A Simple Coverage-Based Locator for Multiple Faults

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Mario Bertschler
V8tech
Millennium Park 4
A-6890 Lustenau
mario.bertschler@v8tech.com

Abstract

Fault localization helps spotting faults in source code by exploiting automatically collected data. Deviating from other fault locators relying on hit spectra or test coverage information, we do not compute the likelihood of each possible fault location by evaluating its participation in failed and passed test cases, but rather search for each failed test case the set of possible fault locations explaining its failure. Assuming a probability distribution of the number of faults as the only other input, we can compute the probability of faultiness for each possible fault location in presence of arbitrarily many faults. As the main threat to the viability of our approach we identify its inherent complexity, for which we present two simple bypasses. First experiments show that while leaving room for improvement, our approach is already feasible in practical cases.

1. Introduction

With the growing popularity of regression testing tools such as JUNIT, the writing of test cases prior or parallel to programming is becoming commonplace. The existence of well-designed test suites allows one to work on a program while at the same time maintaining some level of confidence that it continues to work as expected. However, while finding out that changes to a program make one or more test cases fail is a fully automated task, this is only the first step in the debugging process: the second step, the search for the code causing the failure, is far more demanding and, much to the harm of productivity, still mostly manual. With the popularity of regression testing, the interest in automatic fault localization is therefore also growing.

A particular breed of fault locators, called hit spectra based or coverage based fault locators [1][6][7][11] [16], exploits information gathered from passed and failed test cases. The general idea behind this is that program units under test (hereafter referred to as UUTs) executed by more failed and fewer passed test cases are more likely faulty than others. Most fault locators of this kind compute the likelihood of faultiness for each UUT in isolation, i.e., independent of other UUTs. However, this leaves important information unexploited: for instance, if a failed test case executes only a single UUT, this UUT must be faulty, no matter in how many passed test cases it participates.

Our contribution to fault localization is the following. We assume a simple fault model according to which each failed test case must have a cause in at least one of its UUTs. However, a UUT assumed as faulty in order to explain a failed test case need not cause other test cases to fail also; yet it can, giving us alternative explanations with varying numbers of faults. By assuming a probability distribution of the number of faults, and by counting the membership of each UUT in all explanations, we get a probability distribution of the faultiness of UUTs. Figuratively speaking, this adds another dimension to coverage based fault localization.

The remainder of this paper is organized as follows. After briefly sketching the general problem of fault localization and recapitulating the basics of its coverage-based form in Sections 2 and 3, we present our approach and its underlying fault model in some detail in Section 4. The implementation and its application are sketched in Section 5; its evaluation in Section 6 primarily investigates the main threat to the viability of our approach, which we deem comes from its inherent complexity. A discussion and comparison with related work conclude our contribution.

2. Fault localization

Fault localization is the attempt to track down logical programming errors in source code. Logical errors are more difficult to localize than syntactic or semantic

Table 1. Relating UUTs to their test cases and actual and predicted faultiness

UUT	TEST CASES						FAULTINESS	
	FAILED			PASSED			PREDICTED	ACTUAL
	t_1	…	t_m	t_{m+1}	…	t_a		
u_1	$x_{1,1}$	…	$x_{1,m}$	$x_{1,m+1}$	…	$x_{1,a}$	$L(u_1)$	f_1
…	…	…	…	…	…	…	…	…
u_n	$x_{n,1}$	…	$x_{n,m}$	$x_{n,m+1}$	…	$x_{n,a}$	$L(u_n)$	f_n

errors, since they do not manifest themselves as violations of the rules of the programming language. Their detection, therefore, requires some source of knowledge outside the program and the language definition.

Fault locators are rarely binary; instead, they provide a numeric estimate of the likelihood (or suspiciousness) of a certain source location of being faulty. The developer is then presented with one or more lists of possible fault locations, with the most likely locations listed first. The performance of a fault locator can be measured as the average number of fault locations that need to be inspected before a fault is found.

Elsewhere [16], we have divided fault localization methods into ones that exploit a priori hints at faultiness (such as program metrics; e.g. [2][5][9][10][12][14][17][18]), and ones that exploit a posteriori information, i.e., that interpret data in light of certainty that a program is incorrect (including the data collected during the proof of incorrectness; e.g. [1][4][5][6][7][8][11][13][16]). The body of literature presenting new fault locators of either kind is continuously growing, with evaluations of their capability showing that presented fault locators excel previous ones, either generally or under certain conditions. However, it can be postulated that no single fault locator is, or likely ever will be, optimal, but that instead suitable fault locators must be carefully selected per project and their results be combined to produce satisfactory results [14]. After all, fault localization is a detective process, having to collect and combine evidence of faultiness from all available sources.

3. Coverage based fault localization

Coverage based [11] (also called hit spectra based [1]) fault locators compute the likelihood of units of execution (i.e., statements, blocks, methods, etc.) being faulty given their participation in a number of passing and failing test cases (or more generally: successful program runs and ones in which an error has been detected). The data required to compute this likelihood can be derived from a table like Table 1, which relates test cases $t_1, \ldots, t_a$ with the UUTs $u_1, \ldots, u_n$. In this table, the set of available test cases is partitioned into ones that fail ($t_1, \ldots, t_m$) and ones that pass ($t_{m+1}, \ldots, t_a$). A table entry $x_{i,j}$ with value 1 means that UUT u_i is executed by test case t_j, a value of 0 means that u_i is not executed by t_j.

