Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-Based Fault Locators

Friedrich Steimann, Marcus Frenkel
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Hagen, Germany
steimann@acm.org, marcus.frenkel@feu.de

Rui Abreu
Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
rui@computer.org

ABSTRACT

Resuming past work on coverage-based fault localization, we find that empirical assessments of its accuracy are subject to so many imponderables that they are of limited value. To improve on this situation, we have compiled a comprehensive list of threats to be considered when attempting such assessments in the future. In addition, we propose the establishment of theoretical lower and upper bounds of fault localization accuracy that depend on properties of the subject programs (including their test suites) only. We make a suggestion for a lower bound and show that well-known fault locators do not uniformly perform better.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids*

General Terms: Measurement, Performance, Experimentation

Keywords: Fault localization, Threats to validity

1. INTRODUCTION

Ever since the first publications on coverage-based fault localization [8, 9, 15, 22], an empirical evaluation of the diagnostic accuracy, or of the (theoretical) effort of localizing a fault, has been a *conditio sine qua non*. Although accuracy measures, which are usually based on rankings of program elements to be inspected in search for a fault, may be questioned as indicators of the *usefulness* of fault locators [21], they are certainly necessary for evaluating their *performance*, in a theoretical setting at least. Since utility depends on performance (who would deny the value of a fault locator with perfect accuracy?), we assume that researchers will continue to try and improve the accuracy of fault localization, unless a theoretical upper bound for the accuracy is found which makes fault localization unappealing for practical application. Since such a bound is currently not in sight, we will likely see empirical assessments of fault localization accuracies in the scientific literature for some time to come.

In this paper, we summarize the problems and pitfalls that we have encountered during our own attempts to get the evaluation of fault locators right. In the organization of our presentation, we do not separate between our own observations and those made by others (to which we give references whenever we are aware of them); where observations coincide, the results presented here should be seen as independent support for, or reproduction of, the results obtained by others. A contribution that, as we believe, stands out alone however is the establishment of an absolute lower bound of the accuracy of fault locators, for which we show that some fault locators established in the literature fail to perform better in a significant number of cases. With this, we hope to set a useful frame of reference for the ad hocery that we are seeing in the evaluation of fault locators until this day.

The remainder of this paper is organized as follows. Section 2 sets the stage by introducing the terms and definitions on which our work rests. Section 3 describes the setup for the experiments we have conducted to identify and quantify the threats to validity and value that are the subject of this paper, and which are described in Section 4 and 5, resp. Section 6 briefly summarizes related work.

NB: In this paper, the adequacy of any particular fault locator, or the superiority of one fault locator over another, is not the topic — our objective is to show how performance is generally susceptible to the design of the evaluations assessing it.

2. TERMS AND DEFINITIONS

We introduce and define a number of terms as we will use them in this work. Other work may define or use the same terms differently.

A *test case* is a repeatable execution of some part of a program, supplying it with specified input and matching its observable behaviour (including its output) with the expected behaviour. A *test suite* is a collection of test cases whose result is independent of the order of their execution.

A *proband* (elsewhere called a *subject* [4] or an *object program* [16, 29]) is a program *together with its test suite* that we use for evaluating the performance of coverage-based fault locators.

The *granularity* of fault localization defines the units of the probands to which faults are ascribed. Typical granularities found in the literature are *class* [10] *method* [27], *block* [1, 2], *branch* [24], or *statement* [15]. A *unit under test* (UUT) is a unit of the proband (at the chosen granularity level) that is covered by at least one test case.

A *test coverage matrix* (TCM) is a mathematical abstraction of a test suite and the UUTs it covers. The abstraction has the form of a binary matrix whose rows correspond to the UUTs and whose columns correspond to the test cases. Each element of the matrix indicates whether a test case covers (i.e., executes) a UUT (1) or not (0). Figure 1 shows a generic TCM in which the test cases have been divided (by permutation of columns) into failed and passed ones.[1]

[1] [1–3] uses the transposed matrix, and expresses the division of test cases into passed and failed ones as a separate error vector. In this paper, we stick to the presentation used by [15–18].

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07...$15.00
http://dx.doi.org/10.1145/2483760.2483767

UUTs	Test Cases					
	failed			*passed*		
	t_1	…	t_m	t_{m+1}	…	t_a
u_1	$x_{1,1}$	…	$x_{1,m}$	$x_{1,m+1}$	…	$x_{1,a}$
…	…	…	…	…	…	…
u_n	$x_{n,1}$	…	$x_{n,m}$	$x_{n,m+1}$	…	$x_{n,a}$

Figure 1: General structure of a test coverage matrix (TCM)

A *failed test coverage matrix* (FCM) reduces a TCM to the columns corresponding to failed test cases, and to the rows that are covered by these test cases. Note that neither a TCM nor a FCM has rows or columns whose elements are all 0.

We refer to as *coverage-based fault localization* any activity that points to potential faults in a proband, *solely* using information contained in a TCM or some subset thereof (including a FCM).

2.1 Basic Assumptions of Coverage-Based Fault Localization

The feasibility of coverage-based fault localization rests on a number of assumptions.

1. Faults are intermittent, i.e., faulty UUTs are covered by both failing and passing test cases. (This is sometimes also called coincidental correctness [28].)
2. Every failed test case executes at least one fault whose execution causes the failure.
3. The prior probability distribution of faultiness is unknown (and therefore not exploited by fault localization).
4. For the purpose of measuring the performance of a fault locator, we assume that upon inspection, a programmer always recognizes a faulty UUT as such, and recognizes no UUT considered to be correct in the context of the evaluation — falsely or rightly — as faulty.

While not all of these assumptions may hold in practice, rejecting any of them means questioning the adequacy of coverage-based fault localization, which is not the topic of this paper.

2.2 Fault Localization Modes

In the single-fault case, accuracy of fault locators is usually evaluated by visiting the UUTs in the order suggested by the locator, until the fault is found. When evaluating the accuracy of fault locators in presence of multiple faults, we distinguish two modes:

1. *one-at-a-time mode*: one fault is identified and fixed, and then the fault localization process is repeated (including a re-run of the test suite and a re-computation of the TCM); and
2. *many-at-a-time mode*: as many faults are identified and fixed as are known to be contained in the program, before the fault localization process is repeated.

The question of how many faults are known to be contained in a program turns out to be a tricky one: while the total number of faults will be known in an evaluation setting in which the faults themselves are known, in a practical application of fault localization it will not be (so that an evaluation should not be based on the localization of all existing faults, at least not at a time; cf. Section 4.4.3).

2.3 Accuracy Measures

The diagnostic accuracy of a fault locator is usually measured in two steps:

1. by computing a ranking of UUTs from a TCM and
2. by determining the position of the first fault (in the one-at-a time mode) or the position of the first n faults (in the many-at-a-time mode) in that ranking.

Note that for the second step, the faulty UUTs must be known. In the many-at-a-time mode, the TCM may be partitioned into independent smaller ones first so as to localize faults in each partition separately, which may lead to better localization results [3, 17, 18, 26, 27].

The above general evaluation scheme leaves two main degrees of freedom:

1. how to deal with ties, i.e., how to rank UUTs that have received the same score, where one of them is faulty and the subject of localization (called a *critical tie* in [30] or an *ambiguity group* in [13]); and
2. how to convert the rank of the faulty UUT to a measure of accuracy.

