Towards Raising the Failure of Unit Tests to the Level of Compiler-Reported Errors

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

Thomas Eichstädt-Engelen
LG Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
te@eichstaedt.net

Martin Schaaf
101tec.com
Mansfelder Straße 13
D-06108 Halle
Martin.Schaaf@feu.de

Abstract. Running unit tests suites with contemporary tools such as JUNIT can show the presence of bugs, but not their locations. This is different from checking a program with a compiler, which always points the programmer to the most likely causes of the errors it detects. We argue that there is enough information in test suites and the programs under test to exclude many locations in the source as reasons for the failure of test cases, and further to rank the remaining locations according to derived evidence of their faultiness. We present a framework for the management of fault locators whose error diagnoses are based on data about a program and its test cases, especially as collected during test runs, and demonstrate that it is capable of performing reasonably well using a couple of simple fault locators in different evaluation scenarios.

Keywords. *Regression testing, Debugging, Fault localization.*

1 Introduction

Continuous testing [7, 23, 24] is a big step forward towards making the detection of logical errors part of the edit/compile/run cycle: whenever a resource has been edited and saved, it is not only checked by the compiler for absence of syntactic and semantic (i.e., type) errors, but also — by running all relevant unit tests — for absence of certain logical errors. However, presently a failed unit test is presented to the programmer as just that — in particular, no indication is given of where the error is located in the source code. By contrast, the compiler names not only the syntactic or semantic rule violated by an incorrect program, it also tries to point the programmer to the place in the source code where the error occurred. Would not the same be desirable for logical errors detected by failed unit tests?

A first simple approach to solving this problem explicitly links each test case to one or more methods under test (MUTs). Whenever a test case fails, the reason for the failure must be sought among the designated MUTs. Using JAVA and JUNIT, test cases (which are implemented as methods in JUNIT) can be linked to MUTs via a corresponding method annotation. An integrated development environment (IDE) such as ECLIPSE can then be extended to link failed unit tests directly with potential error locations in the source code.

In our own prior work, we have implemented this approach as a plug-in to ECLIPSE, named EZUNIT [4, 19]. It supports the programmer with creating and maintaining `@MUT` annotations by performing a static call graph analysis of test methods, producing all candidate MUTs. A filter can be set to exclude calls to libraries (such as the JDK) or other parts of a project known or assumed to be correct, minimizing the set of potential MUTs to select from; the annotations themselves can be used for quick navigation between a test and its MUTs and vice versa. Whenever a unit test fails, the corresponding MUTs are marked by a red `T` in the gutter of the editor, and a corresponding hint is shown in ECLIPSE's Problem view [4, 19]. The programmer can thus navigate directly to potential fault locations in the source code, much like (s)he can for compiler-reported errors. However, the setting up and maintenance of `@MUT` annotations is tedious and error-prone.

In this paper, we describe how we advanced our work on logical error spotting (hereafter referred to as *fault localization*) by automatically narrowing down the set of possible fault locations using information that is present in the program and its tests. In particular, we

- present a framework that can incorporate arbitrary a priori fault predictors (such as program metrics) into its reasoning,
- show how information obtained from the execution of unit test suites can be used to compute a posteriori possibilities of fault locations, and
- demonstrate how making precise control flow information available can increase the accuracy of fault localization.

Especially for the utilization of precise control flow information, which is usually expensive — if not impossible — to obtain, we exploit a characteristic property of JUNIT test cases, namely that their traces do not change unless either a called method or a test case itself is changed. This allows us to record the trace of a test run once (using an available program tracing tool) and keep it until invalidated by a change of source code. That this is worth its effort, and more generally that our approach to fault localization is feasible, is shown by a systematic evaluation which is also presented.

The remainder of this paper is organized as follows. First, we describe the problem we are trying to solve, arguing why we believe that it is indeed a relevant problem. In Section 3 we present a number of fault locating strategies we have implemented and tested our framework with. In Section 4 we show the results of a systematic evaluation of the fault locators, which uses historical bugs found in archives of open source code bases, error seeding to inject faults in correct programs, and practical experience of professional programmers. Section 5 describes the architecture of EZUNIT as an ECLIPSE plugin, and how it offers extension points for additional fault locators. Other fault locators we considered, but did not implement, are presented in Section 6; a discussion with related work concludes our presentation.

2 Problem

When first presenting our approach to practitioners, we heard objections of the kind "when I see a failed unit test, I know exactly where to look for the error". While this

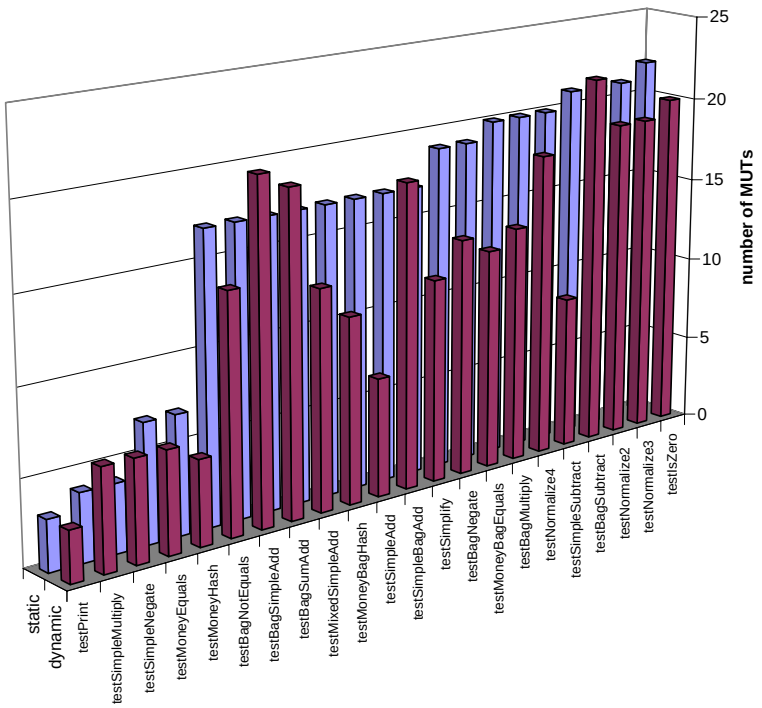


Figure 1. Number of methods called by each unit test of JUNIT's Money example, as derived from tracing (dynamic, front row) and a program analysis (static, rear row). Both direct and indirect calls are counted. See Section 5.1 for an explanation of the differences.

may be true in certain situations (for instance when only few changes were made since the last successful run of a test suite), we doubted the general validity of the claim and measured the number of methods called in the course of the execution of test runs. Figure 1 shows the number of methods called by each test case of a simple test suite (the Money example from the JUNIT distribution). Considering that the test cases contained in MoneyTest are examples for beginners, the common assumption that test cases are usually so simple that there is no doubt concerning the reason for a failure is clearly relativized.

