SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Prioritizing Tests for Software Fault Diagnosis

Alberto Gonzalez-Sanchez[1]*, Éric Piel[1], Rui Abreu[2],
Hans-Gerhard Gross[1], Arjan J.C. van Gemund[1]

[1]*Department of Software Technology, Delft University of Technology. The Netherlands.*
[2]*Department of Informatics Engineering, Faculty of Engineering, University of Porto, Portugal.*

SUMMARY

During regression testing, 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 our benchmark set, the results of our test case prioritization approach show reductions of up to 60% of the overall combined cost of testing and debugging, compared with the next best technique. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: test prioritization; testing; diagnosis

1. INTRODUCTION

Software regression testing is a time-consuming but rather important task for improving software reliability. Two main steps can be distinguished: (1) testing to find failures (i.e., detect the presence of faults), and (2) finding the root causes of the failures (faults, defects, bugs). Whereas the former is what is commonly known as “testing”, the latter is commonly denoted as “fault diagnosis”, “fault localization” or “debugging”.

Given the significant cost associated with regression testing, test *prioritization* has emerged as predominant technique to reduce testing cost, by trying to find failures as soon as possible [1, 2, 3, 4, 5, 6, 7, 8]. The main motivation behind test prioritization is that, the sooner failures are found, the sooner diagnosis can commence.

The debugging phase can make use of automatic *fault localization* techniques, which help to significantly reduce the manual debugging effort needed, as shown in [9, 10, 11, 12]. Fault localization algorithms use the information provided by tests executed in the testing phase to deduce a list of program elements (e.g., functions, statements) that are highly suspect to be at fault (the

*Correspondence to: Department of Software Technology, Delft University of Technology. Mekelweg 4, 2628CD Delft, The Netherlands. E-Mail: a.gonzalezsanchez@tudelft.nl

Contract/grant sponsor: POSEIDON project of the Embedded Systems Institute (ESI), Eindhoven, The Netherlands. Partially Supported by the Dutch Ministry of Economic Affairs; contract/grant number: BSIK03021

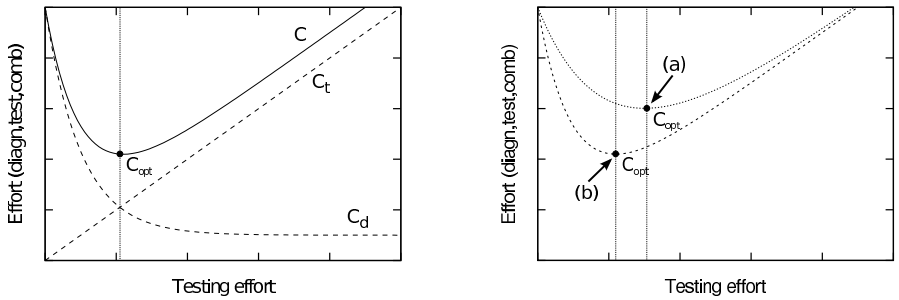

Figure 1. Test cost C_t and diagnosis cost C_d combined (left) and effect of test orders in the optimum (right)

diagnosis). Engineers then manually inspect the components top-down in search for the faults. The better information is provided by tests, the least work will be wasted in the manual inspection. What test prioritization techniques overlook, in general, is that the quality of the result of the fault localization phase depends on the quality of the information provided by the testing phase. Testing should be done with diagnosis in mind, from the first test.

The plot on the left hand side of Figure 1 illustrates the cost sources associated to regression testing and debugging. $C_t(N)$ denotes the aggregate time cost of testing, where N is the number of tests. $C_d(N)$ denotes the time cost associated with the diagnostic work performed by the software developer to debug the actual defects. $C_t(N)$ has a linear shape as a function of the number of tests N, whereas $C_d(N)$ has a geometric decreasing shape [10, 13]. The overall time cost of the combined testing and diagnosis process, C, can be modeled by

$$C(N) = C_t(N) + \alpha \cdot C_d(N) \tag{1}$$

where α expresses the possible factor between the cost of the regression tests C_t and the cost of the manual inspection performed by the developer (typically high). After an initial decreasing phase, C reaches its optimum point C_{opt}. Past this point, new tests will not provide enough new diagnostic information to compensate their cost, and C will increase. The location of the optimum point depends on how fast C_d decreases, and the value of α. The effect of two different prioritization techniques can be observed on the plot on the right hand side of Figure 1.

Recent work on the combination of test prioritization and fault localization [14, 15, 16] has highlighted the fact that, while test prioritization minimizes the delay between testing and diagnosis $C_t(N)$, it does not maximize diagnostic information yield $C_d(N)$. The reason is that test prioritization aims at high code coverage, whereas diagnosis aims at partially revisiting already covered code to further exonerate or indict defective components. The net effect of existing test prioritization techniques is that the overall cost of the combined process of testing and debugging, given defective code, is not minimized. This suboptimal situation corresponds to scenario (a) on the right hand side plot in Figure 1.

In software development processes with high code production rates, the probability of introducing at least one defect between subsequent revisions is considerable. Consequently, the probability of having to apply diagnosis after regression testing is high. This calls for a prioritization technique (i.e., *diagnostic prioritization*) that puts emphasis on *fault localization* performance from the first test, rather than failure detection performance. The goal should be to reduce the overall QA cost, i.e., the combined cost of testing and debugging, by performing a trade-off between testing and debugging effort. This corresponds to scenario (b) in Figure 1.

Shifting the focus of test choice from failure detection to fault diagnosis is bound to cause an increase in testing cost, if measured according to usual test cost metrics such as APFD [6]. However, this increase is negligible once considered in the context of combined cost C, since usually the cost of applying a test is orders of magnitude lower than manually inspecting a component (i.e., α in Equation 1 is very large). Furthermore, even if $\alpha \leq 1$ (i.e., executing a test is as expensive as manual inspection), it must be taken into account that a manual inspection will discard *only one* component,

DOI: 10.1002/spe

whereas a test execution has the potential to discard *half* of the components if the test is properly chosen.

In this paper we present and evaluate a novel, dynamic approach to diagnostic prioritization, which aims to minimize the combined cost of testing and diagnosis. Unlike static test prioritization, in diagnostic prioritization the tests are dynamically selected based on the actual pass/fail results of previously executed tests (which varies per regression cycle), leading to higher diagnostic performance per test, yielding the much steeper $C_d(N)$ curve in scenario (b). In particular, we make the following contributions:

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

The current paper represents novel work in the domain of test prioritization and diagnosis, and is an extension of our preliminary work in diagnostic prioritization presented in [14]. The following extensions have been added:

1. We refine our definition of *diagnostic prioritization*. Unlike in [14], we take into account the possibility that not all test cases that cover the fault will result in a failure (i.e., the possibility of false negatives or *coincidental correctness*.
2. We perform the evaluation of our technique under these new assumptions on three different false negative rate scenarios (low, high, very high) on the Siemens programs [17] and extend it to four larger and more realistic programs from the SIR repository [18]. Our results show up to 60% reduction of the overall combined cost of testing and debugging, when compared to the next best performing technique.
3. We present a discussion on the practical applicability issues of diagnostic prioritization, with emphasis on the estimation of the input parameters (fault probability, false negative rate) and the extent to which the accuracy of diagnostic prioritization is affected by erroneous estimations.

The paper is organized as follows. In Section 2, we describe the main concepts of fault diagnosis and the diagnosis algorithm used in our experiments. Section 3 surveys the existing prioritization techniques with which we will compare our approach. In Section 4, we describe why current prioritization techniques fall short for fault localization. Section 5 introduces diagnostic prioritization and the information gain heuristic. Our evaluation goals and experimental setup are described in Section 6, while the results are presented and discussed in Section 7. Section 8 discusses a number of practical considerations, including the sensitivity of our technique to erroneous parameter estimations. Related work is surveyed in Section 9. Section 10 presents our final conclusions and future work directions.

2. FAULT DIAGNOSIS

The objective of fault diagnosis is to pinpoint the precise location of a number of faults in a program (bugs) by observing the program's behavior given a number of tests. Although there is a large number of different diagnosis techniques (see Section 9), our work is based on Bayesian diagnosis, well-known from Model-Based Diagnosis, an area within AI. Compared to other, statistical approaches such as Tarantula [11], Ochiai [9], and alternative techniques [19, 20, 21, 22, 23, 24, 25], Bayesian diagnosis is founded on probability theory, and is the only technique that can serve as base for our test prioritization heuristic search function, described in Section 4.

The following inputs are involved in diagnosis:

DOI: 10.1002/spe

- A finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_j, \ldots, c_M\}$ of components (typically source code statements) that 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 set of *false negative rates* (FNR) $0 \leq h_j \leq 1$ (for *health*) for each component c_j. The value of h_j expresses the probability c_j will **not cause** a failure when covered in a test iff c_j is faulty. This is also known as *intermittency* or *coincidental correctness* rate.
- 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 $N \times M$ test coverage matrix, $A = [a_{ij}]$, where $a_{ij} = 1$ if test t_i involves component c_j, and 0 otherwise.

Bayesian fault diagnosis is aimed at obtaining a set of fault candidates $D = \langle d_1, \ldots, d_k \rangle$. Each candidate d_k is a subset of the components that, at fault, explain the observed failures. For instance, $d = \{c_1, c_3, c_4\}$ indicates that c_1 **and** c_3 **and** c_4 are faulty, and no other component. As most previous work [15, 11, 16], for the scope of this paper, we will assume that the system under test contains exactly **one** fault. The j subindices designate members in $\mathcal{C}$, whereas k designates members of D. However, under the single fault assumption $|D| = |C|$, therefore these two indices can be used interchangeably, as $d_k \equiv \{c_k\}$.

Due to the limited number of tests, the number of possible diagnostic explanations is typically very high. The output of fault diagnosis, i.e., a *diagnosis*, is a component *ranking*, i.e., a list of component indices, ordered by the *likelihoods* of each component being faulty. As already mentioned, in this paper we consider a Bayesian approach to obtain this ranking. Therefore the likelihoods are the fault probabilities output by the Bayesian reasoning process.

2.1. Diagnostic Ranking by Bayesian Reasoning

In the case of Bayesian approaches, the likelihood of a diagnosis corresponds to the posterior probability of a component being faulty, given the outcome of the executed test, $\Pr(d_k|o_i)$, for a particular diagnosis d_k. As there can only be one correct explanation, all the individual probabilities add up to 1.

