Prioritizing Tests for Software Fault Localization

Alberto Gonzalez-Sanchez Eric Piel Hans-Gerhard Gross Arjan J.C. van Gemund

Delft University of Technology, Software Technology Department
Mekelweg 4, 2628 CD Delft, The Netherlands
Email: {a.gonzalezsanchez,e.a.b.piel,h.g.gross,a.j.c.vangemund}@tudelft.nl

***Abstract*—Test prioritization techniques select test cases that maximize the confidence on the correctness of the system when the resources for quality assurance (QA) are limited. In the event of a test failing, the fault at the root of the failure has to be localized, adding an extra debugging cost that has to be taken into account as well. However, test suites that are prioritized for failure detection can reduce the amount of useful information for fault localization. This deteriorates the quality of the diagnosis provided, making the subsequent debugging phase more expensive, and defeating the purpose of the test cost minimization.**

In this paper we introduce a new test case prioritization approach that maximizes the improvement of the diagnostic information per test. Our approach minimizes the loss of diagnostic quality in the prioritized test suite. When considering QA cost as the combination of testing cost and debugging cost, on the Siemens set, the results of our test case prioritization approach show up to a 53% reduction of the overall QA cost, compared with the next best technique .

I. Introduction

Critical and high-availability systems, such as air traffic control systems, systems of the emergency units, and banking applications, are becoming more and more complex and dynamic. The number and complexity of the components that form the systems is growing. Moreover, in the case of Systems of Systems, or Service Oriented Architectures components may not be available until deployment time, e.g., third party external services. Components can be even unknown at deployment time.

The *quality assessment* (QA) phase of these kind of systems was traditionally performed either in a separate, identical copy of the system, or by taking the system offline. Lately, *run-time testing* is emerging as the solution for the validation and acceptance testing of the above systems. Run-time testing is a testing method that has to be conducted and performed in-vivo in the final execution environment of a system [4], [11], [16].

The amount of resources available during the QA phase of the software life-cycle is limited. In run-time testing this is further exacerbated by the fact that tests will interfere with the operations of the systems [4]. Consequently, the cost of the QA phase needs to be minimized, while maximizing the confidence in the integrated system.

Many approaches have been aimed at minimizing testing cost by prioritizing tests with the objective of *failure detection*, i.e., of detecting the presence of faults as early in the testing process as possible [3], [14], [15]. What these approaches usually do not consider is the fact that once the presence of a fault has been detected (test phase), developers have to find the actual location of the fault (debugging phase) with the information produced by the tests.

The debugging phase can make use of automatic *fault localization* techniques which help to significantly reduce the debugging effort needed, as shown in [1], [10], [18]. However, the quality of the result of fault localization techniques depends on the information provided by the testing phase.

The information provided by tests can be improved by selectively adding more test cases [2]. However, the usual practice is to reduce the number of tests to save testing time, not to increase it. Previous work has shown how test suites that are reduced or prioritized for failure detection can decrease the amount of useful information for the fault localization [8], [17]. This will deteriorate the quality of the diagnosis provided by the fault localization algorithm, leading to a longer subsequent debugging phase, partially defeating the purpose of the test cost minimization.

This poses the question of whether there exists a prioritization technique putting emphasis on *fault localization* performance rather than failure detection performance. The goal should be to reduce the overall QA cost (testing and debugging) and not trade testing for debugging effort.

This paper presents such a technique and make the following contributions:

1) We present an analysis of *why* failure detection prioritization deteriorates the performance of fault localization algorithms, which motivates our alternative approach.
2) We introduce a prioritization strategy for fault localization, contrasting with existing approaches whose goal is failure detection. Our approach performs *on-line* prioritization depending on the outcome of the tests based on diagnostic *information gain*.
3) We evaluate our technique on the Siemens programs in a semi-synthetic setting, comparing it to existing

prioritization techniques in terms of both fault localization and failure detection performance. Our results show up to a 53% reduction of the overall QA cost, when compared to the next best performing technique in the Siemens set.

The paper is organized as follows. In Section II, we describe the main concepts of fault diagnosis and the diagnosis algorithm used in our experiments. Section III surveys the existing prioritization techniques with which we will compare our approach. In Section IV, we describe why current prioritization techniques fall short for fault localization. Section V introduces *diagnostic prioritization* and the information gain heuristic. Our evaluation goals and experimental setup are described in Section VI, while the results are presented and discussed in Section VII. Related work is surveyed in Section VIII. Section IX presents our final conclusions and future work directions.

II. Fault Diagnosis

The objective of fault diagnosis is to pinpoint the precise location of a fault in a program (a bug) by executing tests and observing the program's behavior. Diagnosis can be achieved by statistical or probabilistic approaches, for example Spectrum-based Fault Localization (SFL) [1], [10], which are lightweight and based on coverage information. Therefore, we will use SFL as our diagnosis technique.

A. Diagnostic Process

For compatibility with the test selection algorithms in the following sections, we will define the diagnostic process as the process of obtaining a set of diagnostic explanations $D = \{d_1, \ldots, d_k\}$ from binary test outcomes and the components involved in the tests. Each explanation d_k is a subset of the components in the system, which, at fault, explain the observed failures. As most previous work [8], [10], [17], for the scope of this paper, we will assume that only one fault is present.

The following inputs are involved in diagnosis:

- A finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_j, \ldots, c_M\}$ of components (typically source code statements) which are potentially faulty.
- A corresponding set of prior fault probabilities p_j for each component. These priors represent the knowledge available before any test is executed.
- A finite set $\mathcal{T} = \{t_1, t_2, \ldots, t_i, \ldots, t_N\}$ of tests with binary outcomes $O = (o_1, o_2, \ldots, o_i, \ldots, o_N)$, where $o_i = 1$ if test t_i failed, and $o_i = 0$ otherwise.
- A $N \times M$ coverage matrix, $A = [a_{ij}]$, where $a_{ij} = 1$ if test t_i involves component c_j, and 0 otherwise.