In a way, the problem of blame assignment for failed unit tests is like the problem of medical diagnosis (or any kind of diagnosis for that matter): a single or a set of symptoms must be mapped to possible diagnoses, where the only causal knowledge available is the mapping of diagnoses to symptoms. Transferred to testing: a single or a set of failed unit tests must be mapped to possible fault locations, where the faults and their locations cause the unit tests to fail. Unfortunately, this causality (as a mapping) is only seldom injective.

EZUNIT can be viewed as attempting such a diagnosis. However, its first version accommodated only for binary answers: a method is either included as a possible reason for failure, or it is not. With no other information given, the developer has to look

at a whole (and unordered) set of locations in order to find and fix an error. For instance, as suggested by Figure 1 the number of methods to be considered (if no filters or specific `@MUT` annotations are set) is 15.6 (static) and 13.1 (dynamic) on average.

Fortunately, there is more information in a program and its test cases, so that heuristics can be developed that rate one fault location as more likely than another. For example, complexity measures can be used to assign a priori probabilities of faultiness to methods; information from past test runs, error reports, and error fixes can be collected and evaluated using statistical methods; and so forth. In order to exploit such information, the binary fault localization framework of EZUNIT must be extended to collect graded evidence, to combine individual results (as delivered by different fault locators in use) by appropriate operators, and to present its suggestions to the developer in adequate form.

3 Fault Locators

With the term *fault locator* we mean an algorithm that computes the possibility of a piece of source code containing a logical error. We use the term *possibility* here in an informal sense as a measure of uncertainty distinct from probability. In particular, we do not require the computed possibilities for exclusive alternatives to add up to 1, as probability theory dictates — indeed, it is entirely feasible that several mutually exclusive fault locations have a possibility of 1, meaning that it is completely possible for either to host the error. However, just like a probability of 0, a possibility of 0 definitely rules out fault locations, and higher possibility values are indicative of stronger support for a fault location in the available data. Thus our possibilities could be interpreted as possibilities in the sense of possibility theory [9]; yet, we have no pretensions of being formal here.

Fault locators always have a granularity associated with them [15, 26]. Generally, granularity levels can range from the whole program to a single expression; yet the extremes are rather useless in most practical settings — knowing that there is an error in the program helps only little with finding it, and having hundreds of statements with non-zero possibilities of being fault locations (which is what is to be expected with most location strategies currently available) is not helpful either. For EZUNIT, we constrain ourselves to defining fault locators that compute possibilities for whole methods rather than single statements or lines in the source code. Besides saving the developer from being flooded with uncontrollable amounts of data, this allows us to utilize certain information that can be assigned to whole methods, but not to single statements.

Clearly, only methods that are actually called by a unit test can contribute to its success or failure.[1] If non-execution of a method was the reason for the failure of a test, then this must be ascribed to an executed method (including the test case itself) whose fault it was that the method did not get called. On the other hand, if non-

[1] In JAVA, code can exist outside of methods, for instance in variable initializers. We consider this code as part of the constructors of the class, which are called for test object creation.

execution of a method was the reason for the success of a test (because it was faulty and should have been called, but was not), then there is no way to detect this (unless the uncalled method was listed in the test's `@MUT` annotation as described in the Introduction; however, we do not pursue this fault detection strategy, which was detailed in [4, 19], here).

One might deduce from this that only methods actually called by a failed unit test, and thus only failed unit tests, need be analyzed. However, as will be seen, some fault locators exploit the information that certain unit tests have passed, and which methods were called by these tests.

The evaluators we have devised and experimented with can be divided into two categories. The first category relies on prior possibilities, i.e., on evidence that is independent of the outcome of testing and of the fact whether a method was called by a test case that failed. The second category assumes no prior possibilities of faultiness, but instead collects information from the actual successes and failures of a test run.

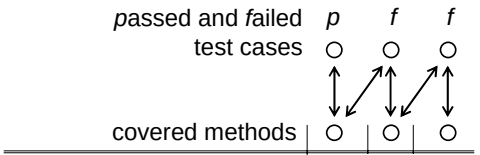
3.1 Fault Locators Based on Prior Knowledge

Assuming that some methods of a program are more error-prone than others, there is a prior possibility of faultiness. This prior possibility can be a derived property of a method (for instance its complexity) or it can be ascribed by extraneous factors, such as its author's confidence in its correctness, or the number of fixes it already needed. Note that in our setting it makes little sense to asses a priori possibility of error freeness by determining test coverage: since tests can fail (and indeed our aim is to exploit the information gained from failed tests), we will not count good test coverage as an a priori sign of freeness from faults.

For the evaluation presented in Section 4 , we have used two simple complexity measures of methods: lines of code (LOC) and McCabe's cyclomatic complexity (CC), which measures the number of linearly-independent paths through a method (basically by counting the number of branches and loops, adding 1) [18]. To map these complexity measures to possibility values, we have normalized LOC by the longest method in the program, and bounded and normalized CC with 10. Note that this way, every method of a program is a possible fault location (i.e., its a priori possibility is greater than 0), but the possibility of simple methods (such as setters and getters) is rather low. Any other complexity measure could also have been used (and indeed such has been done by others; see Section 7), but since our primary focus is on exploiting information obtained from the execution of unit tests, and more generally on the presentation of EZUNIT as a framework that can combine any prior knowledge with evidence obtained ex post, LOC and CC should be seen as placeholders only.

3.2 Fault Locators Based on Posterior Knowledge

The prior possibility of a fault location is relativized by posterior information available after a test suite has been run: the possibility of being the reason for a failure of methods that were not called by a test case that failed drops to zero. Apart from this, with no other information given the ranking of possible fault locations as suggested



Fault Locator	Possibility		
Failure Ratio (FR)	0.5	1	1
Failure Accountability (FA)	0.25	1	0.5

Figure 2. A posteriori possibility measures taking number of passed and failed tests into account.

by the computed prior possibilities remains unchanged; in particular, of all methods contributing to failed tests, the one with the highest prior possibility of being the reason for failure is also the one with the highest posterior possibility.

