Entropy-Based Test Generation for Improved Fault Localization

José Campos* Rui Abreu* Gordon Fraser† Marcelo d'Amorim‡

*Faculty of Engineering of University of Porto, Portugal
jose.carlos.campos@fe.up.pt, rui@computer.org,
†University of Sheffield, United Kindom
gordon.fraser@sheffield.ac.uk,
‡Federal University of Pernambuco, Brazil
damorim@cin.ufpe.br

***Abstract*—Spectrum-based Bayesian reasoning can effectively rank candidate fault locations based on passing/failing test cases, but the diagnostic quality highly depends on the size and diversity of the underlying test suite. As test suites in practice often do not exhibit the necessary properties, we present a technique to extend existing test suites with new test cases that optimize the diagnostic quality. We apply probability theory concepts to guide test case generation using entropy, such that the amount of uncertainty in the diagnostic ranking is minimized. Our ENTBUG prototype extends the search-based test generation tool EVOSUITE to use entropy in the fitness function of its underlying genetic algorithm, and we applied it to seven real faults. Empirical results show that our approach reduces the entropy of the diagnostic ranking by 49% on average (compared to using the original test suite), leading to a 91% average reduction of diagnosis candidates needed to inspect to find the true faulty one.**

***Index Terms*—Fault localization; Test case generation.**

I. INTRODUCTION

Programmers make mistakes, therefore testing and debugging are inevitable parts of software development. Spectrum-based fault localization is a popular automated approach to assist programmers in debugging [5], [6], [21], [26], [30], [35]. It takes as input code coverage information for a given test suite, and produces a list of statements ranked in order of fault suspiciousness. Unfortunately, the technique is fundamentally imprecise: it is possible that a tester will need to inspect several suspicious statements in the rank to find the faulty one. In fact, a recent study [30] involving 68 developers failed to establish the usefulness of the approach as it only showed improvement for experienced programmers on simple code. Considering this result, is spectrum-based fault localization a dead-end avenue of research?

We conjecture that the answer is no. Research in spectrum-based fault localization has continuously advanced over the past few years. Many of its notorious initial limitations are no longer a problem:

- it can identify multiple faults [2], [27], [37], [38];
- it can aggregate faults scattered across the code [2], [27], [37], [38];
- it can quantify confidence of the diagnosis [14], [21].

C	Subject: Triangle	T						*new tests*		
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
	`class Triangle {...`									
	`static int type(int a, int b, int c){`									
c_1	`int type = SCALENE;`	1	1	1	1	0	0	1	1	1
c_2	`if ( (a==b) && (b==c) )`	1	1	1	1	0	0	1	1	1
c_3	`type = EQUILATERAL;`	1	0	0	0	0	0	0	1	0
c_4	`else if ( (a*a) == ((b*b)+(c*c)) )`	0	1	1	1	0	0	1	0	1
c_5	`type = RIGHT;`	0	0	1	0	0	0	1	0	0
c_6	`else if ( (a==b) || (b==a) ) /*FAULT*/`	0	1	0	1	0	0	1	0	1
c_7	`type = ISOSCELES;`	0	1	0	0	0	0	0	0	0
c_8	`return type; }`	1	1	1	1	0	0	1	1	1
	`static double area(int a, int b, int c){`									
c_9	`double s = (a+b+c)/2.0;`	0	0	0	0	1	1	1	1	1
c_{10}	`return Math.sqrt(s*(s-a)*(s-b)*(s-c));}...}`	0	0	0	0	1	1	1	1	1
Test case outcome (pass=0, fail=1):		0	0	0	0	0	0	1	0	1
Number of fault candidates:		6						4	2	1

Figure 1: Triangle class adapted from [4] with tests and coverage matrix; *type* classifies triangles based on the side lengths, and *area* calculates the area of the triangle. Test suite T produces an unusable ranking, but adding new test cases t_7, t_8, t_9 gradually narrows down the candidate statements until only c_6 remains.

Despite all these advances, there remains one fundamental limitation, which we argue is one of the main confounding factors for the usefulness of spectrum-based fault localization: *the dependency on the quality of the existing test suite* [23].

Figure 1 shows a variation [4] of the well-known triangle example. There is a fault at statement c_6: method *type* declares the predicate $b == a$ but the correct condition should be $b == c$. Assume we have received a bug report for the `Triangle` class. Using BARINEL [2] results in a ranking with a very low confidence score, and the tester would need to inspect six statements ($c_3, c_5, c_6, c_7, c_9, c_{10}$) prior to locating the fault. Note that none of the existing tests (t_1 – t_6) fails; even with a failing test the confidence score might be low, and thus the fault localization is unreliable and not helpful for the developer.

To overcome this limitation, this paper proposes ENTBUG, a novel technique to generate test cases specifically guided by diagnostic accuracy. Based on our knowledge of how the existing test cases cover the statements of the Triangle class, ENTBUG generates a new test case t_7, which is the test case that improves the confidence in the ranking the most over the existing test suite; test case t_7 fails. Adding t_7 to the original

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

suite reduces the average number of statements for inspection to four. Generating test case t_8 reduces the average number of statements for inspection to two, and finally, adding test case t_9 precisely pinpoints the faulty statement c_6 in the top of the rank, with high confidence.

To produce these new test cases automatically, our technique borrows ideas from probability theory. Given a diagnostic report, ranked by probability of diagnosis candidates being the true fault explanation, the uncertainty in the ranking can be quantified using *entropy* — a measure of uncertainty in a random variable [25]. To increase the quality of the ranking, we need test cases that maximize the diagnostic precision of the underlying fault localization algorithm, this way decreasing the entropy in the ranking [24], [25], [40]. In order to achieve this, ENTBUG uses a search-based technique to automatically generate tests that reduce uncertainty in rank reports. This paper makes the following contributions:

- An entropy-based strategy to optimize the quality of ranking reports;
- A fitness function based on entropy to guide search-based test generation;
- A prototype implementation of the described approach on top of the EVOSUITE [17] test generation tool;
- An evaluation of the approach using seven real faults;

To the best of our knowledge, this is the first attempt to generate tests based on the entropy of a test suite. ENTBUG was able to pinpoint the faulty statement in all real-world bugs we have considered in our benchmark. Compared to the original test suite, we observed an average reduction of 49% on the uncertainty of the diagnostic ranking, delivering an expressive reduction on the number of candidates that one needs to inspect before finding the fault by 91% on average. Comparing with random approach, ENTBUG reduces the uncertainty of the diagnostic ranking in 4.2 more times, and 2.5 times the number of candidates to explore until finding the faulty one.

II. BACKGROUND

ENTBUG is based on a spectrum-based reasoning approach to multiple fault localization [2]. In this section we summarize the background information of spectrum-based reasoning and automated test generation.

A. Spectrum-Based Reasoning

Spectrum-based reasoning is an approach to fault localization founded on probability theory. The main principles underlying the technique are based on Model-Based Diagnosis (MBD) [13], [15], [16], [21], [27], [39], which uses *logical reasoning* to find faults.

