Test Case Analytics: Mining Test Case Traces to Improve Risk-Driven Testing

Tanzeem Bin Noor, Hadi Hemmati
Department of Computer Science
University of Manitoba
Winnipeg, Canada
{tanzeem, hemmati}@cs.umanitoba.ca

***Abstract*—In risk-driven testing, test cases are generated and/or prioritized based on different risk measures. For example, the most basic risk measure would analyze the history of the software and assigns higher risk to the test cases that used to detect bugs in the past. However, in practice, a test case may not be exactly the same as a previously failed test, but quite similar. In this study, we define a new risk measure that assigns a risk factor to a test case, if it is similar to a failing test case from history. The similarity is defined based on the execution traces of the test cases, where we define each test case as a sequence of method calls. We have evaluated our new risk measure by comparing it to a traditional risk measure (where the risk measure would be increased only if the very same test case, not a similar one, failed in the past). The results of our study, in the context of test case prioritization, on two open source projects show that our new risk measure is by far more effective in identifying failing test cases compared to the traditional risk measure.**

Keywords*—*Risk; Execution trace; Testing; Bug; Risk-driven testing; Similarity; Test case prioritization.

I. Introduction

In risk-driven testing, test cases are generated and or prioritized (for example in the context of regression testing), by their degree of riskiness. Riskiness can be defined differently in different contexts. For example, in the context of safety-critical systems the severity of a failure can define how risky its detecting test case is. In the context of web services with thousands/millions of users, on the other hand, the likelihood of the failing test scenarios to be experienced by a typical user may be a more important risk factor. However, in general, a basic risk measure in risk-driven testing is defined as the probability of the test being failed in the past. In other word, if a test case detects a bug in previous releases, one should make sure that the test case is executed in the current release (after the new modifications), as well. This is very common practice in regression testing, where the goal is prioritizing test cases so that the more effective tests (in this case more risky ones) are being executed first. Test case prioritization is very important in practice for software companies, specially when continuous integration and rapid release demands fast development paces.

The typical risk measure [1] basically goes through the history of the software and checks whether any of the current test cases used to fail (detect a bug) in any of the previous releases. If so, the riskiness of the test case would be incremented per failing occurrence. The problem with this approach is that in many situations the test case that detects a bug is not exactly the same as any of the previously failing test cases. However, it is quite similar to some test cases, in terms of the sequence of methods being called (the test scenario).

Therefore, in this paper, we define a new risk measure that assigns a riskiness value to test cases when their execution trace (mined by Daikon tool [2]) is similar to the execution traces of any of the past failing test cases. We have evaluated our measure, in the context of test case prioritization, on two open source Java projects Commons Lang and Joda Time and the results show that our approach is always by far better than the old-fashion risk measure.

The rest of this paper is organized as follows: section II mentions our motivation; our proposed risk measure has been presented in section III. We have explained our experiments and results in section IV. Section V states some of the related works. Finally, section VI concludes the paper and mentions our future work.

II. Motivation

Traditionally, a test case would be considered as risky if it failed in the previous releases [1]. Now assume a test case, such as testLang747 (a test case from the latest version of Project Lang, explained in Section IV) in Fig. 1, that is just added to the current test suite and fails. It is obviously not risky according to the traditional definition of test riskiness, since it did not exist in the previous releases, to fail. However, there are some test cases in the past that are quite similar to this test case and they failed, e.g., the test case in Fig. 2. Given that testLang747 is risky (actually failed), it would be nice to have a risk measure for test cases that do not only look at exact occurrences of the test case in the past, but rather look at its similar cases as well.

The key question here is "how do we identify such similar test cases?". In this paper, we represent each test case with a sequence of method calls, which are extracted from their execution traces. For instance, the sequences for the two test cases of Fig. 1 and 2 are shown in Fig. 3(a) and Fig. 3(b). As it can be seen, $math.NumberUtils.createNumber(java.lang.String)$, $StringUtils.isBlank(java.lang.CharSequence)$ and $math.NumberUtils.isAllZeros(java.lang.String)$

```
@Test
public void testLang747() {
    assertEquals(Integer.valueOf(0x8000), NumberUtils.createNumber("0x8000"));
    assertEquals(new BigInteger("8000000000000000", 16), NumberUtils.createNumber("0x8000000000000000"));
    ....
    assertEquals(new BigInteger("FFFFFFFFFFFFFFFF", 16), NumberUtils.createNumber("0xFFFFFFFFFFFFFFFF"));
    assertEquals(Long.valueOf(0x80000000000000L), NumberUtils.createNumber("0x00800000000000000"));
    assertEquals(Long.valueOf(0x8000000000000000L), NumberUtils.createNumber("0x0800000000000000"));
    ...
    assertEquals(Long.valueOf(0x7FFFFFFFFFFFFFFFL), NumberUtils.createNumber("0x07FFFFFFFFFFFFFF"));
    assertEquals(new BigInteger("8000000000000000", 16), NumberUtils.createNumber("0x0008000000000000000"));
    assertEquals(new BigInteger("FFFFFFFFFFFFFFFF", 16), NumberUtils.createNumber("0x0FFFFFFFFFFFFFFF"));
}
```