Initially, for each component c_k, the probability of each explanation is $\Pr(d_k) = p_k$. The value of p_k represents the knowledge available before any test is executed. Component priors are typically derived from defect density data.

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:

$$\Pr(d_k|o_i) = \frac{\Pr(o_i|d_k)}{\Pr(o_i)} \cdot \Pr(d_k) \tag{2}$$

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, given by

$$\Pr(o_i = 1|d_k) = 1 - \Pr(o_i = 0|d_k) = a_{ik} \cdot (1 - h_k) \tag{3}$$

where h_k is the false negative rate of the faulty component. If $h_k = 0.1$, this means that, if c_k was the fault, a 10% of the test cases that cover c_k would not produce a failure, i.e., an erroneous result.

$\Pr(o_i)$ 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

$$\Pr(o_i) = \sum_{d_k \in D} \Pr(o_i|d_k) \cdot \Pr(d_k) \tag{4}$$

and need not be computed directly.

DOI: 10.1002/spe

	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

2.2. Residual Diagnostic Cost

The final diagnosis after N observations D_N is returned to the user as the basis to find the actual fault. Typically the user finds the fault by inspecting each candidate in the ranking in descending order according to the updated diagnostic probabilities.

In the following, we define C_d as the number of components the developer has to examine until finding the real fault d_* [10]. It corresponds to the position of d_* in the ranking. Because multiple explanations can be assigned the same probability, the value of C_d is averaged between the ranks of explanations that share the same probability, amongst which the real fault d_* is located.

$$C_d = \frac{|\{k : \Pr(d_k) > \Pr(d_*)\}| + |\{k : \Pr(d_k) \geq \Pr(d_*)\}| - 1}{2} \tag{5}$$

The above model for C_d is similar to existing diagnostic performance metrics [11, 22].

There are two ways of reducing diagnostic cost. One can try to develop better techniques to reduce the residual diagnosis effort C_d by reducing the number of candidates, or 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 C_d) is the main focus of this paper.

2.3. Example Diagnosis

Table I shows an example faulty program [4], eight tests ($t_1 ... t_8$), and their respective 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: $d_k \equiv \{c_k\}$. We will consider $h_k = 0.1$ for all components. 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 candidates containing covered statements (including d_6) are updated by

$$\Pr(d_k|o_1) = \frac{\Pr(o_1 = 1|d_k) \cdot \Pr(d_k)}{\Pr(o_1)} = \frac{\frac{9}{10} \cdot \frac{1}{13}}{\frac{99}{130}} = \frac{1}{11}$$

The candidates corresponding to statements that were not covered are updated by

$$\Pr(d_k|o_1) = \frac{\Pr(o_1 = 1|d_k) \cdot \Pr(d_k)}{\Pr(o_1)} = \frac{0 \cdot \frac{1}{13}}{\frac{99}{130}} = 0$$

DOI: 10.1002/spe

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. This is only true under the single fault assumption used in this paper.

After applying test t_2, no failure occurs. The probabilities of the 9 covered statements $(d_1, d_2, d_3, d_4, d_5, d_7, d_9, d_{10}, d_{13})$ are then updated by

$$\Pr(d_k|o_2, o_1) = \frac{\frac{1}{10} \cdot \frac{1}{11}}{\frac{29}{110}} = \frac{1}{29}$$

and the 2 uncovered statements (c_6, c_8) by

$$\Pr(d_k|o_2, o_1) = \frac{1 \cdot \frac{1}{11}}{\frac{29}{110}} = \frac{10}{29}$$

The normalizing value $\frac{29}{110}$ comes from adding up the 9 numerators of the covered statements and the 2 non-covered statements: $9 \cdot \frac{1}{10} \cdot \frac{1}{11} + 2 \cdot \frac{1}{11} = \frac{29}{110}$ (Equation 4).

The next test to be applied is t_3, which fails. The probabilities for $d_1, d_2, d_3, d_4, d_5, d_{13}$ are updated to

$$\Pr(d_k|o_3, o_2, o_1) = \frac{\frac{9}{10} \cdot \frac{1}{29}}{\frac{144}{290}} = \frac{1}{16}$$

The probability of d_6 is updated to

$$\Pr(d_k|o_3, o_2, o_1) = \frac{\frac{9}{10} \cdot \frac{10}{29}}{\frac{144}{290}} = \frac{10}{16}$$

As d_8 is not being covered, its probability is updated to 0. From this point on, d_6 is ranked on top, which means no diagnostic effort is wasted and the diagnosis is finished from a practical point of view, i.e. it is still ambiguous but $C_d = 0$. The subsequent tests will only reaffirm this diagnosis by bringing $\Pr(d_6)$ very close to 1, i.e., they will make the diagnosis perfect, i.e. unambiguous. The complete evolution for the 8 test cases is shown in Table II. Empty cells correspond to zero probabilities, whereas 0.00 represent very small non-zero probabilities.

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}	C_d
		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_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.03	0.03	0.03	0.03	0.03	0.34	0.03	0.34	0.03	0.03			0.03	0.038
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.06	0.06	0.06	0.06	0.06	0.63							0.06	0.000
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.06	0.06	0.06	0.06	0.06	0.63							0.06	0.000
t_5	0	1 1 1 1 1 0 1 1 1 0 1 1 1	0.01	0.01	0.01	0.01	0.01	0.94							0.01	0.000
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_7	1	1 1 1 1 1 1 1 1 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000

Table II. Evolution of D for our example system and default test order.

3. 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 achieve this goal, and will be used in the evaluation to compare with our diagnostic prioritization technique.

DOI: 10.1002/spe

3.1. Random

Random prioritization (RND) is the most straightforward prioritization criterion. It orders test cases according to random permutations of the original test suite. Random permutations are used as control in many prioritization experiments [2, 6, 7].

3.2. Statement coverage

Following statement coverage prioritization, the test cases that cover the highest total number of statements are executed first. This approach is based on 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 less useful as it is less likely it will produce a failure [2, 6]. This reasoning conduces to the definition of the *additional coverage* heuristic (ADDST), 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.,

$$\mathcal{H}_{ADDST}(t_i) = \sum_{j=1}^{M} a_{ij} \cdot (1 - cov_j) \tag{6}$$

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

3.3. Fault-exposing potential (FEP)

This criterion adapts ADDST's "binary" approach (i.e., covered, not covered) to a continuous confidence value, to account for the fact that a test may produce a false negative. If a statement has been covered a number of times, the confidence on its correctness increases, making it less likely to be chosen again. We will consider only the additional FEP prioritization heuristic, given by

$$\mathcal{H}_{FEP}(t_i) = \sum_{j=0}^{M} FEP_{ij} \cdot (1 - conf_j) \tag{7}$$

where $conf_j$ is the confidence in statement$j$ being correct. FEP_{ij} is the probability that test i will fail for statement j. The value of $conf_j$ is updated after executing test t_i according to

$$conf'_j = conf_j + FEP_{ij} \cdot (1 - conf_j) \tag{8}$$

In our experiments, the FEP_{ij} value corresponds to the complementary value of the false negative rate: $FEP_{ij} = 1 - h_j$.

3.4. Adaptive Random Testing

ART is a hybrid random-coverage-based test ordering [4]. It selects test cases in two steps, first it samples a group of tests randomly, and from that group it selects the test that maximizes a distance function with the already executed 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 [4] as the most promising one. It is defined as

$$\mathcal{H}_{ART}(t_i) = \min_{t_j \in C}(\delta(t_i, t_j)) \tag{9}$$

where C is the set of already applied tests and δ is the distance function used, in [4] the Jaccard distance.

4. PRIORITIZATION AND DIAGNOSIS

Previous empirical work has shown that early failure detection and fault localization seem to be rather incompatible goals [15, 16]. The evolution of the diagnostic effort C_d, per unit of test effort,

DOI: 10.1002/spe

C_t, is negatively affected by criteria for early failure detection. Random ordering, which has been traditionally considered the baseline prioritization technique [2, 6, 7], was found to perform as good as or better than all other prioritization techniques, except ADDST. However, even for the latter case the random order was better for some subject programs [15].

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 assuming all tests will pass. 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 III shows an example of how traditional test prioritization for failure detection fails for diagnosis, when using the ADDST heuristic for our example system (shown in Table I). We use the Bayesian diagnosis approach from Section 2.1, assuming a fault will cause a 10% of false negatives ($h_k = 0.1$). For clarity, the fourth column shows only the probabilities of diagnostic explanations that are non-zero. Initially, no statement has been covered, and D ranks every candidate with uniform probability.

ADDST selects test t_1 as first test, as it covers the most test cases, and, indeed, t_1 produces the first failure. As a result of the failure, all the c_k covered by t_1 move to the top of the ranking. Unfortunately, the test case covered many statements, so C_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 candidates in D that correspond to statements covered by t_5 drop below the ones that were not covered. The extent of this drop depends on h_j, the lower h_j the lower they will drop. The statements that where not covered remain at the top of the ranking. Full coverage has been reached, so in the third step, the coverage is reset as described in [6], instead of falling back to a random sequence. 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 neither D nor C_d change. This happens also in the fourth step for t_6. Finally, t_7, a test case that covers c_{10} but not c_6 is chosen. As it fails, c_{10}, which was not covered, is assigned a zero probability (since we assumed there is only one fault). c_6 remains as the only most likely explanation. As we can see, ADDST selects 2 tests that provide no information to the diagnosis, independent of their outcome, i.e., resulting in wasted testing effort C_t from the point of view of diagnostic performance C_d.

As a comparison, Table IV shows the optimal test order for unequivocally diagnosing 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 2 depicts the evolution of both approaches.

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}	C_d
		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 0 1 1 1 0 1 1 1	0.03	0.03	0.03	0.03	0.03	0.34	0.03	0.03	0.03	0.34			0.03	0.038
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.03	0.03	0.03	0.03	0.03	0.35	0.03		0.03	0.35			0.03	0.038
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.00	0.00	0.00	0.00	0.00	0.48	0.00		0.00	0.48			0.00	0.038
t_7	1	1 1 1 1 1 1 1 1 0 0 0 0 1	0.01	0.01	0.01	0.01	0.01	0.93	0.01						0.01	0.000
t_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000

Table III. Evolution of D for the ADDST heuristic for our example system.