A *component* is a program statement. A *fault candidate* is a set of components that together explain a fault. Let the symbol C denote the set of program components and let the symbol D denote a set of fault candidates. For instance, $D = \langle\{c_1, c_3, c_4\}\rangle$ indicates that components c_1, c_3, and c_4 are *simultaneously* at fault, and no other. The approach sorts the faulty candidates in D by the probability of each candidate to explain fault. Spectrum-based reasoning is comprised of two phases: candidate generation and candidate ranking.

1) Candidate Generation: The problem of finding fault candidates can be defined in terms of the widely-known Minimal Hitting Set (MHS) problem [13]. The precise computation of MHS is highly demanding [19], restricting its direct usage for diagnosis. However, in practice, previous research has found that precise computation of D is not necessary [1]. STACCATO is a low-cost heuristic for computing a relevant set of multiple-fault candidates. To illustrate STACCATO, consider the `Triangle` class from Figure 1 and the test suite T. As all test cases in T pass, STACCATO yields a diagnostic report containing all components in the program, i.e., $D = \langle\{c_1\}, \{c_2\}, \{c_3\}, \{c_4\}, \{c_5\}, \{c_6\}, \{c_7\}, \{c_8\}, \{c_9\}, \{c_{10}\}\rangle$. But if we consider T augmented with the failing test t_7, STACCATO reports the following diagnostic report $D = \langle\{c_1\}, \{c_2\}, \{c_4\}, \{c_5\}, \{c_6\}, \{c_8\}, \{c_9\}, \{c_{10}\}\rangle$. Note that components c_3 and c_7 are not considered valid candidates because they have not been covered by any failing tests. For detailed information on STACCATO, interested readers may refer to [1].

2) Candidate Ranking: The candidate generation phase may result in an extensive list of diagnosis candidates. As not all candidates have the same probability of being the true fault explanation, techniques have been devised to assign a probability to each diagnosis candidate d_k. Each candidate d_k is a subset of the system components that, when at fault, can explain the faulty behavior. The probability a diagnosis candidate being the true fault explanation $\Pr(d_k|obs)$, given a number of observations obs, is computed using Bayesian probability updates. An observation obs is a tuple (a_{i*}, e_i).

Executing each test case t_i from the test suite T, the probability of each candidate is updated following Bayes' rule

$$\Pr(d_k|obs) = \Pr(d_k) \cdot \prod_{obs_i \in obs} \frac{\Pr(obs_i|d_k)}{\Pr(obs_i)} \tag{1}$$

$\Pr(d_k)$ is the *a priori* probability of the candidate (i.e., the probability before any test is executed), defined as

$$\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|} \tag{2}$$

where p is the a priori probability of a component being faulty. The prior probability given no observations is an approximation to 1 fault for every 1000 Lines of Code (LOC), $1/1000 = 0.001$ [12].

$\Pr(obs_i|d_k)$ represents the conditional probability of the observed outcome e_i produced by a test t_i (obs_i), assuming that candidate d_k is the actual diagnosis

$$\Pr(obs_i|d_k) = \begin{cases} 0 & \text{if } e_i \wedge d_k \text{ are inconsistent;} \\ 1 & \text{if } e_i \text{ is unique to } d_k; \\ \varepsilon & \text{otherwise.} \end{cases} \tag{3}$$

where ε is defined as

$$\varepsilon = \begin{cases} \prod_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 0 \\ 1 - \prod_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 1 \end{cases} \tag{4}$$

and a_{ij} represents the coverage of the component j when the test i is executed. As this information is typically not available, the values for $h_j \in [0, 1]$ are determined by maximizing $\Pr(obs|d_k)$ using maximum likelihood estimation [2]. To solve the maximization problem, a simple gradient ascent procedure [9] (bounded within the domain $0 < h_j < 1$) is applied.

$\Pr(obs_i)$ represents the probability of the observed outcome, independently of which diagnostic explanation is the correct one and thus needs not be computed directly. The value of $\Pr(obs_i)$ is a normalizing factor given by

$$\Pr(obs_i) = \sum_{d_k \in D} \Pr(obs_i|d_k) \cdot \Pr(d_k) \tag{5}$$

The BARINEL framework is used in our approach to compute the probabilities of each diagnosis candidate d_k. Further information about the framework and underlying technique can be found in [2]. When compared to other spectrum-based approaches to fault localization [2], BARINEL yields more information rich diagnostic reports due to the fact that it also reasons in terms of multiple faults. To illustrate this approach, the following probabilities are computed for the example in Figure I after executing test suite T (for detailed information about the probabilities, see Table I)

$$d_1 = \{c_1\}, \ \rightarrow \ \Pr(obs|d_1) = 1 \times h_3 \rightarrow 0.099984$$
$$\ldots$$
$$d_{10} = \{c_{10}\}, \ \rightarrow \ \Pr(obs|d_{10}) = 2 \times h_{10} \rightarrow 0.100004$$

After computing the probabilities for each $d_k \in D$, the candidates are ranked and shown to the user in descending order of probability to be the true fault explanation (see Table I).

B. Diagnostic Report Entropy

While debugging, developers can resort to the diagnostic ranking of diagnosis candidates yielded by BARINEL to pinpoint the root cause of observed failures quickly. This can be done by inspecting the ranked candidates in a descending order according to the diagnostic probabilities.

As the diagnostic ranking assigns probabilities to the diagnosis candidates, one may compute an entropy-based score quantifying the reliability of the ranking; this score conveys how confident one can be that the ranking helps finding the fault. The entropy $\mathcal{H}(D)$ [22] intuitively serves to quantify the capability to distinguish candidates in the set D. For example, the value of $\mathcal{H}(D)$ is very high for the set $D = \{0.3, 0.5, 0.5, 0.5\}$ because we have several elements in the set with the same value. Therefore, it is difficult to distinguish which element is more relevant, i.e., which of the candidates in the example with probability 0.5 of being faulty can explain the fault better. The minimum value (i.e., approximate ideal) for $\mathcal{H}$ is zero, in which case all elements in the set can be distinguished from one another. In our context, the maximum value is $\log_2(M)$, where M is the number of components. So, $\mathcal{H}(D)$ can be defined as:

$$\mathcal{H}(D) = -\sum_{d_k \in D} \Pr(d_k) \cdot \log_2(\Pr(d_k)), \quad 0 \leq \mathcal{H} \leq \log_2(M) \tag{6}$$

In the Triangle example we have 10 components $(c_1, \ldots, c_{10})$, and to represent the whole set C, in theory, we need 2^M states

$$2^x \geq M \rightarrow x = \log_2(M) \rightarrow x = \log_2(10) \rightarrow x = 3.322$$

So, the best value of $\mathcal{H}$ is the minimum value (zero) and the worst value is the maximum ($\log_2(\mathcal{H})$), $0 \leq \mathcal{H}(D) \leq 3.322$. Therefore, the entropy for the example is

$$\mathcal{H}(D) = -\Big(0.099984 \cdot \log_2(0.099984) + \ \ldots \\ + \ 0.100004 \cdot \log_2(0.100004)\Big) = 3.322$$

which corresponds to the maximum value. This means that the ranking suffers considerably from uncertainty, and we cannot distinguish which components are most probably at fault. Therefore, we are not able to diagnose the faulty program.