The actual faultiness of a UUT u_i is represented by $f_i \in \{0,1\}$ where a value of 1 means that u_i is faulty, whereas one of 0 means that it is correct. Usually, it remains implicit that $f_i = 0$ here can only mean that u_i is not responsible for the failure of any test case and therefore need not be fixed for the tests to pass; it can nevertheless be faulty. Conversely, $f_i = 1$ means that fixing u_i is mandatory for making test cases pass.[1] The vector $\vec{f} = (f_1 \ldots f_n)^T$ is sometimes called the *gold standard* of the testing; it is usually only available in an evaluation setting, when the locations of existing faults are known (for instance because they have been injected), or after the program has been successfully debugged.

A fault locator L predicts the faultiness of a UUT u_i as a value $L(u_i) \in [0,1]$. For an ideal fault locator, the vector $\vec{L} = (L(u_1) \ldots L(u_n))^T$ is identical to $\vec{f}$; practical fault locators are the better the smaller the difference (or the greater the similarity) between $\vec{L}$ and $\vec{f}$ is (where difference or similarity are appropriately defined functions).

A hit spectrum based fault locator interprets the column vectors of Table 1 (the test cases $t_1 \ldots t_a$) as spectra and computes the similarity of each row vector $\vec{u}_i = (x_{i,1} \ldots x_{i,a})^T$ with the vector representing the outcome associated with each spectrum t_i, consisting of m 1s and $a - m$ 0s (in that order). The simple idea of this is that, the more often a UUT is executed in failed test cases and the less often it is executed in passed test cases, the more likely it is that it is faulty.

Based on this scheme, a number of coverage based fault locators have been defined and evaluated. For instance, the fault locator *suspiciousness* of Tarantula [11] has been defined as

$$suspiciousness(u_i) := \frac{\frac{1}{m}\sum_{j=1}^{m} x_{i,j}}{\frac{1}{a-m} \cdot \sum_{j=m+1}^{a} x_{i,j} + \frac{1}{m}\sum_{j=1}^{m} x_{i,j}}$$

Our own fault locator, named *Failure Accountability* (FA) and defined as

$$FA(u_i) := \left(\sum_{j=1}^{m} x_{i,j}\right)^2 \cdot \left(m \cdot \sum_{j=1}^{a} x_{i,j}\right)^{-1}$$

[1] This ignores, as usual, the possibility that changing another unit can cancel the fault.

is the square of the Ochiai similarity coefficient [1], and has been shown independently in [1] and [16] to yield excellent results in presence of a single fault. However, given that all failed test cases must have executed the single faulty unit (or otherwise they would not have failed), and given that FA/Ochiai punishes units that are not called by all failed test cases [16], this should come as no surprise.

In the context of this paper we are not so much interested in the demonstrated performance of fault locators; instead, we would like to stress that all of the hit spectra and similarity coefficient based fault locators are strictly "horizontal", i.e., their computation of the likely faultiness of u_i depends solely on the entries in the i^{th} row of a table such as Table 1. This is interesting because these fault locators ignore the "vertical" information that can be derived from the columns of the table. Indeed, such information can be amazingly revealing: for instance, if we have a failing test case t_j such that the $x_{i,j}$ are zero for all but one row i', (meaning that only $u_{i'}$ is executed by this test case) it is quite clear that $u_{i'}$ must be faulty, no matter in how many passing test cases it has also been executed. Similarly, if we have that the $x_{i,j}$ are zero for all but two rows i_1 and i_2, we know that u_{i_1} or u_{i_2} must be faulty and also that if one of u_{i_1} and u_{i_2} is known to be not faulty, the other must be. However, based on tests alone, it is usually impossible to know whether a unit is correct — in fact, there is no reason to doubt that both are faulty. *Unless* one believes that two faults are less likely than one.

4. A fault locator assuming multiple faults

Before we present our approach to fault localization, we extend the formalization of the problem begun in the previous section.

Let P be a program consisting of a set of units, $U = \{u_1, \ldots\}$. Let T be a test suite of P consisting of a set of test cases $\{t_1, \ldots\}$; T is partitioned into T_F and T_P, the sets of failing and passing test cases, respectively. Let further c be a function mapping test cases to their UUTs, i.e.

$$c: T \to \wp(U)$$

where $\wp(U)$ denotes the power set of U. We call $c(t)$ the *coverage* of t.

We extend c to sets of test cases so that

$$c: \wp(T) \to \wp(U)$$

where

$$c(T') := \bigcup_{t \in T'} c(t) \quad \text{for all } T' \subseteq T$$

We abbreviate the set of all UUTs covered by failed test cases, $c(T_F)$, as U_F. $n = |U_F|$ is the number of units in U_F; it is also the maximum number of individual faults we can expect to be diagnosed in P given the test suite T.[2] We call the pair (T_F, c) a *fault localization problem*.

4.1. Test case failures and explanations

If a test case t fails, this must have an explanation, and this explanation must be sought among the set of units executed by the test case, $c(t)$.[3] Of course, all executed units could be faulty, but for the explanation of our finding, namely that a test failed, assuming that a single covered unit is faulty suffices.

For the remainder of this work, we are seeking *explanations of observed test failures*, where an explanation is a subset of U_F, the set of units covered by any failed test case. We speak of $\wp(U_F) \setminus \varnothing$ as the set of all *potential explanations* (the "explanation space"; note that assuming no unit as being faulty is ruled out as an explanation), which gives us $2^n - 1$ potential explanations.