While the first works evaluating coverage-based fault locators always ranked the faulty UUT of a critical tie last [16], more recent work has assumed a random ordering, and therefore uses the expected value $(n+1)/2$ for the position of the culprit instead [4, 27, 30][2]. In the multiple fault case, when k faulty UUTs are among a critical tie of n UUTs with identical coefficient, the expected value is given by

$$E(k,n) = \sum_{i=1}^{n-k+1} \binom{n-i}{k-1} \cdot \binom{n}{k}^{-1} \cdot i = \frac{n+1}{k+1}$$

For the assessment of fault locator performance, ranks of faulty UUTs are usually converted to some measure of accuracy. Relatively frequent are measures that put the rank into relation to the program size (in units of the chosen granularity), either as the percentage of units that *need not* be inspected to identify a fault [9, 16, 21, 22], or as the percentage of units that *need* be inspected [2]. However, searching the explanation for failed test cases in code that is not covered by *any* test case, or even not covered by *any failed* test case, not only makes little sense *per se* (cf. Assumption 2 in Section 2.1), but also makes accuracy correlate inversely with overall test coverage, warping all evaluations based on probands whose test coverage is less than 100% (see Section 4.1 for evidence of this).[3] These problems are avoided by using

$$\frac{1}{m}\sum_{j=1}^{m} x_{i,j} \cdot \left(\frac{1}{a-m} \cdot \sum_{j=m+1}^{a} x_{i,j} + \frac{1}{m}\sum_{j=1}^{m} x_{i,j} \right)^{-1} \quad \text{(T)}$$

$$T \cdot \max\left(\frac{1}{a-m} \cdot \sum_{j=m+1}^{a} x_{i,j} \; , \frac{1}{m}\sum_{j=1}^{m} x_{i,j} \right) \quad \text{(T*)}$$

$$\sum_{j=1}^{m} x_{i,j} \cdot \left(m \cdot \sum_{j=1}^{a} x_{i,j} \right)^{-\frac{1}{2}} \quad \text{(O)}$$

Figure 2: Similarity coefficients underlying the fault locators used in this paper. T is suspiciousness of Tarantula [18], T* multiplies suspiciousness with confidence [18], and O is the Ochiai similarity coefficient used by [1–3, 25, 27]. The free variables *a* and *m* are introduced in Figure 1; *i* is the index of a UUT in a TCM.

[2] [4] also showed that assuming the last position introduces a bias, which is avoided by using the expected value.

[3] Note that the coverage of the Siemens test suite, which is often used in evaluations, is 100%.

Table 1: Probands used in this paper

Proband and Version Number	Units† #	UUTs #	TCs #	UUTs/TC #	UUTs/TC σ§
Daikon 4.6.4	14387	1936	157	30	84
Eventbus 1.4	859	338	91	32	22
Jaxen 1.1.5	1689	961	695	186	81
Jester 1.37b	378	152	64	15	12
JExel 1.0.0b13	242	150	335	21	14
JParsec 2.0	1011	893	510	40	39
AC Codec 1.3	265	229	188	7	7
AC Lang 3.0	5373	2075	1666	8	11
Eclipse Draw2d 3.4.2	3231	878	89	82	65
HTML Parser 1.6	1925	785	600	142	93
mean	*2936*	*840*	*440*	*56*	*43*
standard deviation	*4107*	*658*	*463*	*58*	*33*

† here: total number of methods, not counting JUnit test methods

§ standard deviation

absolute measures of accuracy, such as the rank of the faulty UUT (which also better reflects the real effort required [21]).

3. EXPERIMENTAL SETUP

The setup of the experiments used in Sections 4 and 5 to demonstrate that the threats identified in this paper are evident, largely follows the setup discussed in detail in [27]. The following briefly repeats the cornerstones, going into detail only where we differ.

Our probands together with measures of qualities relevant for our endeavour are given in Table 1 (see "Downloads" at the end of this paper for their sources). Units where counted using the Eclipse Java Search facility; this provided more accurate counts than any metrics tool that we tried (whose counts did in fact vary widely, presenting a threat in its own right; see Section 4.5). The list excludes the Apache Commons Collections library used in [27], because it made massive use of abstract tests (see Section 4.3 for the problems this causes), and adds 4 new probands. As has been pointed out in [27], obtaining usable probands is a very time-consuming (and overall rather frustrating) process; the value of a suite of probands such as that of Table 1 should therefore not be underestimated.

To compute TCMs we used the JUnit test suites that came with the probands, and JUnit 4 as the test runner. As entry point for running the test suite, we used the `AllTests()` suite (in case of Jaxen: `JaxenTests()`) if present, and else the test packages. We explicitly made sure that no test case was run twice in the same execution of a test suite.

We used the granularity of methods (rather than the more common statements). Methods have the advantage of providing a natural context of each fault, of being cognitive units directly related to a failed test case (via the call graph), and of being the natural skip/step into units of contemporary debuggers (so that the programmer's choice which UUTs to skip or to debug into can be guided by the output of the fault locator).

The fault locators used are similarity-coefficient based [3]. They use the coefficients shown in Figure 2. The selection of T (Tarantula) and O (Ochiai) was mainly influenced by their popularity (both are frequently used in evaluations; see, e.g., [1–4, 11, 15, 16]); T*, to the best of our knowledge defined and evaluated here for the first time, was added to show that the performance of similarity coefficients can, to a certain extent, be designed (cf. Sections 4.1 and 4.4.1).

From the rankings produced by computing the coefficients, we computed two accuracy measures: *absolute wasted effort*, determined as the number of UUTs that need to be inspected in vain *before* a faulty UUT is found, and *relative wasted effort* (introduced in [3]), computed as the fraction of all methods (as the units of granularity) of a proband that are inspected in vain before the fault is found. For tie breaking, we use the expected value as described in Section 2.3. For reasons given in [21, 27], in particular because using relative wasted effort introduces the bias mentioned in Section 2.3 and demonstrated in Section 4.1, absolute wasted effort will mostly be used; it will be referred to only as *wasted effort*.

Table 2. Fault injectors (mutation operators) used

Name	Function
Negate Decision (ND)	negate condition in an if or while statement
Replace Constant (RC)	replace integer constant C by 0, 1, –1, C+1 or C–1
Delete Statement (DS)	delete a statement
Replace Operator (RO)	replace an arithmetic, relational, logical, bitwise logical, increment/decrement or arithmetic-assignment operator by an operator from the same class (adapted to Java)
Assign Null (AN)	replace rhs of assignment with null
Return Null (RN)	replace return expression with null

Faulty versions of the probands were obtained using fault injection (see Section 4.2 for the justification). For this, the fault injectors shown in Table 2 were employed: we added to the first four, which were previously used in the evaluation of mutation analysis itself [5] and in the mutation-based evaluation of fault locators [4, 20], two fault injectors used in our own previous work [27], mainly because we suspected that they provoke different faulty behaviour and that this impacts accuracy (Section 4.2.2).

Each fault injector was used to create as many as possible, but no more than 100 random injections of a single fault into each proband such that the injection made at least one test case of the proband fail. This gave us up to 600 detectable fault injections per proband. In the single fault case, all injections were used to produce one faulty version of the proband each; in the multiple fault cases, we used the geometric progression 2, 4, 8, 16, 32 for *n*, the number of faults, and randomly drew 1000 samples of *n* injections for each *n*, making sure that in each sample, the faults were located in different methods. Note that this procedure does not produce an equal probability distribution of the faultiness of UUTs; instead, larger methods (containing more statements) are likely to be faulty more often. While assuming such a probability distribution to be realistic may appear simplistic, the alternative, an equal number of fault injections in each method, would mean that many aberrant control flows would never be provoked, simply because the smallest method of a proband limits the number of faults injected into all larger ones.