Due to the limited number of tests, the number of possible diagnostic explanations is typically very high. Therefore, it is necessary to *rank* diagnostic explanations by the likelihood of that diagnostic explanation being the correct one, for example by using statistical similarity coefficients, or by using a Bayesian approach as we will explain in the next subsection.

	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	Prior
c_0		0	0	0	0	0	0	0	0	
c_1	`main() {`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_2	`int let, dig, other, c;`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_3	`let = dig = other = 0;`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_4	`while(c = getchar()) {`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_5	`if ('A'<=c && 'Z'>=c)`	1	1	1	1	1	1	1	0	$^1/_{13}$
c_6	**`let += 2; /* FAULT */`**	1	0	1	1	0	0	1	0	$^1/_{13}$
c_7	`elif ('a'<=c && 'z'>=c)`	1	1	0	1	1	1	1	0	$^1/_{13}$
c_8	`let += 1;`	1	0	0	0	1	0	1	0	$^1/_{13}$
c_9	`elif ('0'<=c && '9'>=c)`	1	1	0	1	1	1	0	0	$^1/_{13}$
c_{10}	`dig += 1;`	1	1	0	1	0	0	0	0	$^1/_{13}$
c_{11}	`elif (isprint(c))`	0	0	0	0	1	1	0	0	$^1/_{13}$
c_{12}	`other += 1;}`	0	0	0	0	1	0	0	0	$^1/_{13}$
c_{13}	`printf("%d %d %d\n", let, dig, others);}`	1	1	1	1	1	1	1	1	$^1/_{13}$
	Test case outcomes	1	0	1	1	0	0	1	0	

Table I
Faulty program and Fault Diagnosis inputs

B. Diagnostic Ranking by Bayesian Reasoning

In the case of Bayesian approaches, the likelihood of an explanation corresponds to the posterior probability of that diagnostic being correct, given the outcomes of the executed tests, $\Pr(d_k|o_i, o_{i-1}, \ldots)$, for a particular diagnosis d_k. As there can only be one correct explanation, all the individual probabilities add up to 1.

For each test case, the probability of each diagnostic explanation $d_k \in D$ is updated depending on the outcome o_i of the test, following Bayes' rule:

$$\Pr(d_k|o_i, o_{i-1}, \ldots) = \frac{\Pr(o_i|d_k) \cdot \Pr(d_k|o_{i-1}, \ldots)}{\Pr(o_i)} \quad (1)$$

In this equation, $\Pr(o_i|d_k)$ represents the probability of the observed outcome, if that diagnostic explanation d_k is the correct one. It is related to the intermittency of the fault, i.e., whether the component always causes a failure when used in a test, or only in some cases. Although for software it is quite common to have intermittent faults, for the purpose of this paper (which focuses on prioritization) for simplicity we will assume that a faulty statement in a program will always generate a failure if covered, thus $\Pr(o_i = 1|d_k) = 1 - \Pr(o_i = 0|d_k) = a_{ik}$.

$\Pr(o_i)$ represents the probability of the observed outcome, independently of which diagnostic explanation is the correct one. The value of $\Pr(o_i)$ is a normalizing factor that is given by

$$\Pr(o_i) = \sum_{d_k \in D} \Pr(o_i|d_k) \cdot \Pr(d_k|o_{i-1}, \ldots) \quad (2)$$

C. Diagnostic Example

Table I shows an example faulty program [7], eight tests, and their statement coverage (the matrix A is transposed for the sake of readability).

As we assume a single fault is present, each explanation in D corresponds to one code statement: $\forall d_k \in D, d_k = \{c_k\}$.

Consequently, the initial probability of each diagnostic candidate corresponds to the prior probability of each component: $\forall d_k \in D, \Pr(d_k|i=0) = p_k = \frac{1}{13}$.

After applying test t_1, we observe a failure. The probabilities of all the covered statements c_j (including c_6) are updated by

$$\Pr(d_j|o_1) = \frac{a_{1,j} \cdot \Pr(d_j|i=0)}{\Pr(o_1)} = \frac{1 \cdot \frac{1}{13}}{\frac{11}{13}} = \frac{1}{11}$$

The statements which were not covered are updated by

$$\Pr(d_j|o_1) = \frac{a_{1,j} \cdot \Pr(d_j|i=0)}{\Pr(o_1)} = \frac{0 \cdot \frac{1}{13}}{\frac{11}{13}} = 0$$

Their zero value follows from the fact that, if they were not involved in the test, and the test failed, it is impossible that these statements are faulty.

After applying test t_2, no failure occurs. The probabilities of the covered statements which are not already 0 are then updated by

$$\Pr(d_j|o_2, o_1) = \frac{(1 - a_{2,j}) \cdot \Pr(d_j|o_1)}{\Pr(o_2)} = \frac{0 \cdot \frac{1}{11}}{\frac{2}{11}} = 0$$

and the untouched statements by

$$\Pr(d_j|o_2, o_1) = \frac{(1 - a_{2,j}) \cdot \Pr(d_j|o_1)}{\Pr(o_2)} = \frac{1 \cdot \frac{1}{11}}{\frac{2}{11}} = \frac{1}{2}$$

The last test applied is t_3, which fails. The only covered component with non-zero probability is c_6, and it is updated by

$$\Pr(d_6|o_3, o_2, o_1) = \frac{a_{3,6} \cdot \Pr(d_6|o_2, o_1)}{\Pr(o_3)} = \frac{1 \cdot \frac{1}{2}}{\frac{1}{2}} = 1$$

and the probability of c_8 is therefore 0. The remaining tests have no influence on the diagnosis.

D. Residual Diagnostic Cost

A diagnostic process is divided in two phases, testing-based diagnosis (outlined above) and *residual diagnosis*. During testing, test cases are applied to collect observations in order to refine the initial diagnosis D_0. During the residual diagnosis phase, the final diagnosis after N observations D_N is returned to the user as the basis to find the real fault. Typically the user finds the fault by inspecting each candidate in descending order according to the updated diagnostic probabilities.