Although simple, this example demonstrates that maximizing the probability of failure does not maximize the information that the diagnostic algorithm receives. In fact, as test cases that

DOI: 10.1002/spe

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}	C_d
		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	0.03	0.03	0.03	0.03	0.03	0.32	0.03	0.03	0.03	0.32	0.03	0.03	0.03	0.038
t_7	1	1 1 1 1 1 1 1 1 1 0 0 0 0 1	0.05	0.05	0.05	0.05	0.05	0.55	0.05	0.05					0.05	0.000
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.01	0.01	0.01	0.01	0.01	0.85	0.01	0.08					0.01	0.000
t_1	1	1 1 1 1 1 1 1 1 1 1 1 0 0 1	0.01	0.01	0.01	0.01	0.01	0.85	0.01	0.08					0.01	0.000
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.01	0.01	0.01	0.01	0.01	0.93	0.01						0.01	0.000
t_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000

Table IV. 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}	C_d
		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 0 0 0 0 0 0 0 0 1	0.04	0.04	0.04	0.04	0.40	0.40							0.04	0.038
t_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.01	0.01	0.01	0.01	0.08	0.86							0.01	0.000
t_5	0	1 1 1 1 1 0 1 1 1 0 1 1 1	0.00	0.00	0.00	0.00	0.01	0.98							0.00	0.000
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_1	1	1 1 1 1 1 1 1 1 1 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_7	1	1 1 1 1 1 1 1 1 1 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000

Table V. Evolution of D for the IG heuristic for our example system.

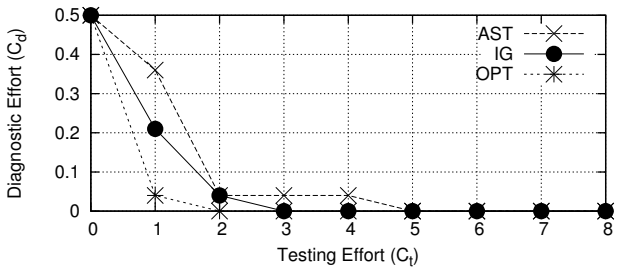

Figure 2. $C_d(C_t)$ for three prioritization approaches

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.

In contrast, for test prioritization for diagnosis, we need test cases that revisit partially the already tested system, to introduce diversity in the coverage and reduce the set of suspect components.

5. 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 inspired by research in *sequential diagnosis* of hardware systems, where algorithms exist to diagnose systems with permanent [26] and intermittent [12] failures. A preliminary study on diagnostic prioritization for software was presented in [14].

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.

DOI: 10.1002/spe

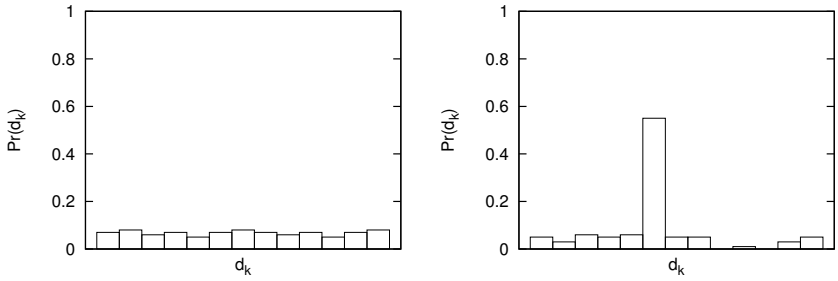

Figure 3. Very ambiguous diagnosis (left) and very precise diagnosis (right).

For diagnosis, the best tests are those that, at each step, maximize the reduction of diagnostic cost C_d. However, since we do not know the correct diagnosis (d_*) a different heuristic has to be used.

In general, good diagnoses are those in which one of the candidates has a much larger probability than the rest, i.e, it stands out over all other candidates. Figure 3 illustrates what we want to achieve. The plot on the left shows a poor diagnosis where no clear candidate exists. Our objective is to produce a diagnosis like the one on the right hand side plot.

A reduction in the ambiguity of the diagnosis can be seen as an increase in diagnostic information, i.e., a reduction of the information entropy of the candidate set D. Applying this reasoning, at each decision step in the test sequence, the test yielding the highest average information gain is chosen. The information gain heuristic [27], IG, is defined as

$$\mathcal{H}_{IG}(D, t_i) = \mathrm{H}(D) - \Pr(o_i = 0) \cdot \mathrm{H}(D|o_i = 0) - \Pr(o_i = 1) \cdot \mathrm{H}(D|o_i = 1) \tag{10}$$

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

$$\mathrm{H}(D) = -\sum_{d_k \in D} \Pr(d_k) \cdot \log_2(\Pr(d_k)) \tag{11}$$

$D|o_i = 0$ represents the updated diagnosis if test t_i passes, and $D|o_i = 1$ if it fails. In the case when any $\Pr(d_k|o_i) = 0$, H can still be calculated, as $\lim_{x \to 0} = x \cdot \log_2 x = 0$.

The rationale of the IG 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 (C_t) needed to reach a diagnosis is related to the number of binary tests needed to separate the candidates.

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

The pseudocode in Algorithm 1 describes all the steps in the information gain prioritization procedure. At every decision step, the test that maximizes IG is chosen, and then executed by the RUNTEST command. Once the result is available, the probabilities of each of the candidates are updated. To avoid repeating a test, its row in A is removed by REMOVEROWIN. Table V shows the evolution of D and $\Pr(d_k)$ in our example, for each test selected by the algorithm. The plot in Figure 2 depicts the evolution of C_d with respect to C_t compared to ADDST and the optimal solution.

5.1. Time/Space Complexity

Conceptually, when considering all the possible test outcomes, a test suite prioritized for diagnosis is a tree, in contrast to off-line prioritization techniques using a static list. Figure 4 shows an example

DOI: 10.1002/spe

Algorithm 1 Greedy Information Gain Prioritization

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

of such a tree. Circular nodes contain the updated sets of diagnostic candidates D at each point in the decision process, and rectangular nodes represent which test is applied at each step.

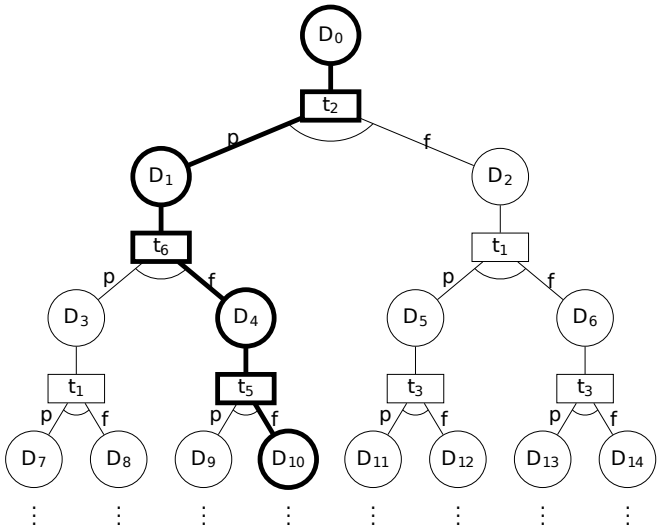

Figure 4. Decision tree of the IG-prioritized test suite

Although the complete tree has up to $O(2^N)$ nodes, and therefore the complexity of diagnostic prioritization is $O(M \cdot N \cdot 2^N)$, when calculated on-line, only the path corresponding to the observed test outcomes has to be calculated. This is marked with thicker lines in Figure 4. Consequently, the algorithmic complexity of the information-gain approach is $O(M \cdot N^2)$, similar to the ADDST heuristic. Comparison with the worst case $O(M^3 \cdot N)$ complexity of ART [15] 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.

6. EXPERIMENTAL SETUP

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

Question 1: What is the evolution of diagnostic effort (C_d) with respect to testing effort (C_t) for the information gain heuristic IG? How does IG compare to RND, ADDST, FEP and ART?

Question 2: What is the fault detection performance of the sequences produced by IG? Since IG is not aimed at maximizing fault probability, we expect an impact on the failure detection performance. Our aim is to quantify this impact.

DOI: 10.1002/spe

Question 3: What is the best prioritization technique, taking into account the overall combined cost of testing and diagnosis? This third question is the most relevant. Our hypothesis is that, even if failure detection performance is impacted, it will be greatly compensated by the return in terms of diagnostic cost savings.

For our study, we use a set of seven test programs known as the Siemens set [17], and 4 programs from the Software Infrastructure Repository (SIR) [18]. Each program has a set of test inputs that ensures full code coverage. Table VI provides more information about the programs (for more detailed information refer to [17] and [18]). Although the Siemens set and SIR programs were 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
`space`	9126	150	ADL Compiler
`gzip`	7933	210	Data Compression
`sed`	7125	370	Stream Editor
`grep`	13287	809	String Matching

Table VI. Set of programs and versions used in the experiments

The coverage matrix A of each program is obtained by instrumenting each of the programs with `Zoltar` [28] to obtain the statements covered by each test case. Type and variable declarations and other static code, which are not instrumented by `Zoltar`, are excluded from diagnostic rankings and effort calculations.

Each program is provided with a number of seeded faults (real in the case of `space`). The number and distribution of those faults are not enough to obtain statistically significant results in some cases, given that diagnostic prioritization is designed for best *average* performance among the whole universe of potential faults. Therefore we opt for a semi-synthetic approach, using the original test matrices, but simulating a bigger sample of faults and error vectors than the ones provided by the Siemens set. The test outcomes are obtained by randomly choosing a faulty statement with uniform probability. The simulated error vector is obtained by using the faulty statement's column in A. Every time the fault c_k is covered, an error is simulated from a Bernoulli distribution with probability $p = 1 - h_k$. We simulate three different FNR scenarios: low with 10% FNR ($h_k = 0.1$), high with 50% FNR ($h_k = 0.5$), and very high with 90% FNR ($h_k = 0.9$).

To answer Question 1, we measure and plot the evolution of C_d with respect to C_t for the first 100 tests of each program's prioritized test suite, for 10 runs for 100 simulated sample faults (1000 runs in total per technique). We compare the RND, ART, ADDST, FEP and IG heuristics in terms of the normalized area S under the C_d curve and above the diagnostic asymptote (the C_d value when the complete test suite has been executed), according to

$$S = \frac{1}{100} \cdot \sum_{i=1}^{100} (C_d(i) - C_d(N)) \tag{12}$$

We prefer this metric to, e.g., the value of C_d after an arbitrary number of tests or the number of tests required to reach an arbitrary C_d. The reason for this is that we want to avoid the definition of arbitrary performance thresholds.