If more than one test case fails, an explanation is needed for each. Sometimes, we will find that assuming the same unit as faulty explains all failed test cases.[4] In this case, we have that the explanation consists of a single unit $u \in U_F$ such that

$$\forall t \in T_F : u \in c(t)$$

In fact, there may be several such units so that the set of all possible explanations consisting of a single unit, E_1, is defined as

$$E_1 = \{ \{u \in U_F\} \mid \forall t \in T_F : u \in c(t) \}$$

Any u with $\{u\} \in E_1$ then explains all failed test cases. In case there is no such unit, however, we must assume more than one unit to be faulty to explain all findings (failed test cases). In fact, there is always a trivial explanation, namely assuming all units as faulty:

$$E_n = \{U_F\}$$

(recall that n stands for $|U_F|$, the number of units exe-

[2] Of course, there can be more faults in a program, especially if a single unit contains several; nevertheless, in our setting we can only expect each unit to be diagnosed as faulty once. Also, recall that we are not searching for faults in units not executed by at least one failed test case, simply because we have no evidence of them being faulty.

[3] We do not consider omission faults here. However, if UUTs are on the level of methods or higher, most omissions can be blamed on some executed unit.

[4] In fact, this is invariably the case if a single fault has been injected into the program under test that passed all test cases prior to the injection. Such is frequently done for the purpose of evaluating fault locators (cf. the discussions in Sections 3 and 6).

cuted by failed test cases). However, this explanation is not very satisfactory, since the difference of its vector representation, $(1 \;\; \dots \;\; 1)^T$, to the gold standard, the vector $\vec{f}$ from Table 1, is likely huge, and the actual number of faulty units is likely much smaller. In fact, any set $e \in \wp(U_F)$ such that

$$\forall t \in T_F : e \cap c(t) \neq \emptyset$$

is also an explanation of all failed test cases. We can therefore define the set of all explanations, E, as

$$E = \{\, e \in \wp(U_F) \mid \forall t \in T_F : e \cap c(t) \neq \emptyset \,\}$$

Note that E_1 and E_n are subsets of E, and that generally E is partitioned into n subsets E_k defined as

$$E_k = \{\, e \in E \mid |e| = k \,\} \quad \text{for each } 1 \leq k \leq n$$

In the worst case, i.e., when each test case covers all units,

$$|E_k| = \binom{n}{k} \quad \text{for all } 1 \leq k \leq n$$

and

$$|E| = 2^n - 1$$

which means that the set of explanations equals the set of potential explanations. In practice, however, we are likely to have fewer explanations.

It is important to note that for any explanation $e \in E$, e' such that $e \subset e' \subseteq U_F$ is necessarily also an explanation, since assuming additional units as faulty does not invalidate an explanation. Conversely, if e is no explanation, no subset of e can be. We will exploit this regularity for our algorithmic considerations in Section 4.4.

To illustrate matters, consider the following table of three failed test cases covering three units of P (zeros have been omitted for better readability):

U_F	T_F		
	t_1	t_2	t_3
u_1	1	1	
u_2		1	1
u_3			1

The set of explanations of these findings is

$$\{\, \{u_1, u_2\}, \{u_1, u_3\}, \{u_1, u_2, u_3\} \,\}$$

while the elements of

$$\{\, \{u_1\}, \{u_2\}, \{u_3\}, \{u_2, u_3\} \,\}$$

are ruled out as explanations (none of them explains all failures).

4.2. Deriving evidence of individual faultiness from the set of explanations

At first glance, it may seem that collecting *all* explanations does not help with localizing faults, since assuming that all program units are faulty is always an explanation (so that every unit is a member of at least one explanation). However, some units may be member of more explanations than others: for instance, in the above example

$$|\{e \in E \mid u_i \in e\}|$$

is 3 for u_1 and 2 for u_2 and u_3. If we normalize the results by the total number of explanations, $|E|$, we get the relative frequencies of the participation in explanations of each unit; in fact,

$$L(u_i) := \frac{|\{e \in E \mid u_i \in e\}|}{|E|} \tag{1}$$

is a fault locator (cf. Table 1) producing 1, 2/3, and 2/3 for u_1, u_2, and u_3, respectively, which directs us to inspecting u_1 for faultiness first, and second either u_2 or u_3.[5]

It is instructive to see how this result differs from the usual "horizontal" evaluation of coverage-based fault locators, according to which (in absence of information regarding passed test cases) u_1 and u_2 are equally more likely to be faulty than u_3 (they both participate in two failed test cases, whereas u_3 participates in only one). This ignores that u_1 must be faulty (or otherwise t_1 would not fail), while u_2 need not (u_3 can fail instead). Also, note that while assuming u_2 to be faulty does explain t_2 and t_3, nothing we know of suggests that the faultiness of u_2 shows in both test cases. Therefore, unless we know that the number of faults is limited to 2, we must also take the possibility that all three units are faulty into consideration; otherwise, we would have missed a possible actual fault combination. This logical reasoning is inherent in our fault locator.

At first glance, it may seem unfair that participation in explanations with small cardinalities is given the same weight as participation in explanations with large cardinalities (such as U_F): in our computation of relative frequencies (Eq. (1)), all participations count equally. However, as noted above participation in an explanation implies participation in all supersets, which are also necessarily explanations, so that membership in small explanations automatically yields high counts.

[5] In fact, we can derive from the solution set E that there must be at least two (the cardinality of the smallest explanation) faults in P, one being u_1. However, we leave exploitation of this fact, which in the general case is less trivial than one might expect, to another paper.