The residual diagnosis cost, W, is the manual *work* that has to be performed by the developer, who has to inspect (debug) each of the d_k explanations in D_N top down, until he or she finds the real fault d_*.

In the following, we define W as the fraction of components the developer has to examine until finding the real fault d_* [1], according to

$$W(d_*) = \frac{\tau}{M-1} \cdot 100\% \tag{3}$$

where τ is the position of d_* in the ranking. Because multiple explanations can be assigned the same probability, the value of τ is averaged between the ranks of explanations that share the same probability, amongst which the real fault d_* is located.

$$\tau = \frac{|j : \Pr(d_j|o_i) > \Pr(d_*|o_i)|}{2} + \frac{|j : \Pr(d_j|o_i) \geq \Pr(d_*|o_i)| - 1}{2} \tag{4}$$

There are two ways of reducing diagnostic cost. One can try to develop better techniques to reduce the residual diagnosis effort W, by reducing the number of candidates, or by improving the ranking so that the real explanation d_* ranks higher.

One can also try to reduce testing cost, by executing only a subset of the tests. Prioritizing $\mathcal{T}$ in such a way that the executed subset of $\mathcal{T}$ yields the highest diagnostic accuracy (minimizing W) is the main focus of this paper.

III. Test Case Prioritization

Test case prioritization techniques order test cases with respect to a given goal, so that those tests with the highest utility (which bring the test process closer to its goal), are given higher priorities and therefore are executed earlier in the testing process.

A *failure* is a deviation of the expected behavior of a program, caused by a fault. The most common prioritization goal is to increase the rate of failure detection. It means, tests are executed in an order such that failures occur as early as possible in the testing process, so that confidence in the presence or absence of faults is reached faster. The following prioritization techniques have been proposed in order to achiever this goal proposed.

Random: this is the most straightforward prioritization criterion, which orders test cases according to random permutations of the original test suite. Random permutations are used as control in many prioritization experiments [3], [14], [15].

Statement coverage: the test cases that will cover the highest total number of statements are executed first, under the assumption that the more statements are covered by a test, the higher is the probability of triggering a failure. If a statement is covered without producing a failure, covering it again is meaningless as it will not produce a failure either [3], [14]. This reasoning conduces to the definition of the *additional coverage* heuristic, where test cases are selected iteratively in terms of the additional coverage they yield, taking into account all the test cases that were already executed, i.e.,

$$\mathcal{H}_{add-st}(t_i) = \sum_{j=1}^{M} a_{ij} \cdot (1 - cov_j) \tag{5}$$

where $cov_j = 1$ if statement j has been covered so far.

Adaptive Random Testing: ART is a hybrid random-coverage-based test ordering [7]. It selects test cases in two steps, first it selects a group of tests randomly, and from that group it selects the test which maximizes a distance function with the already selected test cases. This distance function can be either the minimum distance with all executed tests, the maximum distance, or the average distance. In this paper we will compare with the minimum distance heuristic, as it was cited [7] as the most promising one. It is defined as

$$\mathcal{H}_{art-mxmn}(t_i) = \min_{t_j \in C}(\delta(t_i, t_j)) \quad (6)$$

where C is the set of already applied tests and δ is the distance function used, in [7] the Jaccard distance.

IV. Prioritization and Diagnosis

Previous empirical work has shown that early failure detection and fault localization seem to be rather incompatible goals [8], [17]. The evolution of the diagnostic effort W, per unit of test effort, T, is negatively affected by criteria for early failure detection. Random ordering, which has been traditionally considered the baseline prioritization technique [3], [14], [15], was found to perform as good as or better than all other prioritization techniques, except $\mathcal{H}_{add-st}$. However, even for the latter case the random order was better for some subject programs [8].

The main reason for the poor diagnostic performance of existing prioritization techniques is that they perform *off-line* prioritization, in such a way that tests maximize the probability of failing. This approach may be appropriate for regression testing, but not for fault diagnosis. When performing fault diagnosis, if a test has failed, the components covered by the test become important suspects. However, many regression prioritization algorithms will choose a next test that covers *different* components, whereas from the diagnostic point of view, the next test case should help differentiate between the *current* suspects. Therefore a test order independent of the outcomes of the tests cannot be used. The order has to be adapted *on-line*, depending on the output of the previous tests.

Table II shows an example of this situation when performing additional-statement prioritization. We use the Bayesian diagnosis approach from Section II-B, once more assuming a fault always triggers a failure. The initial probability of each diagnostic candidate p_j is also uniformly distributed. For clarity, the fourth column shows only the probabilities of diagnostic explanations which are non-zero. Initially, no statement has been covered, and D ranks every component with uniform probability.

The additional-statement heuristic selects test t_1 as first test, as it covers the most test cases, and, indeed, t_1 finds the first failure. As a result of the failure, all the c_j covered by t_1 move to the top of the ranking. Unfortunately, the test case covered many statements, so $|D|$ does not decrease too much.

In the second step, test t_5 is selected because it provides the highest additional coverage, and passes. Because it passed, the updated probability of those candidate explanations in D which were covered by t_5 drops to 0 (following the permanent fault assumption of Section II-B). The statements which where not covered remain at the top of D.

Full coverage has been reached, so in the third step, the coverage is reset as described in [14], instead of opting for a random order. Test t_4 provides the highest coverage, and indeed fails. However, it covered both c_6 and c_{10}, so it provides no extra information and D does not change. This happens also in the fourth step for t_6.

Finally, in the fifth step, a test case that covers c_{10} but not c_6 is chosen. As it passes, c_{10}, which was covered, is assigned a probability of 0, and c_6 remains as only (and correct) explanation.

As we can see, $\mathcal{H}_{add-st}$ has the problem that 2 tests provide no information to the diagnosis independent of their outcome, i.e., a complete waste of effort.