C. Automated Test Generation

The basis for a diagnosis is a test suite, and often the existing test suite is not optimized for producing high-quality diagnostic reports. Hence, is important to generate tests to improve diagnosis. There are many available test generation techniques [7]; Search-Based Software Testing (SBST) [28] is well suited for our scenario, as it has the distinct ability to optimize test cases and test suites based on custom non-functional properties. For example, SBST can generate test suites optimized for coverage and size at the same time, and SBST is also well suited to address test generation in a diagnosis context [10], [33].

SBST uses meta-heuristic algorithms to generate test cases for user-informed objectives. Global search algorithms such as Genetic Algorithms (GAs) are popular for domains where the neighbourhood of a candidate solution is very large, such as for example for unit tests represented as sequences of method calls. The EVOSUITE tool [17] uses GA to generate test suites for Java classes with respect to a given coverage criterion. The GA starts with an initial population of randomly-generated candidate solutions. A *fitness function* determines, for each individual of the population, a numerical value estimating its distance to the optimal solution. Individuals are selected from the population; those with better fitness value have higher probability of being selected. Selected individuals "evolve" according to pre-defined operators, and form a new population. This procedure continues until it either finds an optimal solution, or runs out of resources (e.g., it reaches timeout).

Debugging is usually seen as a complementary activity to software testing, typically done afterwards. However, there are some test generation techniques that aim to improve debugging [8], [10]. In addition, test generation — in particular, search-based testing — can also be helpful for failure reproduction [34] as well as debugging [33].

III. ENTROPY-BASED TEST GENERATION WITH ENTBUG

This section presents ENTBUG, a novel approach to improve fault diagnosis. ENTBUG receives a test suite as input and produces additional test cases for that test suite, such that the entropy in the diagnosis is reduced.

A. Calculating Entropy

Spectrum-based fault localization heavily relies on the diversity of coverage information across passing and failing test cases. The variety and number of test cases are two major factors to determine uncertainty in the ranking. Previous work [24] has shown that *variety* and *size* are directly related to $\bar{\rho}$, the *density of the coverage matrix* [21], [24], and that this metric can be used as a proxy for entropy. The density of a coverage matrix is the average percentage of components covered by test cases. It is defined as follows:

$$\bar{\rho} = \frac{1}{N} \cdot \sum_{i=1}^{N} \rho(t_i) \wedge 0 \leq \bar{\rho} \leq 1$$

where $\rho(t_i)$ refers to the coverage density of a test case t_i

$$\rho(t_i) = \frac{|\{j \mid a_{ij} = 1 \wedge 1 \leq j \leq M\}|}{M} \tag{7}$$

where N and M denote the number of test cases and the number of components, respectively. Low values of $\bar{\rho}$ mean that test cases exercise small parts of the program (sparse matrices), whereas high values mean that test cases tend to involve most components of the program (dense matrices). Intuitively, neither too-low nor too-high values of $\bar{\rho}$ are positive. Considering the example from Figure 1, we have $\rho(t_1) = \frac{4}{10}, \ldots, \rho(t_6) = \frac{2}{10}$. Consequently, the coverage density is $\bar{\rho} = \frac{0.4+\ldots+0.2}{6} \rightarrow \bar{\rho} = 0.400$, i.e., the test cases yield a coverage matrix density of 40% ($\bar{\rho} = 0.400$).

B. Optimal Coverage Matrix Density

Our aim is to reduce the entropy of a test suite, and the drop in entropy is known as *information gain* [25]. The information gain that a (new) test case provides is determined by the reduction of the size of the top-ranked suspect set. Assuming there are $|D|$ top-ranked suspects, a test t_i with coverage density $\rho(t_i)$ reduces the top-ranked set to $|D| \cdot \rho(t_i)$ components if t_i fails, and to $|D| \cdot (1-\rho(t_i))$ if t_i passes. Under these conditions, it has been previously demonstrated [24] that the information gain can be modeled as follows:

$$IG(\bar{\rho}) = -\bar{\rho} \cdot \log_2(\bar{\rho}) - (1-\bar{\rho}) \cdot \log_2(1-\bar{\rho}) \tag{8}$$

For our running example, the value of IG is equal to $-0.400 \cdot \log_2(0.400) - (1-0.400) \cdot \log_2(1-0.400) = 0.971$. The value of IG is optimal for $\bar{\rho} = 0.5$. Hence, a technique that is able to generate test cases such that the coverage density of the matrix is $\bar{\rho} = 0.5$ (provided there is a variety of test cases) will have the capability of reducing the diagnostic ranking entropy, and consequently improve the diagnostic quality of spectrum-based reasoning. Our approach augments the existing test suite with additional test cases with the goal of balancing the density of the coverage matrix.

Algorithm 1 ENTBUG Test Generation

Input: Program Π, Test Suite T, Search Budget Δt, Stopping Condition C
Output: Extended Test Suite T'

```
1:  T' ← T
2:  d ← |0.5 − DENSITY(T)|
3:  δ ← GETFITNESSFUNCTION(T')
4:  while ¬C do
5:      t_c ← EVOSUITE(Π, δ, Δt)
6:      if d > |0.5 − DENSITY(T' ∪ {t})| − ε then
7:          T' ← T' ∪ {t_c}
8:          d ← |0.5 − DENSITY(T')|
9:          δ ← GETFITNESSFUNCTION(T')
10:     end if
11: end while
12: return T'
```

C. Generating Tests Guided by the Coverage Matrix Density

The coverage matrix density $\bar{\rho}$ gives us a measurable goal to guide test generation. As we can measure the effect but cannot *construct* suitable test cases systematically, this is an ideal application for search-based software testing (SBST). In SBST, an optimization goal is formulated as a *fitness function*, and then efficient meta-heuristic search algorithms are guided by the fitness function to generate tests.

A fitness function takes as input a candidate solution, and calculates a numerical value representing the quality of the candidate, such that there is a strict ordering. In our case, the optimal solution is a test case that leads to $\bar{\rho} = 0.5$, consequently our fitness function for test case t for a given test suite T is:

$$\text{fitness}(t) = |0.5 - \bar{\rho}(T \cup \{t\})| \tag{9}$$

This fitness function turns the problem into a minimization problem, i.e., the optimization aims to achieve a fitness value of 0, which is the case if a solution is found such that $\bar{\rho} = 0.5$.

D. Entropy-Based Test Suite Extension

Algorithm 1 illustrates ENTBUG's test-generation procedure. The goal of the algorithm is to extend a potentially empty test suite with test cases for improving the diagnosis. It takes as input the program Π, the original test suite T, the search budget Δt one wants to invest in generating each individual test, and Boolean condition C which evaluates to true once the process should stop (e.g., timeout, fixed number of test cases, etc.). It produces an extension of T as output.

The density entailed by T, which is used to guide the test generation process, is calculated using the DENSITY function (Line 2) and the difference to 0.5 is stored in d to later compare if a generated test improves $\bar{\rho}$. Then, EVOSUITE is called using the fitness function δ to generate a test case using Δt as the search budget (Line 5). EVOSUITE returns the test case that gives the maximum diagnostic information, i.e., the one that minimizes the fitness function (Line 5). However, it may be the case that EVOSUITE fails to find a solution that improves

$\bar{\rho}$ and in that case t_c is ignored. If t_c improves $\bar{\rho}$ (within an accepted error margin ϵ), then it is added to the test suite T', a new fitness function δ is created, and the value of d is updated (Lines 6 – 10). These steps are repeated until stopping condition C holds.