4. THREATS TO THE VALIDITY

4.1 Heterogeneity of Probands

In practice, programs and their test suites do not only vary greatly in terms of size (cf. Table 1), but also in terms of the quality of test suites that they provide. In fact, even in open source projects, test cases such as

```
/**
* Test the constructor. This is just for coverage
*/
@Test public void testConstructor() {
   new CalendarUtils();
}
```

Table 3. Dependence of outcome on probands and accuracy measure

PROBAND	WASTED EFFORT					
	absolute			*relative*		
	T	T*	O	T	T*	O
Daikon	111.3	92.2	93.1	0.8%	0.6%	0.6%
Eventbus	13.0	3.9	4.7	1.5%	0.5%	0.5%
Jaxen	47.8	7.4	14.6	2.8%	0.4%	0.9%
Jester	4.4	2.1	2.5	1.2%	0.6%	0.7%
JExel	13.1	3.3	5.8	5.4%	1.3%	2.4%
JParsec	15.1	1.8	2.7	1.5%	0.2%	0.3%
AC Codec	5.8	2.5	2.9	2.2%	1.0%	1.1%
AC Lang	3.0	0.7	0.9	0.1%	0.0%	0.0%
Draw2d	31.6	19.0	20.6	1.0%	0.6%	0.6%
HTML Parser	19.7	4.1	5.6	1.0%	0.2%	0.3%
mean	*26.5*	*13.7*	*15.3*	*1.7%*	*0.5%*	*0.7%*
std. dev.	*31.1*	*26.7*	*26.6*	*1.4%*	*0.4%*	*0.6%*

Table 4. Dependence of the outcome on sample size

PROBAND	MEAN AND STANDARD DEVIATION OF WASTED EFFORT OF T* FOR 10 SAMPLES OF SIZE					
	10		*100*		*300*	
	∅	σ	∅	σ	∅	σ
Daikon	105.0	65.0	91.2	8.1	92.3	1.4
Eventbus	3.4	1.7	3.7	0.8	3.9	0.2
Jaxen	7.2	4.8	7.5	0.8	7.5	0.7
Jester	2.7	1.3	2.0	0.3	2.1	0.1
JExel	1.9	2.8	3.5	0.8	3.3	0.4
JParsec	1.6	1.0	1.7	0.5	1.7	0.3
AC Codec	1.8	1.4	2.4	0.5	2.5	0.2
AC Lang	0.6	0.4	0.7	0.2	0.7	0.1
Draw2d	21.6	7.6	17.6	2.0	19.6	0.8
HTML Parser	2.8	2.1	3.9	1.3	4.1	0.5
mean	*14.9*	*8.8*	*13.4*	*1.5*	*13.8*	*0.5*
std. dev.	*30.6*	*18.8*	*26.4*	*2.3*	*26.7*	*0.4*

(from JExel, `CalendarUtilsTest.java`) are not uncommon. (In this particular case, the constructor was not even implemented, so that the value of the test case for fault localization is zero.) Other probands contain mostly generated stubs for test cases (JHotDraw is an example of this) and are therefore useless for fault localization.

Even when excluding inept probands, the result of fault localization still depends considerably on their selection. Table 3 contrasts the accuracy results for each proband of Table 1 and each of our chosen three fault locators, applied to all single fault versions (obtained as described in Section 3). As can be seen, both absolute and relative wasted effort vary widely with the proband, consistently across all three used fault locators. In fact, both for absolute and relative wasted effort, outcome varies more per proband than it varies per fault locator.

Consequence Unless classes of probands with similar properties (similar TCM sizes and structures) can be identified (e.g., libraries, frameworks, applications), averages will likely be not representative of any individual proband, and thus of limited value for the assessment of accuracy for practical purposes (cf. also [12]). As an immediate consequence, we decided to break down all data presented in paper per proband, except Table 5, where this would have exceeded space limits.

Table 3 also shows how the relationship between absolute and relative accuracy (here: wasted effort) depends on the individual proband. For instance, while Daikon is by far the worst proband for demonstrating accuracy in absolute terms, it fares quite well when relative accuracy is used. This is caused by the fact that in Daikon, only 13% of all methods are covered by JUnit test cases. Conversely, the worst project for relative accuracy, JExel, performs much better in absolute terms — its test coverage is 62%. This demonstrates how relative wasted effort is biased towards low coverage (cf. Section 2.3).

The reason why both T* and O perform so much better than T is that they contain a factor rewarding UUTs that are covered by larger fractions of failed test cases (see Figure 2). In fact, in the single fault case the faulty UUT must be covered by *all* failed test cases (Assumption 2 of Section 2.1), and indeed, for all and only the UUTs covered by all failed test cases this factor is 1.[4]

[4] This property is also exploited in [3] to come up with the optimal similarity for single fault programs, which are however not representative of real programs.

4.2 Faulty Versions and Fault Injection

The previous section gave a first impression of the variability in fault localization accuracy. An even higher variability must be expected when localization accuracy is broken down to individual faults: Some faults may be ranked at first position, while others may be ranked down low. In absence of probands with numbers of faulty versions (of real faults) large enough to arrive at stable averages, fault injection seems inevitable, even if it comes with its own threats. Before turning to the threats of fault injection, we inquire into the number of faulty versions needed to obtain representative results.

4.2.1 Influence of Sample Size

Generally, not all test case failure inducing faults can be injected into a program — their number will be too large. Therefore, a selection needs to be made. However, this selection may not be representative and therefore, particularly if used by many evaluations (such as the Siemens suite), may introduce a bias towards the unrepresentative.

To demonstrate the effect of sample size (that is, the number of faulty versions used in an evaluation) on the result of the evaluation, we repeated evaluations based on 10, 100, and 300 randomly chosen faulty versions of each proband, collecting the mean and standard deviation of the average accuracy (in terms of wasted effort using T*). The results for 10 repeats are shown in Table 4. As can be seen, the mean varies randomly (e.g., from 21.6 to 17.6 to 19.6 for Draw2d), converging to the values of Table 3 (whose sample size is 600). This observation is corroborated by the standard deviation, suggesting that a sample size of 10 largely leads to random results, while a sample size of 300 is already fairly stable, even for the largest project, Daikon.

Consequence In absence of a reliable statistical model for computing required sample ("poll") sizes, that is, how many different faulty versions need to be randomly created so that the results are representative within a reasonable margin of error, we suggest that experiments like the above are performed first. If the variation is too large, sample size should be increased.

4.2.2 Choice and Distribution of Injectors

As the left of Table 5 shows, the choice of injector has a considerable impact on diagnostic accuracy, suggesting that the frequency distribution of injector use in generating the faulty version will influence outcome. Thus, if equal distribution is not possible (probands usually provide different numbers of opportunities for injection for different injectors), outcomes are not comparable.

Table 5. Dependence of outcome on choice of Fault Injectors

FAULT INJECTOR	WASTED EFFORT			IMPACT ON COVERAGE				
				overall		*exceptions only*		
	T	T*	O	Δ[§]	σ[†]	#[$]	Δ	σ
ND	28.3	12.7	14.5	28%	9%	2510	39%	13%
RC	14.6	4.1	6.8	27%	20%	1707	34%	19%
DS	34.3	20.1	22.5	21%	10%	683	32%	14%
RO	28.1	11.8	14.8	22%	8%	1391	32%	13%
AN	20.7	10.7	11.3	35%	8%	1394	37%	8%
RN	13.3	3.7	4.4	33%	7%	7670	39%	10%
mean	*23.2*	*10.5*	*12.4*	*28%*	*10%*	*2559*	*35%*	*13%*
std. dev.	*7.7*	*5.6*	*5.9*	*5%*	*5%*	*2348*	*3%*	*3%*

[§] mean reduction in coverage per test case caused by injector

[†] standard deviation of Δ

[$] number of exceptions thrown that were not expected by the test cases

One explanation of why and how different injectors impact accuracy differently is found in the right of Table 5: while there is a general tendency of fault injectors to reduce the coverage of test cases, it is not equally distributed. As was to be expected, replacing an expression with null (injectors AN and RN) leads to the greatest reductions in coverage (measured as the percentage by which coverage is reduced), which can, to a significant part, be ascribed to an increase of the number of unexpected exceptions, causing a greater than average decrease in coverage. A similarly large increase in the number of exceptions, leading to a comparable reduction in coverage, is caused by negating a decision (ND), which typically includes guards (such as tests for not null before dereferencing, instanceof before downcasts, etc.). Additionally, negating a condition of an if statement without and else branch leads to a shorter trace, reducing coverage. While in general this should impact the diagnostic accuracy positively (since there are fewer UUTs to select from), one should be aware that the results are not representative of faults that affect coverage differently. For instance, the introduction of statements will likely increase coverage.