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 RND, ART, ADDST, FEP and IG heuristics. Following [6] 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

DOI: 10.1002/spe

fault F_i is first revealed in test N_{ffi}, the APFD value of such test suite is given by

$$APFD = 1 - \frac{N_{ff1} + N_{ff2} + \ldots + N_{ffm}}{nm} + \frac{1}{2n} \tag{13}$$

In order to answer Question 3, we calculate the minimum possible combined cost of the detection and residual diagnosis of each fault, C_{opt}, as can be seen in Figure 1. We assume that the test cost and (absolute) residual diagnosis cost can be modeled according to $C = C_t + C_d(C_t)$, i.e., we ignore relative differences in test cost and absolute residual diagnosis cost (i.e., $\alpha = 1$ in Equation 1).

7. RESULTS

7.1. Question 1: Fault Localization Performance

Table VII shows the score values for each program and heuristic. The value marked in bold indicates the best performing technique or techniques with a 95% confidence according to the Bonferroni mean separation test. To further illustrate our results, Figure 5 shows the evolution of C_d with respect to the number of executed tests C_t, per program, for the $\bar{h} = 0.5$ scenario.

IG is consistently better than any other technique for every program and every false negative rate. No other technique achieves the improvement rate of IG. It can also be seen how in general, as the FNR increases, the difference between IG and RND narrows, as choices become more and more uncertain.

In our experiments we observe how ADDST, and especially FEP, are the worst in terms of C_d evolution, being even worse than RND. However, this would seem to contradict previous literature [15], where random sequences were generally worse than ADDST except for one case. The reason is that the work in [15] was analyzing the performance exclusively for the small set of faults present in the Siemens set for each program. As pointed out in [6], the original faults in Siemens are located in hard to reach areas (i.e., covered only by a few tests), which favors ADDST and FEP. However, in our simulations faults can be *anywhere* in the program's code, eliminating the bias that favors ADDST and FEP.

Our experimental results are consistent with our theory in Section 4, that maximizing failure probability hinders the improvement of C_d because it causes the execution of tests that cover an excessive number of components.

The plot for `schedule2` in Figure 5 depicts an interesting case where A is extremely dense (including tests with full coverage). This makes FEP perform extremely poorly because it chooses first tests that provide no diagnostic information at all.

In most cases, the order created by ART is better than RND 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, providing a slight advantage over RND.

Apparently contradicting our predictions, the performance of ADDST and FEP is actually better than RND for the larger SIR programs. Furthermore, it is not significantly different from IG in some of the $h = 0.9$ cases. The answer to this apparent contradiction resides in the shape and distribution of statement coverage in the test matrix A, and will be discussed in the Section 7.4.

In summary, based on the plots, and the statistical analysis, we conclude that IG is most suitable for the purpose of fault localization, given its performance and robustness to test matrices of differing sizes and shapes and false negative rates. In the next section we will see how this implies a trade-off with failure detection.

7.2. Question 2: Failure Detection Performance

Table VIII shows the averaged APFD scores for each heuristic. The value marked in **bold** indicates the best performing technique or techniques with a 95% confidence according to the Bonferroni mean separation test. This is complemented with Figure 6, where the average, minimum and maximum APFD of each technique are depicted.

DOI: 10.1002/spe

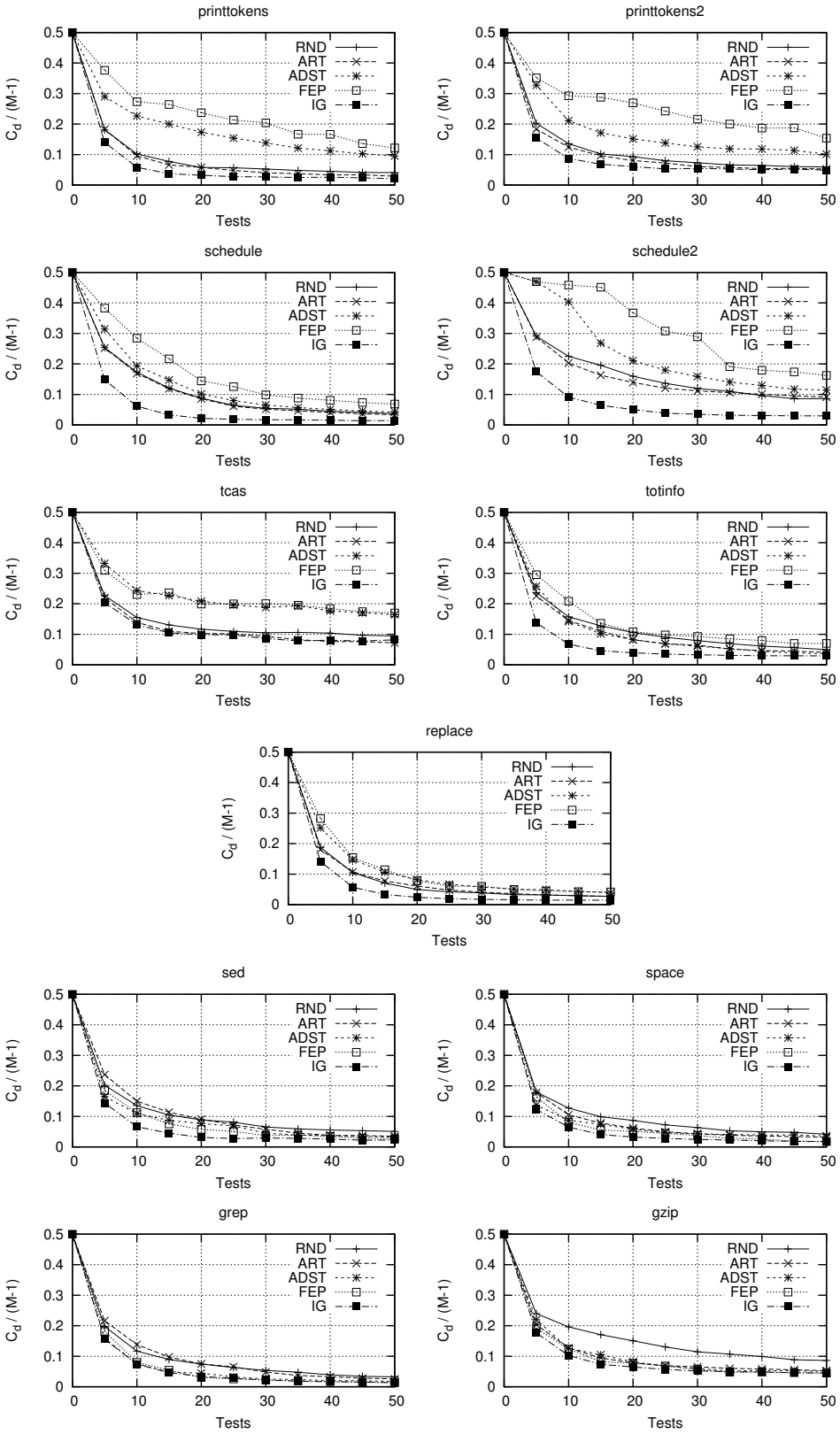

Figure 5. $C_d(N)$ for the various prioritization approaches

DOI: 10.1002/spe

Program	h	RND	ART		ADDST		FEP		IG	
print_tokens	0.1	0.036	0.029	-18.5%	0.076	+112.3%	0.097	+169.1%	**0.023**	-36.9%
	0.5	0.056	0.050	-10.5%	0.112	+101.0%	0.135	+143.1%	**0.035**	-36.8%
	0.9	0.133	0.128	-3.8%	0.207	+55.1%	0.173	+29.7%	**0.096**	-28.1%
print_tokens2	0.1	0.059	0.053	-11.0%	0.093	+57.5%	0.117	+98.9%	**0.045**	-24.2%
	0.5	0.085	0.075	-11.6%	0.116	+36.7%	0.152	+79.7%	**0.060**	-28.7%
	0.9	0.138	0.137	-0.9%	0.198	+43.2%	0.176	+27.6%	**0.115**	-16.5%
replace	0.1	0.026	0.024	-8.3%	0.035	+34.8%	0.040	+55.0%	**0.015**	-43.4%
	0.5	0.046	0.047	+1.2%	0.062	+35.1%	0.069	+49.4%	**0.028**	-38.8%
	0.9	0.127	0.140	+10.2%	0.160	+25.5%	0.155	+21.7%	**0.103**	-19.4%
schedule	0.1	0.042	0.040	-4.1%	0.046	+9.0%	0.052	+25.0%	**0.016**	-61.2%
	0.5	0.065	0.063	-3.7%	0.077	+19.2%	0.103	+59.2%	**0.029**	-56.1%
	0.9	0.151	0.141	-6.3%	0.152	+0.8%	0.172	+14.1%	**0.103**	-31.8%
schedule2	0.1	0.070	0.063	-9.4%	0.090	+29.5%	0.107	+53.8%	**0.022**	-68.8%
	0.5	0.106	0.105	-1.1%	0.155	+46.4%	0.219	+106.0%	**0.040**	-62.2%
	0.9	0.135	0.135	-0.2%	0.166	+22.5%	0.208	+53.6%	**0.100**	-26.3%
tcas	0.1	0.065	0.064	-1.4%	0.106	+63.5%	0.106	+63.9%	**0.057**	-12.8%
	0.5	0.093	0.089	-4.8%	0.164	+76.7%	0.167	+79.5%	**0.084**	-10.1%
	0.9	0.146	0.146	+0.0%	0.181	+23.7%	0.181	+23.6%	**0.135**	-7.8%
tot_info	0.1	0.038	0.036	-6.2%	0.039	+0.6%	0.048	+25.3%	**0.024**	-37.1%
	0.5	0.072	0.063	-13.4%	0.063	-12.5%	0.088	+21.4%	**0.038**	-47.4%
	0.9	0.145	0.150	+3.6%	0.126	-13.2%	0.148	+2.5%	**0.106**	-27.1%
space	0.1	0.039	0.028	-26.9%	0.018	-52.8%	0.018	-53.6%	**0.014**	-63.8%
	0.5	0.061	0.049	-19.3%	0.044	-28.4%	0.037	-39.4%	**0.031**	-49.0%
	0.9	**0.160**	0.169	+5.8%	0.162	+1.0%	**0.158**	-1.1%	**0.150**	-6.4%
grep	0.1	0.034	0.030	-11.4%	0.020	-40.6%	0.019	-42.3%	**0.015**	-55.6%
	0.5	0.056	0.055	-1.8%	0.040	-27.3%	0.036	-35.9%	**0.030**	-46.3%
	0.9	0.167	0.215	+28.8%	0.181	+8.8%	**0.137**	-17.9%	**0.131**	-21.5%
gzip	0.1	0.069	0.045	-34.0%	0.032	-54.0%	0.031	-54.7%	**0.027**	-60.9%
	0.5	0.098	0.068	-30.7%	0.060	-39.1%	**0.053**	-45.7%	**0.050**	-48.5%
	0.9	0.194	0.165	-15.0%	0.163	-16.2%	**0.140**	-27.9%	**0.132**	-32.3%
sed	0.1	0.041	0.031	-24.3%	0.025	-39.0%	0.025	-38.4%	**0.019**	-52.7%
	0.5	0.067	0.059	-11.4%	0.052	-21.8%	0.051	-23.2%	**0.037**	-44.0%
	0.9	0.168	0.182	+8.2%	0.159	-5.6%	0.149	-11.4%	**0.138**	-18.3%