E. Revisiting the Triangle Example

To measure the success of a diagnosis technique we use the *diagnostic quality* C_d, which is a numerical value that estimates the number of components the tester needs to inspect to find the fault [2], [21]. Note that C_d cannot be computed prior to computing the ranking: one does not know the actual position of true-fault candidates in the ranking beforehand. Because multiple explanations can be assigned with the same probability, the value of C_d for the real fault d_* is the average of the ranks that have the same probability:

$$\theta = |\{d_k | \Pr(d_k) > \Pr(d_*)\}|,\ 1 \leq k \leq M$$

$$\phi = |\{d_k | \Pr(d_k) \geq \Pr(d_*)\}|,\ 1 \leq k \leq M \qquad (10)$$

$$C_d = \frac{\theta\ +\ \phi\ -\ M_f}{2}$$

A value of 0 for C_d indicates an ideal diagnostic report where all M_f faulty components appear on top of the ranking, i.e., there is no wasted effort in inspecting other components. On the other hand, $C_d = M - M_f$ indicates that the user needs to test all $M - M_f$ healthy components until reaching the M_f faulty ones — this is the worst-case outcome. For the `Triangle` example, the value of C_d is 4.0 ($= \frac{3+6-1}{2}$) after executing test suite T. This means that on average one needs to inspect 4 components to find the fault.

Table I shows the progress of the diagnostic report when new tests are added to the test suite using EntBug. Each column shows a different stage in the augmentation process. Numbers in bold-font face under column "Pr(d_k)" indicate components which appear at the same or higher position as the reaul faulty component d_*. The final diagnostic report obtained with the suite $T + \{t_7, t_8, t_9\}$ is 0 (i.e., best). At the bottom of the figure, one can observe three metrics that, as explained before, strongly correlate with C_d: $\bar{\rho}$, IG, and $\mathcal{H}$. While C_d is dependent on d_*, which is unknown prior to finding the faults, the other metrics are independent. They are very important for two reasons: (i) they provide a confidence value on the quality of the diagnostic report, which is critical for the user to build trust on the report, and (ii) they provide a means to automate test generation. In particular, we observed that it is possible to derive from $\bar{\rho}$ the definition of a generation.

Table I: Impact of additional test cases on fault localization for the triangle example.

Ranking	T		$T + \{t_7\}$		$T + \{t_7, t_8\}$		$T + \{t_7, t_8, t_9\}$	
	d	Pr(d_k)	d	Pr(d_k)	d	Pr(d_k)	d	Pr(d_k)
1	{**c3**}	**0.1000140**	{**c5**}	**0.2903310**	{**c5**}	**0.3766646**	{**c6**}	**0.3478261**
2	{**c5**}	**0.1000140**	{**c6**}	**0.1720448**	{**c6**}	**0.2232045**	{c_4}	0.1739130
3	{**c7**}	**0.1000140**	{**c9**}	**0.1720448**	{c_4}	0.1098354	{c_9}	0.1739130
4	{**c6**}	**0.1000040**	{**c10**}	**0.1720448**	{c_9}	0.1098354	{c_{10}}	0.1739130
5	{**c9**}	**0.1000040**	{c_4}	0.08466051	{c_{10}}	0.1098354	{c_1}	0.04347826
6	{**c10**}	**0.1000040**	{c_1}	0.03629137	{c_1}	0.02354154	{c_2}	0.04347826
7	{**c4**}	0.09999400	{c_2}	0.03629137	{c_2}	0.02354154	{c_8}	0.04347826
8	{**c1**}	0.09998400	{c_8}	0.03629137	{c_8}	0.02354154		
9	{**c2**}	0.09998400						
10	{**c8**}	0.09998400						
C_d	4.0		2.0		1.0		⟩⟩ **0.0** ⟨⟨	
$\bar{\rho}$	0.400		0.457		0.475		**0.500**	
IG	0.971		0.995		0.998		**1.000**	
$\mathcal{H}$	3.322		2.651		2.445		**2.437**	

IV. Evaluation

We have conducted an empirical study to evaluate the extent to which EntBug is capable of improving diagnostic quality compared to a baseline technique that, to the best of our knowledge, incorporates all recent advances in spectrum-based fault localization [2]. As a sanity check, we have also compared EntBug with random test generation.

A. Experimental Setup

We have implemented EntBug using EvoSuite to drive test generation, and evaluated it on a set of real faults. A particular difficulty to address in our evaluation setup is the need to create test oracles: the test-generation procedure needs to decide whether a test it generates passes or not. The automated generation of test oracles is challenging [18], [29], [31]. This has to do with the fact that the behaviour of the software has to be known so that the right oracles are added to the test cases. As the oracle problem is orthogonal to this paper, we mitigated this problem by using the version of the subject after the fix was applied. Let P to be the faulty program and P' its fixed version, a test case t *passes* if $P'(t) == P(t)$; it *fails* otherwise. EvoSuite adds regression oracles to the tests that make it possible to make this comparison automatically.

To investigate how the diagnosis improves over time, we configured the GA in EvoSuite to run for 10 seconds per attempt (see Δt from Algorithm 1), and measured the diagnostic accuracy of the evolving test suite at five time intervals: 1, 2, 5, 10 and 30 minutes. To account for the randomness of the test generation, we repeated all experiments for 10 times, and all values reported are averaged.

As a sanity check, we also evaluated whether the test cases produced by EntBug improve the diagnosis over the same number of random test cases. To do this, we used EvoSuite to produce a fixed number of random tests. For example, if an original test suite is composed by 5 test cases, and at the end of 1 minute EntBug generated 10 additional test cases, we also generated 10 random test cases and compared the impact on the diagnosis.

B. Subjects

The requirements for choosing the subjects used in our evaluation are as follows: (1) the programs should be developed in Java, (2) the fault must be documented, and (3) the fix should be available to validate if EntBug is able to identify the exact place of the fault.

We chose the vending machine example used in previous work [11], and selected six new faults from four large open-source programs. For each subject, we analyzed recent bug

Table II: Details of subject programs.

Subject	Revision	Classes	LOCs	Original Test Suite / Test Cases Used	Bug ID	Affected Class
Vending Machine	-	2	54	1 / 1	-	VendingMachine
Apache Commons Codec	928,140	24	2,998	303 / 77	99	Base64
Apache Commons Compress	768,548	62	7,365	121 / 26	114	TarUtils
Apache Commons Math	1,244,107	537	61,921	3,541 / 197	835	Fraction
Apache Commons Math	1,416,643	588	69,520	4,318 / 26	938	Line
Apache Commons Math	1,416,643	588	69,520	4,318 / 12	939	Covariance
Joda Time	1,070	188	23,964	2,921/ 169	-	BasicDayOfYearDateTimeField

reports, and selected those reports where the fix represents a change in only one statement (single-fault programs). We use the fixed version P' of a faulty version P to evaluate whether or not ENTBUG pinpoints the exact location of the bug in the report, i.e., we check if ENTBUG effectively isolates the faulty candidate on the top of the ranking. Note that the same subjects have also been used in previous studies (e.g., [33]), but we use different faults to demonstrate that ENTBUG works regardless of whether the fault causes an undeclared exception or a wrong output.