However, running a test suite does not only rule out methods from having contributed to possible failures, it also provides its own graded evidence for methods as fault locations, which can be interpreted as independent posterior possibility (i.e., one that is *not* derived from prior possibility). The argumentation goes as follows.

As suggested by Figure 1, a test case usually calls several methods, and each method is called by several test cases. Methods that are called exclusively by test cases that pass cannot be held accountable for failures in the test suite, so that their possibility can be set to 0 (see above). At the opposite extreme, for a method that is called exclusively by failed test cases, there is no posterior evidence against its faultiness, so that the possibility can be set to 1. For all other cases, the possibility is a value between 0 and 1, for instance as calculated by the ratio of participations in failed test cases to the total number of participations in test cases. We call the corresponding measure *failure ratio* (FR) and define it formally as follows:

Let T be the set of test cases in a test suite, and M be the set of methods called by this suite. We then define a function c: $T \to \wp(M)$ (where is $\wp(.)$ stands for the powerset) such that $c(t)$ computes the set of MUTs for a $t \in T$, and say that test case t covers the methods in $c(t)$. We further define a pair of functions p: $M \to \wp(T)$ and f: $M \to \wp(T)$ such that $p(m)$ computes the set of passed test cases from T that cover m, i.e.,

$$p(m) = \{t \in T \mid m \in c(t) \wedge passed(t)\}$$

and $f(m)$ computes the set of failed test cases accordingly. The FR value for a method $m \in M$ is then defined as

$$\mathrm{FR}(m) = \frac{|f(m)|}{|f(m)| + |p(m)|}$$

As can be seen from Figure 2, FR is insensitive to the absolute number of failed tests that a MUT presumed as faulty can explain. In particular, it ranks a method that is called by all failed test cases and none that passed the same as one that is called by only one failed test case, where there are more test cases that failed. We have therefore devised a second fault locator that takes the contribution to total failures into account. This fault locator, which we call *failure accountability* (FA), is defined as

$$\mathrm{FA}(m) = \mathrm{FR}(m)\frac{|f(m)|}{\left|\bigcup_{m' \in M} f(m')\right|}$$

It yields a value of 1 for a method m if and only if all test cases covering m and no other test cases fail. Observe how FA ranks a method covered by one passed and one failed test below one covered by one failed test only, and both below one covered by all failed tests (Figure 2).

The computation of posterior possibilities as described above depends critically on the availability of information which methods have been called. As discussed in more detail in Section 5.1, the methods called in the course of a test run can be determined by a control flow analysis performed at compile time, or by tracing the program. As will be seen, the preciser the information is, the better is the expected result.

4 Evaluation

We have evaluated EZUNIT in a number of ways:

1. For archived versions of programs known to have bugs undetected by their accompanying test suites, we have applied EZUNIT using test suites from successor versions of the programs which made sure that the bugs had been fixed, and recorded how well each fault locator was able to spot the bugs.
2. We have used error seeding to inject errors into programs extensively covered by unit tests, and recorded how well each fault locator was able to spot the errors.
3. We have used EZUNIT in a commercial software development setting and observed its precision and usability.

For each kind of evaluation, we ranked the possible fault locations according to their possibility values as computed by EZUNIT separately for each of the four fault locators described above. Assuming that faultiness of n MUTs with same possibility values is equally distributed, we computed their rank (relative to their predecessors) as $(n + 1)/2$ so that the rank always represents the *expected* number of method lookups required until the fault is found.

4.1 Evaluation Based on Flawed Historical Releases

Prior to JUNIT 3.8, its `Money` example had a small bug: when comparing two money bags with unequal currencies for equality, a null pointer exception was raised. JUNIT 3.8 added a test case `testBagNotEquals()` unveiling the error, and fixed it. Incidentally

Table 1. Number of locations to search before fault is found (halves are due to same possibility, and thus equal ranking, of several methods). Possib. locat. counts the methods called by test cases that failed (thus being fault locations to be taken into consideration given the test suite).

Project		Fault Location (Method)	No. of Tests		Possib. Locat.		Rank (Expected Necessary Lookups)							
							LOC		CC		FR		FA	
			p	*f*	stat	dyn	stat	dyn	stat	dyn	stat	dyn	stat	dyn
JUnit 3.7 Money		contains	21	1	23	13	8½	4½	8½	4½	1½	2½	1½	2½
JUnit 4.3		format	294	4	8	5	2	2	1	1	6½	2	6½	2
BeanUtils 1.6		copyProperty	338	6	196	29	2½	1	1	3½	8	3	2	1
Apache Commons Codec 1.1	before	soundex	62	9	14	14	2	2	2½	2½	4	5	2	2
	after 1st fix		68	3	15	9	2	1	2	1	12½	7	12½	6½
	after 2nd fix		70	1	9	9	1	1	1	1	6½	6½	6½	6½
	before	getMapping-Code	62	9	14	14	7	7	7½	6½	4	5	2	2
	after 1st fix		68	3	15	9	5	3	4½	3	12½	7	12½	6½
	after 2nd fix		70	1	9	9	3	3	3	3	6½	6½	6½	6½
	before	setMax-Length	62	9	14	14	11½	11½	11½	11½	4	5	6	7
	after 1st fix		68	3	15	9	12	6½	12	6½	1	1	1	1
	after 2nd fix		70	1	9	9	6½	6½	6½	6½	1	1	1	1
	before	decode-Base64	62	9	14	14	1	1	1	1	13½	13	11½	14
	after 1st fix		68	3	15	9	1	1	1	1	8½	5	6½	5
	after 2nd fix		70	1	9	9	–	–	–	–	–	–	–	–
	before	discard-Whitespace	62	9	14	14	3	3	7½	6½	13½	14	11½	14
	after 1st fix		68	3	15	9	3	3	7½	6½	8½	4	6½	3½
	after 2nd fix		70	1	9	9	–	–	–	–	–	–	–	–

(and contrary to the claims mentioned in Section 2), the location of the actual error was less than obvious from the failed test case: in class MoneyBag, method

```
private boolean contains(Money aMoney) {
  Money m= findMoney(aMoney.currency());
  return m.amount() == aMoney.amount();
}
```

lacked a test for m being not null, but this method is only one out of 14 methods invoked by the test case (cf. Figure 1).