As a comparison, Table III shows the optimal test order for a fault in c_6. With just one test case, the set of candidates is drastically reduced. The next test case finalizes the diagnosis by using a test case that bisects D. The order of the remaining tests is irrelevant for the diagnosis, as none will provide more information. The plot in Figure 1 depicts the evolution of both approaches.

Although simple, this example shows that maximizing the probability of a failure does not maximize the information that the diagnostic algorithm receives. In fact, as test cases that cover many statements are those with the highest failure probability, those tests will not provide much useful information because the number of remaining diagnostic candidates will not decrease substantially.

V. Diagnostic Prioritization

In the following we will present *diagnostic prioritization*, an on-line greedy prioritization approach that takes into account the observed test outcomes to determine the next test case. Our work is motivated by research in *sequential diagnosis* of hardware systems, where algorithms exist to diagnose systems with permanent [12] and intermittent [13] faults.

Diagnostic prioritization uses the same inputs as traditional test prioritization and fault localization techniques in software engineering: component set $\mathcal{C}$, prior fault probabilities p_j, tests $\mathcal{T}$ and coverage matrix A. Additionally, a special component c_0 is added to represent the special condition that no other component is faulty (fault-free system). No test can check the fault-free component c_0 directly, therefore $a_{i0} = 0$ for all i.

High-utility tests are those tests which, at each step, maximize the reduction of diagnostic cost on average, considering all possible diagnostic candidates d_k, and *both* possible test

Test	o_i	Covered Statements	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	W
		0 0 0 0 0 0 0 0 0 0 0 0 0	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.500
t_1	1	1 1 1 1 1 1 1 1 1 1 0 0 1	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09			0.09	0.357
t_5	0	1 1 1 1 1 1 1 1 1 1 1 1 1 *						0.50		0.50						0.038
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1 *						0.50		0.50						0.038
t_6	0	1 1 1 1 1 1 1 0 1 1 1 0 1						0.50		0.50						0.038
t_7	1	1 1 1 1 1 1 1 1 0 0 0 0 1 *						1.00								0.000
t_2	0	1 1 1 1 1 1 1 1 1 1 0 0 1 *						1.00								0.000
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1 *						1.00								0.000
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1 *						1.00								0.000

(*) Step after which coverage is reset.

Table II
EVOLUTION OF D FOR THE $\mathcal{H}_{add-st}$ HEURISTIC FOR OUR EXAMPLE SYSTEM.

Test	o_i	Covered Statements	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	W
		0 0 0 0 0 0 0 0 0 0 0 0 0	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.500
t_5	0	1 1 1 1 1 0 1 1 1 0 1 1 1						0.50				0.50				0.038
t_7	1	1 1 1 1 1 1 1 1 1 1 0 1 1 1						1.00								0.000
t_6	0	1 1 1 1 1 1 1 1 1 1 0 1 1 1						1.00								0.000
t_1	1	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000
t_4	1	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000
t_2	0	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000
t_3	1	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000
t_8	0	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000

Table III
OPTIMAL EVOLUTION OF D FOR c_6 IN OUR EXAMPLE SYSTEM.

Test	o_i	Covered Statements	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	W
		0 0 0 0 0 0 0 0 0 0 0 0 0	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.500
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.14	0.14	0.14	0.14	0.14	0.14							0.14	0.214
t_8	0	1 1 1 1 1 1 0 0 0 0 0 0 1					0.50	0.50								0.038
t_2	0	1 1 1 1 1 1 1 0 1 1 0 0 1						1.00								0.000
t_1	1	1 1 1 1 1 1 1 1 1 1 1 0 0 1						1.00								0.000
t_4	1	1 1 1 1 1 1 1 1 1 1 1 0 0 1						1.00								0.000
t_5	0	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000
t_6	0	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000
t_7	1	1 1 1 1 1 1 1 1 1 1 1 1 1 1						1.00								0.000

Table IV
EVOLUTION OF D FOR THE $\mathcal{H}_{IG}$ HEURISTIC FOR OUR EXAMPLE SYSTEM.

outcomes: pass and fail. This reduction in diagnostic cost can be seen as an increase in diagnostic information, i.e., a reduction of the information entropy of the candidate set D.

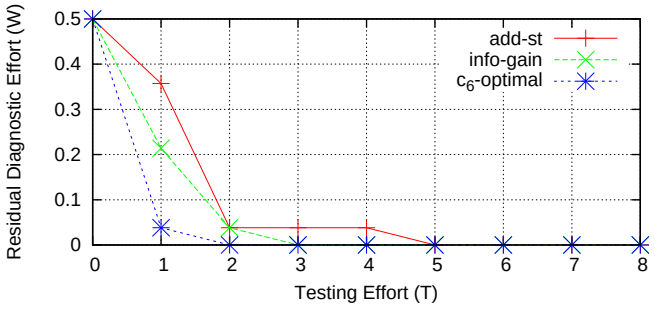

Figure 1. $W(T)$ for three prioritization approaches

Applying this reasoning, at each decision step l in the test sequence, the test yielding the highest average information gain is chosen. The information gain heuristic [9], IG, is defined as

$$\begin{aligned}\mathcal{H}_{IG}(D,t_i) = \mathrm{H}(D) \\ - \Pr(o_i=0)\cdot \mathrm{H}(D|o_i=0) \\ - \Pr(o_i=1)\cdot \mathrm{H}(D|o_i=1)\end{aligned} \quad (7)$$

where $\mathrm{H}(D)$ is the information entropy of the diagnostic candidate set D, defined as

$$\mathrm{H}(D) = -\sum_{d_k\in D} \Pr(d_k|o_i,\ldots)\cdot\log_2(\Pr(d_k|o_i,\ldots)) \quad (8)$$

In the case when any $\Pr(d_k|o_i) = 0$, H can still be calculated, as $\lim_{x\to 0} = x\cdot\log_2 x = 0$.

In Equation 7, $D|o_i = 0$ represents the updated diagnosis if test t_i passes, and $D|o_i = 1$ if it fails.