Table II provides details about our experimental subjects. For each subject we only used those test cases that executed the class containing the fault, therefore Table II shows both the total number of test cases in the original test suite[1] and the number of test cases actually used. It has also the bug identifier of each bug report used.

C. Summary of Results

Empirical results indicate that ENTBUG improves the ability of a test suite to diagnose a problem. In fact, ENTBUG was able to pinpoint the location of every bug we considered in our study.

Figure 2 shows the evolution of $\bar{\rho}$, $C_{d_{EntBug}}$ and $C_{d_{random}}$ over time during test generation. The y-axis at the left-hand side denotes the domain of $\bar{\rho}$ while the y-axis at the right-hand side denotes the domain of C_d. The x-axis represents the number of test cases from the original test suite plus the test cases generated on each time interval. Recall that the ideal scenario for diagnosis is one where $\bar{\rho}$ approximates to 0.5 and C_d approximates to 0. Figure 3a shows the increase in number of tests for each of the subjects over time, and Figure 3b demonstrates how ENTBUG is able to reduce entropy during evolution.

In the following we discuss each subject in detail, show how ENTBUG helps to locate each corresponding fault, and compare this result with random test generation.

D. Vending Machine

Vending Machine is a small proof-of-concept example comprised of two classes and one test case only. Its purpose is to check whether there is enough credit that allows a user to buy a product. If needed, the user may chip more money in.

[1]The original test suite of each subject does not contain the test cases which were submitted when a patch was committed.

Listing 1: Vending Machine fix.

```
// Class : vendingmachine.VendingMachine
@@ -45,7 +45,7 @@
  public void vend() throws Exception { ...
    this.currValue -= COST;
-   if (this.currValue == 0) {
+   if ((this.currValue - COST) <= 0) {
      this.enabled = false;
    } ...
```

Vending Machine fails when the user credit reaches a negative value and the branch was not taken: the `enabled` variable stays true, and the user has permission to buy more products (even with a negative credit).

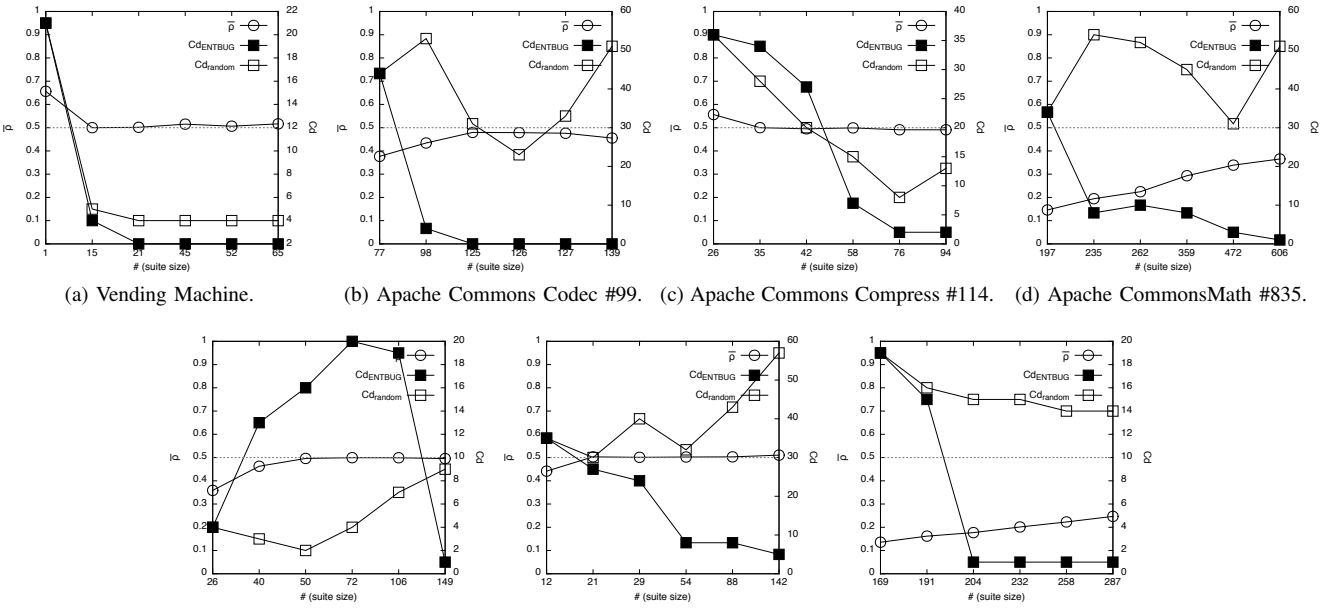
Using the original test suite, and despite the fact that an observed failure is documented, the spectrum-based reasoning approach to fault localization leads to 39% of the code to be inspected to find the fault. This has to do with the fact that the original test suite has a high value $\bar{\rho}$ as it only contains one passing test. Consequently, the cost C_d to diagnose this program is relatively high, as Figure 2a indicates.

Augmenting the test suite with new test cases during 1 minute reduces the number of statements to inspect to find the fault (4 on average), as well as reduces the value of $\bar{\rho}$ from 0.656 to 0.500. During this time interval, ENTBUG increased the size of the original test suite from 1 to 15. Figure 3a shows the increase in the number of test cases. At the same time the information gain (IG) obtained with the new test cases increases and the entropy decreases from 5.000 to 3.381.

After 2 minutes and 20 test cases generated, ENTBUG reduces the statements that need to be inspected to just 2 (on average). After 5 minutes ENTBUG obtains a test suite that produces a $\bar{\rho}$ value of 0.515 (0.015 away from the optimal value) and an IG value of 0.999 (very close to the ideal value of 1.0). In turn, the entropy of the ranking is 2.398 representing a reduction of 52.04% compared to that of the original test suite, which has an entropy of 5.000. See Figure 3b.

ENTBUG pinpoints as faulty the statements at lines 44 and 45. As one can see from Listing 1, the fault localization technique is accurate when singling the statement 45 out as faulty. The statement 44 also appears in the top of the ranking because it is always executed with statement 45; the underlying technique cannot distinguish between components that have the same execution pattern.

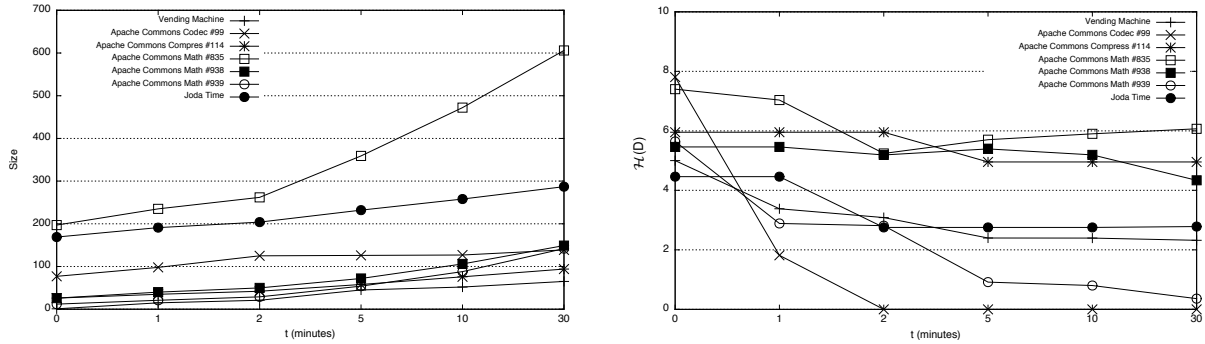
Comparing the results with random test generation, ENTBUG was twice as precise in terms of the number of statements that need to be explored until finding the faulty one.