Running the test suite of MoneyTest from JUnit 3.8 on the flawed implementation of the Money example from JUnit 3.7, the fault locators based on a priori possibility presented in Section 3 (LOC and CC) perform rather poorly (see Table 1, first row). Indeed, as it turns out the problem of the method was that it was too short and its cyclomatic complexity too low: in JUnit 3.8, the flaw was fixed by inserting

```
if (m == null) return false;
```

in the method body. By contrast, the fault locators based on posterior information (passed and failed tests cases) ranked the flawed method highly.

In another example, a flaw was detected in JUnit 4.3 in the method static String format(String, Object, Object) of class org.junit.Assert: the contained lines

```
String expectedString= expected.toString();
String actualString= actual.toString();
```

cause a null pointer exception whenever expected or actual are null. JUNIT 4.3.1 had this fixed by writing

```
String expectedString= String.valueOf(expected);
String actualString= String.valueOf(actual);
```

instead. The fault is covered by four new test cases, which all fail when executed on JUNIT 4.3. Copying these test cases to the test suite of JUNIT 4.3 and running it with EZUNIT produced the result shown in the second row of Table 1: the culprit method was ranked 1st, 2nd and 6½th out of 8 statically and 1st and 2nd out of 5 dynamically. Note that the performances of FR and FA are poor in the static case because format is included in the static call graph of several passing test cases even though it is factually not called by these cases, which has a lowering effect on FR and FA (cf. Section 3.2).

In a third example, APACHE's BeanUtils 1.6.1 fixed a bug of its predecessor, version 1.6, in method copyProperty(Object, String, Object) from class BeanUtils (problem and fix are complex and not detailed here). Running the test suites of version 1.6.1 on the source code of version 1.6 with JUNIT produces three errors and three failures; running them with EZUNIT ranks copyProperty highly among the large number of possible fault locations (see Table 1). The perfect hit of FA is due to the fact that copyProperty is the only method whose faultiness can explain all failed tests (i.e., that was called by all failed test cases). Again, this result is diluted by static call graph computation, which includes methods not called.

All previous examples have in common that they contain only a single bug, which is unveiled by one or more test cases designed to detect this one bug. To see how our approach performs when there are more bugs we resort to a final example.

The APACHE Commons Codec 1.2 fixed the following list of bugs of its predecessor, version 1.1 [2]: [2]

1. Modified Base64 to remedy non-compliance with RFC 2045. Non-Base64 characters were not being discarded during the decoding.
2. Soundex: The HW rule is not applied; hyphens and apostrophes are not ignored.
3. Soundex.setMaxLength causes bugs and is not needed.

The bugs manifest themselves as follows:

1. Class binary.Base64 in version 1.1 contained the method

```
public static byte[] decodeBase64(byte[] base64Data) {
  // RFC 2045 suggests line wrapping at (no more than) 76
  // characters -- we may have embedded whitespace.
  base64Data = discardWhitespace(base64Data);
  …
```

   which was replaced by

```
public static byte[] decodeBase64(byte[] base64Data) {
  // RFC 2045 requires that we discard ALL non-Base64 characters
  base64Data = discardNonBase64(base64Data);
  …
```

   in version 1.2. The flaw is covered by two new test cases in class Base64Test.

[2] There are in fact two more [2], but one appears to be not one of version 1.1 (which did not include the methods said to be fixed), and the other (use of a number literal instead of a variable) was not unveiled by any of the added test cases.

2. Fixing this bug required substantial changes to methods getMappingCode and soundex in class Soundex, including the introduction of several new methods. The fix is covered by six new test cases in class SoundexTest.
3. The loop conditional count < maxLength in method String soundex(String) of class Soundex erroneously used the maximum length, which was later corrected to count < out.length. It is questionable whether being able to call setMaxLength with a parameter value unequal to out.length was the flaw, or if the method soundex itself was flawed (cf. the bug description above).

After first running the test suite of Commons Codec 1.1 extended with the relevant tests of version 1.2, 9 failed tests are reported (see Table 1). Since overall, soundex is ranked highest (it is in fact involved in 7 of the failed test cases, leading to a high FA), we assume that the developer finds bug 2 from the above list first. Note that fixing it involves changes to soundex and getMappingCode, the latter also being ranked highly — in fact, both and a third method have identical posterior possibilities so that the expected number of lookups until the bug is found is calculated as 2.

Rerunning the test suite after the bug has been fixed produces the ranking labelled with "after 1st fix". Since the posterior possibilities based rank of setMaxLength has changed to 1, we assume that the developer inspects this method next. However, setMaxLength is a plain setter that is obviously correct. Therefore, we assume that the developer proceeds with the other possible fault locations and detects the bug in decodeBase64 (bug 1 from the above list). Note that the bug might have been considered as being one of discardWhiteSpace (whose call from decodeBase64 was replaced by one of a new method discardNonBase64; see above); this is also on the list, but overall ranked lower. Also note that soundex and getMappingCode still appear on the lists, despite the previous bug fix.

After having fixed the second problem, the test suite is rerun again, leading to the rows labelled with "after 2nd fix". As can be seen, decodeBase64 and discardWhiteSpace have now been ruled out as possible fault locations, and while setMaxLength has remained in front, the ranks of soundex and getMappingCode have increased. This is indeed where the final bug (bug 3) is located.

Overall, using tracing to detect the MUTs (and thus the possible failure locations) leads to visibly better results than static call graph analysis. The few cases in which it does not are due to the calling of library methods whose source code was unavailable for program analysis. This is typically the case for assertEquals and related methods from the JUNIT framework, which are called from tests cases and which invoke the equals method which is often overridden in classes under test, but whose invocation remains undetectable for the program analysis (due to the unavailability of the source code of assertEquals; cf. Section 5.1).

4.2 Evaluation Based on Error Seeding

Evidence collected from our above described experiments is somewhat anecdotal in character, and it is unclear whether and how it generalizes. To increase belief in the feasibility of our approach, a more systematic evaluation is needed. Such an evalua-

Table 2. The nine code changes performed by JESTER, as used for our evaluation

#	BOTH WAYS		#	FROM	TO
1., 2.	`==`	`!=`	7.	`if (…)`	`if (true || …)`
3., 4.	`false`	`true`	8.	`if (…)`	`if (false && …)`
5., 6.	`++`	`--`	9.	`<number>`	`<number + 1>`

tion is generally difficult; however, in our special case it can be obtained by a relatively simple mechanism, namely by injecting errors into otherwise correct programs and seeing whether EZUNIT can locate them accurately.