Fig. 1. Failing test in latest version

```
@Test
public void testStringCreateNumberEnsureNoPrecisionLoss() {
    String shouldBeFloat = "1.23";
    String shouldBeDouble = "3.40282354e+38";
    String shouldBeBigDecimal = "1.797693134862315759e+308";

    assertTrue(NumberUtils.createNumber(shouldBeFloat) instanceof Float);
    assertTrue(NumberUtils.createNumber(shouldBeDouble) instanceof Double);
    assertTrue(NumberUtils.createNumber(shouldBeBigDecimal) instanceof BigDecimal);
}
```

Fig. 2. Failing test in previous version

```
math.NumberUtilsTest.testLang747()
math.NumberUtils.createNumber(java.lang.String)
StringUtils.isBlank(java.lang.CharSequence)
math.NumberUtils.createInteger(java.lang.String)
. . . . .
StringUtils.isBlank(java.lang.CharSequence)
math.NumberUtils.isAllZeros(java.lang.String)
. . . . . .
StringUtils.isBlank(java.lang.CharSequence)
math.NumberUtils.createInteger(java.lang.String)
```

(a) Test case trace of Fig. 1

```
math.NumberUtilsTest.testStringCreateNumberEnsureNoPrecisionLoss()
math.NumberUtils.createNumber(java.lang.String)
StringUtils.isBlank(java.lang.CharSequence)
math.NumberUtils.isAllZeros(java.lang.String)
math.NumberUtils.createFloat(java.lang.String)
math.NumberUtils.createNumber(java.lang.String)
StringUtils.isBlank(java.lang.CharSequence)
math.NumberUtils.isAllZeros(java.lang.String)
math.NumberUtils.createFloat(java.lang.String)
```

(b) Test case trace of Fig. 2

Fig. 3. Execution traces

methods are the same in the two test case traces, which makes the two test cases similar.

In the next section, we introduce a new test case risk measure based on the trace similarities, to capture the historical risk factor that is not extractable using the traditional risk measure, as explained in this section.

III. SIMILARITY BASED RISK MEASURE

To overcome the limitation explained in the previous section and empower test case risk measures, we introduce the Similarity-Based Risk Measure (SBRM). In a nutshell, SBRM works on the execution sequence level of test cases. The current implementation of SBRM is on the unit test level but it can easily be extended to component and system level testing as well.

The key idea of SBRM is representing each test case by a sequence of methods that are being called when the test is executed on the system under test. These sequences can be extracted dynamically (for example using Daikon [2]), or statically). Once the tests are encoded as sequences, the riskiness of a test case would be a function of its similarity to the failing sequences (test cases) in the past.

There are several similarity functions that one can use to define the risk measure. The main difference between these functions is whether they account for the order of the method calls in the sequence or not (i.e., they take it as a Set not a Sequence). In this paper, we use a very basic function which does not account for the method call orders nor for their position in the sequence. The function simply looks at the past failing sequences and counts the overlap of their method calls with the method calls of the test case under study. The total risk measure of a test case is the sum of all these occurrences.

For example, assume we are ranking a pair of test cases (T2 and T3) in the current release of a software (for instance as part of test prioritization in the regression testing context). Based on the history, we find a failing test case T1, which calls method m1, m2, m3, m2, m4, sequentially, from the source code. So, its trace would be T1: $< m1, m2, m3, m2, m4 >$. We identify m1, m2, m3 and m4 as risky methods and the total risk factor of these methods are 1, 2, 1 and 1, respectively. Now, in the current release, assume T2's trace is $< m0, m1, m2, m3, m4, m5, m6 >$ and T3 has the following trace $< m7, m3, m4, m8, m1, m8 >$. The method m1, m2, m3 and m4 from T2 also appeared in the buggy trace T1. So, the corresponding risk factors of T2 are $< 0, 1, 2, 1, 1, 0, 0 >$ (assuming there is no other failing test case in the past with a method call overlap with T2). Thus the overall riskiness of T2 is 5. Similarly, T3's risk factors, with the same assumption, are $< 0, 1, 1, 0, 1, 0 >$ and the overall riskiness is 3. Thus we would prioritize T3 to T2, in a risk-driven prioritization.