Consequence Given the different profiles of the different fault injectors, one should consider evaluating fault localization accuracy separately for each fault injector. Otherwise, their frequency of use must be clearly documented. Also, it seems that in the context of evaluating coverage-based fault localization accuracy, the question of whether injected faults are representative of real faults [4, 20] is strongly influenced by the question of whether they have the same effect on coverage.

4.2.3 Faults Violating Basic Assumptions

As discussed in Section 2.1, coverage-based fault localization rests on a number of basic assumptions. As we found out during the debugging of our experiments, however, fault injection may produce faulty versions of programs that violate these assumptions, which may have a significant impact on evaluation. In particular, we found that Assumption 2 can be violated.

A test case not executing a faulty unit may nevertheless fail if it depends on state it does not control. This may happen, for instance, if the test depends on a static field whose value is either set by previous test cases (which would be a fault in test suite design, since tests should be independent of the order of their execution; cf. Section 2) or is set by a method invoked during static initialization performed at class loading time. Indeed, if a statically initializing expression calls a method and if a fault is injected into this method, this faultiness (or, rather, its effect on the static fields of the class) may lead to the failure of test cases that never execute the method. We call these test case failures *spurious* (since they cannot be ascribed to the test case's UUTs as taken from the TCM).

Table 6. Number of spurious failures and how this impacts accuracy of T*

PROBAND	SPURIOUS FAILURES	WASTED EFFORT *spurious failures*	
	#	*included*	*excluded*
Daikon	23	125.1	92.2
Eventbus	160	5.8	3.9
Jaxen	8	11.3	7.4
Jester	0	2.1	2.1
JExel	20	6.6	3.3
JParsec	27	4.3	1.8
AC Codec	1	2.5	2.5
AC Lang	10	2.6	0.7
Draw2d	20	23.8	19.0
HTML Parser	27	4.9	4.1
mean	*29.6*	*18.9*	*13.7*
std. dev.	*44.5*	*35.9*	*26.7*

Table 6 shows the number of spurious failures introduced by our fault injection procedure, and how it affects accuracy in terms of wasted effort. As can be seen, the relatively few spurious failures in most probands have a noticable impact on accuracy anyway, which is explained by the fact that the failure of a test case not covering a fault will not contribute to increasing its similarity coefficient and, hence, its rank. Given that spurious failures violate the assumptions of coverage-based fault localization, we have excluded them in all evaluations except that of Table 6. One should be aware, however, that while doing so is easy in the context of evaluation, it will not be in practical applications of fault localization, where the locations of faults are generally unknown.

Consequence Unless one is willing to accept the existence of spurious failures (and to regard the corresponding poor performance of fault locators as real), one has to make sure that test cases are completely self-contained, that is, do not depend on external (including static) factors.

4.2.4 Accidental Fault Injection into Test Cases

When using a test framework such as JUnit, in which test cases are coded as specially tagged, but otherwise normal, methods, faults may be accidentally injected into test cases. While this can be easily excluded for the tagged entry methods, and also for code in dedicated test directories, it works no longer if test cases use helper methods that are not automatically classifiable as such, or use production code in their assertions, or use mock objects, etc.

On the other hand, that a test case fails does not necessarily mean that its covered UUTs are flawed — it can also mean that the test case itself is faulty. In that case, searching the culprit in the UUTs only is useless and affects localization accuracy adversely. Instead, one could argue that localizing the fault in the tests contributes to assessing fault localization accuracy just like localizing any other, "regular" fault, so that distinguishing between test methods and tested methods is pointless.

However, methods representing test cases are different from other methods in that they are executed only once in the run of a test suite, and are thus uniquely associated with a failure or a pass. This may lead to extreme rankings (at either end of the scale) — in particular, in case of a failure it may give a test method a higher rank than any of the methods it tests (which may also be executed by passing test cases). Since such a ranking would likely be considered an evaluation artefact, we excluded test methods from both injection and tracing. However, since there is no automated way of telling test helper methods from methods under test, the

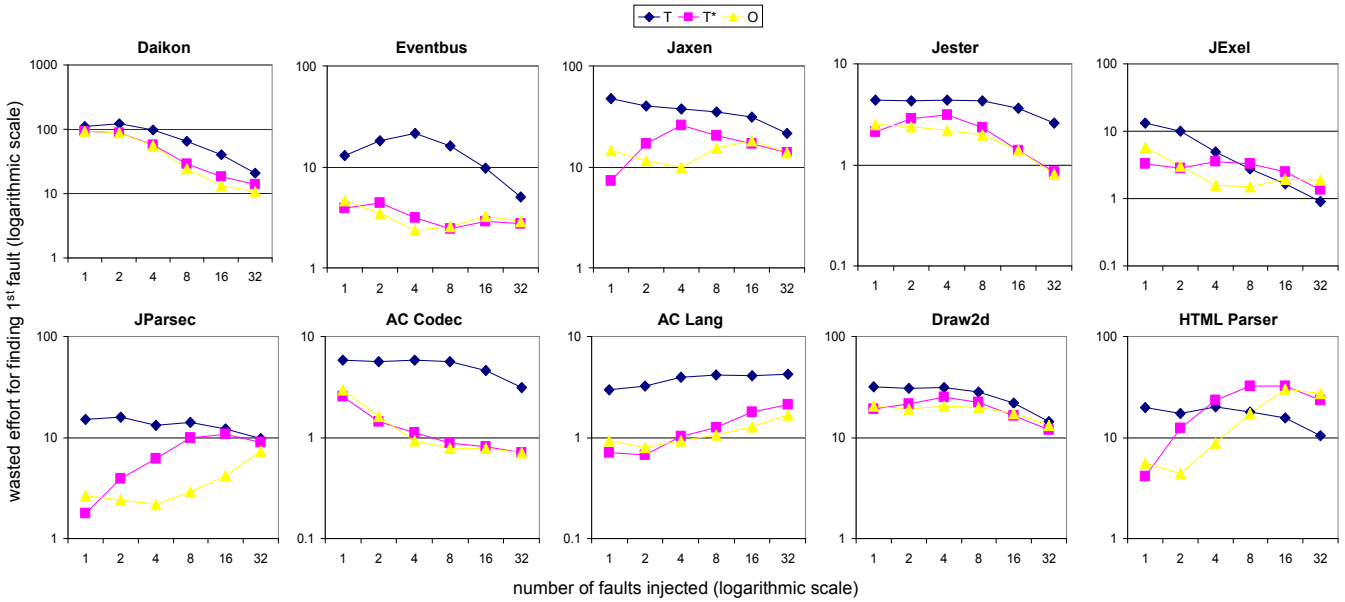

Figure 3: Dependence of the wasted effort for finding the first fault on the number of injected faults

former were included. Unfortunately, for the very same reason we were not able to investigate the effect this had on accuracy.

4.3 Tracing

To obtain TCMs, some kind of tracing is required. Obviously, tracing must not interfere with test suite execution, which is usually not the case unless this execution depends on timing conditions (tracing usually slows execution down). However, the problems of tracing that we have observed are of a different nature.

If a test framework such as JUnit is used, the identification of executed test cases may be surprisingly difficult. This is due to the use of abstract tests, theories [23], and parameterized tests.

In JUnit, abstract tests are concrete test methods defined in an abstract superclass that invoke abstract stubs overridden in concrete subclasses. Since test methods are inherited by subclasses, JUnit executes these test cases once per subclass; yet, for the tracer, this appears like the execution of one big test case. Similarly, JUnit theories [23] and parameterized tests comprise several test cases under the umbrella of one test method. In a TCM, this materializes as a single column (test case), potentially with increased coverage, that either succeeds or fails as a whole. Obviously, in the general case this single test case contributes less to fault localization than dividing it into its components would do, thereby yielding worse accuracy results than could be achieved.

Unfortunately, there is no way of dividing such multi-test cases automatically, and because of countless complications that we encountered, we aborted our attempts to do it manually for the Apache Commons Collections library, whose tests are almost entirely of the problematic kind. Therefore, we cannot present numbers suggesting how abstract tests and their kin affect fault localization accuracy.

