# Data Mining and Cross-checking of Execution Traces: A re-interpretation of Jones, Harrold and Stasko test information visualization (Long version)

Tristan Denmat, Mireille Ducassé, Olivier Ridoux

▶ **To cite this version:**

Tristan Denmat, Mireille Ducassé, Olivier Ridoux. Data Mining and Cross-checking of Execution Traces: A re-interpretation of Jones, Harrold and Stasko test information visualization (Long version). [Research Report] PI 1743, 2005, pp.21. ⟨inria-00000566⟩

**HAL Id: inria-00000566**
**https://inria.hal.science/inria-00000566v1**

Submitted on 3 Nov 2005

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

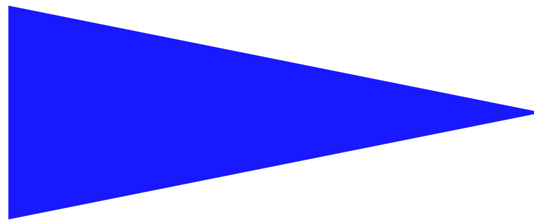
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

I R I S A

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1743



# DATA MINING
# AND CROSS-CHECKING OF EXECUTION TRACES
## A RE-INTERPRETATION OF JONES, HARROLD AND STASKO TEST INFORMATION VISUALIZATION
## (LONG VERSION)

TRISTAN DENMAT , MIREILLE DUCASSÉ AND OLIVIER RIDOUX

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

# Data Mining and Cross-checking of Execution Traces

## A re-interpretation of Jones, Harrold and Stasko test information visualization

## (Long version)

Tristan Denmat * , Mireille Ducassé ** and Olivier Ridoux ***

Systèmes symboliques
Projet Lande

Publication interne n ° 1743 — Août 2005 — 21 pages

**Abstract:** The current trend in debugging and testing is to cross-check information collected during several executions. Jones et al., for example, propose to use the instruction coverage of passing and failing runs in order to visualize suspicious statements. This seems promising but lacks a formal justification. In this paper, we show that the method of Jones et al. can be re-interpreted as a data mining procedure. More particularly, the suspicion indicator they define can be rephrased in terms of well-known metrics of the data-mining domain. These metrics characterize *association rules* between data. With this formal framework we are able to explain limitations of the above indicator. Three significant hypotheses were implicit in the original work. Namely, 1) there exists at least one statement that can be considered as faulty ; 2) the values of the suspicion indicator for different statements should be independent from each others; 3) executing a faulty statement leads most of the time to a failure. We show that these hypotheses are hard to fulfill and that the link between the indicator and the correctness of a statement is not straightforward. The underlying idea of association rules is, nevertheless, still promising, and our conclusion emphasizes some possible tracks for improvement.

**Key-words:** Software Engineering, Testing and Debugging, Learning, Knowledge acquisition

*(Résumé : tsvp)*

This report is a long version of the eponymous article published in the proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA, November 2005

* IRISA/INSA, Tristan.Denmat@irisa.fr
** IRISA/INSA, Mireille.Ducasse@irisa.fr
*** IRISA/Universit de Rennes 1, Olivier.Ridoux@irisa.fr

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

# Fouille de données et recoupement de traces d'exécution
## Une ré-interprétation de la visualisation d'information de test de Jones, Harrold et Stasko
## (Version longue)

**Résumé :** La tendance actuelle en débogage et test de programmes est de recouper des informations rassemblées lors de plusieurs exécutions. Jones et al., par exemple, proposent d'employer la couverture d'instructions calculée pour des exécutions réussissant et échouant afin de visualiser des instructions suspectes. Ceci semble prometteur mais il manque une justification formelle. Dans cet article, nous montrons que la méthode de Jones et al. peut être réinterprétée comme un procédé de fouille de données. Plus particulièrement, l'indicateur de suspicion qu'ils définissent peut être reformulé en termes de métriques bien connues en fouille de données. Ces métriques caractérisent des *règles d'association* entre les données. Avec ce cadre formel nous pouvons expliquer des limitations de l'indicateur mentionné ci-dessus. Trois hypothèses significatives étaient implicites dans le travail original. A savoir, 1) il existe au moins une instruction qui peut être considérée comme défectueuse ; 2) les valeurs de l'indicateur de suspicion pour différentes instructions doivent être indépendantes les unes des autres ; 3) exécuter une instruction défectueuse conduit la plupart du temps à un échec. Nous prouvons qu'il est difficile de satisfaire ces hypothèses et que le lien entre l'indicateur et la correction d'une instruction n'est pas direct. L'idée fondamentale des règles d'association est, néanmoins, prometteuse, et notre conclusion dessine quelques voies possibles d'amélioration.