TABLE I. PROJECTS UNDER STUDY

Project	#Versions	#Test Methods
Commons Lang	65	2245
Joda Time	27	4130

IV. EVALUATION

In this section, we explain our dataset, experiment design and results.

A. Dataset

In this paper, we use two projects from *defects4j* database [3]. The database provides 357 bugs and 20,109 tests from 5 different open-source Java projects. All the bugs are real, reproducible and have been isolated in different versions [4]. There is a buggy version and a fixed version of the program source code, for each bug. The buggy source code is modified in the fixed version to remove the bug. The test cases are the same in both the buggy and the fixed version. However, there is at least one test case (a Junit test method) in each version that fails on the buggy version but passes on the fixed version.

In this paper, we study two projects (Table I) out of the five projects of *defect4j*. Note that the number of versions is equal to the number of bugs, in each project, due to the way *defect4j* isolates each bug in a separate version. In this paper, we have analyzed the latest 10 versions for each of these projects. Note that for the sake of reproducibility, all the raw data of the experiment is available online. [1]

B. Experiment Design and Setup

We run all test cases (all the test methods of all the test classes) that come with each of the 10 most recent versions. For a given version, all the previous versions are considered as history. We make two history database for two risk measures, studied in this paper.

First, for the traditional risk measure, we extract the failing test cases from each version and keep the failing counts per test method. For the SBRM, we extract the traces of each of the failing test methods and keep the failing counts for each internal method call. Note that for each version the history is made based on the previous versions, thus the counts would be different per version.

We have used daikon [2] tool to produce the execution traces. We run the JUnit tests individually using the daikon Chicory command. As we are only interested in the method calls of the source code (not the internal method calls of other referenced library, e.g., JUnit), some more options have been set to the daikon Chicory command.

When a test fails, a list of risky methods have been generated from its execution trace those work as historical data for the later versions. The overall riskiness of current test cases have been calculated based on the similarity between current release tests trace and buggy traces from the history (list of risky methods) as mentioned in section III. Finally, the tests have been sorted in their descending order of overall riskiness and ranked accordingly (test prioritization).

[1] http://sealab.cs.umanitoba.ca/wp-content/uploads/2015/01/Dataset.zip

Though we have generated the trace dynamically by running the tests from all versions, the trace could be statically generated for the current version, specially the newly added tests, to avoid executing them before prioritization. Another design decision was that, for each version under study, the traces have been collected only for all the test methods in failing test classes. This is just to keep the experiment size small, since the entire test suite was quite large. However, the approach can be extended to the entire test classes, if necessary, without any validity threat.

C. Results

Table II shows the results of our experiment on the Commons Lang project the Joda Time project. The table represents the rank of the failing test method(s) out of all the test methods in the class, using the two risk measures (SBRM and the traditional).

For both of the projects, Version 11 (V11) is the initial release and V1 is the latest release. As V11 is the first release, so no risk measure is available for this version based on the history. However, V10 has V11 as its history and similarly V1 gets information from V11 to V2, as history. The 'N/A' value in the last column of the tables means the risk measure can not be used for ranking because of the lack of data from history. This usually happens in the traditional risk measure, where the failing test method has been newly added in the current version and therefore, it has no failing history in the previous versions. It can also happen in both traditional and SBRM measures when the failing test's risk value is 0.

The first observation from the results is that SBRM definitely outperforms the traditional risk measure given that in most cases the failing test is newly added and the traditional approach is not even able to rank the test methods. However, only in two occasions, V10 and V7 of project Commons Lang, SBRM fails to rank.

Looking at the results we see that not only SBRM is able to rank in most cases but also it ranks quite well. In most cases, the failing test method is among the top 5 risky test methods. This is quite interesting and has potential use cases in test case selection and prioritization. Finally, we can also see that the results of SBRM is better for the more recent versions, where there is more history available. For example, in V6 the failing test is ranked 33 out of 93 tests in its failing test class using SBRM. However, the failing test is ranked 4 out of 81 tests in the latest release, V1. This is again promising since our measure is essentially a history-based approach.