In fact, if a single unit explains all failed tests, its relative frequency is guaranteed to be greater than 1/2.[6]

4.3. Exploiting the probability distribution of the number of faults

Adding up membership in explanations as above implicitly assumes that all explanations are equally likely: given that there are $2^n - 1$ potential explanations, the probability of each is $(2^n - 1)^{-1}$. The number of faults is therefore distributed binomially: the probability that there are k faulty units is

$$\binom{n}{k} \cdot \frac{1}{2^n - 1}$$

(which sums up to 1 over all $1 \le k \le n$). However, this probability assignment is a prior one, ignoring that some of the potential explanations are de facto no explanations of the observations. Also, the binomial distribution may not model the actual distribution of the number of faults adequately — other distributions may be more realistic. Given any such distribution of the number of faulty units, $P_F(k)$, and assuming that alternative explanations with same cardinality (i.e., with the same number of faulty units) are equally probable, we can compute the probability of a unit u_i as being faulty (in the above sense, i.e., that fixing it is a prerequisite to making test cases pass), $P_U(u_i)$, as

$$P_U(u_i) = \sum_{1 \le k \le n} \sum_{e \in E_k : u_i \in e} \frac{P_F(k)}{|E_k|}$$

This has a couple of interesting implications:

1. Given the distribution $P_F(1) = 1$ (implying that $P_F(k) = 0$ for $k > 1$), P_U of any unit u_i in the above example is 0, meaning that there is no explanation conforming to P_F (which was saying that there is precisely one fault, which we know is false).
2. Given the distribution $P_F(2) = 1$, we get $P_U(u_1) = 1$ and $P_U(u_2) = P_U(u_3) = 1/2$: knowing that there are precisely two faulty units, we can conclude that u_1 and one of u_2 and u_3 (with equal probability) must be faulty.

However, while the above two distributions lead to results that are intuitive, the following does not: assuming a fault distribution (1/4, 1/2, 1/4) for fault numbers (1, 2, 3) yields 3/4 for u_1 and 1/2 for both u_2 and u_3. This is contrary to the fact that u_1 is contained in all explanations and therefore must be assumed to be faulty, i.e., $P_U(u_1) = 1$.

[6] This is so because every member of a base set is an element of half of all members of the base set's powerset, and because the empty set is excluded from all counts.

The error, again, is in the assumption of a prior probability distribution that is inconsistent with our observations: given that we already know that there is no explanation with cardinality 1, assigning this case a non-null probability does not model our findings adequately. Any prior probability distribution of the number of faults must therefore be turned into a posterior probability distribution that assigns impossible events zero probability. There are several ways to do this; perhaps the simplest is to override all probabilities of impossible events with zero and linearly scale the remaining probabilities so that they sum up to one:

$$P'_F(k) := \begin{cases} 0 & \text{if } E_k = \varnothing \\ P_F(k) \cdot \left(\sum_{1 \le j \le n : E_j \ne \varnothing} P_F(j) \right)^{-1} & \text{else} \end{cases}$$

This gives us

$$P_U(u_i) = \sum_{1 \le k \le n : E_k \ne \varnothing} \frac{\left| \{ e \in E_k \mid u_i \in e \} \right|}{\left| E_k \right|} P'_F(k) \quad (2)$$

In the above example, this leads to $P_F = (0, 2/3, 1/3)$, giving us the probabilities $P_U = (1, 2/3, 2/3)$, which is consistent with our observations (and intuition).

The observant reader will have noticed that P_U is now identical to the relative frequencies computed in Section 4.2. This is so because our above probability distribution P_F models equal probability of all possible explanations (1/3 for each), which was the underlying assumption of the calculations of Section 4.2. Other probability distributions will lead to other results.

Our fault model can be further improved by replacing the assumed equal probability distribution among explanations of same cardinality with one that takes other evidence into account. For instance, if one agrees that because it is covered by more failing test cases, u_2 is more likely to be faulty than u_3 (the reasoning of "horizontal" fault locators), we can conclude that $\{u_1, u_2\}$ is more likely the reason of failure than $\{u_1, u_3\}$. In fact, we could replace the probability distribution of the number of faults and its breaking down to explanations by a probability distribution of potential explanations derived from any other source; our approach would still add some value, since it selects the possible combinations and combines them to compute individual probabilities. However, the derivation and discussion of such probability distributions is not our topic here.

4.4. Mastering complexity

The problem with our approach is its inherent complexity: the number of potential explanations is exponential in the number of program units covered. Computing the intersection of each potential explanation with the coverage of each test case adds a linear factor, leading to a total complexity of

$$O\left(2^{|U_F|} \cdot |T_F|\right)$$

for a naive implementation (where the cost of set intersection is assumed to be constant). Relief comes from various practical considerations.

4.4.1. Partitioning. Test suites can fall into partitions that can be treated completely independently of each other. In particular, if we are able to partition T_F into p mutually disjoint subsets $T_1, \ldots, T_p$ such that

$$c(T_i) \cap c(T_j) = \varnothing \quad \text{for all } 1 \le i < j \le p$$

we can treat each $(T_i, c|_{T_i})$ as its own fault localization problem whose solution is completely independent of all others. Note that such a partition, if it exists, is unique and can be computed in time quadratic in the number of test cases. A simple algorithm doing this is the following:

```
function partitions begin
    partitions := $\varnothing$
    for each $t$ in $T_F$ do begin
        newPart := { $t$ }
        for each part in partitions do
            if $c(part) \cap c(newPart) \neq \varnothing$ then begin
                newPart := newPart $\cup$ part
                partitions := partitions \ { part }
            end
        partitions := partitions $\cup$ { newPart }
    end
end
```

where each partition is a set of test cases and c is the coverage function from above.