Table VII. C_d performance results (lower is better)

Failure detection performance can be explained analytically by modeling the number of tests that need to be executed until the first failure occurs, N_{ff}, by a geometric distribution, $X \sim Geo(p)$, whose expected value is $E[X] = p^{-1}$. The objective of FEP and ADDST is to choose tests with maximum failure probability, ideally $p = 1.0$. Therefore approximately 1 test is needed on average ($N_{ff} \approx 1$). On the other hand, IG tends to select test cases that balance the probability of passing and failing, ideally $p = 0.5$, and therefore on average needs 2 tests ($N_{ff} = 2$).

It can be clearly seen how ADDST, and especially FEP, are the best performing techniques. This is expected, as the assumptions under which FEP and ADDST were devised, i.e., maximization of failure probability, are the most favorable. The failure detection performance of IG is lower than FEP and ADDST and slightly higher than ART and with a lower dispersion. ART has a better performance than random and a lower dispersion, consistent with [4]. Again, this is caused by the coverage distance kept between each test.

The APFD performance of IG for the SIR programs is similar to FEP. This is again caused by the shape and sparsity of the matrices, and will be discussed in Section 7.4.

In summary, when considering early failure detection as the main goal, FEP and ADDST are more suitable for this purpose than IG. This result is consistent with previous literature [6]. However, given the small difference with IG, and the fact that for the large programs and very high FNR scenarios IG behaves essentially like FEP, IG is a suitable alternative. Furthermore, in the cases where FEP and ADDST clearly outperform IG, the trade-off of APFD for C_d is compensated greatly, as we will see in the next section.

DOI: 10.1002/spe

Program	h	RND	ART		ADDST		FEP		IG	
print_tokens	0.1	0.941	0.965	+2.6%	**0.998**	+6.0%	**0.998**	+6.1%	0.984	+4.6%
	0.5	0.904	0.938	+3.8%	0.978	+8.2%	**0.989**	+9.4%	0.959	+6.0%
	0.9	0.673	0.714	+6.2%	0.813	+20.8%	**0.874**	+30.0%	0.789	+17.4%
print_tokens2	0.1	0.972	0.983	+1.1%	**0.997**	+2.6%	**0.998**	+2.6%	0.987	+1.5%
	0.5	0.943	0.962	+2.0%	0.972	+3.1%	**0.987**	+4.7%	0.963	+2.1%
	0.9	0.644	0.663	+3.1%	0.648	+0.6%	**0.738**	+14.6%	**0.708**	+10.0%
replace	0.1	0.946	0.955	+1.0%	**0.991**	+4.8%	**0.993**	+5.0%	**0.983**	+4.0%
	0.5	0.912	0.920	+0.9%	**0.963**	+5.6%	**0.986**	+8.1%	0.961	+5.4%
	0.9	0.658	0.651	-1.0%	0.639	-2.9%	**0.783**	+19.1%	0.706	+7.3%
schedule	0.1	0.989	0.993	+0.5%	**0.998**	+1.0%	**0.999**	+1.1%	0.987	-0.2%
	0.5	0.976	0.978	+0.2%	0.981	+0.5%	**0.990**	+1.4%	0.962	-1.5%
	0.9	0.790	0.767	-2.9%	0.752	-4.8%	**0.848**	+7.3%	0.745	-5.7%
schedule2	0.1	0.992	**0.996**	+0.4%	**0.999**	+0.7%	**0.999**	+0.7%	0.981	-1.1%
	0.5	0.980	0.980	-0.1%	**0.988**	+0.8%	**0.989**	+0.9%	0.957	-2.4%
	0.9	**0.771**	0.757	-1.8%	**0.784**	+1.7%	**0.790**	+2.5%	0.702	-8.9%
tcas	0.1	0.950	0.973	+2.5%	**0.995**	+4.8%	**0.996**	+4.8%	**0.990**	+4.2%
	0.5	0.923	0.945	+2.4%	**0.978**	+5.9%	**0.977**	+5.9%	0.959	+3.9%
	0.9	0.692	0.704	+1.6%	**0.738**	+6.7%	**0.760**	+9.8%	0.724	+4.6%
tot_info	0.1	0.985	0.988	+0.3%	**0.995**	+1.0%	**0.995**	+1.0%	0.981	-0.4%
	0.5	0.958	0.958	+0.0%	**0.979**	+2.3%	**0.981**	+2.4%	0.951	-0.7%
	0.9	0.686	0.657	-4.2%	**0.752**	+9.7%	**0.763**	+11.3%	0.689	+0.5%
space	0.1	0.768	0.814	+6.0%	**0.942**	+22.6%	**0.944**	+22.9%	**0.934**	+21.6%
	0.5	0.717	0.759	+5.9%	0.797	+11.1%	**0.864**	+20.6%	**0.848**	+18.3%
	0.9	0.486	0.485	-0.2%	0.504	+3.7%	**0.548**	+12.6%	**0.548**	+12.6%
grep	0.1	0.865	0.894	+3.4%	**0.954**	+10.4%	**0.958**	+10.8%	**0.948**	+9.6%
	0.5	0.812	0.830	+2.3%	**0.896**	+10.3%	**0.915**	+12.6%	**0.905**	+11.5%
	0.9	0.515	0.450	-12.6%	0.498	-3.4%	**0.616**	+19.6%	**0.612**	+18.8%
gzip	0.1	0.733	0.847	+15.5%	**0.951**	+29.8%	**0.951**	+29.9%	**0.947**	+29.2%
	0.5	0.658	0.751	+14.0%	**0.784**	+19.0%	**0.796**	+21.0%	**0.798**	+21.1%
	0.9	0.392	0.454	+15.9%	0.459	+17.3%	**0.519**	+32.4%	**0.514**	+31.2%
sed	0.1	0.876	0.938	+7.0%	**0.989**	+12.8%	**0.989**	+12.8%	**0.982**	+12.1%
	0.5	0.797	0.874	+9.7%	0.926	+16.2%	**0.947**	+18.8%	**0.938**	+17.6%
	0.9	0.508	0.515	+1.4%	0.569	+12.1%	**0.624**	+22.7%	**0.608**	+19.6%

Table VIII. Failure detection performance (APFD) for the Siemens and SIR programs (higher is better)

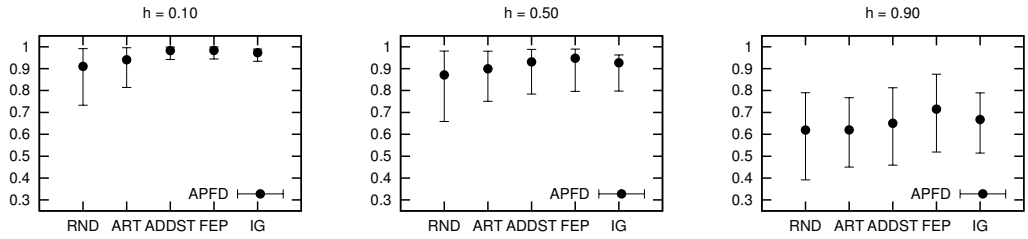

Figure 6. Failure detection performance (average, minimum and maximum APFD)

7.3. Question 3: Best Combined Performance

Table IX shows the average combined costs according to C_{opt} per program, and the improvement with respect to RND. The value marked in **bold** indicates the best performing technique or techniques with a 95% confidence according to the Bonferroni mean separation test.

In our case, considering the QA cost as a whole, the number of tests required to reveal the presence of a fault as measured by APFD 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. This is reflected in the fact that ADDST and FEP have an increased cost over RND for the Siemens programs of up to a 100%. Although faults are detected very early, the diagnostic information gain is very limited given the extremely high coverage of the Siemens suites. Despite the fact that IG needs more tests to detect the presence of a fault, this is more than compensated by