Consequence Possible peculiarities of the test suites and test frameworks should be carefully considered; if present, averaging may be contraindicated.

4.4 Multiple Faults

In practice, assuming only a single fault in a program is unrealistic. Thus, any evaluation of the performance of a fault locator under the single fault assumption has limited practical value. On the other hand, increasing the number of faults poses its own threats to the validity of evaluations.

4.4.1 Fake Accuracy

As the number of faults increases, one (the first) should be easier detected even without the help of a fault locator, simply because the density of faults increases. This may lead to fake fault locator accuracy. On the other hand, fault locators rewarding the capability of a single UUT to explain all failed test cases (such as T* and O; cf. Section 4.1) may be confounded by the presence of many faults, and hence perform worse. Both effects can be observed in Figure 3 (but note that focusing on the first fault means selection of the smallest wasted effort; Figure 5, which shows the average wasted effort with growing numbers of faults for the case of T*, is more in line with the above theoretical considerations, demonstrating an initial increase overlaid by a growing decrease).

Consequence Evaluations of the accuracy of fault locators in finding the first fault should also evaluate the dependence on the number of faults present in the program.

4.4.2 Fault Masking

For evaluations in many-at-a-time mode, one injected fault may mask another injected fault in that it causes the other fault to get no longer executed (while it would be executed were it not for the first fault; note that avoiding this in the generation process of faulty versions is hard). The number of faults that a TCM actually covers (and that can reasonably be located) may therefore be smaller than the number of faults actually present in a proband. Table 7 shows how the number of masked faults increases with the number of faults injected in our probands. Unfortunately, we have no way of assessing the impact of masked faults on localization accuracy, as they may not even occur in the TCMs.

Consequence Because of the possible existence of masked faults, evaluations should not depend on the localization of all faults injected into the probands. How many faults should be localized is addressed by the following threat.

4.4.3 Present vs. Derivable Numbers of Faults

Since in practice it is unknown how many faults precisely are covered by a FCM (or a TCM for that matter; see above), it is unrealistic to measure the performance of a fault locator in identify-

Table 7. Dependence of the number of masked and known faults on the total number of injected faults, and effect on wasted effort (using T*) per located fault

PROBANDS	NUMBER OF FAULTS INJECTED																			
	2				4				8				16				32			
	# non-masked faults	w. e. non-masked	# known faults	w. e. known	# non-masked faults	w. e. non-masked	# known faults	w. e. known	# non-masked faults	w. e. non-masked	# known faults	w. e. known	# non-masked faults	w. e. non-masked	# known faults	w. e. known	# non-masked faults	w. e. non-masked	# known faults	w. e. known
Daikon	1.8	93.1	1.5	84.4	3.2	71.0	2.2	61.2	5.4	49.7	3.2	40.6	8.9	36.2	4.7	28.5	14.2	21.0	7.0	18.4
Eventbus	2.0	20.0	1.2	6.5	3.8	25.5	1.6	8.2	6.9	18.6	2.2	7.9	12.1	10.6	3.0	5.9	19.5	5.7	4.4	3.8
Jaxen	1.9	103.6	1.1	32.1	3.6	99.1	1.3	44.7	6.3	71.6	1.6	36.5	9.5	44.7	2.2	25.9	13.3	27.0	3.0	17.6
Jester	1.9	8.9	1.5	6.3	3.7	10.9	2.3	7.6	6.6	9.3	3.3	7.0	11.2	6.3	4.8	5.3	17.2	3.7	6.9	3.6
JExel	1.9	14.9	1.3	6.6	3.6	13.2	1.8	8.2	6.7	8.5	2.5	7.5	11.7	4.7	3.4	4.9	18.9	2.5	5.3	2.9
JParsec	2.0	38.3	1.7	26.1	4.0	45.6	2.8	39.7	7.9	40.7	4.7	37.0	15.4	28.0	8.1	26.8	29.7	18.2	13.9	17.1
AC Codec	2.0	6.9	1.9	6.5	3.9	9.7	3.6	8.5	7.7	10.0	6.5	7.7	15.0	8.2	10.8	5.6	27.9	5.4	17.0	3.8
AC Lang	2.0	6.5	2.0	5.4	4.0	9.4	3.9	7.6	8.0	10.0	7.4	8.1	15.8	11.6	13.8	7.8	31.3	12.7	24.9	7.1
Draw2d	1.8	55.2	1.2	27.2	3.5	64.9	1.4	36.3	6.1	52.0	1.9	34.0	9.9	32.2	2.6	23.9	14.7	18.3	3.4	15.0
HTML Parser	2.0	81.1	1.4	39.4	3.7	77.2	2.0	70.6	6.8	50.9	3.0	61.4	11.5	30.4	4.7	41.8	19.3	16.3	7.6	23.0
mean	*1.9*	*42.9*	*1.5*	*24.0*	*3.7*	*42.6*	*2.3*	*29.3*	*6.8*	*32.1*	*3.6*	*24.8*	*12.1*	*21.3*	*5.8*	*17.6*	*20.6*	*13.1*	*9.3*	*11.2*
std.dev.	*0.1*	*35.9*	*0.3*	*23.5*	*0.2*	*31.7*	*0.8*	*23.2*	*0.8*	*22.2*	*1.9*	*18.5*	*2.4*	*13.8*	*3.7*	*12.6*	*6.3*	*7.9*	*6.7*	*7.3*

ing all (non-masked) faults at once — who would expect a programmer to search for further faults when it is unclear whether there are actually more? Thus, to be realistic, an evaluation in many-at-a-time mode can only involve localization of as many faults as there is evidence of. The question that arises, then, is, for how many of the contained faults does a FCM provide evidence?

Taking Assumption 2 in Section 2.1 as a starting point, namely that every failed test case must cover at least one fault, a minimum number of faults that can be derived from a FCM is given by the maximum number of test cases in the FCM whose coverages (as sets of UUTs) are mutually disjoint (an instance of the *maximum set packing* problem; see Figure 4 for an example). Table 7 shows the average minimum number of faults (computed using a greedy algorithm approximating maximum set packing) and how it depends on the number of injected faults. As can be seen, the number of known faults is considerably smaller than the number of injected faults (it drops to less than one third on average for 32 injected faults) and also than the number of non-masked faults (less than half on average for 32 injected faults). At the same time, the effort wasted, per fault, on locating the known number of faults (in many-at-a-time mode) decreases (see also Figure 5). Given that the known number of faults is significantly smaller than the number of injected, non-masked faults, this effect can be ascribed to an increased density of faults (see above).

Consequence When evaluating the accuracy of fault locators in many-at-a-time mode, one should limit the search to the number of faults that can be deduced from the FCM alone (without knowledge of how many faults have been injected or executed).

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 4. FCM providing hard evidence for 4 faults; test cases constituting a maximum set packing are highlighted.

4.5 Uncontrollable Factors

As already noted in Section 3, tools used in the evaluation process may be inaccurate. For instance, obtaining accurate measures of the probands (such as their size) depends on the definition of the metric (which varies from tool to tool and is sometimes unclear) and on the quality of its implementation; obtaining correct TCMs depends on the tracing tool used; and so on. Also, the programs written specifically for the evaluation may be flawed which, so the state of the art, cannot be excluded.

Last but not least, despite all care taken, an evaluation process may be subject to random factors. For instance, improper parallelization, either in the probands or in the evaluation environment, may be a source of randomness, for example when data races occur (we actually experienced problems of this kind with running the HTML Parser test suites in Eclipse). Also, a coverage-based fault locator that is sensitive to the ordering of the rows or columns of a TCM may produce random results if the TCM is stored internally using hash maps or sets. Since some of these effects may be caused by the environment, they may be impossible to rule out with certainty.