The division into partitions has the potential to cut down complexity considerably: if we obtain p partitions with coverage sizes $|c(T_i)|$, we get a search space of

$$2^{|c(T_1)|} + \ldots + 2^{|c(T_p)|}$$

instead of

$$2^{|c(T_F)|}$$

for the original problem. Accordingly, if each partition i has m_i explanations, we get a total of

$$m_1 + \ldots + m_p$$

explanations, instead of

$$m_1 \cdot \ldots \cdot m_p$$

as for the original problem (which enumerates all possible combinations of independent, partial explanations). Last but not least, due to the complete independence, fault localization in partitions can be performed in parallel, speeding the process up further on today's (and even more so tomorrow's) hardware.

4.4.2. Pruning. Despite the potential for considerable complexity cut-downs through partitioning, within each partition we are left with exponential complexity. However, here we can take advantage of the fact that if some $e \subset U_F$ is not an explanation, none of its subsets is, either (see Section 4.1). We can exploit this to compute solutions using a simple branch-and-bound algorithm, which is defined as follows.

Let $\vec{u}_i$ represent the test cases executing u_i, i.e., ith row vector of a table such as Table 1 (e.g., $\vec{u}_1 = (1 \quad 1 \quad 0)^T$ in the above example). Let further $\vec{T}_F$ be the vector that contains for every failed test case t the number of units it covers, $|c(t)|$, so that

$$\vec{T}_F = \sum_{1 \le i \le n} \vec{u}_i$$

In the above example, $\vec{T}_F = (1 \quad 2 \quad 2)^T$. The set of all explanations, E, is then computed by calling the recursive procedure *search* with actual parameters $\vec{T}_F$, U_F, and U_F, where *search* is defined as follows:

```
procedure search($\vec{T}$, $U$, $L$) begin
    $E := E \cup \{U\}$
    for each $u_i$ in $L$ do
        if $\min(\vec{T} - \vec{u}_i) = 0$ then
            $L := L \setminus \{ u_i \}$
    for each $u_i$ in $L$ do begin
        $L := L \setminus \{ u_i \}$
        search($\vec{T} - \vec{u}_i$, $U \setminus \{ u_i \}$, $L$)
    end
end
```

where L is the list of UUTs to be tried for subtraction, E is initialized to the empty set and where min selects the least element of a vector.

The algorithm starts with the largest explanation, U_F, and branches for smaller potential explanations created by subtracting single units. It bounds when it detects that such a potential explanation cannot explain all failed test cases, which is the case whenever the number of remaining potential explanations for any single test case, $\min(\vec{T} - \vec{u}_i)$, has reached zero. The bookkeeping in L saves *search* from retrying to subtract UUTs in lower levels, and also from finding duplicate solutions (by avoiding different permutations of the same set of subtractions). It is easy to see that *search* always terminates: at least one element of $\vec{T}_F$ is decremented in every step until the first reaches zero. In fact, the complexity of the algorithm is $O(n \cdot |E|)$; however, in the worst case, the algorithm never bounds, which is the case when every test case covers every method (so that the corresponding table is completely filled with 1s and $|E| = 2^n$). Generally, the more densely the table is populated, the more solutions are there, and the longer the search will take to complete.

5. Implementation

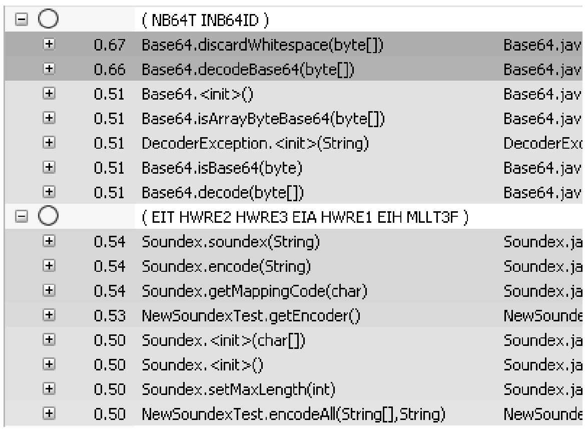
We have implemented the fault locator described here as a plugin to our fault localization framework EzUnit [4][16]. EzUnit provides the infrastructure for a priori and a posteriori fault locators, by granting them access to the source code of the program as well as to the traces and results of regression tests, and by accepting the results of each installed fault locator, allowing their combination with the results of other fault locators, and presenting them to the developer in various forms. EzUnit is itself a plugin to the Eclipse IDE and uses JUnit as its regression testing framework. It provides a specially designed fault localization view that allows immediate access to all identified fault locations, in the order of their likelihood. Traces of executed test cases are collected using JIP[7]. Currently, EzUnit supports only methods as UUTs.

To give an impression of how our fault locator plugged into EzUnit works in practice, Figure 1 shows a screenshot of its application on a real project, Apache Commons Codec 1.1, which contained three bugs that were unveiled by the test suite of its successor version 1.2 [16]. The assumed probability distribution of the number of faults is the one of Section 4.2, i.e., we have simply computed the relative frequencies of participation in explanations.