Prepared using speauth.cls
DOI: 10.1002/spe

Program	h	RND	ART		ADDST		FEP		IG	
print_tokens	0.10	39.984	33.493	-16.2%	71.799	+79.6%	80.482	+101.3%	**23.694**	-40.7%
print_tokens	0.50	55.751	51.418	-7.8%	92.600	+66.1%	100.279	+79.9%	**35.649**	-36.1%
print_tokens	0.90	106.126	108.209	+2.0%	143.941	+35.6%	122.225	+15.2%	**81.155**	-23.5%
print_tokens2	0.10	55.738	50.693	-9.1%	70.732	+26.9%	84.858	+52.2%	**39.435**	-29.2%
print_tokens2	0.50	72.831	66.606	-8.5%	90.113	+23.7%	113.134	+55.3%	**51.902**	-28.7%
print_tokens2	0.90	110.392	109.328	-1.0%	144.369	+30.8%	127.497	+15.5%	**93.334**	-15.5%
replace	0.10	34.212	30.466	-10.9%	41.490	+21.3%	39.000	+14.0%	**18.303**	-46.5%
replace	0.50	47.322	51.855	+9.6%	61.667	+30.3%	58.399	+23.4%	**32.469**	-31.4%
replace	0.90	103.780	113.413	+9.3%	125.410	+20.8%	115.732	+11.5%	**91.477**	-11.9%
schedule	0.10	36.843	35.652	-3.2%	35.252	-4.3%	36.192	-1.8%	**16.817**	-54.4%
schedule	0.50	51.050	50.166	-1.7%	56.735	+11.1%	69.870	+36.9%	**28.278**	-44.6%
schedule	0.90	103.609	100.759	-2.8%	104.575	+0.9%	109.436	+5.6%	**81.226**	-21.6%
schedule2	0.10	43.867	39.972	-8.9%	55.025	+25.4%	57.051	+30.1%	**20.083**	-54.2%
schedule2	0.50	61.612	57.682	-6.4%	77.072	+25.1%	86.177	+39.9%	**32.087**	-47.9%
schedule2	0.90	89.861	85.545	-4.8%	92.685	+3.1%	105.775	+17.7%	**73.278**	-18.5%
tcas	0.10	27.843	26.186	-6.0%	36.908	+32.6%	36.735	+31.9%	**23.850**	-14.3%
tcas	0.50	36.251	35.652	-1.7%	48.520	+33.8%	49.482	+36.5%	**31.085**	-14.3%
tcas	0.90	58.005	58.095	+0.2%	63.191	+8.9%	61.558	+6.1%	**55.745**	-3.9%
tot_info	0.10	37.838	35.975	-4.9%	36.270	-4.1%	42.405	+12.1%	**22.154**	-41.4%
tot_info	0.50	61.647	52.176	-15.4%	51.907	-15.8%	62.925	+2.1%	**33.676**	-45.4%
tot_info	0.90	100.772	103.413	+2.6%	90.214	-10.5%	103.082	+2.3%	**75.929**	-24.7%
space	0.10	201.949	165.473	-18.1%	112.843	-44.1%	106.193	-47.4%	**83.717**	-58.5%
space	0.50	289.821	225.540	-22.2%	182.047	-37.2%	174.910	-39.6%	**158.270**	-45.4%
space	0.90	875.710	861.108	-1.7%	1042.716	+19.1%	**764.285**	-12.7%	**723.306**	-17.4%
grep	0.10	244.828	201.240	-17.8%	161.051	-34.2%	148.324	-39.4%	**118.051**	-51.8%
grep	0.50	339.495	270.074	-20.4%	232.842	-31.4%	207.171	-39.0%	**168.296**	-50.4%
grep	0.90	1222.752	1515.065	+23.9%	1577.514	+29.0%	**952.354**	-22.1%	**867.989**	-29.0%
gzip	0.10	364.962	288.359	-21.0%	248.951	-31.8%	243.604	-33.3%	**218.291**	-40.2%
gzip	0.50	540.282	427.220	-20.9%	**378.113**	-30.0%	**368.421**	-31.8%	**373.046**	-31.0%
gzip	0.90	1297.090	980.563	-24.4%	985.323	-24.0%	**885.160**	-31.8%	**867.261**	-33.1%
sed	0.10	193.300	147.613	-23.6%	130.750	-32.4%	130.025	-32.7%	**120.188**	-37.8%
sed	0.50	303.200	262.788	-13.3%	243.287	-19.8%	**230.550**	-24.0%	**220.688**	-27.2%
sed	0.90	879.475	910.825	+3.6%	811.788	-7.7%	791.400	-10.0%	**694.500**	-21.0%

Table IX. Average combined performance C_{opt}

the improved quality of diagnostic information, which at the optimal point can be over a 60% better than RND.

As for Question 1, for the SIR programs we observe that the FEP and ADDST do not cause an increased cost but a significant reduction with respect to RND. The reason is again caused by the shape and sparsity of the matrices, and will be discussed in the next section.

7.4. Theoretical Analysis

Our experiments have revealed there is a difference between the behavior of the techniques for the Siemens programs, and for the SIR programs. In this section we detail the reason behind this difference.

To explain why for the SIR programs the performance of FEP and ADDST is close to IG, we must consider the relationship between failure probability, $\Pr(o_i)$ (which is directly correlated to coverage) and information gain. The plot in Figure 7 shows all the possible spectrum of IG values. However, not all of the possibilities are available depending on the shape of the matrix.

Siemens programs are small, and have very few functionalities and simple inputs. This makes generating test cases that simultaneously cover most of the functionality (hence, the code) a relatively easy task. This results in very dense test matrices, with very high failure probabilities even if the FNR is high.

On the other hand, SIR programs are large and contain many different functionalities and complex inputs. When creating test cases, testers choose inputs that test each of these functionalities individually. This is done because it makes this (usually manual) task simpler, and because in case

Prepared using speauth.cls
DOI: 10.1002/spe

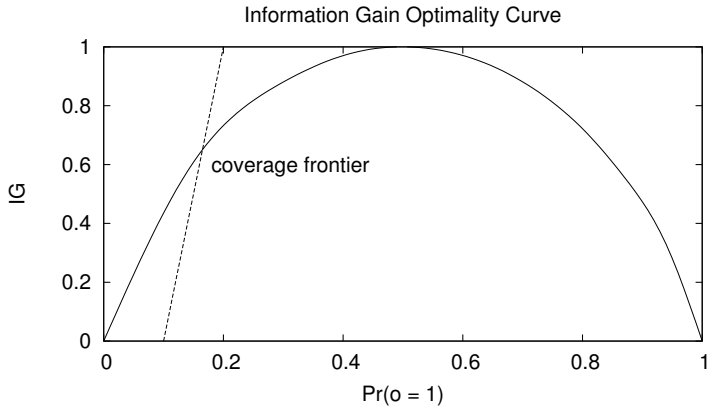

Figure 7. Information Gain as a function of failure probability

of failure, it can be directly related to the faulty functionality as it is being tested in isolation. The resulting test matrices are close to diagonal.

In the case of the dense test matrices of the Siemens programs or the SIR programs for low FNR, many of the tests will be located *after* the optimal IG value. In fact, many of them will have such a high failure probability that the IG they provide will be almost zero. In their effort to maximize failure probability, FEP and ADDST choose tests that fall far from the optimal point, severely impacting their performance.

However, in the case of the SIR programs combined with a very high FNR ($h = 0.9$), most of the tests will be located *before* the optimal IG value. This creates a *coverage frontier*, with the consequence that no test can provide an IG value close or past the theoretical optimum. This prevents FEP and ADDST from choosing non-optimal tests. In fact, it will cause FEP and ADDST to choose tests close to optimal tests, since the best test will always be the test with the highest failure probability (hence closer to the optimal) equaling the performance of IG in some cases. This is also the reason why in Question 2 the APFD performance of IG and FEP are so similar for the SIR programs.

7.5. Threats to Validity

The validity of our experiments is threatened by the subjects used in our study. One must consider whether the test matrices A are representative of real programs, in both size and test composition. Even though the Siemens set is widely used in literature, there are doubts to whether the programs and test suites represent reality. For this reason we have included the larger programs of the SIR repository. Their larger sizes and realistic test matrices strengthen the validity of our results.

Simulation of faults has enabled us to obtain a greater sample of faults per program. Without additional faults, our experiments would not have enough statistical significance. However, it also affects the validity of our results. Firstly, one must consider whether the distribution of faults (p_j) and false negative rates (h_j) used in our experiments is valid. Secondly, as we used the same p_j and h_j as input for the prioritization algorithm, our results show the performance of diagnostic prioritization when it has the best information available. This is something to take into account for its practical application. A validation involving a large sample of real or mutation faults would strengthen the validity of our experiments by showing the performance of IG when faced with real faults and imperfect FNR estimations. In the next section we will comment on how to obtain more realistic values on practical settings, which can be used to obtain experimental results with stronger validity, and what is the effect of errors in these estimations.

In previous literature, interface, type and variable declarations are considered in the component ranking and the C_d metric, although their likelihoods are in most cases 0 because of the limitations on the code instrumentation, which causes $\forall i\, a_{ij} = 0$. This is especially true in Spectrum-based

DOI: 10.1002/spe

techniques [9, 10, 11]. As they are located at the bottom of the ranking, the number of inspected components (the numerator in the diagnostic effort formula used in Section 7.1) does not change. However, the denominator can be made arbitrarily large by adding static lines of code improving C_d. Although this affects all techniques equally, the relative differences between them can be made arbitrarily small. By not taking into account these columns in A where instrumentation is defective we make the comparison of algorithms fairer.

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 IG. Test cost increases linearly with each test, and the return of that investment is an inverse exponential. To bias C towards test cost-centered techniques (especially ADDST, FEP), the cost of a test would have to be disproportionately larger than the cost of a manual inspection. On the other hand, if α was extremely large, the performance gain would be comparable to the values obtained in Question 1.

8. PRACTICAL CONSIDERATIONS

In this section we will discuss some practical applicability issues of diagnostic test prioritization. Concretely, we will comment on how to obtain the input parameters needed for the algorithm and how to control the overhead caused by online prioritization, since the test choice has to be done during the testing phase for every testing phase.

8.1. Determination of the Input Parameters

Crucial to the applicability of diagnostic prioritization is the determination of all the required inputs. In this section we will comment on how to practically obtain the test matrix A, prior fault probabilities p_k, and fault intermittencies h_k.

For our study, coverage matrices A were obtained by instrumenting each of the programs at statement level with `Zoltar`, a spectrum-based fault localization tool set [28]. However, it must be taken into account that the coverage of a test input can vary between regression cycles. This will not affect diagnostic accuracy as diagnosis is performed *a posteriori* when the updated coverage is already available. However, it can affect the accuracy of prioritization heuristics such as FEP, ART or IG as the coverage of a test case is needed *a priori*. This deviation should be taken into account, by using techniques for estimating the updated coverage of a test input [29]. This situation is generally overlooked in test prioritization literature [2, 4, 6].

Prior fault probabilities, p_k, are typically derived from defect density data. It is safe to assume equal valued priors. The prior distribution has been shown not to be very critical to diagnostic performance [10] as it is adjusted during the diagnosis process by the Bayesian update formula (Equation 2). On the other hand, the FNR h_k is not corrected during diagnosis, and, in fact, the accuracy of the diagnosis itself depends greatly on accurate FNR estimations [10, 30]. False negative rates in software diagnosis are intimately connected with the concept of *testability*, as defined by Voas and Miller [31]: *the degree to which software reveals faults during testing*. Testability can be modeled by the so-called propagation, infection, execution approach (PIE) [32]. For the estimation of the FNR value h_k, a variation of the PIE testability can be used, following the approach in [6]. The cost of this study is expensive in terms of execution time. Fortunately however, the high cost of FNR analysis is amortized over many code commit and regression test cycles. Other approaches based on static analysis, such as [33, 34, 35, 36, 37] have been suggested, however their practical application for diagnosis or prioritization has not been demonstrated.