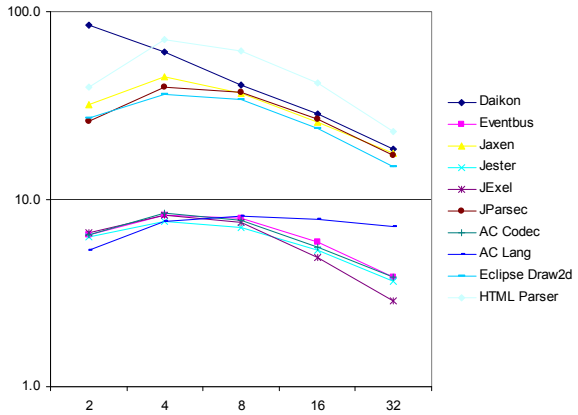

Figure 5: Dependence of wasted effort per known fault on number of injected faults for T* (logarithmic scales)

Consequences To detect random factors, every evaluation should be repeated by their authors several times in an identical setting. If random factors exist, but cannot be avoided, their effect must be quantified or, if possible, eliminated by averaging over repeats.

5. THREATS TO THE VALUE

Even results that are valid within the scope in which they have been obtained may be of limited value, for the following reasons.

5.1 Unrealistic Assumptions

One threat to the value of an empirical assessment of fault localization is that the assumptions underlying the evaluation are too strong to accord with practice. For instance, Assumption 2 in Section 2.1, while easy to ensure in an evaluation setting in which all (injected) faults are known, is difficult to guarantee in practice, so that fault locators will be applied to test suites with spurious failures (cf. Section 4.2.3). Also, while one might argue that test suites with interdependencies between individual test cases or with external dependencies should not be used, they are used in practice, where fault localization is ultimately to be applied.

5.2 Language Idiosyncrasies

Not only the individual probands, but also their programming languages and test frameworks have an impact on outcome (cf. the discussions of Sections 4.2.4 and 4.3). While this does not invalidate studies that are carried out in — and for — a particular language and testing environment, their value is limited in that their results cannot be carried over to other environments. Note that this also holds for the experiments reported on here (which are based on Java and JUnit); yet, that evaluations are subject to idiosyncrasies of the environment seems universally valid.

5.3 Unset Expectations

The accuracy of fault locators is usually not assessed relative to some gold standard that suggests what a good result is. Taking user expectations as a surrogate here may be interesting from a pragmatic point of view, but it is less helpful in a research context in which boundaries are still being pushed. Indeed, an observation such as "programmers will stop inspecting statements, and transition to traditional debugging, if they do not get promising results within the first few statements they inspect" [21, p. 207] does not tell much about the performance of a fault locator (other than it obviously failed to meet user expectation), unless it is clear how difficult it really is to locate the given faults. A more objective measure of performance would be desirable here.

So, what does it mean if a given fault locator produces a wasted effort of x in a given setting? Surely, we can compare performances (but, given the above threats to the validity, it should be clear that most assessments are not comparable), but what does it mean in absolute terms? In fact, even an evaluation showing that the wasted effort is (close to) zero does not mean much without knowing how difficult it was to achieve — given an ideal test suite, a perfect diagnostic accuracy may be easily obtained.

What we really need to judge the performance of a fault locator are lower and upper bounds, that is, a range of accuracies that we can reasonably expect. By analogy, when presented with a new algorithm, how else do we judge its performance if not by comparing it to one that is easily achieved (e.g., $O(n^2)$ in the case of sorting) and the theoretical optimum (e.g., $O(n \cdot \log n)$)?

5.3.1 Lower Bounds

In a single-fault setting, Assumptions 2 and 3 of Section 2.1 directly provide a lower bound of the accuracy that needs to be passes by a fault locator to be useful: if its wasted effort is on average greater than $(n+1)/2-1$, where n is the number of UUTs covered by the failed test case with the lowest coverage (and 1 is subtracted because the measure is *wasted* effort), it is useless, since one could use a random inspection order of the n UUTs instead. In particular, if the shortest failed test case covers only a single UUT, the lower bound of wasted effort is 0. In a multi-fault setting, if for some reason it were known that the failed test case covers k faulty units, the fault locator should perform better than $(n+1)/(k+1)-1$ (see Section 2.3); however, firm knowledge of this fact cannot usually be expected.[5] Table 8 therefore shows how fault locators T, T*, and O fare in a single fault setting, compared to $(n+1)/2-1$ as the lower bound (where n is defined as above). As can be seen, on average T performs better than the lower bound in half of all probands, and even for probands for which this is not the case, it still performs better in 59% of all localizations (averaged over these probands). Thus, by comparison with the lower bound, T is a reasonable fault locator. Somewhat surprisingly, however, even though both T* and O perform significantly better, both perform worse than the lower bound in non-negligible numbers of cases, indicating that there is room for improvement.

Table 8. Accuracy of fault locators compared to a theoretical lower bound (L. B.) in a single-fault setting

Proband	L. B.	T		T*		O	
	w. e.	$\Delta^{\dagger}$	$\Phi^{\S}$	*w. e.*	Φ	*w. e.*	Φ
Daikon	126.5	15.3	19%	34.3	4%	33.4	5%
Eventbus	11.7	-1.4	32%	7.8	7%	7.0	10%
Jaxen	63.8	16.0	33%	56.5	5%	49.3	17%
Jester	6.1	1.8	24%	4.0	6%	3.6	11%
JExel	10.7	-2.4	45%	7.4	6%	4.9	16%
JParsec	11.9	-3.2	34%	10.1	3%	9.2	7%
AC Codec	4.5	-1.4	62%	1.9	21%	1.5	27%
AC Lang	2.6	-0.3	31%	1.9	4%	1.7	7%
Draw2d	45.9	14.2	31%	26.9	5%	25.3	8%
HTML Parser	51.0	31.3	25%	46.9	2%	45.4	6%
mean	*33.5*	*7.0*	*34%*	*19.8*	*6%*	*18.1*	*11%*
std.dev.	*37.5*	*11.0*	*11%*	*19.0*	*5%*	*17.7*	*6%*

† difference in wasted effort; > 0 means better, < 0 means worse

§ fraction of times locator performed worse than lower bound

In many-at-a-time mode, the same lower bound is applicable in principle, but varies with the number of faults to be detected at a time. If this number is the evident number of faults computed by maximum set packing (as in Section 4.4.3), the test cases constituting the packing can be used directly to compute a lower bound for each fault as $(n_i-1)/2$ (where n_i is the number of UUTs covered by test case t_i; note that each test case in the packing must cover one fault, which is the one to be detected). If n, the number of faults to be detected at a time, is less, a tighter lower bound can be established, by searching n non-overlapping failed test cases under the additional constraint that the coverages of the test cases should be as small as possible (a weighted set packing problem).

Since it has been observed that applying coverage-based fault locators to complete TCMs leads to poor results in many-at-a-time mode (e.g., [17, 18, 27]; but cf. Section 4.4.1 for how this effect is counteracted by increasing the numbers of faults significantly), we do not locate all known faults in a single FCM (as we did in Table 7). Instead, we first transform each FCM into a block diagonal matrix as described in [26, 27], giving us n partitions in

[5] Instead, if we have some estimate of the total number of faults in a program, we can compute a probability of the fact that the faulty test case covers k faulty UUTs. However, since the test case was chosen for its small coverage, chances are low that it covers more than one fault, so that we will leave it at that.