The rationale for this heuristic is that H is an estimation of both the remaining tests towards an unambiguous diagnostic, and the residual diagnostic cost if testing would stop at the given state. Under ideal conditions, diagnostic prioritization performs a binary search, bisecting the set of candidates after each test. Therefore, the number of tests (T) needed to reach a diagnostic is related to the number of binary tests needed to separate the candidates.

Furthermore, H and W are both monotonically decreasing after each test. Ideally, after each test, D contains half the number of candidates with non-null probabilities, reducing W in half and H by 1 bit. Therefore, a decrease in H also represents a reduction in residual diagnostic cost W, even when their correlation is not so strong.

Algorithm 1 Diagnostic Prioritization

$D \leftarrow (\{c_0\}, \{c_1\}, \ldots, \{c_M\})$
for all $d_k \in D$ **do**
  $\Pr[d_k] = p_j$
for $l \leftarrow 1, N$ **do**
  $i(l) = \arg\max(A, \mathcal{H}_{IG}(D, t_i))$
  $o_{i(l)}$ = RUNTEST($t_{i(l)}$)
  for all $d_k \in D$ **do**
    $\Pr[d_k]^l = \frac{\Pr(o_{i(l)}|d_k) \cdot \Pr[d_k]^{l-1}}{\Pr(o_{i(l)})}$
  REMOVEROWIN(A, $i(l)$)
return SORT(D, Pr)

The pseudocode in Algorithm 1 describes all the steps in the information gain prioritization procedure. Table IV shows the evolution of D and Pr in our example, for each test selected by the algorithm, and the plot in Figure 1 depicts the evolution of W with respect to T compared to $\mathcal{H}_{add-st}$ and the optimal solution.

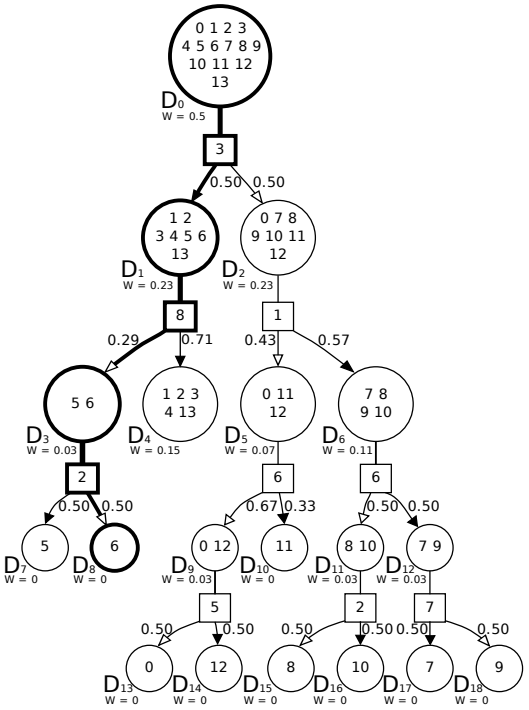

Figure 2. Optimal test sequence of the example system as tree, including fault-free candidate c_0

Conceptually, when considering all the possible test outcomes, a test suite prioritized for diagnosis is a tree, in contrast with off-line prioritization techniques using a static list. Figure 2 shows the complete tree for the system in Table I. Circular nodes contain the top-ranked candidates at each point in the decision process, and rectangular nodes represent which test is applied. The leaf nodes represent states where no test can improve the diagnostic, either because an unambiguous diagnosis has been reached, or because no test can refine the diagnostic any further. The average diagnostic effort (W) is annotated next to each D state. The probability of the outcome of each test is annotated next to the outgoing arrows from tests. An empty arrowhead represents a passed test, and a filled arrowhead represents a failed test.

Although the complete tree has up to $O(2^N)$ nodes, when calculated on-line, only the branches corresponding to the observed test outcomes have to be calculated. In our example system, this is marked with thicker lines in Figure 2. Consequently, the algorithmic complexity of the information-gain approach is $O(MN^2)$, similar to the $\mathcal{H}_{add-st}$ heuristic. Comparison with the worst case $O(M^3N)$ complexity of ART [8] depends on the relative size of M and N. In the benchmark suite used in our experiments N is much bigger than M, therefore ART has a somewhat lower cost.

VI. Experimental Setup

In order to evaluate the applicability of diagnostic prioritization, we address the following questions.

Question 1: What is the evolution of diagnostic effort (W) with respect to testing effort (T) for the information gain heuristic $\mathcal{H}_{IG}$? How does $\mathcal{H}_{IG}$ compare to random order and those generated by $\mathcal{H}_{add-st}$ and $\mathcal{H}_{art-mxmn}$?

Question 2: What is the fault detection performance of the new ordering produced by $\mathcal{H}_{IG}$?

Question 3: What is the best prioritization technique, taking into account the overall combined cost of testing and diagnosis?

For our study, we use a set of test programs known as the Siemens set [5]. The Siemens set is composed of seven programs. Each program has a set of test inputs that ensures full code coverage. Table V provides more information about the programs in the package (for more detailed information refer to [5]). Although the Siemens set was not assembled with the purpose of testing fault diagnosis techniques, it is typically used by the research community as the standard set of programs to test their techniques.

Program	LOC	Tests	Description
print_tokens	563	4130	Lexical Analyzer
print_tokens2	509	4115	Lexical Analyzer
replace	563	5542	Pattern Matcher
schedule	412	2650	Priority Scheduler
schedule2	307	2710	Priority Scheduler
tcas	173	1608	Aircraft Control
tot_info	406	1052	Information Measure

Table V
SET OF PROGRAMS AND VERSIONS USED IN THE EXPERIMENTS

The provided faults with each program in the set are not enough to obtain statistically significant results in some cases, given that diagnostic prioritization is designed for best *average* performance among the whole set of potential faults. Therefore we opt for a semi-synthetic approach, using the original spectra, but simulating a bigger sample of faults than

the ones provided by the Siemens set. The test outcomes are obtained by randomly choosing a faulty statement with uniform probability, and using its execution pattern (column in A) as test outcomes. Every time the fault is covered, an error is produced.