To do this, we have integrated the error seeding algorithm of JESTER [12, 17] into the evaluation part of EZUNIT. JESTER's approach is to systematically alter JAVA programs in such a way that their syntax remains intact, but their logic likely changes. If such a change does not cause a test failure, JESTER suggests that a unit test be added that detects the changed (and thus presumably flawed) logic.

Contrary to JESTER's focus, we are not interested in changes that pass the test suites, but only in those that cause tests to fail. Knowing which changes JESTER performed, we can thus evaluate our fault locators for their ability to spot the error inducing method (which must always be the one changed by JESTER if all unit tests passed successfully prior to the change). If the test suite passes in spite of a change performed by JESTER, the change is of no use for our evaluation purposes; it must be undone and the next one tried.

We have implemented this evaluation procedure as part of EZUNIT so that it runs completely automatically. The procedure is sketched as follows:

1. Check whether the test suite passes initially.
2. Until all possible errors have been injected:
3. Inject a single error into a known method.
4. Run the test suite using EZUNIT and its implemented fault locators.
5. If the test suite fails:
6. For each fault locator, determine and record the position of the changed method in its diagnosis (rank as described above).
7. Undo the change.

The changes JESTER is capable of performing are listed in Table 2. Note that the set of manipulations is rather limited — basically, Boolean expressions are changed (affecting control flow), integer increment and decrement are swapped, and numbers are incremented by 1. More complex code manipulations, especially of objects, would be desirable, but such changes are tricky, and since code manipulation is not an essential part of our work, only of its evaluation, we did not pursue this further. One problem of JESTER's manipulation is that it can introduce infinite loops, so that all unit tests had to be equipped with timeouts (a new feature of JUNIT 4). Infinite recursion can also be introduced (and cannot be caught by timeouts, since its causes stack overflows very quickly); however, during our experiments this never happened.

Applied to the `Money` example from the JUNIT distribution, the evaluation produced the results shown in Table 3. As can be seen, trace-based FA performs better than any other fault locator: in more than half of all cases, the flawed method is ex-

Table 3. Evaluation results using error seeding for the `Money` example from the JUNIT distribution. Each row counts the number of hits at the corresponding ranks.

RANK	STATIC (CALL GRAPH)				DYNAMIC (TRACE)			
	LOC	CC	FR	FA	LOC	CC	FR	FA
1–1½	3	7	4	4	5	5	7	14
2–2½	7	3	0	1	0	4	1	3
3–3½	0	2	0	0	4	3	4	4
4–4½	3	3	1	0	3	2	2	0
5–5½	2	3	0	0	0	6	2	1
6–6½	3	0	0	0	4	1	0	1
7–7½	0	0	1	6	4	0	1	2
8–8½	0	0	0	0	1	2	1	0
9–25	4	4	16	11	6	4	9	2
average	5.8	5.4	11.5	10.5	6.9	5.7	6.6	3.4

pected to be found among the first two suggestions and on average, the injected error is expected to be found in the 3.4th method. A more comprehensive evaluation drawing from the projects listed in Table 4 produced the results shown in Figure 3: according to this evaluation, dynamic FA ranked the flawed method first in 60% of all cases, and in a total of 76%, no more than two lookups need be expected.

The impressive performance of dynamically determined FA should not be overrated. It is mostly due to the fact that in our special evaluation scenario, there is always precisely one flawed method in the program (so that this method must account for all failed test cases), and that FA degrades all methods whose assumed faultiness does not explain all test case failures (see Section 3.2). Only if the flawed method, *m*, participates in more successful test cases than a competitor, *m′*, (so that FR(*m*), which is one factor of FA(*m*), is lower than FR(*m′*)), it can be the case that FA does not rank *m* highest.[3] It follows that a definition of FA that ignores FR would perform better, but only as long as there is only one error in the program.[4]

The better than expected performance of the fault locators based on a priori possibilities (LOC and CC) is due to an inherent bias of our evaluation procedure: given the complexity metrics (Section 3.1) and code changes performed by JESTER (Table 2), longer methods and in particular those with more if statements (i.e., methods that are more complex by definition) are likely to be changed more frequently and thus are more often the sources of (injected) faults. Since the more complex methods inherently lead the lists of possible fault locations derived from the complexity-based fault locators, the hit rate of these locators must be expected to be higher than a random selection. This is particularly true for CC: every if statement not only increases it by 1,

[3] Example: Given 2 failed test cases, if *m′* is called by 1 test case, which fails, and *m* is called by 5 test cases, of which 2 fail, FA(*m′*) is computed as 0.5, compared to 0.4 for FA(*m*).

[4] Note that it still would not be perfect since there can be several methods with identical FA value.

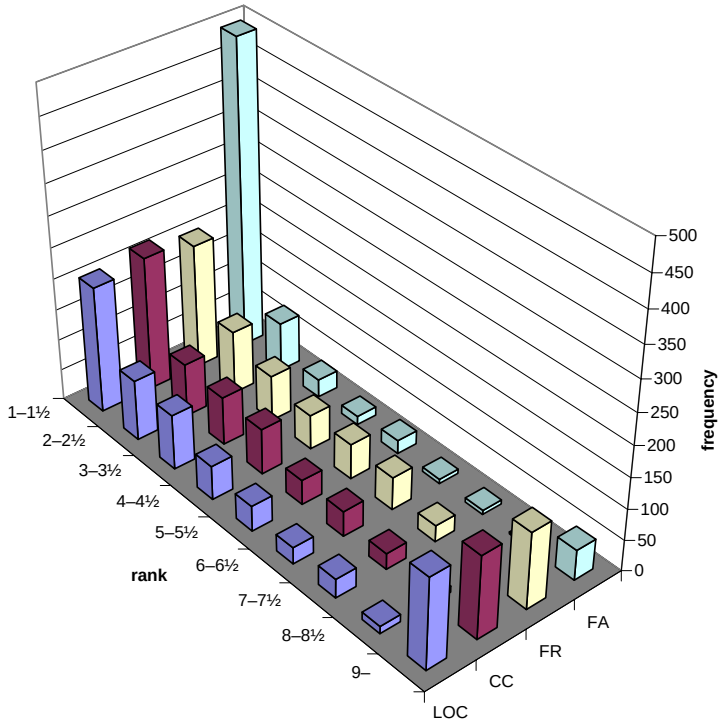


Figure 3. Hit rates of the fault locators of Section 3 based on tracing, applied to the projects of Table 4 (excluding MATH and JUNIT).

it also leads to at least two code changes that are almost certainly errors (change 7 and 8 from Table 2, and often also change 1 or 2). Assuming that programmers tend to make more errors the longer or the more complex a method is, the bias may reflect a natural condition; yet, the first example of Section 4.1 suggests that this is not generally the case.