8.2. Sensitivity to Estimation Errors

The diagnostic precision of Bayesian diagnosis and diagnostic prioritization depends on how well the h_k values are estimated (similar to the estimation problem in FEP). The average number of

DOI: 10.1002/spe

		$h = 0.10$			$h = 0.50$			$h = 0.90$		
Program	σ	RND	IG		RND	IG		RND	IG	
print_tokens	0.10	0.037	0.022	-40.9%	0.066	0.041	-38.0%	0.181	0.173	-4.1%
	0.25	0.040	0.024	-38.6%	0.074	0.049	-33.4%	0.213	0.213	-0.0%
	0.50	0.043	0.032	-25.1%	0.090	0.071	-21.7%	0.220	0.254	**+15.6%**
print_tokens2	0.10	0.049	0.036	-27.1%	0.084	0.063	-25.4%	0.206	0.205	-0.3%
	0.25	0.052	0.040	-22.4%	0.102	0.072	-29.6%	0.277	0.271	-1.9%
	0.50	0.066	0.049	-25.5%	0.124	0.105	-15.1%	0.301	0.310	**+2.9%**
replace	0.10	0.028	0.018	-35.5%	0.056	0.035	-37.9%	0.202	0.184	-9.3%
	0.25	0.029	0.018	-37.4%	0.059	0.038	-34.7%	0.228	0.208	-8.8%
	0.50	0.033	0.021	-37.3%	0.077	0.051	-33.5%	0.232	0.222	-4.5%
schedule	0.10	0.045	0.018	-59.6%	0.079	0.035	-56.2%	0.261	0.194	-25.7%
	0.25	0.047	0.019	-59.8%	0.113	0.042	-62.8%	0.330	0.250	-24.3%
	0.50	0.048	0.020	-58.4%	0.146	0.074	-49.4%	0.310	0.245	-20.9%
schedule2	0.10	0.074	0.028	-62.3%	0.125	0.058	-53.4%	0.275	0.216	-21.3%
	0.25	0.072	0.028	-60.9%	0.155	0.066	-57.4%	0.305	0.242	-20.4%
	0.50	0.092	0.033	-63.7%	0.171	0.085	-50.5%	0.319	0.269	-15.7%
tcas	0.10	0.076	0.065	-13.7%	0.124	0.112	-9.1%	0.223	0.223	**+0.1%**
	0.25	0.078	0.072	-8.3%	0.138	0.133	-4.2%	0.264	0.284	**+7.6%**
	0.50	0.094	0.090	-4.4%	0.155	0.149	-4.0%	0.270	0.273	**+1.3%**
tot_info	0.10	0.045	0.027	-39.1%	0.088	0.051	-42.3%	0.248	0.209	-15.5%
	0.25	0.045	0.029	-36.5%	0.099	0.058	-41.5%	0.279	0.239	-14.2%
	0.50	0.048	0.031	-35.2%	0.138	0.096	-30.6%	0.262	0.256	-2.4%
space	0.10	0.043	0.016	-62.1%	0.067	0.035	-47.2%	0.194	0.213	**+9.7%**
	0.25	0.044	0.017	-62.0%	0.071	0.039	-45.0%	0.210	0.253	**+20.6%**
	0.50	0.044	0.020	-55.5%	0.084	0.062	-26.9%	0.230	0.261	**+13.8%**
grep	0.10	0.036	0.017	-54.1%	0.055	0.029	-47.0%	0.227	0.183	-19.7%
	0.25	0.034	0.016	-52.1%	0.063	0.032	-48.5%	0.266	0.243	-8.9%
	0.50	0.035	0.019	-46.8%	0.076	0.050	-34.2%	0.271	0.259	-4.3%
gzip	0.10	0.076	0.036	-53.0%	0.106	0.064	-39.8%	0.248	0.202	-18.4%
	0.25	0.079	0.036	-53.9%	0.111	0.067	-39.6%	0.266	0.218	-18.0%
	0.50	0.184	0.095	-48.6%	0.120	0.083	-31.4%	0.281	0.234	-16.8%
sed	0.10	0.045	0.023	-49.4%	0.076	0.041	-46.5%	0.219	0.200	-8.6%
	0.25	0.045	0.022	-51.0%	0.081	0.045	-44.0%	0.251	0.236	-6.1%
	0.50	0.151	0.087	-42.6%	0.089	0.053	-40.2%	0.266	0.271	**+1.8%**

Table X. Sensitivity of IG to errors in the estimation of FNR

tests needed to exonerate a healthy component is $1/(1-h_k)$, which starts affecting diagnostic performance when h_k is close to 1, since small errors in h_k translate into very large errors in the number of tests. Since our simulation experiments were performed using correct h_k values, the validity of our results is threatened since the practical applicability of the method is not guaranteed.

We performed a simple experiment using the matrix of all our evaluation programs, by adding random noise to h_k with a deviation of $\sigma = 10, 25, 50\%$, averaged over 500 runs. Three different average FNR scenarios were used: low $\bar{h} = 0.1$, high $\bar{h} = 0.5$, and very high $\bar{h} = 0.9$.

Table X shows the results of our sensitivity study. Values in **bold** indicate the cases where IG is affected up to the point that it performs worse than RND.

The results show that for low and intermediate FNR, IG is not affected to a great extent, although it shows an increasing trend with noise. On the other hand, the choices made by IG can be severely impacted by even small errors in the estimation of h_k if the real FNR is very high. This is can be seen in the extremely poor performance of IG for `tcas` and `space`. The construction of the matrix has, again, an influence. The robustness of IG to error is higher in some cases, such as `schedule2` shows.

8.3. Reduction of the Prioritization Overhead

As on-line prioritization has to be recalculated for each test cycle, the time overhead imposed by the algorithm is a critical success factor in this approach to QA.

For the coverage matrix of `print_tokens` ($N = 4130 \times M = 563$), selecting a test takes in our (non-optimized) experimental platform approximately 1s of CPU time. For comparison, ART takes

DOI: 10.1002/spe

an average of 20ms per test. This overhead can be partially compensated if 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.

Another simple approach for reducing complexity is to collapse tests with identical coverage into a single one. Even though this approach provides a degree of reduction (in the test matrices used in Section 6 the matrix is reduced to approximately of a 50% of its rows), this only represents halving the time required per test case.

Finally, just as ART uses random *sampling* of the test suite *per step*, a similar approach can be used for IG. The reason is that there is a large redundancy in the test matrix A due to an excessive number of tests providing almost no information gain. A random sample of tests reduces this redundancy while maintaining performance as good as if no sampling was performed. The reduction in test selection time achievable by random sampling depends on the sampling method used, but is of at least an order of magnitude or more. We performed a trial run where we sampled 100 tests per step (from tests matrices of up to 5,000 tests), yielding a 50-fold reduction of the time required per test case, reducing the time it takes to select a test using IG to a mere 20ms, similar to ART, while IG's diagnostic performance was not affected.

9. RELATED WORK

The influence of test-suite extension, reduction, modification, and prioritization on fault detection and diagnosis has received considerable attention.

Hao *et al.* [38] study the effect of reduction of redundant tests on the Tarantula fault detection algorithm [11], concluding that reduction may actually have a beneficial effect on diagnosis. However, their conclusions partially contradict those of Yu *et al.* [16] for a larger sample of programs, who conclude that test-suite reduction, especially coverage-based reduction, has a negative influence on the quality of the diagnosis. In particular, Jiang *et al.* [15] show how some prioritization techniques are worse than random sequencing and that those that are better, do not provide a significant improvement.

In contrast to the works cited above, which use coverage-based heuristics to reduce the size of the test suite, diagnostic prioritization employs a heuristic to select the best test case to optimize diagnostic accuracy.

Baudry *et al.* [39] propose enhancing test suites by adding new tests that increase the number of *dynamic basic blocks* (i.e., sets of components that are always covered together), thus reducing diagnostic uncertainty. The additional tests created by this technique are subject to prioritization as well, and thus there is no guarantee they will be ordered appropriately, unless diagnostic prioritization is used.

Test case prioritization is a mature and active area of research whose most common goal is to increase failure detection rate. Harrold *et al.* [3] and Wong *et al.* [8] proposed to use reduction and prioritization as a means of controlling the size of regression test suites. The failure detection effectiveness of different coverage-based prioritization techniques was studied by Rothermel *et al.* [2], who also proposed the FEP heuristic [6] that has already been discussed throughout the paper. Cost-cognizant test prioritization techniques have been proposed [40, 7, 41] that take into account that test cases may have variable costs. Elbaum *et al.* propose a set of guidelines [1] to aid in selecting the most cost-effective prioritization technique depending on program attributes. Li *et al.* [5] study different search algorithms for coverage-based prioritization in order to avoid the local minimal of greedy strategies [6]. Jiang *et al.* [4] propose a hybrid random and coverage-based prioritization technique (ART), which has already been discussed throughout this paper. Kim and Baik propose a fault-aware test case prioritization (FATCP) [42] that combines a diagnostic algorithm with coverage-based test prioritization with the goal of producing failures as soon as possible, under the assumption that faulty components that were fixed are less likely to be faulty again in a subsequent cycle. Unlike our diagnostic prioritization algorithm, all traditional prioritization research centers around faster fault detection (APFD). As shown in this article,

DOI: 10.1002/spe

focusing on a faster rate of fault detection may have a very negative impact on fault localization cost.

Automated fault-localization techniques also aim at minimizing diagnostic cost when failures occur during the testing phase. Statistical approaches include the Tarantula tool by Jones *et al.* [11], Ochiai by Abreu *et al.* [9], the Nearest Neighbor technique by Renieris *et al.* [22], Sober by Liu *et al.* [21], CBI by Liblit and his colleagues [20], and CrossTab by Wang *et al.* [25]. Approaches to statistical fault localization need not be limited to the statement level, work that considers execution paths and dependences includes [19, 23, 24, 43].

Recently, Abreu *et al.* [10] proposed Barinel, which aimed to combine the best of reasoning and statistical approaches. Although differing in the way they derive the fault ranking, all techniques are based on measuring the coverage information and failure pattern of a program (also known as its spectrum). Nica *et al.* [44] use mutation testing to reduce the amount of diagnosis candidates in the ranking.