(a) Vending Machine. (b) Apache Commons Codec #99. (c) Apache Commons Compress #114. (d) Apache CommonsMath #835.

(e) Apache Commons Math #938. (f) Apache Commons Math #939. (g) Joda Time.

Figure 2: Evolution of $\bar{\rho}$ and C_d for every subject.



(a) Test suite size evolution. (b) Entropy value evolution.

Figure 3: Evolution of the test suites size and $\mathcal{H}$ for every subject.

E. Apache Commons Codec #99

Apache Commons Codec[2] provides an API of common encoders and decoders such as Base64, Hex and URLs.

As described in the *major bug* 99[3], the method `encodeBase64String` of the class `Base64` fails because it chunks the parameter `binaryData`. This means that the second parameter of the method `newStringUtf8` called on method `encodeBase64String` should be `false` and not `true`.

Listing 2: Apache Commons Codec fix for bug 99.

```
// org.apache.commons.codec.binary.Base64
@@ -667,7 +667,7 @@
  public static String encodeBase64String(byte[] binaryData) {
-   return StringUtils.newStringUtf8(encodeBase64(binaryData,
      true));
+   return StringUtils.newStringUtf8(encodeBase64(binaryData,
      false));
  } ...
```

[2]Apache Commons Codec project homepage http://commons.apache.org/proper/commons-codec/, 2013.

[3]Apache Commons Codec Bug 99 https://issues.apache.org/jira/browse/CODEC-99, 2013

The original test suite requires a considerable effort to find the fault (44 statements on average), when using spectrum-based reasoning. After generating 21 new test cases (1 minute) we are able to reduce C_d to 4 statements. As can be seen in Figure 2b, this represents a reduction of 90.91% on the diagnostic effort. Applying ENTBUG for an extra minute results in more 27 test cases (see Figure 3a), and we are able to pinpoint exactly the faulty line described in Listing 2. This corresponds (as expected) to a value of $\bar{\rho} = 0.480$, very close to the optimal value, a total information of 0.999, and that the faulty statement appears in top of the ranking as the most likely faulty component in the program.

Note that after 10 minutes there is a slight decrease in the value of $\bar{\rho}$. Recall that ENTBUG's test generation uses an error

margin ϵ to calculate fitness improvement (see Algorithm 1). As such it can happen that a test case can be admitted for inclusion in the test suite even without improving fitness.

Although random test cases achieve a minimum that represents a reduction of 47.13% (when comparing to the original test suite), the diagnostic effort is 23 times as high as on the result achieved by ENTBUG (see Figure 2b).

F. Apache Commons Compress #114

The Apache Commons Compress[4] library defines an API for working with the most popular compressed archives such as ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200 and bzip2.

Listing 3: Apache Commons Compress fix for Bug 114.

```
// org.apache.commons.compress.archivers.tar.TarUtils
@@ -95,11 +95,11 @@
  for (int i = offset; i < end; ++i) { ...
-   result.append((char) buffer[i]);
+   result.append((char) (b & 0xFF));//Allow for sign extension
  } ...
```

The reported *major bug* 114[5] explains that the project Apache Commons Compress fails when the class `TarUtils` receive as input a *tarfile* which contains files with special characters. A simple fix to resolve the encoding problem is to guarantee that the name of the files are treated as unsigned.

The original test suite of Apache Commons Compress project has an IG close to the perfect value, 0.991 (this means a $\bar{\rho}$ value of 0.556). However, the cost to diagnose this project in the beginning is 36 statements on average (as seen in Figure 2c). This is explained by the diversity of the test suite: although being able to entail a $\bar{\rho}$ near to the optimal value, the tests are not diverse.

After the first minute of test generation the size of the original test suite increases from 26 test cases to 35 test cases (see Figure 3a), and the value of C_d is reduced to only 34 statements on average. Compared to the initial effort to diagnose the program, this represents a reduction of only two statement on average. If we continue generating new test cases for 1 more minute (7 new test cases) the effort to find the faulty statement is 27 statements on average. In this period, as we can see in Figure 2c, random generation has a slightly better result. At the interval of 5 minutes, we have an effort of only 7 statement and at 10 minutes we pinpoint with precision of $C_d = 2$ (this means a reduction of 94.37%) the faulty statements described in Listing 3, with a best value of $\bar{\rho} = 0.491$. At the same time, random generation only achieved a reduction in terms of C_d around 79%.

G. Apache Commons Math #835

The Apache Commons Math[6] is a library that provides self-contained mathematics and statistics functions for Java.

[4]Apache Commons Compress project homepage http://commons.apache.org/proper/commons-compress/, 2013.

[5]Apache Commons Compress Bug 114 https://issues.apache.org/jira/browse/COMPRESS-114, 2013.

[6]Apache Commons Math project homepage http://commons.apache.org/proper/commons-math/, 2013.

Listing 4: Apache Commons Math fix for Bug 835.

```
// org.apache.commons.math3.fraction.Fraction
@@ -594,7 +594,7 @@
  public double percentageValue() {
-   return multiply(100).doubleValue();
+   return 100 * doubleValue();
  } ...
```

The Bug 835[7] reports a failure when the `percentageValue()` method of the `Fraction` class multiplies a fraction value by 100, and then converts the result to a double. This causes an overflow when the numerator is greater than `Integer.MAX_VALUE/100`, and even when the value of a fraction is far below this value. A change in the order of multiplication, i.e., first convert a fraction value to a double and then multiply that value by 100, resolved the overflow problem.

The cost to diagnose the Apache Commons Math project with the original test suite for bug 835 is on average 34 statements (as we can see in Figure 2d).

For the bug 835, which starts with a coverage density of 0.146, ENTBUG takes more time to achieve the perfect $\bar{\rho}$ value. After 10 minutes of random generation, the number of candidates to inspect was reduced from 34 to 31, whereas ENTBUG reduced the diagnostic effort in the same time by 92.24% (which represents only 3 remaining components) on average. But, only after 30 minutes of generation, we achieve a C_d value of 1 (almost perfect) and a $\bar{\rho}$ value of 0.365 for bug 835 (Listing 4). In this period of time, the test suite increases its size (see Figure 3a) from 197 (number of original test cases that touch the fault class) to 606 test cases. We conjecture that the slow performance is because EVOSUITE produces large numbers, which are important for this scenario, with only a low probability.

For this particular subject, as we can see in Figure 3b, the value of entropy decreases in the first two minutes of generation (29.18%). However, at 5 minutes (where we are able to reduce the cost to diagnose in 77.61%) the entropy that characterizes the ranking increases from 5.242 to 5.702. The reason why this happened is that at the first two minutes almost every (71 out of 79) candidate in the ranking has the same probability (0.011); at 5 minutes the probabilities still are the same but higher (0.034). For these two intervals the number of candidates in the ranking is equal, 79 in total.