4.3 Evaluation in Practice

Using EZUNIT in routine software development, we found that it was most helpful when used by the author of the tests and the methods being tested. This is somewhat disappointing, since the goal of EZUNIT is to point the programmer to so few possible error locations that intimate knowledge of test cases or MUTs would not be necessary to find and fix the bug. And yet, without having a least basic understanding of tests and their MUTs, spotting the bug within a method marked as a possible fault location was found to be difficult. Our developers then tried to understand the test by analysing the test case and methods it calls from the bottom up (and this without the information provided by EZUNIT); and in this effort to understand the case, the bug was

usually found. Surely, the tracing information cached by EZUNIT could be used to replay a failed test case without setting break points and executing the program in the debugger, but we did not explore this possibility further.

Also, EZUNIT proved of limited practical value when the test suites were run immediately after a MUT that was known to have passed all tests previously had been changed (continuous testing): as long as the developer remembered correctly what he had done, he went back to the changes immediately rather than seek assistance from the suggestions offered by EZUNIT. If he forgot, the recently changed locations in the source code could be brought to his attention by other means (see Section 7); yet again, we did not pursue this further.

Things were different, however, when a test written by a developer failed some time after work on the tested functionality had been finished (regression testing): in these cases, the developer usually had an idea of where to look for the fault (given his understanding of the tests and the program), but often found that this idea was misleading. The list of ranked fault locations was then helpful for choosing the next location to look at (recall that, as mentioned in Section 2, developers are often unaware of the exact methods invoked by a test case); in fact, we found that the cases in which the bug was not among the top ranked possible fault locations were rather rare.

5 Architecture of EZUNIT

EZUNIT is implemented as a plugin to the ECLIPSE Java Development Tools (JDT) that itself offers points for extension. If tracing is to be used to determine the exact set of methods called by each test case, it requires the Test and Performance Tools Platform (TPTP) [10], which offers an API for tracing programs from within the IDE, to be installed. For the evaluation of the tracing information and its presentation in the UI, EZUNIT implements its own test run listener (the standard way of extending the JUNIT framework). EZUNIT adds possible fault locations to the Problems view of ECLIPSE (in which the compiler-reported errors are shown) and sets corresponding gutter markers [4, 19]; the current version also comes with its own view, which presents the detailed diagnosis (including the values of all fault locators) to the user.

5.1 Call Graph Computation

Inheriting from its predecessor, EZUNIT can perform a static call graph analysis of test cases. For this, it constructs and traverses the method call graph, taking overriding of methods and dynamic binding into account. On its traversal through the graph, EZUNIT applies filtering expressions provided by the user to exclude library and other methods from being considered as potential fault locations. Note that since libraries may call user-defined methods through dynamic binding (via so-called hook methods), static analysis cannot stop once it enters a library. Yet, if the source code of li-

Table 4. Performance data for JUNIT, EZUNIT with static call graph analysis, and EZUNIT with tracing; both for full and incremental tracing; times in seconds.

PROJECT	NO. OF TEST CASES	AVG. NO. OF MUTS		JUNIT	EZUNIT			
					full trace		incremental	
		stat	dyn		stat	dyn	stat	dyn
JUNIT Money	22	16	13	0.09	13	4	7.1	4.0
JUNIT	308	38	9	1.02	541	35	36.9	26.3
BEANUTILS	336	62	9	1.48	1363	11	210	4.5
CODEC	191	11	6	0.72	127	107	6.7	34.0
MAIL	75	12	10	0.01	58	11	4.7	14.0
MATH	1022	39	10	35	12018	5604	282	485

braries is unavailable, such calls must remain undetected.[5] Another deficiency of static analysis is that it cannot detect reflective method calls.

Since static call graph analysis is imprecise (it usually contains methods that are never called) and also incomplete (because of reflection, and also because it requires the availability of source code), replacing it with a dynamic call graph analysis seems worthwhile. However, analysis of dynamic control flow is usually computationally expensive. And yet, in the special case of unit testing things are extremely simplified: since each test run sets up precisely the same fixture and executes only methods whose behaviour does not depend on random (including user input) or extraneous state[6], every run has exactly the same trace, so that dynamic control flow analysis becomes trivial. In fact, unless the test cases or the MUTs are changed, caching the trace of a single test run is sufficient. The cache of a test case is invalidated by each of the following events:

1. The test method is changed.
2. The body of one of the MUTs of the test method is changed.
3. The signature of one of the MUTs of the test method is changed, or the MUT is overridden by a new method, or an overloading of the method is added.
4. A MUT of the test method is deleted.

The latter two events take the possibility of a change of method binding into account. Note that the information which methods have changed is also the basis of continuous testing [7, 23, 24] and certain debugging techniques [8, 30], and in fact we have exploited this information for fault location itself [11]; however, we do not pursue these approaches here.

To get an impression of the computational overhead induced by tracing, in comparison to that induced by static control flow analysis and to plain JUNIT, we have measured the execution time required for a number of test suites. The results are presented in Table 4. All times were obtained on a contemporary PC with an Intel Centrino Duo T5600 processor run at 1.83GHz, with 2 GB of RAM. The incremental

[5] This explains why the number of called methods counted during tracing may surpass that computed through program analysis; cf. Figure 1.

[6] Mutual independence of test cases and independence of their order of execution is a general requirement of unit testing.

times were obtained after random changes to the MUTs had been made and represent averages (explaining why the ratio of *static* to *dynamic* does not follow that of the full traces).