The coverage matrix A is obtained by instrumenting each of the programs with `Zoltar` [6] to obtain the statements covered by each test case. Type and variable declarations and other static code which is not instrumented were always assigned $a_{ij} = 0$ in previous literature. For our experiments, we reverse this convention, assigning $a_{ij} = 1$ for static code to avoid conflicts with the special a_{i0} column (See Section VII-D).

To answer Question 1, we measure and plot the evolution of W with respect to T for the first 100 tests of each program's prioritized test suite, for 500 simulated sample faults. We compare the random, $\mathcal{H}_{art-mxmn}$, $\mathcal{H}_{add-st}$, and $\mathcal{H}_{IG}$ heuristics.

With respect to Question 2, the test case in which the first failure occurs is stored, for each of the prioritized test suites. We compare the occurrence of the first failure for the random, $\mathcal{H}_{art-mxmn}$, $\mathcal{H}_{add-st}$, and $\mathcal{H}_{IG}$ heuristics. Following [14] we calculate the APFD measure to evaluate the rate of fault detection for the prioritized test suites. For a test suite with n tests and a set of m faults, where each fault F_i is first revealed in test T_{ffi}, the APFD value of such test suite is given by

$$APFD = 1 - \frac{T_{ff1} + T_{ff2} + \ldots + T_{ffm}}{nm} + \frac{1}{2n} \quad (9)$$

In order to answer Question 3, we calculate the combined cost of the detection and residual diagnosis of each fault. We assume that the test cost and (absolute) residual diagnosis cost can be added according to

$$C = T_{ff} + M \cdot W(T_{ff}) \quad (10)$$

where T_{ff} is the test where the first failure happens, and that the diagnostic process (debugging phase) starts the moment a failure is revealed. Note that we ignore relative differences in test cost and residual diagnosis cost.

VII. Results

A. Question 1: Fault Localization Performance

Figure 3 shows the evolution of W with respect to the number of executed tests T, averaged for all programs and per program as well. As can be seen, $\mathcal{H}_{IG}$ is consistently better than any other technique for every program, reaching the lower asymptote (the point where no other test can provide more diagnostic information) in less than 10 tests, for every program. No other technique achieves this improvement rate.

Consistent with [8], random orderings are the worst of all orders. The order created by $\mathcal{H}_{art-mxmn}$ is consistently better than random because it chooses tests always at a certain distance to the already applied ones. By doing this, the chance of choosing a test that bisects the current set of diagnostic candidates increases. The orders created by $\mathcal{H}_{add-st}$ do have a good initial performance, but after a few tests the progress stops, and W decreases very slowly.

The plot for `schedule2` in Figure 3 depicts an interesting case where A is extremely dense (including tests with full coverage). This makes $\mathcal{H}_{add-st}$ work extremely poorly because it will choose such tests, which add no diagnostic information at all, first. Also $\mathcal{H}_{art-mxmn}$ does not differ with random because it is difficult to keep a significant distance with the previous tests. Only $\mathcal{H}_{IG}$ is prepared to deal with this situation, and makes the most out of the available pool of tests.

In summary, based on the plots, we conclude that $\mathcal{H}_{IG}$ is most suitable for the QA purpose of fault localization. In the next section we will see how this implies a trade-off with failure detection.

B. Question 2: Failure Detection Performance

Figure 4 shows the averaged APFD scores for each heuristic, with their maximum and minimum values. By using permanent faults in our simulation, the values of the APFD scores are greater than in previous work, where intermittent faults were used. However, the differences in failure detection performance between each technique remain.

In Figure 4, it can be clearly seen how $\mathcal{H}_{add-st}$ is the best performing technique, in terms of mean APFD score and dispersion among programs. This is expected, as the assumptions under which $\mathcal{H}_{add-st}$ was devised are completely met in our experiment. The failure detection performance of $\mathcal{H}_{IG}$ is lower than $\mathcal{H}_{add-st}$ and slightly lower than $\mathcal{H}_{art-mxmn}$, although with a lower dispersion. $\mathcal{H}_{art-mxmn}$ has a better performance than random and a lower dispersion, consistent with [7]. Again, this is caused by the coverage distance kept between each test.

Theoretically, the number of tests until the first failure occurs can be modeled in the ideal case by a geometric distribution $X \sim G(p)$, whose expected value is $E[X] = p^{-1}$. The objective of $\mathcal{H}_{add-st}$ is choosing tests with maximum failure probability, ideally $p = 1.0$, and therefore approximately 1 test is needed on average ($T_{ff} \approx 1$). On the other hand, $\mathcal{H}_{IG}$ tends to select test cases which balance the probability of passing and failing, ideally $p = 0.5$, and therefore on average needs 2 tests ($T_{ff} = 2$).

In summary, when considering early failure detection as the main goal, $\mathcal{H}_{add-st}$ is more suitable for this purpose.

C. Question 3: Best Combined Performance

Table VI shows the average combined costs according to Equation 10 per program, at the point where the first failure occurs ($T = T_{ff}$), and the improvement with respect to a random order.