The actual bugs in version 1.1 were the following (see [16] for a more detailed description):

1. Method decodeBase64 called the inappropriate method discardWhitespace, which was replaced by calling discardNonBase64 (fix covered by two new test cases).
2. Methods getMappingCode and soundex in class Soundex were flawed, which required substantial changes to both methods, plus the introduction of several new ones (fix covered by six new test cases for the original methods).
3. Method soundex of class Soundex used a bad loop bound. The documentation of the bug and its fix leaves it open whether choosing the wrong variable for the loop bound or being able to set this variable (through method setMaxLength) was the flaw.

[7] http://jiprof.sourceforge.net/



```
⊟ ○        ( NB64T INB64ID )
   ⊞  0.67  Base64.discardWhitespace(byte[])         Base64.jav
   ⊞  0.66  Base64.decodeBase64(byte[])              Base64.jav
   ⊞  0.51  Base64.<init>()                          Base64.jav
   ⊞  0.51  Base64.isArrayByteBase64(byte[])         Base64.jav
   ⊞  0.51  DecoderException.<init>(String)          DecoderExc
   ⊞  0.51  Base64.isBase64(byte)                    Base64.jav
   ⊞  0.51  Base64.decode(byte[])                    Base64.jav
⊟ ○        ( EIT HWRE2 HWRE3 EIA HWRE1 EIH MLLT3F )
   ⊞  0.54  Soundex.soundex(String)                  Soundex.ja
   ⊞  0.54  Soundex.encode(String)                   Soundex.ja
   ⊞  0.54  Soundex.getMappingCode(char)             Soundex.ja
   ⊞  0.53  NewSoundexTest.getEncoder()              NewSounde
   ⊞  0.50  Soundex.<init>(char[])                   Soundex.ja
   ⊞  0.50  Soundex.<init>()                         Soundex.ja
   ⊞  0.50  Soundex.setMaxLength(int)                Soundex.ja
   ⊞  0.50  NewSoundexTest.encodeAll(String[],String) NewSounde
```

Figure 1. Screenshot of the results of our fault locator as presented through EzUnit. The UUTs are methods.

Our partitioning algorithm of Section 4.4.1 identified two independent fault localization problems, both shown in Figure 1. Within each partition, Eq. (1) produced the probabilities driving the ranking, using the algorithm of Section 4.4.2 to compute all possible explanations. Although the results cannot be generalized, it is encouraging to see that our fault locator ranks all actual fault locations highly.

6. Feasibility of our approach

Since the choice of the probability distribution of the number of faults influences outcome critically (cf. Section 4.3), it is impossible to systematically evaluate our fault localization procedure independently of any such distribution — invariably, the evaluation would be one of the adequacy of the assumed distribution as much as one of our method. What we could have done, though, is to evaluate our method together with a set of alternative fault distributions and measure how they all compare to other fault locators. However, the number of publically available benchmarks with multiple faults is quite limited (in fact, the most popular of evaluation benchmarks, the so-called Siemens suite, has only single faults and multi-fault versions of it are not standardized) so that there is no good evaluation basis we could compare our approach against.

On the other hand, we maintain that there is no magic in our fault localization method that needs to be justified empirically: fault localization is based on a simple fault model, namely that every failed test requires an explanation in the form of at least one of its UUTs assumed as being faulty, where it is possible that the same UUTs serve as explanations for several failed tests. Whether this is actually the case is generally unpredictable (that a UUT causes the failure of one test case does not imply that it does the same in another), and in any case would require further modelling. The only other assumption our approach relies on is that the fault must be among the UUTs — in languages such as JAVA, however, with complex binding rules depending, among other things, on visibility, it is possible that the fault is in a unit not executed: for instance, insufficient visibility of a method may have caused the wrong method to be executed, where the fault is on the side of the insufficiently visible method.

We consider the main threat to our fault locator to be its inherent complexity, which is exponential in the size of the number of UUTs. However, we have presented two algorithms with the potential to cut down complexity considerably, leaving the original problem size only for the worst case. The question, then, is, what can we expect in practice?

To get an impression of this, we have applied our fault locator to a selection of open source projects known to have good test coverage. Since all these projects are fault-free in the sense that all test cases pass, and since we want to measure performance in presence of multiple faults, we apply a simple trick and assume that all test cases fail. This gives us for each project the largest and most densely populated tables that could be caused by faults, leading to the smallest number of partitions in each case, so that our results are an upper bound of the complexity to be expected in any realistic multiple fault scenario in one of those projects.

Table 2 summarizes our findings of computing the partitions and explanations for the selected sample projects. As can be seen, all test suites fall into a number of partitions of moderate size. Since EZUNIT processes each partition in a separate thread (that can be aborted by the user), debugging can begin with the partitions for which all explanations have been computed. Note that we have omitted partitions with a single test case, since their explanations are trivial to compute.

As concerns time complexity, small partitions are explained quickly (milliseconds to a few seconds), whereas large ones are currently intractable (aborted after reaching 2^{24} explanations). While the latter was to be expected, it is interesting to see that $|E|$ tends to approach 2^{n-1} (where $n = |U_F|$) for large n, meaning that half of all potential explanations are actual explanations. With hindsight, this was also to be expected, since the larger the set U_F, the more the fraction of elements of $\wp(U_F)$ that contain an explanation (and thus are themselves explanations) approaches 0.5 (cf. Section 4.2). This has some important consequences for our future work (see below).

Table 2. Numbers of partitions (p) and their sizes (number of test cases, UUTs, and explanations) for selected test projects (only the p' partitions with more than one test case are shown). Note that all numbers are derived for the worst case, i.e., when all test cases fail.