To our surprise, dynamic tracing does not generally perform worse than static call graph computation — quite the contrary, in most cases it performs significantly better. This is due to the rather slow implementation of call graph analysis, which builds on ECLIPSE JDT's search facilities for finding the methods called. Also, overriding and dynamic binding of methods lead to significant branching in the static call graphs, since all possible method bindings need to be considered. That this is indeed a problem can be seen from the significantly larger number of MUTs statically derived for JUNIT and BeanUtils, when compared to the actual, dynamically determined MUTs. In fact, it turned out that dynamic tracing is slower only when the MUTs contain loops or (mutually) recursive calls; this is particularly the case in the MATH project.

Compared to the test suite execution times required by JUNIT, EZUNIT is slow. In fact, the time required for a full trace (the first run of a test suite) can become so long that it makes running unit tests an overnight job. On the other hand, the incremental build times seem acceptable in all cases so that, once a full trace has been obtained, work should not be hindered unduly.

5.2 Adding New Fault Locators

EZUNIT is an extension to ECLIPSE that itself offers an interface for extension. This interface (an extension point in ECLIPSE terminology) is used by the Fault Locator Manager to invoke the fault locators plugged in.

New fault locators that are implemented as plug-ins for the extension point must implement an interface that declares methods through which the Fault Locator Manager can feed the plug-ins with the tracing and other useful data (including the failed and successful tests cases, as well as comprehensive information about the execution of methods as collected by the profiling of TPTP). The plug-in must then compute its possibility values for each MUT and store it in a map that can be queried by the Manager. The EZUNIT framework turns these possibility values into a ranking and presents them to the developer.

In our current implementation, EZUNIT combines the possibilities obtained by each fault locator into a single aggregate value. Initially, this value is the unweighted average of the possibility values delivered by each locator. However, the user can vote for a locator by selecting it in the view, which increases its weight. This can be seen as a first implementation of a simple learning strategy (see Section 6).

6 Other Possible Fault Locators

There are various ways to improve the specificity of fault location, some more, some less obvious. The following list is not exhaustive.

Fault Location Based on Dynamic Object Flow Analysis The most obvious, but also perhaps the most difficult, extension to our framework is an analysis of dynamic object flow. Since test failures are usually triggered by the failure of an assert* method in which a found parameter is compared with an expected one, a backward analysis of where the unexpectedly found parameter comes from would be extremely helpful to locate the error. However, available techniques for such an analysis, such as dynamic program slicing [1, 16], prove difficult in practice [29] (cf. Section 7). Nevertheless, we expect the greatest potential for narrowing error sources to come from such analyses.

Fault Location Based on Violated Program Assertions Design by contract guards method calls with preconditions and method returns with postconditions. Blame assignment under design by contract is simple: if the precondition is violated, it is the caller's fault; if the postcondition is violated, it is the fault of the called.

Design by contract combines well with unit testing [5, 21]. Rather than waiting for a method to be called during the normal course of a program, test cases can force it to execute. The postcondition of a method under test may replace the test oracle (delivering the expected result), but then, postconditions are usually less specific, and having an independent oracle can help debug postconditions (in case a result is classified as false by a unit test, but passed the postcondition of the method under test, it may be that the postcondition is not sensitive enough).

More important in our context is the simple fact that the failure of a precondition narrows potential culprits to the methods executed before the call, and that of a postcondition to the methods executed inside the called method (including itself). Assuming that all pre- and postconditions in a program are correct, EZUNIT can be extended to catch failed assertions and exploit the stack trace or, if dynamic tracing is switched on, the trace to exclude the methods that cannot have contributed to the violation.

Fault Location Based on Subjective Prior Assessment As mentioned in Section 3.1, prior possibility measures such as complexity can be complemented by a subjective estimation, or confidence, of the error-proneness of a method. This can be added to MUTs using a corresponding annotation, which can be treated just like a derived prior possibility.

Fault Location Based on Learned Combinations of Fault Locators In this paper, we have evaluated our fault locators separately. However, EZUNIT currently implements one ad hoc aggregation of their individual votes (see Section 5.2). Indeed, it could be assumed that the best approaches are those that mix a number of different approaches, sometimes in other than obvious ways. It is therefore conceivable to apply machine learning strategies to find a combination of different (and differently weighted) locators that yields better results than any individual one (but see [20] for why this may be of limited value). This could replace for the simple feedback loop currently implemented in EZUNIT, which allows the programmer to mark the locator that actually caused the error, making the selection and combination of locators adapt to the specifics of the programmer or the project being worked on.

7 Related Work

There is a significant amount of work in the literature on predicting presence and locations of errors in programs based on a priori knowledge, such as program metrics (e.g., [3, 20]). While fault predictions of this kind are useful to direct verification and testing efforts (especially in settings in which resources for these activities are limited [3, 13, 20]), as we have argued they can also be used in combination with concrete knowledge of the presence of specific faults, to track down their locations ex post.

A posterior possibility indicator much related to ours has been suggested and evaluated in [14]. It uses the fraction of failed test cases that executed a statement (note the finer granularity) and that of passed test cases to compute a measure similar to our FR. This measure is coded as colour of the statement (varying from red for 1 to green for 0) and complemented by brightness as a measure of belief in (or evidence for) the correctness of the colour, computed as the maximum of the two above fractions. By contrast, we have used the fraction of failed test cases that executed a method as an integral part of FA, to account for the explanatory power of a single potential fault for the total set of failed test cases. Adding the fault locator suggested in [14] to EZUNIT would require an extension to a second dimension, the support (or credibility) visualized as brightness; however, this makes ranking, and thus a quantitative comparison of performance, difficult. Also, we would have to change granularity from method to statement level, invalidating our a priori possibility based fault locators.

As an enhancement of JUNIT, our approach is somewhat related to David Saff's work on continuous testing [23, 24]. Continuous testing pursues the idea that test execution, like compilation, can be performed incrementally and in the background. Whenever a developer changes something and triggers a (successful) compilation, all tests whose outcome is possibly affected by that change are automatically rerun. Thus, like our own work Saff's raises unit testing to the level of syntactic and semantic (type) checking, yet it does so in an *orthogonal dimension*: continuous testing is about *when tests are executed*, our work is about *how the results are interpreted and presented*. It should be interesting to see whether and how the two approaches can be combined into one, particularly since the mutual dependency of testing and program units under test is common to both of them.