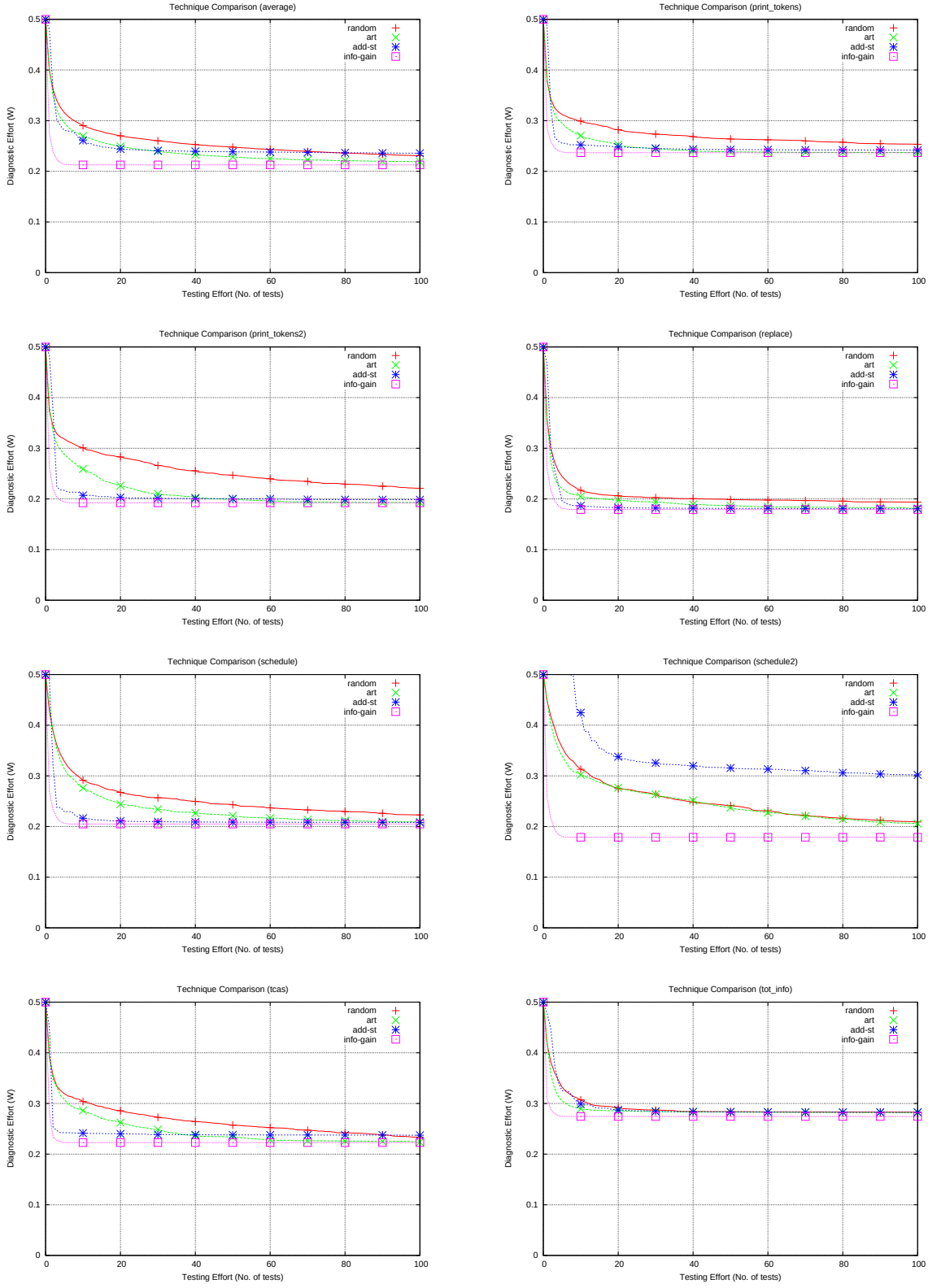

Figure 3. $W(T)$ for the various prioritization approaches (Siemens Set)

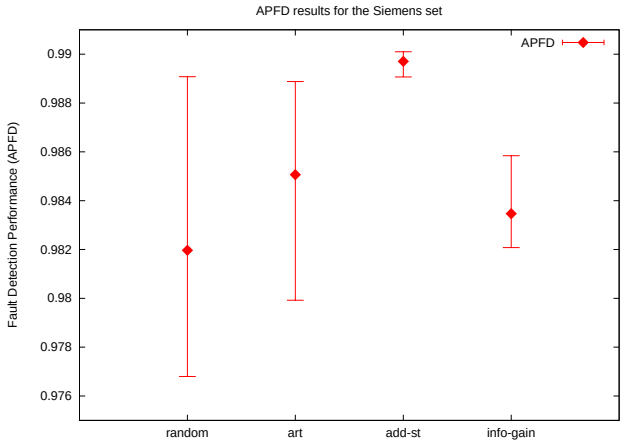

Figure 4. APFD results for the Siemens set

In our case, considering the QA cost as a whole, the number of tests required to reveal the presence of a fault T_{ff} is not the most relevant term, because in general, testing is an automated process whereas debugging is a manual, cognitive process, and therefore much more costly.

$\mathcal{H}_{add-st}$ has an increased cost over random orders, because although faults are detected very early, the diagnostic information gain is very limited. Despite the fact that $\mathcal{H}_{IG}$ needs more tests to detect the presence of a fault, this is more than compensated by the improved diagnostic information provided.

From the data in our experiments, we conclude that the $\mathcal{H}_{IG}$ order is the most appropriate for the global purpose of reducing combined QA cost, with an average cost reduction of 39% with respect to the combined cost of randomly ordered tests.

Program	Rand	$\mathcal{H}_{art-mxmn}$		$\mathcal{H}_{add-st}$		$\mathcal{H}_{IG}$	
	C	C	ΔC	C	ΔC	C	ΔC
print_tokens	210.5	210.6	+0.1%	275.2	+30.7%	143.3	-32.0%
print_tokens2	187.5	189.6	+1.1%	247.5	+32.0%	109.4	-41.6%
replace	195.2	193.0	-1.1%	262.8	+34.6%	116.7	-40.2%
schedule	176.9	177.3	+0.2%	202.9	+14.7%	95.0	-46.3%
schedule2	138.1	136.7	-1.0%	154.5	+11.8%	64.2	-53.5%
tcas	69.3	69.1	-0.3%	78.9	+13.7%	44.3	-36.1%
tot_info	169.9	174.8	+2.9%	195.1	+14.9%	118.8	-30.0%

Table VI
AVERAGE COMBINED COST $C = T_{ff} + M \cdot W(T_{ff})$

As on-line prioritization has to be performed for each test, the time overhead imposed by the algorithm is a critical success factor in this approach to QA. For the coverage matrix of `print_tokens` (4130×563), selecting a test takes in our (non-optimized) experimental platform approximately 1s of CPU time. For comparison, ART takes an average of 20ms per test. This overhead can be avoided because the next case can be pre-computed in parallel with the test being executed. It must be taken into account that it is necessary to speculatively pre-compute the next test for both possibilities of the yet unknown outcome, which requires twice the time.