Table 9. Accuracy of fault localization using block diagonal partitioning of the TCM and T* compared to a lower bound in many-at-a-time mode, localizing one fault per partition

PROBAND	2				4				8				16				32			
	# p.[$]	*l. b.*	*Δ*[†]	*Φ*[§]	*# p.*	*l. b.*	*Δ*	*Φ*	*# p.*	*l. b.*	*Δ*	*Φ*	*# p.*	*l. b.*	*Δ*	*Φ*	*# p.*	*l. b.*	*Δ*	*Φ*
Daikon	2.0	93.4	10.9	23%	3.8	49.7	-4.0	39%	7.1	31.2	-2.1	48%	12.7	22.2	0.2	53%	21.4	11.0	0.3	47%
Eventbus	1.0	7.3	2.9	20%	1.1	4.2	1.0	21%	1.1	2.1	-0.3	35%	1.2	1.3	-1.4	59%	1.4	0.5	-1.6	77%
Jaxen	1.9	47.4	30.4	16%	3.3	32.0	6.0	30%	5.4	19.0	-1.4	40%	7.7	9.2	-7.2	56%	10.6	3.0	-9.1	77%
Jester	1.3	4.8	1.8	19%	1.5	2.9	0.2	32%	1.7	2.0	0.1	33%	2.2	1.4	0.2	32%	3.4	1.0	0.1	40%
JExel	1.5	5.9	3.0	17%	2.1	3.0	-0.2	28%	3.0	1.5	-1.0	35%	4.2	0.8	-0.5	34%	5.9	0.5	-0.1	37%
JParsec	1.1	6.2	2.3	19%	1.2	2.1	-3.4	46%	1.3	0.8	-6.7	64%	1.5	0.5	-6.0	71%	1.8	0.5	-3.6	80%
AC Codec	1.0	4.0	1.9	22%	1.0	3.8	1.7	20%	1.1	2.7	1.1	22%	1.1	1.7	0.6	22%	1.3	0.8	0.3	24%
AC Lang	1.3	2.5	1.7	6%	1.4	2.4	1.6	6%	1.9	2.1	1.3	7%	2.6	1.8	0.9	10%	4.1	1.3	0.5	16%
Draw2d	1.4	30.9	9.4	24%	2.0	17.2	-6.5	48%	2.8	9.0	-11.0	64%	4.0	4.3	-8.8	78%	5.7	2.1	-6.2	80%
HTML Parser	1.1	33.8	20.9	16%	1.3	11.3	-10.4	34%	1.7	3.9	-19.8	55%	2.4	1.6	-16.0	74%	4.1	1.2	-6.4	83%
mean	*1.4*	*23.6*	*8.5*	*18%*	*1.9*	*12.8*	*-1.4*	*30%*	*2.7*	*7.4*	*-4.0*	*40%*	*4.0*	*4.5*	*-3.8*	*49%*	*6.0*	*2.2*	*-2.6*	*56%*
std. dev.	*0.3*	*27.7*	*9.3*	*5%*	*0.9*	*15.2*	*4.5*	*12%*	*1.9*	*9.5*	*6.4*	*17%*	*3.5*	*6.4*	*5.3*	*22%*	*5.8*	*3.0*	*3.3*	*25%*

[$] number of partitions

[†] difference in wasted effort; > 0 means better, < 0 means worse

[§] fraction of times locator performed worse than lower bound

which faults can be located independently. In each partition, we locate one fault (using T*), and contrast the wasted effort so obtained with that suggested by the lower bound as defined above. The results of this experiment are shown in Table 9. As can be seen, while T* still performs better than the lower bound for 2 injected faults (yielding 1.4 partitions on average across all project averages), accuracy drops consistently towards and below the lower bound for higher numbers. Thus, like for the single-fault case, our (theoretical) lower bound suggests that there is (this time significant) room for improvement (see Figure 6).

Surely, this experiment does not only shed light on the performance of T*, but also on the used partitioning algorithm. In fact, as can be seen by comparing the number of known faults from Table 7 with the number of partitions from Table 9, better partitioning seems possible. However, we did not compare the algorithms described in [17, 27] with the lower bound obtained from a block diagonal matrix here, since they depend on additional assumptions [27], which makes their evaluation difficult.

NB: Both for the one-at-a-time (Table 8) and for the many-at-a-time mode (Table 9), the specified lower bound depends on qualities of the probands (programs *and* test suites) exclusively so that it is valid across all fault locators.

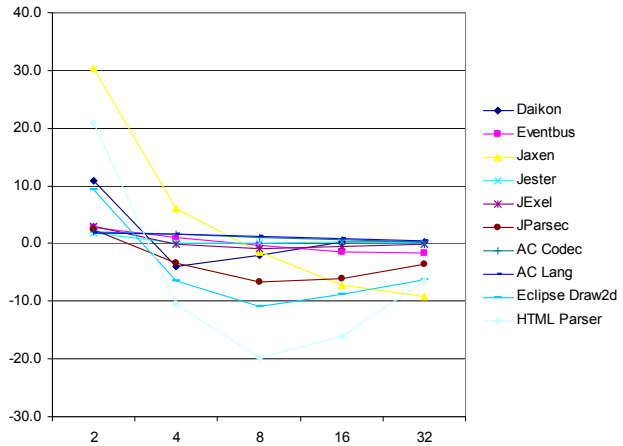

Figure 6: Performance of T* compared to lower bound, and how it depends on the number of faults injected; measured as savings in wasted effort per partition (cf. Table 9 for how number of partitions depends on number of injected faults).

5.3.2 Upper Bounds

Upper bounds of fault locator performance seem more difficult to establish. Intuitively, a fault locator cannot extract more information out of a FCM than it contains. For instance, if all rows of a FCM are identical, a fault locator cannot perform better than a random selection from the units explaining a failure. More generally, it could be argued that the best achievable performance of any coverage-based fault locator depends on the entropy of the TCM to which it is applied. How precisely this is the case is not obvious, though.

Note that like for the lower bounds, we are looking for an absolute upper bound of fault localization accuracy, i.e., one that depends on qualities of the probands alone. Establishing lower and upper bounds of diagnostic accuracy for a specific fault locator is a different question, one that should be addressed by the researchers suggesting use of their fault locators.

6. RELATED WORK

We are not the first to question the validity and value of coverage-based fault localization studies. Ali, Andrews, Dhandapani and Wang have tested some underlying assumptions, but mostly found that they hold [4]. In particular, no evidence has been found that fault injection via mutation should not be used [4, 20]. Our focus has been much broader, however. A recent study [12] supports our observation that large, randomly drawn corpuses (sets of probands) are inevitable to achieve (external) validity. And yet, no matter how valid results may be, Parnin and Orso have questioned the sanity of advocating coverage-based fault localization [21], based on a first empirical user study; however, we maintain that achieving high accuracy in automated fault localization is too important a goal to be given up early.

On a more technical side, the general adequacy of coverage-based fault localization is challenged by the confounding effects of existing program dependences. For instance, the fault in a condition of an `if-then-else` statement may lead to the execution of the `else` branch in all failed test cases, ranking the statements in this branch higher than the faulty condition, which is also executed by passing test cases. Recent work [6, 14] has therefore focused on reducing confounding biases using additional information (such as program dependence graphs) not contained in the pure TCMs, on which our work rests exclusively.

The set of similarity coefficients that can be used in coverage-based fault localization is virtually unlimited. Zhang et al. performed a comprehensive comparative analysis [32], but as we

have shown, the performance of fault locators depends on many factors, some of which (such as number of faults present) even having the potential to invert comparative performance rankings. [30] evaluates three different tie breaking strategies (to be used when several UUTs, one of them being faulty, share the same rank); however, each examined strategy defines itself a ranking and is therefore subject to the same criticisms.

Given that performing empirical evaluations is subject to at least the pitfalls comprised in this paper, theoretical evaluations of diagnostic accuracy promise more reliable results. Naish et al. have designed a sample program that serves as a model to the fault localization problem [19]. Using this model, they show which different similarity coefficients must have the same accuracy in the single fault case, and which must perform best. Xie et al. have performed a theoretical analysis of similarity coefficients (called risk evaluation formulas) for arbitrary probands [29] — they show that a group of five out of 30 similarity coefficients must outperform the rest, but their reasoning likewise relies on certain strong assumptions (such as presence of a single fault only) that we must reject as unrealistic for practical application.

Entropy (which we hope to be the key to setting up an upper bound for diagnostic accuracy) has been used already for prioritizing the execution of test cases [13, 31]: Intuitively, when resources are limited, as soon as the execution of a test case has flagged the presence of a fault, execution order of the remaining test cases should reflect their expected diagnostic information gain. Surely, accuracy can be increased as long as pending test cases can add information, but how information content — or entropy — translates to an achievable diagnostic accuracy seems still an open question.