H. Apache Commons Math #938

The *major bug* 938[8] explains that the method `revert` from the class `Line` only maintains 10 digits of precision for the field direction. This becomes a bug when the line's position is evaluated far from the origin. A possible fix is creating a new instance of `Line` and then reverting its direction.

Using the original test suite, the cost incurred to diagnose the Apache Commons Math project for bug 938 is 4 statements.

[7]Apache Commons Math Bug 835 https://issues.apache.org/jira/browse/MATH-835, 2013.

[8]Apache Commons Math Bug 938 project homepage https://issues.apache.org/jira/browse/MATH-938, 2013.

Listing 5: Apache Commons Math fix for Bug 938.

```
// org.apache.commons.math3.geometry.euclidean.threed.Line
@@ -84,7 +84,9 @@
  public Line revert() {
-   return new Line(zero, zero.subtract(direction));
+   final Line reverted = new Line(this);
+   reverted.direction = reverted.direction.negate();
+   return reverted;
  } ...
```

Figure 2e shows that the cost increase as more tests are added, rather than decreasing, even though $\bar{\rho}$ improves. This is because the generated test cases were all passing (and therefore, there is no evidence for exonerating/blaming components). After 2 minutes, random generation achieves a close to perfect C_d – taking advantage of the fact that random generated test cases failed, too. However, after generating test cases for 30 minutes, and with sufficient pass/fail test cases in the suite, C_d reaches the minimum value of of 1 statement to inspect on average and $\bar{\rho} = 0.496$. This is a 50% improvement over the cost when using the random approach.

I. Apache Commons Math #939

Listing 6: Apache Commons Math fix for Bug 939.

```
// org.apache.commons.math3.stat.correlation.Covariance
@@ -279,15 +279,15 @@
  private void checkSufficientData(final RealMatrix matrix)
      throws MathIllegalArgumentException {
    int nRows = matrix.getRowDimension();
    int nCols = matrix.getColumnDimension();
-   if (nRows < 2 || nCols < 2) {
+   if (nRows < 2 || nCols < 1) {
      throw new MathIllegalArgumentException(
        LocalizedFormats.INSUFFICIENT_ROWS_AND_COLUMNS,
          nRows, nCols); ...
```

The specification of the class `Covariance` states that it only takes a single-column matrix (i.e., N-dimensional random variable with $N = 1$) as argument and returns a $1 - by - 1$ covariance matrix. However, the method `checkSufficientData` throws an `IllegalArgumentException` (see *major bug* 939[9] for detailed information) when the constructor of the class receives a $1 - by - M$ matrix.

The cost to diagnose the Apache Commons Math project for the bug 939 is on average 35 statements with the original test suite (as we can see in Figure 2f).

To pinpoint the faulty location for bug 939 (described in Listing 6) EntBug needs 30 minutes. This represents a $\bar{\rho}$ value of 0.510, a gain of 93.60% in terms of ranking entropies, and 130 new test cases. On the other hand, as with the previous subjects, random test generation performed 5.8 times worse when comparing to the C_d value returned by EntBug. For this subject, EntBug may not achieve the perfect value of $C_d = 0$ because of the structure of the class `Covariance`. The `checkSufficientData` function (more properly the line 279) responsible for the bug 939, is a private function. This function is only executed by the constructor of the class `Covariance`. Therefore, those components that appears at the top of the ranking are the statements of the `Covariance` constructor and the `checkSufficientData` function.

[9]Apache Commons Math Bug 939 https://issues.apache.org/jira/browse/MATH-939, 2013.

J. Joda Time

Joda Time[10] is a library for advanced *date* and *time* functionalities for the Java language.

Listing 7: Joda Time fix.

```
// org.joda.time.chrono.BasicDayOfYearDateTimeField
@@ -90,7 +90,7 @@
  protected int getMaximumValueForSet(long instant, int value)
      {
    int maxLessOne = iChronology.getDaysInYearMax() - 1;
-   return value > maxLessOne ? getMaximumValue(instant) :
-     maxLessOne;
+   return (value > maxLessOne || value < 1) ?
+     getMaximumValue(instant) : maxLessOne;
  } ...
```

The class `BasicDayOfYearDateTimeField` provides methods to perform time calculations for a day of a year. Joda Time bug[11] was related to the method `getMaximumValueForSet`, which returns an incorrect value. The fix of this bug consists of validating if the value of the variable `value` is between the maximum and the minimum value of the range or not.

Due to the nature of the Joda Time project (e.g., the source code is structured in large hierarchies of classes consisting of many one-line methods) the original test suite has a coverage density of only 0.136. With the original test suite, the cost to diagnose the fault described in Listing 7 is 19 statements on average. However, already after two minutes and 35 new test cases (see Figure 2g for more details), the value of C_d drops to only one statement. At the same time, the ranking entropy was reduced by 38.27%. After half an hour of generating new test cases, the value of $\bar{\rho}$ increases to 0.247. For this subject, and considering the 30 minutes of experiments, random generation achieved values of C_d between 14 to 16 statements on average. So, considering the total of time (30 minutes), this means, that EntBug performed 4 times better than random generation.

K. Threats to Validity

Construct Validity: The performance of EntBug has been evaluated using the C_d metric, which measures diagnostic effort in terms of the position of the fault in the diagnostic report. This metric assumes that developers traverse the ranking, but that may not be the case in practice [30]. However, we argue that developers are more likely to traverse the ranking if the precision is increased.

External Validity: Despite being real, large and open source software systems, we have only considered five subjects with seven (single-) faults in our empirical experiment. Therefore, it is possible that the results for a different set of programs with different characteristics and even with *multiple-faults* may produce different results.

[10]Joda Time project homepage http://joda-time.sourceforge.net/, 2013.

[11]Joda Time bug corrected at https://joda-time.svn.sourceforge.net/svnroot/joda-time/trunk@1071, 2013.

Internal Validity: Eventual faults in the implementation of EntBug or in the underlying test case generation EvoSuite may invalidate the results. To mitigate this threat, we have not only thoroughly tested our scripts but also manually checked a large set of results. Furthermore, all experiments were repeated multiple times to account for the randomness of the test generation, and we verified the results between runs for consistency.

V. Related Work

Although there is a large body of work on automated test generation and debugging in general, there have only been few attempts at using test generation in the context of debugging.

Baudry *et. al* [10] proposed an approach to improve diagnostic accuracy using a bacteriological algorithm (similar to a GA) to select test cases from a test suite. The criterion for test selection they proposed estimates the quality of a test for diagnosis. Their selection procedure attempts to find an optimal balance between the size of a test suite and its contribution to diagnosis. The goal of their work is similar to ours, but our contributions are complementary: One could use EntBug for test generation and the algorithm they proposed for test selection. It remains to be evaluated if such a combination would improve the diagnostic report's accuracy.

Artzi *et al.* [8] use a specialized concrete-symbolic execution [20], [36] to improve fault localization. The principle of their customized algorithm is highly similar to that of the Nearest Neighbors Queries algorithm [32] proposed by Renieris and Reiss. However, instead of selecting tests that are similar to a given failing test, Artzi *et al.* generate such tests. One important difference between our work and theirs is in the assumptions: While we make no assumption about the previous test suite, they assume there is at least one fault-revealing test to seed the search. However, in practice it is possible that no fault-revealing test exists in the test suite. An important technical difference between our approaches is that their algorithm uses as input a single fault-revealing test and generates passing tests that minimizes observed differences for that particular test. However, it has been shown that multiple fault-revealing tests can help improve diagnostic accuracy [3].