D. Threats to Validity

We perform our experiments in a permanent fault setting, which is not very common in software. With respect to fault intermittency, although the numerical values of the results on Questions 1 and 2 are different from literature, the conclusions drawn are consistent with work where intermittent faults were used [8], [14], [17].

Modifying the columns in A where, for any i, $a_{ij} = 0$ to $a_{ij} = 1$ obeys to practical reasons. From a practical point of view, those statements usually correspond to interface, type and variable declarations. Although they are not 'executed' in test cases, they influence every single run. Therefore, we consider that every test case is checking them. This change does not affect W significantly as a fault in executable code will always rank above static statements. If the fault is located in static code, having $a_{ij} = 0$ would send the fault to the bottom of the ranking, whereas with $a_{ij} = 1$ it will be kept as a plausible diagnostic explanation, improving W independently of the prioritization algorithm used

Simulation of faults has enabled us to obtain a greater sample of faults per program, but it also affects the validity of our results. As we used the simulated fault distribution as input for the prioritization algorithm, our results show the performance of diagnostic prioritization when it has the best prior information available, something to take into account for its practical application.

With regard to our results in Question 3, the construct validity of the formula for C has to be considered. Our formula considers that the cost of a test is equal to the cost of manually inspecting a component (which can be seen as a sort of 'test' as well). Manual inspection (debugging) is usually much more expensive than just testing, which means that our formula is actually pessimistic in terms of the cost improvement we obtain with $\mathcal{H}_{IG}$.

VIII. Related Work

The information gain heuristic was first proposed to solve the problem of *sequential diagnosis* of hardware systems [9]. Algorithms for solving sequential diagnosis exactly which can be applied to systems with permanent [12] and intermittent [13] faults do exist.

As mentioned earlier, our work was motivated by previous empirical evidence that test suite prioritization and reduction [3], [14], [15] techniques have a negative impact on the diagnostic quality provided by fault localization algorithms [8], [17]. In [2], the diagnostic quality that a test suite provides is enhanced by adding new test cases that increase the number of *dynamic basic blocks* (DBB). DBBs are blocks of code that have different execution patterns (i.e., their corresponding columns in A are different). Blocks with similar columns will always rank together, increasing residual diagnostic effort. This enhancement is complementary to our technique, as it provides a lower W limit, whereas

our approach ensures that such limit is reached in the fewest possible tests.

IX. Conclusions and Future Work

In this paper, we have introduced a specific diagnostic prioritization of test cases that reduces the loss of diagnostic information to a minimum. Our experiments have shown that in terms of diagnostic information gain per test case, diagnostic prioritization is the best existing technique. This comes at the price of a reduced first failure detection performance with respect to additional-coverage techniques. However, when considering the overall combined cost of both testing and manual residual diagnosis, our experiments have shown cost reduction of up to 53% with respect to the next best performing technique.

In future work we will extend the validation of our approach to larger systems with intermittent faults, a more realistic scenario in software. We will also explore the performance of our approach at different levels of granularity, such as interface, and component-level granularities.

Acknowledgments

The authors wish to thank Rui Abreu for his invaluable feedback and their partners in the Poseidon project in the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] R. Abreu, P. Zoeteweij, and A. van Gemund. Spectrum-based multiple fault localization. In *24th International Conference on Automated Software Engeneering (ASE'09)*, pages 88–99. IEEE Computer Science, November 2009.

[2] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *28th International Conference on Software Engineering (ICSE'06)*, pages 82–91, Shanghai, China, 2006.

[3] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28:159–182, 2002.

[4] A. González, E. Piel, and H.-G. Gross. A model for the measurement of the runtime testability of component-based systems. In *Software Testing Verification and Validation Workshop, IEEE International Conference on*, pages 19–28, Denver, CO, USA, 2009. IEEE Computer Society.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE '94*.

[6] T. Janssen, R. Abreu, and A. van Gemund. Zoltar: A toolset for automatic fault localization. In *24th International Conference on Automated Software Engeneering (ASE'09) - Tools Track*, pages 658–660. IEEE Computer Society, November 2009. Best Demo Award.

[7] B. Jiang, Z. Zhang, W. Chan, and T. Tse. Adaptive random test case prioritization. In *24th International Conference on Automated Software Engeneering (ASE'09)*, Los Alamitos, USA, 2009. IEEE Computer Society.

[8] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *33rd Annual IEEE International Computer Software and Applications Conference*, pages 99–106, Seattle, Washington, USA, 2009.

[9] R. Johnson. An information theory approach to diagnosis. In *Proceedings of the 6th Symposium on Reliability and Quality Control*, pages 102–109, 1960.

[10] 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 - ICSE '02*, page 467, Orlando, Florida, 2002.

[11] C. Murpy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *ICST '09: Proceedings of the 2nd international Conference on Software Testing*. IEEE Computer Society, 2009.

[12] K. Pattipati and M. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. In *Proceedings IEEE International Symposium on Intelligent Control*, pages 291–296, Arlington, VA, USA, 1988.

[13] V. Raghavan, M. Shakeri, and K. Pattipati. Test sequencing algorithms with unreliable tests. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 29(4):347–357, 1999.

[14] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[15] A. M. Smith and G. M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 461–467. ACM Press, Mar. 2009.

[16] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka. The MORABIT approach to runtime component testing. In *30th Annual International Computer Software and Applications Conference*, pages 171–176, Sept. 2006.

[17] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *International Conference on Software Engineering (ICSE 2008)*, pages 201–210, Leipzig, Germany, May 2008.

[18] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *23rd International Conference on Machine Learning (ICML '06)*, pages 1105–1112, New York, NY, USA, 2006. ACM.