generalize the results to other applications. However, our case study is based on two real-world industry applications with real deviation faults, which might be considered as representative. In addition, even if the applications themselves might be representative, the selection of the first 20 formulas by the company, might have been biased. However, random selection of the second 20 formulas reduced that threat. In addition, we believe that the approach is quite generic for spreadsheet application migration and thus we encourage replication of this study.

In terms of conclusion validity, we have conducted each experiment 30 times and reported medians and statistical significance results. Ideally we would like to rerun the experiment with more executions to gain more confidence in the results.

We keep internal validity threats as low as possible, by using a common framework (EvoSuite) to implement all techniques. However, evolutionary algorithms for our objective may be required and there might have been errors in the tools and scripts.

Another threat here is the fact that a real bug may be missed if a technique can detect a minor deviation due to the bug but could not find a scenario that results in an above threshed deviation. Such cases will be simply missed by our approach due to ignoring deviations less than the threshold. However, the alternative (reporting all deviations regardless) would potentially end up in a lot of false positives. Thus we opted for the approach detailed in Section 3.

Finally, in terms of construct validity, we have reduced the threat by defining very basic and clear measures for deviations and success rates. We also validated our solutions by the company experts. However, the domain experts may have given incorrect answers.

5 RELATED WORK

Differential testing [15] such as regression testing and N-version testing aims to demonstrate the behavioral differences between two versions of a program executed with the same test inputs. Evans and Savoia [4] presented differential testing with the intention of detecting more changes as compared to regression testing alone. It generates tests for both original and alternative systems and compare both versions with these two test suites. Furthermore, Tao Xie *et. al* [24] extended differential testing for object-oriented programs. They proposed a framework called *Diffut* in which it simultaneously executes methods of two versions of the program with the same inputs and compare their outputs.

Another related work in the context of automated regression testing is BERT (BEhavioral Regresstion Testing) [14]. BERT tries to provide more insight to the developers as compared to the traditional regression tests by focusing on subset of code and identifying the behavioral differences between two versions of a program. *DiffGen* is another approach presented by Taneja and Xie [23] which tries to reveal the behavioral differences of two version of Java programs by instrumenting the code and adding new branches to expose the differences between two version of class if these branches are covered by test generation tools.

McMinn [17] introduced a novel pseudo-oracle by utilizing testability transformation. Testability transformation is source-to-source program transformation to make the program under test more testable [10, 11]. Basically, the original program, which has no test oracle, will be automatically transformed into another program with the same functionality. Then the test cases aim at fining differences in the outputs of the two programs. In a recent work, Matthew Patrick et al. [21] utilized pseudo-oracles in their new search-based technique for testing various implementations of stochastic models with the intention of maximizing the differences between the original implementation and its respective pseudo-oracle. They used Kolmogorov-Smirnov tests to compare the distributions of outputs from each implementation and concluded that their technique reduces the testing effort and also enables discrepancies, where they could have been overlooked.

6 CONCLUSIONS

Detecting deviation failures in financial applications is difficult because of the potentially large input domain of financial formulas. We have introduced a new search-based approach to address this problem by generating tests to maximize the discrepancies between the newly implemented program (Java implementation) and its legacy version (an Excel spreadsheet). Our approach explores not only the final outputs of formulas but also the respective sub-formulas. The exploration time of each sub-formula level and the global stopping criterion can be tuned (t seconds).

We have evaluated our proposed techniques on two complex pension product calculators, *APP1* and *APP2*, using real financial deviation failures. The new proposed approach outperforms random and branch coverage test generation approaches. Furthermore, we compared both of our proposed approaches, multi-level sub-formulas search-based approach with the Output-based Search approach, in which spreadsheet-based approach produced up to 23.3% better detection rate as well as better fault localization. Finally, we showed that our results is quite cost-effective in practice, given the higher validated success rates (86.7%) compared to the expensive manual testing (57.1%). In terms of future work, one direction is to automatically map the spreadsheet formulas to Java methods, to fully automated the proposed test generation process. Another direction, is to investigate other methods of designing an SST, for example, by choosing to explore sub-formulas differently. Finally, we would also like to more thoroughly evaluate the fault localization aspect of SST compared to other localization techniques.

ACKNOWLEDGEMENTS

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada [RGPIN/05108-2014].

REFERENCES

[1] Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2017)*. 263–272.

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[3] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Communications of the ACM* 56, 2 (2013), 82–90.

[4] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007): Companion Papers*. ACM, 549–552.

[5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*. 416–419.

[6] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[7] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2013. Improving search-based test suite generation with dynamic symbolic execution. In *International Symposium on Software Reliability Engineering (ISSRE 2013)*. 360–369.

[8] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2014. Extending a search-based test generator with adaptive dynamic symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA 2014)*. 421–424.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. 213–223.

[10] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. 2002. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*. 1359–1366.

[11] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

[12] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *International Conference on Software Testing, Verification and Validation (ICST 2015)*. 1–12.

[13] Investopedia. 2017. Pension Pillars. (2017). http://www.investopedia.com/terms/p/pensionpillar.asp

[14] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated behavioral regression testing. In *International Conference on Software Testing, Verification and Validation (ICST 2010)*. IEEE, 137–146.

[15] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

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

[17] Phil McMinn. 2009. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Genetic and Evolutionary Computation Conference (GECCO 2009)*. ACM, 1689–1696.

[18] Phil McMinn. 2011. Search-based software testing: past, present and future. In *International Conference on Software Testing, Verification and Validation (ICST 2011) Workshops*. 153–163.

[19] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *International Conference on Object-Oriented Programming Systems and Applications (OOPSLA 2007) Companion*. ACM, 815–816.

[20] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, 75–84.

[21] Matthew Patrick, Andrew P Craig, Nik J Cunniffe, Matthew Parry, and Christopher A Gilligan. 2016. Testing stochastic software using pseudo-oracles. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*. 235–246.

[22] Sohon Roy, Felienne Hermans, and Arie van Deursen. 2017. Spreadsheet testing in practice. In *International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*. 338–348.

[23] Kunal Taneja and Tao Xie. 2008. DiffGen: Automated regression unit-test generation. In *International Conference on Automated Software Engineering (ASE 2018)*. 407–410.

[24] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. 2007. Towards a framework for differential unit testing of object-oriented programs. In *International Workshop on Automation of Software Test (AST 2007)*. 5–5.

[25] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *International Conference on Dependable Systems & Networks (DSN 2009)*. 359–368.