A preliminary version of our work is described in [14], that reveals the high potential of (IG-based) dynamic prioritization. However, in that work we assumed permanent failures, which is completely unrealistic for real software systems. Diagnostic prioritization is often applied to hardware systems, where it is known as Sequential Diagnosis, where tests can produce false negatives (intermittent faults) [12]. Typically, only one fault is assumed present in the system. Sequential diagnosis can be enhanced by the use of models and test case generation (instead of choosing from a fixed test suite), greatly increasing its performance [13, 45]. However, the practical application of these approaches depends on the availability of accurate models, which seldom exist for software systems due to their complexity and constant evolution.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a specific prioritization technique of test cases, dubbed diagnostic prioritization, 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 technique. This comes at the price of a slightly reduced APFD failure detection performance with respect to additional-coverage techniques, although not in all cases. However, when considering the overall combined cost of both testing and manual residual diagnosis, our experiments have shown cost reduction of up to 60% with respect to the next best performing technique.

In future work we will extend the validation of our approach to larger systems with multiple 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. We will also consider different strategies to overcome errors in the estimation of the input parameters (h_k, p_k) of IG, since our sensitivity study showed that IG can be affected negatively in some cases. Additionally, we will consider the fact that different tests may have different costs, and propose new heuristics to address this situation.

ACKNOWLEDGEMENTS

The authors wish to thank 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. We would also like to thank our anonymous reviewers for their helpful comments and insights, which greatly improved the quality of this paper.

REFERENCES

1. Elbaum S, Rothermel G, Kanduri S, Malishevsky AG. Selecting a cost-effective test case prioritization technique. *Software Quality Control* 2004; **12**(3):185–210, doi:http://dx.doi.org/10.1023/B:SQJO.0000034708.84524.22.
2. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28**(2):159–182, doi:http://dx.doi.org/10.1109/32.988497.

3. Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology* 1993; **2**(3):270–285, doi:http://doi.acm.org/10.1145/152388.152391.
4. Jiang B, Zhang Z, Chan WK, Tse TH. Adaptive random test case prioritization. *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2009; 233–244, doi:http://dx.doi.org/10.1109/ASE.2009.77.
5. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 2007; **33**(4):225–237, doi:http://dx.doi.org/10.1109/TSE.2007.38.
6. Rothermel G, Untch RJ, Chu C. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948, doi:http://dx.doi.org/10.1109/32.962562.
7. Smith AM, Kapfhammer GM. An empirical study of incorporating cost into test suite reduction and prioritization. *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, ACM: New York, NY, USA, 2009; 461–467, doi:http://doi.acm.org/10.1145/1529282.1529382.
8. Wong WE, Horgan JR, London S, Bellcore HA. A study of effective regression testing in practice. *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, IEEE Computer Society: Washington, DC, USA, 1997; 264.
9. Abreu R, Zoeteweij P, van Gemund AJC. On the accuracy of spectrum-based fault localization. *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, IEEE Computer Society: Washington, DC, USA, 2007; 89–98.
10. Abreu R, Zoeteweij P, Gemund AJCv. Spectrum-based multiple fault localization. *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2009; 88–99, doi:http://dx.doi.org/10.1109/ASE.2009.25.
11. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, ACM: New York, NY, USA, 2002; 467–477, doi: http://doi.acm.org/10.1145/581339.581397.
12. Raghavan V, Shakeri M, Pattipati K. Test sequencing algorithms with unreliable tests. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* jul 1999; **29**(4):347 –357, doi: 10.1109/3468.769753.
13. Feldman A, Provan G, Van Gemund A. Fractal: efficient fault isolation using active testing. *IJCAI'09: Proceedings of the 21st international joint conference on Artifical intelligence*, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2009; 778–784.
14. Gonzalez-Sanchez A, Piel E, Gross HG, van Gemund AJ. Prioritizing tests for software fault localization. IEEE Computer Society: Los Alamitos, CA, USA, 2010; 42–51, doi: http://doi.ieeecomputersociety.org/10.1109/QSIC.2010.28.
15. Jiang B, Zhang Z, Tse TH, Chen TY. How well do test case prioritization techniques support statistical fault localization. *COMPSAC '09: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference*, IEEE Computer Society: Washington, DC, USA, 2009; 99–106, doi: http://dx.doi.org/10.1109/COMPSAC.2009.23.
16. Yu Y, Jones JA, Harrold MJ. An empirical study of the effects of test-suite reduction on fault localization. *ICSE '08: Proceedings of the 30th international conference on Software engineering*, ACM: New York, NY, USA, 2008; 201–210, doi:http://doi.acm.org/10.1145/1368088.1368116.
17. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *ICSE '94: Proceedings of the 16th international conference on Software engineering*, IEEE Computer Society Press: Los Alamitos, CA, USA, 1994; 191–200.
18. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.* 2005; **10**(4):405–435, doi:http://dx.doi.org/10.1007/s10664-005-3861-2.
19. Baah GK, Podgurski A, Harrold MJ. The probabilistic program dependence graph and its application to fault diagnosis. *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, ACM: New York, NY, USA, 2008; 189–200, doi:http://doi.acm.org/10.1145/1390630.1390654.
20. Liblit B. Cooperative debugging with five hundred million test cases. *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, ACM: New York, NY, USA, 2008; 119–120, doi: http://doi.acm.org/10.1145/1390630.1390632.
21. Liu C, Yan X, Fei L, Han J, Midkiff SP. Sober: statistical model-based bug localization. *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM: New York, NY, USA, 2005; 286–295, doi:http://doi.acm.org/10.1145/1081706.1081753.
22. Renieris M, Reiss SP. Fault localization with nearest neighbor queries. *Automated Software Engineering, International Conference on* 2003; **0**:30, doi:http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240292.
23. Santelices R, Jones JA, Yu Y, Harrold MJ. Lightweight fault-localization using multiple coverage types. *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2009; 56–66, doi:http://dx.doi.org/10.1109/ICSE.2009.5070508.
24. Wang X, Cheung SC, Chan WK, Zhang Z. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2009; 45–55, doi: http://dx.doi.org/10.1109/ICSE.2009.5070507.
25. Wong E, Wei T, Qi Y, Zhao L. A crosstab-based statistical method for effective fault localization. *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, IEEE Computer Society: Washington, DC, USA, 2008; 42–51, doi:http://dx.doi.org/10.1109/ICST.2008.65.
26. Shakeri M, Raghavan V, Pattipati K, Patterson-Hine A. Sequential testing algorithms for multiple fault diagnosis. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* jan 2000; **30**(1):1 –14, doi: 10.1109/3468.823474.

27. Johnson R. An information theory approach to diagnosis. *Symposium on Reliability and Quality Control*, 1960.
28. Janssen T, Abreu R, Gemund AJCv. Zoltar: A toolset for automatic fault localization. *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2009; 662–664, doi:http://dx.doi.org/10.1109/ASE.2009.27.
29. Chittimalli PK, Harrold MJ. Recomputing coverage information to assist regression testing. *IEEE Transactions in Software Engineering* 2009; **35**(4):452–469, doi:http://dx.doi.org/10.1109/TSE.2009.4.
30. De Kleer J. Diagnosing multiple persistent and intermittent faults. *IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence*, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2009; 733–738.
31. Voas JM, Miller KW. Software testability: The new verification. *IEEE Software* 1995; **12**(3):17–28, doi: http://dx.doi.org/10.1109/52.382180.
32. Voas JM. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering* 1992; **18**(8):717–727, doi:http://dx.doi.org/10.1109/32.153381.
33. Freedman RS. Testability of software components. *IEEE Transactions in Software Engineering* 1991; **17**(6):553–564, doi:http://dx.doi.org/10.1109/32.87281.
34. Offutt AJ, Hayes JH. A semantic model of program faults. *SIGSOFT Software Engineering Notes* 1996; **21**(3):195–200, doi:http://doi.acm.org/10.1145/226295.226317.
35. Voas JM, Miller KW. Semantic metrics for software testability. *Journal of Systems and Software* 1993; **20**(3):207–216, doi:http://dx.doi.org/10.1016/0164-1212(93)90064-5.
36. Woodward MR, Al-Khanjari ZA. Testability, fault size and the domain-to-range ratio: An eternal triangle. *SIGSOFT Software Engineering Notes* 2000; **25**(5):168–172, doi:http://doi.acm.org/10.1145/347636.349016.
37. Zhao L. A new approach for software testability analysis. *ICSE '06*, ACM: New York, NY, USA, 2006; 985–988, doi:http://doi.acm.org/10.1145/1134285.1134469.
38. Hao D, Zhang L, Zhong H, Mei H, Sun J. Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study. *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, IEEE Computer Society: Washington, DC, USA, 2005; 683–686, doi: http://dx.doi.org/10.1109/ICSM.2005.43.
39. Baudry B, Fleurey F, Le Traon Y. Improving test suites for efficient fault localization. *ICSE '06: Proceedings of the 28th international conference on Software engineering*, ACM: New York, NY, USA, 2006; 82–91, doi: http://doi.acm.org/10.1145/1134285.1134299.
40. Elbaum S, Malishevsky A, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2001; 329–338.
41. Zhang L, Hou SS, Guo C, Xie T, Mei H. Time-aware test-case prioritization using integer linear programming. *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM: New York, NY, USA, 2009; 213–224, doi:http://doi.acm.org/10.1145/1572272.1572297.
42. Kim S, Baik J. An effective fault aware test case prioritization by incorporating a fault localization technique. *ESEM '10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM: New York, NY, USA, 2010; 1–10, doi:http://doi.acm.org/10.1145/1852786.1852793.
43. Chilimbi TM, Liblit B, Mehra K, Nori AV, Vaswani K. Holmes: Effective statistical debugging via efficient path profiling. *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society: Washington, DC, USA, 2009; 34–44, doi:http://dx.doi.org/10.1109/ICSE.2009.5070506.
44. Nica M, Nica S, Wotawa F. Does testing help to reduce the number of potentially faulty statements in debugging? *Testing Practice and Research Techniques*, *Lecture Notes in Computer Science*, vol. 6303, Bottaci L, Fraser G (eds.). Springer Berlin / Heidelberg, 2010; 88–103, doi:http://dx.doi.org/10.1007/978-3-642-15585-7.
45. Pietersma J, van Gemund A, Bos A. A model-based approach to sequential fault diagnosis - a best student paper award winner at ieee autotestcon 2005. *Instrumentation Measurement Magazine, IEEE* april 2007; **10**(2):46 –52, doi:10.1109/MIM.2007.364961.