Both approaches can profit from change impact analysis, which can identify the set of tests whose execution behaviour may have changed due to a (set of) change(s) to a software system. CHIANTI [22] is such a change impact analysis tool that is also able to identify the set of (atomic) changes that affect a failed test case. CHIANTI uses static analysis in combination with dynamic or static call graphs. JUNIT/CIA [27] is an extension to CHIANTI that not only identifies the affecting changes for a failing test case, but also classifies them according to their likelihood of failure induction (green, yellow, and red). However, it always requires a previous successful test run and a change history, which we do not.

Delta debugging [30] is a general debugging technique that works by bisecting an input (source code, program input, or program state) recursively until the error is isolated. One particular variant of delta debugging (named DDCHANGE [8]) compares the

last version of a program that passed all unit tests with one whose changes relative to this version make unit tests fail. In its current implementation, it makes no prior assumptions about where the error might be located (which is theoretically not a big problem because bisection has logarithmic complexity). However, despite this huge theoretical advantage delta debugging as currently implemented is rather slow (and the ECLIPSE plug-ins currently available resisted integration in our framework).

In another instantiation of delta debugging, Cleve and Zeller have used differences in program states of a passing and failed runs of a program to track down error locations in space and time [6]. They do so by first identifying variables that are carriers of erroneous state and then identifying the state transitions through which this state came about. Their approach has been shown to yield better results than any other technique of fault localization known thus far, but its technical demands (monitoring the execution state of programs) are heavy.

It has been shown that dynamic program slicing [1, 16] can help discover faults [28], but the computational effort is enormous. We have not yet investigated the possible performance gains (reduction of complexity) made possible by the special setting of unit testing, but expect that it will make it more tractable.

8 Conclusion

While unit testing automates the detection of errors, their localization is currently still mostly an intellectual act. By providing a framework that allows the integration of arbitrary a priori indicators of fault location with information derived from monitoring the success and failure of JUNIT test suites, we have laid the technical groundwork for a symptom-to-diagnosis mapping for logical programming errors that is tightly embedded in the edit/compile/run-cycle. We have evaluated the feasibility of our approach by providing four sample fault locators and measuring how well these were able to localize errors in three different evaluation settings. The results are promising and warrant continuation of our work.

9 Acknowledgements

The authors are indebted to Philip Bouillon for his contributions to earlier versions of EZUNIT, and to Jens Krinke for his pointers to and discussions with related work.

References

1. H Agrawal, JR Horgan “Dynamic program slicing” *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (1990) 246–256.
2. *Apache Commons Codec* (http://commons.apache.org/codec/changes-report.html).
3. 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.

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. Y Cheon, GT Leavens "A simple and practical approach to unit testing: The JML and JUnit way" in: *ECOOP* (2002) 231–255.
6. H Cleve, A Zeller "Locating causes of program failures" in: *ICSE* (2005) 342–351.
7. *Continuous Testing* (http://groups.csail.mit.edu/pag/continuoustesting/).
8. *DDChange* (http://ddchange.martin-burger.de/).
9. D Dubois, H Prade "Possibility theory, probability theory and multiple-valued logics: a clarification" *Annals of Mathematics and Artificial Intelligence* 32:1–4 (2001) 35–66.
10. Eclipse Test & Performance Tools Platform Project (http://www.eclipse.org/tptp/).
11. T Eichstädt-Engelen *Integration von Tracing in ein Framework zur Verknüpfung von gescheiterten Unit-Tests mit Fehlerquellen* (Bachelor-Arbeit, Fernuniversität in Hagen, 2008).
12. ER Harold "Test your tests with Jester" (www.ibm.com/developerworks/java/library/j-jester/).
13. AE Hassan, RC Holt "The top ten list: Dynamic fault prediction" in: *ICSM* (2005) 263–272.
14. JA Jones, MJ Harrold, JT Stasko "Visualization of test information to assist fault localization" in: *ICSE* (2002) 467–477.
15. S Kim, T Zimmermann, EJ Whitehead Jr., A Zeller "Predicting faults from cached history" in: *ICSE* (2007) 489–498.
16. B Korel, J Laski "Dynamic program slicing" *Information Processing Letters* 29:3 (1998) 155–163.
17. S Lever "Eclipse platform integration of Jester — The JUnit test tester" in: *XP* (2005) 325–326.
18. TJ McCabe "A complexity measure" *IEEE TSE* 2:4 (1976) 308–320.
19. N Meyer *Ein Eclipse-Framework zur Markierung von logischen Fehlern im Quellcode* (Master-Arbeit, Fernuniversität in Hagen, 2007).
20. N Nagappan, T Ball, A Zeller "Mining metrics to predict component failures" in: *ICSE* (2006) 452–461.
21. Parasoft Corp Using *Design by Contract to Automate Java Software and Component Testing* Technical Paper (Parasoft, 2002).
22. X Ren, F Shah, F Tip, BG Ryder, O Chesley "Chianti: A tool for change impact analysis of Java programs" in: *OOPSLA* (2004) 432–448.
23. D Saff, MD Ernst "Reducing wasted development time via continuous testing" in: *ISSRE 2003, 14th International Symposium on Software Reliability Engineering* (2003) 281–292.
24. D Saff, MD Ernst "An experimental evaluation of continuous testing during development" in: *ISSTA 2004, International Symposium on Software Testing and Analysis* (2004) 76–85.
25. M Schaaf *Integration und Evaluation von Verfahren zur Bestimmung wahrscheinlicher Ursachen des Scheiterns von Unit-Tests in Eclipse* (Bachelor-Arbeit, Fernuniversität in Hagen, 2008).
26. A Schröter, T Zimmermann, A Zeller "Predicting component failures at design time" in: *ISESE* (2006) 18–27.
27. M Störzer, BG Ryder, X Ren, F Tip "Finding failure-inducing changes in Java programs using change classification" in: *SIGSOFT '06/FSE-14* (2006) 57–68.
28. X Zhang, N Gupta, R Gupta "A study of effectiveness of dynamic slicing in locating real faults" *Empirical Software Engineering* 12:2 (2007) 143–160.
29. X Zhang, R Gupta, Y Zhang "Cost and precision tradeoffs of dynamic slicing algorithms" *ACM TOPLAS* 27:4 (2005) 631–661.
30. A Zeller "Yesterday, my program worked. Today, it does not. Why?" in: *ESEC / SIGSOFT FSE* (1999) 253–267.