7. CONCLUSION

Not surprisingly, the outcome of evaluations of the accuracy of coverage-based fault locators depends on many factors. Every comprehensive evaluation should therefore take as many factors into account as possible. The result is then a multidimensional table that, for presentation and human interpretation, needs to be projected to a lower dimensionality by using suitable aggregations (usually averaging). Where projections are obtained by setting the sample size for one or more dimensions to 1 (e.g., by basing the evaluation on a single proband, or by considering the single fault case only) one should be aware that the obtained results are by no means generalizable to larger sample sizes.

Maybe the most striking result is that the outcome of fault localization depends more on the proband than it depends on the fault locator itself. For relative assessments of fault locators, it is therefore imperative that they are evaluated on the exact same probands. Often enough, this causes technical problems, for instance because the probands are not publically available (at least not in the same version that has been used in the evaluation; for instance, the faulty versions may be unavailable, which is usually the case when they have been generated using automated fault injection) or because the other fault locator was implemented in a different environment than the current one, subjecting it to different influences that are difficult to detect and avoid. However, most of these problems are accidental, since it would be easy (and would comply with good scientific practice) to store and make available the FCMs on which the fault locators have been evaluated. Factually, however, this is not (yet) the case so that relative comparisons remain questionable.

One of the more positive results that we obtained is that the simple lower bound for localization accuracy that we defined in Section 5.3.1 seems useful — it shows that established fault locators do not perform as well as one could rightfully expect, if not in the average case, in non-negligibly many cases at least.

Overall, however, empirical evaluations can only be a best effort. Attempts to independently reproduce results should therefore be rewarded by the community, for instance by accepting corresponding reports — even if containing only little novelty — as scientific contributions. At the same time, concerted efforts should be made to eliminate all sources of error, for instance by verifying and standardizing all used tools.

A more personal conclusion is that doing evaluations of the given kind is an arduous undertaking. While we estimate the total computing time for all experiments here described at around 225 hours on contemporary PCs, this does not count the many repeats of experiments that became necessary for instance because we picked the wrong performance indicators, did the wrong aggregation, or because we had discovered bugs in our programs. Our final resume thus coincides with Brook's Law of Prototypes [7]

> "Plan to throw one away, you will anyhow."

acknowledging that the following riposte (found ibid.) is also true:

> "If you plan to throw one away, you will throw away two."

8. DOWNLOAD

To foster reproducibility and comparability, we have made all TCMs and algorithms used to compute the presented data available through www.feu.de/ps/prjs/EzUnit/eval/ISSTA13. The website also contains descriptions of the data formats used, as well as descriptions of where the probands can be obtained and how they have been modified.

9. ACKNOWLEDGMENTS

The authors wish to thank Alexandre Perez and all three anonymous reviewers for their valuable suggestions for improvement.

The third author's work is partly financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/116796/2010.

10. REFERENCES

[1] R Abreu, P Zoeteweij, AJC van Gemund "An evaluation of similarity coefficients for software fault localization" in: *Proc. of PRDC* (2006) 39–46.

[2] R Abreu, P Zoeteweij, R Golsteijn, AJC van Gemund "A practical evaluation of spectrum-based fault localization" *Journal of Systems and Software* 82:11 (2009) 1780–1792.

[3] R Abreu, P Zoeteweij, AJC van Gemund "Spectrum-based multiple fault localization" in: *Proc. of ASE* (2009) 88–99.

[4] S Ali, JH Andrews, T Dhandapani, W Wang "Evaluating the accuracy of fault localization techniques" in: *Proc. of ASE* (2009) 76–87.

[5] JH Andrews, LC Briand, Y Labiche "Is mutation an appropriate tool for testing experiments?" in: *Proc. of ICSE* (2005) 402–411.

[6] GK Baah, A Podgurski, MJ Harrold "Mitigating the confounding effects of program dependences for effective fault localization" in: *Proc. of SIGSOFT FSE* (2011) 146–156.

[7] J Bentley *More Programming Pearls: Confessions of a Coder* (Addison Wesley 1988).

[8] MY Chen, E Kiciman, E Fratkin, A Fox, A., EA Brewer "Pinpoint: Problem determination in large, dynamic internet services" in: *Proc. of DSN* (2002) 595–604.

[9] H Cleve, A Zeller "Locating causes of program failures" in: *Proc. of ICSE* (2005) 342–351.

[10] V Dallmeier, C Lindig, A Zeller “Lightweight defect localization for Java” in: *Proc. of ECOOP* (2005) 528–550.
[11] N DiGiuseppe, JA Jones “On the influence of multiple faults on coverage-based fault localization” in: *Proc. of ISSTA* (2011) 210–220.
[12] G Fraser, A Arcuri “Sound empirical evidence in software testing” in: *Proc. of ICSE* (2012) 178–188.
[13] A Gonzalez-Sanchez, R Abreu, HG Gross, AJC von Gemund “Prioritizing tests for fault localization through ambiguity group reduction” in: *Proc. of ASE* (2011) 83-92.
[14] R Gore, PF Reynolds Jr. “Reducing confounding bias in predicate-level statistical debugging metrics” in: *Proc. of ICSE* (2012) 463–473.
[15] JA Jones, MJ Harrold, JT Stasko “Visualization of test information to assist fault localization” in: *Proc. of ICSE* (2002) 467–477.
[16] JA Jones, MJ Harrold “Empirical evaluation of the tarantula automatic fault-localization technique” in: *Proc. of ASE* (2005) 273–282.
[17] JA Jones, MJ Harrold, JF Bowring “Debugging in parallel” in: *Proc. of ISSTA* (2007) 16–26.
[18] JA Jones *Semi-Automatic Fault Localization* PhD Thesis (Georgia Institute of Technology, 2008).
[19] L Naish, H Jie L , K Ramamohanarao “A model for spectra-based software diagnosis” *ACM Trans. Softw. Eng. Methodol.* 20(3):11 (2011).
[20] AS Namin, S Kakarla “The use of mutation in testing experiments and its sensitivity to external threats” in: *Proc. of ISSTA* (2011) 342–352.
[21] C Parnin, A Orso “Are automated debugging techniques actually helping programmers?” in: *Proc of ISSTA* (2011) 199–209.
[22] M Renieris, SP Reiss “Fault localization with nearest neighbor queries” in: *Proc. of ASE* (2003) 30–39.
[23] D Saff “Theory-infected: or how I learned to stop worrying and love universal quantification” in: *OOPSLA Companion* (2007) 846–847.
[24] RA Santelices, JA Jones, Y Yu, MJ Harrold “Lightweight fault-localization using multiple coverage types” in: *Proc. of ICSE* (2009) 56–66.
[25] F Steimann, T Eichstädt-Engelen, M Schaaf “Towards raising the failure of unit tests to the level of compiler-reported errors” in: *Proc. of TOOLS Europe* (2008) 60–79.
[26] F Steimann, M Bertschler “A simple coverage-based locator for mul-tiple faults” in: *Proc of ICST* (2009) 366–375.
[27] F Steimann, M Frenkel “Improving coverage-based localization of multiple faults using algorithms from Integer Linear Programming” in: *Proc. of ISSRE* (2012).
[28] X Wang, SC Cheung, WK Chan, Z Zhang “Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization” in: *Proc. of ICSE* (2009) 45–55.
[29] X Xie, T Chen, FC Kuo, B Xu “A theoretical analysis of the risk evaluation formulas for spectrum-based fault Localization” *ACM TOSEM* (in print).
[30] X Xu, V Debroy, WE Wong, D Guo “Ties within fault localization rankings: Exposing and addressing the problem” *International Journal of Software Engineering and Knowledge Engineering* 21:6 (2011) 803–827.
[31] S Yoo, M Harman, D Clark “Fault localization prioritization: Comparing information theoretic and coverage based approaches” *ACM TOSEM* (in press).
[32] Z Zhang, WK Chan, TH Tse, YT Yu, P Hu “Non-parametric statistical fault localization” *Journal of Systems and Software* 84:6 (2011) 885–905.