Rößler *et. al* [33] introduced a search-based approach to identify fault candidates. Similar to the work of Artzi *et al.* [8], their Bugex tool takes a failing test case as a starting point, determines a set of "facts" (e.g., executed statements, branches, program states, etc.) and then systematically tries to generate variations of the failing test which differ in individual facts. If a passing test differs in only one fact to the failing test, then that fact is assumed to be relevant for diagnosis; if the differing test also fails, then the fact is irrelevant. Bugex is also implemented using EvoSuite, but our approach differs in several aspects: First, we do not assume the existence of a single failing test — we can optimize a test suite also in the presence of no faults or in the presence of multiple faults. Second, Bugex uses a white-box testing approach to minimize facts about structural aspects of a program. In contrast, our approach is only guided by entropy, which means it is applicable to any testing domain.

To the best of our knowledge, the use of entropy to guide test generation is novel. However, entropy has been used to prioritize test cases to improve fault localization [24], [40].

VI. Conclusions and Future Work

This paper presents an approach to generate test cases automatically based on their ability to produce a refined diagnosis, rather than finding a failure as early as possible. The results observed in our empirical evaluation using our EntBug prototype show that the proposed approach can localize the faults with superior accuracy in all seven considered scenarios. Compared to the original test suites, we observed an average reduction of 49% on the uncertainty of the diagnostic ranking, while also reducing the number of candidates that one needs to inspect before finding the fault by 91% on average. Comparison to random tests shows that this improvement does not simply come from the increased test suite size, but that entropy provides guidance to the test generator.

These are encouraging results, yet whether these improvements are sufficient to convince programmers to use spectrum-based fault localization needs to be empirically studied. We therefore plan to perform more experimentation and user studies to assess the usefulness of the results for developers.

Besides studying its practical value, the presented approach offers several avenues for future research. First, our evaluation so far only considered single faults, and we plan to evaluate the performance of our approach in presence of multiple faults. Second, our approach is currently only driven by the observed entropy, and does not take information about the source code into account. Potentially, the efficiency could be increased further by considering structural aspects during test generation, for example by explicitly considering the coverage of existing test cases in the fitness function (e.g., [33]). Third, any approach aiming to improve diagnostic accuracy by generating new tests is limited by the oracle problem; future work will therefore have to consider how to apply the presented approach in practice, and how to best interact with the developer. Finally, our current usage scenario is that of an existing test suite that needs to be improved. We plan to apply and evaluate the ideas presented in this paper also for generating test suites with high diagnostic accuracy in the first place.

More information on EntBug is available at

`http://www.evosuite.org/EntBug`

Acknowledgments

This work is partially funded by the ERDF through the Programme COMPETE, the Portuguese Government through FCT - Foundation for Science and Technology, project reference FCOMP-01-0124-FEDER-020484, and by a Google Focused Research Award on "Test Amplification".

REFERENCES

[1] R. Abreu and A. J. C. van Gemund, "A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis," in *Symposium on Abstraction, Reformulation and Approximation*, ser. SARA '09, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-Based Multiple Fault Localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99.

[3] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental Regression Testing," in *Proceedings of the Conference on Software Maintenance*, ser. ICSM '93. IEEE Comp. Soc., 1993, pp. 348–357.

[5] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the Accuracy of Fault Localization Techniques," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. IEEE Computer Society, 2009, pp. 76–87.

[6] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 520–523.

[7] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey on automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, August 2013.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed Test Generation for Effective Fault Localization," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 49–60.

[9] M. Avriel, *Nonlinear Programming: Analysis and Methods*, ser. Dover Books on Computer Science Series. Dover Publications, 2003.

[10] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving Test Suites for Efficient Fault Localization," in *Proceedings of the 28th International Conference on Software engineering*, ser. ICSE. ACM, 2006, pp. 82–91.

[11] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. NY, USA: ACM, 2011, pp. 221–231.

[12] J. Carey, N. Gross, M. Stepanek, and O. Port, "Software hell," in *Business Week*, 1999, pp. 391–411.

[13] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artif. Intell.*, vol. 32, no. 1, pp. 97–130, Apr. 1987.

[14] J. De Kleer, "Diagnosing multiple persistent and intermittent faults," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, ser. IJCAI '09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 733–738.

[15] A. Feldman, G. Provan, and A. Van Gemund, "Computing minimal diagnoses by greedy stochastic search," in *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, ser. AAAI'08. AAAI Press, 2008, pp. 911–918.

[16] A. Feldman and A. van Gemund, "A two-step hierarchical algorithm for model-based diagnosis," in *Proceedings of the 21st national conference on Artificial intelligence*, ser. AAAI. AAAI Press, 2006, pp. 827–833.

[17] 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*, ser. ESEC/FSE '11. ACM, 2011, pp. 416–419.

[18] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA. ACM, 2010, pp. 147–158.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[20] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[21] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 83–92.

[22] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. J. van Gemund, "Spectrum-Based Sequential Diagnosis," in *Proceedings of the 25th AAAI International Conference on Artificial Intelligence (AAAI'11)*, Aug 2011, pp. 189–196.

[23] A. Gonzalez-Sanchez, H.-G. Gross, and A. J. C. van Gemund, "Modeling the Diagnostic Efficiency of Regression Test Suites," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 634–643.

[24] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. C. van Gemund, "Prioritizing Tests for Software Fault Localization," in *Proceedings of the 10th International Conference on Quality Software*, ser. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 42–51.

[25] R. A. Johnson, "An information theory approach to diagnosis," *Reliability and Quality Control, IRE Transactions on*, no. 1, pp. 35–35, 1960.

[26] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[27] W. Mayer and M. Stumptner, "Evaluating Models for Model-Based Debugging," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 128–137.

[28] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[29] E. Miller and W. E. Howden, *Software Testing and Validation Techniques*, ser. 2nd ed. New York, USA: IEEE Comp. Soc. Press, 1981.

[30] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209.

[31] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the Crowd Solve the Oracle Problem?" in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, ser. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013.

[32] M. Renieris and S. P. Reiss, "Fault Localization With Nearest Neighbor Queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ser. ASE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 30–39.

[33] J. Rößler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 309–319.

[34] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, "Reconstructing Core Dumps," in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, ser. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013.

[35] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 56–66.

[36] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[37] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software Fault Localization Using DStar (D*)," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, ser. SERE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 21–30.

[38] F. Wotawa, "Bridging the Gap Between Slicing and Model-based Diagnosis," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering*, ser. SEKE '08. Knowledge Systems Institute Graduate School, 2008, pp. 836–841.

[39] F. Wotawa, M. Stumptner, and W. Mayer, "Model-Based Debugging or How to Diagnose Programs Automatically," in *Proceedings of the 15th international conference on Industrial and engineering applications of artificial intelligence and expert systems*, ser. IEA/AIE '02. London, UK, UK: Springer-Verlag, 2002, pp. 746–757.

[40] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, p. 19, July 2013.