APACHE COMMONS PROJECT	p/p'	PARTITION SIZES ($|T_F|$, $|U_F|$, $|E|$)
codec	4/3	(5, 3, 2), (7, 7, 32), (54, 41, $>2^{24}$)
lang.mutable	7/7	(5, 7, 62), (6, 10, 577), (14, 22, $2^{21.1}$), (14, 22, $2^{21.1}$), (15, 24, $2^{23.0}$), (15, 25, $>2^{24}$), (28, 43, $>2^{24}$)
mail	5/2	(3, 9, 504), (68, 306, $>2^{24}$)
lang.time	12/5	(2, 2, 2), (52, 13, $>2^{12.6}$), (16, 18, $2^{17.3}$), (7, 27, $>2^{24}$), (41, 145, $>2^{24}$)
lang.BooleanUtils	27/6	(2, 2, 2), (2, 2, 2), (2, 2, 2), (4, 4, 8), (9, 4, 4), (4, 4, 8)

We have omitted from Table 2 a number of other projects, whose partitioning led to several very small partitions and one big partition that was intractable. We have two possible explanations for this phenomenon:

1. Most regression test suites are no unit tests, but integration tests of overlapping parts of the system.
2. Most UUTs rely on a small number of helper units.

A partitioning algorithm accepting a small amount of overlap (i.e., some UUTs that are shared between partitions) might lead to clarification and better partitioning results. Such algorithms exist in the field of integer programming (cf. Section 7.1); however, we have not yet investigated this further.

7. Discussion

Our experiments suggest that our fault locator tends to look at unreasonably many possible explanations. As observed above, this does not only threaten tractability of our approach, it also dilutes the results (with all probabilities approaching 0.5). On the other hand, it is unreasonable to expect as many faults as there are UUTs — most realistic probability distributions of error numbers will approach 0 rather quickly, allowing us to cut off search for explanations early. This however would require a bottom-up search for explanations (starting with a single fault assumption). Pruning seems more difficult here, so that we leave it for future work.

Limiting the number of faults has another interesting effect: it lets us conclude that a unit is not faulty, that is, that it does not have to be fixed in order to make all test cases pass. For instance, given the prior probability distribution

$P_F(1) = 2/3$, $P_F(2) = 1/3$, $P_F(3) = 0$

and the fault localization problem

	t_1	t_2	t_3	t_4
u_1	1	1		
u_2		1	1	1
u_3				1

applying Eq. (2) we get

$P_U(u_1) = 1$, $P_U(u_2) = 1$, and $P_U(u_3) = 0$

meaning that u_1 and u_2 must be faulty, while u_3 cannot. Usually, we will not be able to specify the maximum number of faults with certainty, but if we give high numbers a low probability, low probabilities will be assigned to units that are not necessary to explain a fault.

7.1. Mathematical view

Mathematically, the coefficients $x_{1,1} \ldots x_{n,m}$ of a fault localization problem form a binary matrix with rows representing program units and columns representing test cases. The problem of partitioning (Section 4.4.1) is that of transforming the matrix into a diagonal block matrix through suitable permutations of rows and columns:

$$\begin{bmatrix} x_{1,1} & \cdots & x_{1,m_x} & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ x_{n_x,1} & \cdots & x_{n_x,m_x} & 0 & \vdots & & \vdots \\ 0 & \cdots & 0 & \ddots & 0 & \cdots & 0 \\ \vdots & & \vdots & 0 & z_{1,1} & \cdots & z_{1,m_z} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & z_{n_z,1} & \cdots & z_{n_z,m_z} \end{bmatrix}$$

Each block in this matrix is itself a matrix that can be viewed as an independent fault localization problem (one that is more densely populated with 1s).

The search for explanations of failed test cases (Section 4.4.2) translates to a search for sets of rows of the matrix (or each block of the matrix) such that there is at least one 1 in each column. This also a well-known mathematical problem (with many applications in binary integer programming), called *covering in hypergraphs* [15]. Although we have not yet delved into the details, there appear to be numerous heuristics speeding up the search of solutions considerably, especially if matrices are sparsely populated. Also, there are algorithms that accept a small amount of "dirt", i.e., that separate out a few rows on the right that spoil an otherwise "clean" block structure.

7.2. Logical view

Our fault localization problem can also be formulated in terms of propositional logic, more specifically as a problem of model-based diagnosis [8][18]. The model is simple and constructed as follows.

Assume that for each UUT u_i we have a proposition, $p(u_i)$, expressing that u_i is faulty. For each test case t_j in Table 1 with $1 \leq j \leq m$ the model of its failure is the clause

$$\bigvee_{i:x_{i,j}=1} p(u_i)$$

i.e., the test case fails if and only if at least one of the units u_i it covers is faulty (i.e., $p(u_i)$ is true). Given that the clauses so derived for each column must all be true (recall that we are looking exclusively at failed test cases here), we get a clause set (in conjunctive normal form) that represents a satisfiability (SAT) problem. It has a trivial solution, namely that all $p(u_i)$ are true (meaning that all u_i are faulty), which explains all test failures, but again is not likely to be a good fault locator (in the sense that $\vec{p}$ will not be satisfactorily similar to $\vec{f}$, unless of course all u_i are in fact faulty). The set of all solutions is precisely the set of our solutions, *E*.

Although the SAT problem (and with it our fault localization problem) are known to be NP-complete, algorithms exist that can solve problems with thousands of literals and clauses in acceptable time. We have not yet attempted to adapt any of the available algorithms to our specific problem, though.