In the 10 versions of the Lang project that we studied, exactly one test fails per version. However, multiple tests fail in some of the 10 versions of the Time project. Therefore, in Table II, you can see that 2 tests from different classes in project Time fail in V10. Also, V7, V6, V5 and V3, have more than one failing test method in the same test class. For example, 5 test methods fail in V3 from a class having 40 tests and all of them are ranked among top 5 using SBRM.

V. RELATED WORK

History-based risk measures, which are the focus of this paper, are widely used in the bug prediction literature [5].

TABLE II. TEST CASE RANKS BASED ON THEIR RISKINESS USING SBRM AND THE TRADITIONAL RISK MEASURES. (THE LOWER RANK MEANS HIGHER RISK) – PROJECT LANG AND PROJECT TIME

Project	Version	#Test Methods	SBRM-based Rank	Traditional Rank
Lang	10	27	N/A	N/A
	9	28	25	N/A
	8	14	1	N/A
	7	75	N/A	N/A
	6	93	33	N/A
	5	13	1	N/A
	4	2	1	N/A
	3	81	6	N/A
	2	14	2	N/A
	1	81	4	N/A
Time	10	29	3	N/A
	10	23	3	N/A
	9	45	17	N/A
	8	44	15	1
	7	74	62	N/A
			62	N/A
	6	13	2	N/A
			3	N/A
			4	N/A
			5	N/A
	5	81	66	N/A
			67	N/A
	4	69	36	N/A
	3	40	1	N/A
			2	N/A
			3	N/A
			4	N/A
			5	N/A
	2	72	19	N/A
	1	20	2	N/A

In most cases, the previous defects/bugs is an indicator of some sort of risk and thus is used in the predictive models. For example, Zimmerman *et al.* showed that the number of previous bug fixes correlates with the faults in the future versions [1]. In the context of testing, history-based risks are defined mostly for regression testing [6]. Basically, a risky test case is the one that use to fail in the past. This is very similar to the "previous bug" metric used in the bug prediction studies. Therefore, in this paper we use it as a baseline of comparison (traditional risk).

Risk-driven testing analyses different risk factors on the System Under Test (SUT) and ensures better test process execution in all of its phase, i.e., test planning, design, implementation, execution and evaluation [7]. For example, Shihab *et al.* [8], used a traditional risk metric for creating unit test for legacy systems. Kim and Porter also have used a traditional risk measure for prioritizing test cases in the context of regression testing [6].

In most cases, the risk should be defined explicitly and manually added to the tests or to the models that the tests are going to be generated from. Yoon and Choi have proposed risk measures to be determined by the domain experts from the risk management process and prioritize test cases accordingly [9]. However, it requires manual effort. Kloos *et al.* proposed a model based testing Using Fault-Tree analysis where the test cases are selected and prioritized based on the severity of risk values and the event that triggers the risk [10]. In this paper, we try to automatically extract the risk factor and assign it to the test cases.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have explained how the analysis of test case historical execution traces could help us to introduce a similarity-based risk measure. The measure showed promising results on 10 versions of two open source projects, in terms of risk-driven test prioritization. The similarity-based risk measure introduced in this paper can be implemented in many different ways, depending on the similarity function being used. Though our initial and simple implementation in this paper was very promising, we are planning to investigate other similarity functions, specifically those that account for the the method orders in the trace. In addition, this project is a sub-project of a bigger research on risk-driven model-based testing, where we are planning to extract specification models of the system and augment them with the similarity-based risk measures. Those models can later be used in both risk-driven test generation and prioritization.

REFERENCES

[1] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE, 2007, pp. 9–9.

[2] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[3] R. Just. (2014) Defects4j: A database of existing faults to enable controlled testing studies for java. [Online]. Available: http://homes.cs.washington.edu/~rjust/defects4j/

[4] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[5] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 675–689, 1999.

[6] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 119–129.

[7] M. Felderer, M.-F. Wendland, and I. Schieferdecker, "Risk-based testing," in *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*. Springer, 2014, pp. 274–276.

[8] E. Shihab, Z. M. Jiang, B. Adams, A. E. Hassan, and R. Bowerman, "Prioritizing the creation of unit tests in legacy software systems," *Software: Practice and Experience*, vol. 41, no. 10, pp. 1027–1048, 2011.

[9] H. Yoon and B. Choi, "A test case prioritization based on degree of risk exposure and its empirical study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 02, pp. 191–209, 2011.

[10] J. Kloos, T. Hussain, and R. Eschbach, "Risk-based testing of safety-critical embedded systems driven by fault tree analysis," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 26–33.