7.3. Related work

Our approach to fault localization belongs to the ones based on program spectra, more specifically on hit spectra or test coverage. It seems rather similar to that of TARANTULA [11] and others ([6][7], as compared, e.g., in [1]): the information required to compute fault locations is derived from a table such as Table 1. However, our approach is substantially different in that it takes coverage information of each test case (the columns of Table 1) into account, allowing it to deduce that a UUT must be faulty, which approaches exploiting only the data in rows (the similarity coefficients of [1]) cannot. Also, our method is designed from bottom up to cope with arbitrarily many faults, accepting a possibility distribution of the number of faults as additional input helping to constrain its possible diagnoses.

Work on spectra based fault localization in presence of multiple faults is harder to find. In his Ph.D. thesis [11], Jones showed how multiple faults can dilute the results of Tarantula's fault locator. He presented two heuristics to partition test suites into clusters that, when performing fault localization on each, lead to clearer suspects. However, both heuristics depend on thresholds that need to be tuned for performance and there is no guarantee that different clusters favour different suspects. By contrast, our approach to partitioning is strictly logical and guarantees that the sets of possible fault locations suggested by each partition are disjoint. Unfortunately, this means that depending on the test suite, we may fail to identify different partitions even in presence of multiple faults.

Another recent coverage-based approach assuming multiple faults uses machine learning to obtain what the authors call a rule-based statement ranking (RUBAR) [3]. It requires a prior test case classification in terms of abstract categories of test data as input to the machine learning algorithm, which then produces rules predicting the passing or failure of concrete test cases. The idea of RUBAR is that each learned rule represents a different cause of failure, leading to the assumption of multiple faults. The likelihoods of being faulty for statements covered by the test cases classified by a rule are then computed separately, and results weighted and combined to yield an aggregate likelihood of each statement. By contrast, our approach does not require a prior categorization of test data, but rather derives the minimum number of faults from a simple fault model.

Wotawa et al. have shown how a more knowledgeable model of the program under test can help debug it [18]. As is common for model-based diagnosis [8], they use a component model for statements and expressions and role out iterations to receive a model that is loop-free. Like we do, they compute diagnoses as explanations of observed failures of test cases, where an explanation can contain several assumed fault locations. We believe that their approach has certain scalability problems, though, namely that it will be difficult to extend it to a complete language specification, and that using it will be impractical for larger programs.

8. Conclusion

While more and more approaches to software fault localization are being published, most still rely on an implicit single-fault assumption. By contrast, we have defined a fault locator that takes all possible fault combinations explaining a given set of findings (failed test cases) into account. We master combinatorial complexity by partitioning the problem space and by exploiting certain simple regularities in the solution set. First experiments we have conducted suggest that this can make our approach feasible in practical applications.

9. Acknowledgements

The authors are indebted to Winfried Hochstättler for pointing out the relationship to integer programming and its algorithms.

10. References

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

[2] VR Basili, LC Briand, WL Melo "A validation of object-oriented design metrics as quality indicators" *IEEE Trans. Software Eng.* 22:10 (1996) 751–761.

[3] LC Briand, Y Labiche, X Liu, "Using machine learning to support debugging with Tarantula" in: *ISSRE* (2007) 137–146.

[4] P Bouillon, J Krinke, N Meyer, F Steimann "EzUnit: A framework for associating failed unit tests with potential programming errors" in: *XP* (2007) 101–104.

[5] P Chalin, JR Kiniry, GT Leavens, E Poll "Beyond assertions: Advanced specification and verification with JML and ESC/Java2" in: *FMCO* (2005) 342–363.

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

[7] V Dallmeier, C Lindig, A Zeller "Lightweight defect localization for Java" in: *ECOOP* (2005) 528–550.

[8] J de Kleer, J Kurien "Fundamentals of model-based diagnosis" in: *SafeProcess* (2003).

[9] T Gyimothy, R Ferenc, I Siket "Empirical validation of object-oriented metrics on open source software for fault prediction" *IEEE Transactions on Software Engineering* 31:10 (2005) 897–910.

[10] D Hovemeyer *Simple and Effective Static Analysis to Find Bugs* (PhD Thesis, University of Maryland, 2005).

[11] JA Jones *Semi-Automatic Fault Localization* PhD Thesis (Georgia Institute of Technology, 2008).

[12] S Kim, T Zimmermann, EJ Whitehead Jr., A Zeller "Predicting faults from cached history" in: *ICSE* (2007) 489–498.

[13] C Liu, X Yan, L Fei, J Han, SP Midkiff "SOBER: statistical model-based bug localization" in: *ESEC/SIGSOFT FSE* (2005) 286–295.

[14] N Nagappan, T Ball, A Zeller "Mining metrics to predict component failures" in: *ICSE* (2006) 452–461.

[15] A Shriver "Covering and antiblocking in hypergraphs" Chapter 82 of: *Combinatorial Optimization: Polyhedra and Efficiency* (Springer, 2003).

[16] F Steimann, T Eichstädt-Engelen, M Schaaf "Towards raising the failure of unit tests to the level of compiler-reported errors" in: *TOOLS Europe* (2008) 60–79.

[17] M Störzer, BG Ryder, X Ren, F Tip "Finding failure-inducing changes in Java programs using change classification" in: *SIGSOFT FSE* (2006) 57–68.

[18] F Wotawa, M Stumptner, W Mayer "Model-based debugging or how to diagnose programs automatically" in: *IEA/AIE* (2002) 746–757.