**Mots clés :** Génie logiciel, Test et débogage, Apprentissage, Acquisition de données

# 1 Introduction

The current trend in debugging and testing is to use several executions. Executions are abstracted by spectra [8]. For example, Renieris and Reiss induce from a set of successful spectra, a model of correct execution that they compare to a faulty spectrum [16]. This comparison points out suspicious parts of the faulty execution. The type of spectra they use is instruction coverage and their experiments have shown that the nearest neighbor model of the faulty spectrum is the more relevant to be compared with the faulty run. Podgurski et al. present a method to automatically group faulty spectra with respect to the fault that leads to the failure [15]. This method relies on cluster analysis. They have shown that this approach is effective to help debugging programs. Leon et al. introduce a new approach of testing, namely observation-based testing [14]. This method is based on multivariate visualization of an execution and its originality is that it allows users to graphically compare executions. Thus, users can isolate executions that differ from the common profile by visualizing different graphs. This technique has been tried in different contexts, including evaluating and improving synthetic tests or filtering regression test suites. Ball et al. use model-checking to generate faulty runs of a program, in the sense that these runs violate a specified property [3]. The likely cause of the failure is supposed to be the transitions that appear in the counter-example but not in the correct runs. The originality with regard to usual model-checking is that many counter-examples are generated, each one having a different cause of failure. Zeller and Hildebrandt propose to use delta-debugging to discover causal relations in runs of programs [20]. This technique relies on experimentations, which consist in creating runs specifically designed to confirm or refute an hypothesis. This technique has been used in different contexts, such as isolating failure-inducing inputs of programs.

Jones, Harrold and Stasko propose to cross-check the instruction coverage of passing and failing runs in order to visualize suspicious statements [12]. This seems promising but the suspicion indicator is intriguing. It looks reasonable, but it is not one of the well-known indicators that have been defined in the data mining domain. This asked for an evaluation of the properties of the suspicion indicator. In this paper, we show that the method of Jones et al. can be re-interpreted as a data mining procedure. More particularly, the indicator they define, called $JHS$ in the following, is actually a variant of metrics that are well known in data mining. These metrics characterize *association rules* between data.

The first contribution of this article is to analyze the $JHS$ indicator in terms of well-known data mining indicators for association rules. We show how classical indicators can be used in place of $JHS$. The advantage is that the validity of these indicators is better studied than that of the $JHS$ indicator [1, 4, 7, 9].

With the formal data mining framework we are then able to explain limitations of the $JHS$ indicator. Indeed, the second contribution of this article is to uncover three significant hypotheses that were implicit in the original work.

1. There exists at least one statement that can be considered as faulty and an error has its origin in a single faulty statement.

2. $JHS$ indicators for different statements are independent from each other. I.e., statements independently belong to fail traces.

3. Executing a faulty statement leads most of the time to a failure.

These hypotheses are hard to fulfill. The first one excludes bugs where not a single statement is incorrect on its own. The second one is almost certainly false in complex programs; errors may depend on interactions between modules, or the error is in the control flow. The Independence of indicators is impossible to achieve as many instructions of a given program depend from each other because of control dependencies and data dependencies. Instruction dependences are, indeed, the basis of program dependence analysis [10] and slicing [19, 18], a wide area of research. Due to the third hypothesis, faults in statements that are always executed will never be localized. E.g., faulty statements in an initialization phase are always executed, causing or not causing failures in different runs.

The last contribution is to sketch how the full-fledged formalism of association rules can alleviate the above hypotheses. Therefore, the idea of association rules is still interesting, and our conclusion emphasizes some possible avenues worth exploring.

In the following, Section 2 first describes the work of Jones et al. Section 3 re-interprets the $JHS$ indicator as a data mining procedure metrics. Section 4 discusses in detail three hypotheses which explain several limitations of the approach. Section 5 summarizes the contributions of the current article and emphasizes some possible avenues worth exploring.

## 2 Test Information Visualization

Jones et al. suggest to use the information collected during the executions of different tests of a program to help a programmer locate errors [12]. During the execution of a particular test, it is possible to know which statements have been executed. The intuitive idea of the technique is that if the execution of a particular statement leads most of the time to a failure, and seldom to a success, then it should be considered suspect. To inform the user of the suspicion of a statement, Jones et al. color the statements of a program. Roughly speaking, if a statement is colored in red, then it must be inspected. On the contrary, if a statement is colored in green, then it can be considered as correct.

Equation 1 defines the color of a statement. It is a function of an indicator, called $JHS$ in the following. $JHS$ is defined by Equation 2 as a function of $\%P(i)$ and $\%F(i)$, respectively defined by Equations 3 and 4.

$$color(i) = red + JHS(i) * (green - red) \quad (1)$$

$$JHS(i) = \frac{\%P(i)}{\%P(i) + \%F(i)} \quad (2)$$

$$\%P(i) = \frac{\text{nb of passed tests executing i}}{\text{number of passed tests}} \quad (3)$$

$$\%F(i) = \frac{\text{nb of failed tests executing i}}{\text{number of failed tests}} \quad (4)$$

If a statement $i$ is executed by many of the passed tests and few of the failed tests, then $JHS(i)$ tends towards 1 and the color is almost green. On the opposite, if a statement $i$ is executed by many

| | Statements | | Test cases | | | | | |
|---|---|---|---|---|---|---|---|---|
| | $mid()$ | x | 3 | 1 | 5 | 3 | 5 | 2 |
| | { | y | 3 | 2 | 5 | 2 | 3 | 1 |
| | $int x, y, z, m;$ | z | 5 | 3 | 5 | 1 | 4 | 3 |
| 1 | $read(x, y, z);$ | | • | • | • | • | • | • |
| 2 | $m = z;$ | | • | • | • | • | • | • |
| 3 | $if(y < z)$ | | • | • | • | • | • | • |
| 4 | $if(x < y)$ | | • | • | | | • | • |
| 5 | $m = y;$ | | • | | | | | |
| 6 | $else\, if(x < z)$ | | • | | | | • | • |
| 7 | $m = y;$ | | • | | | | | • |
| 8 | $else$ | | | | • | • | | |
| 9 | $if(x > y)$ | | | | • | • | | |
| 10 | $m = y;$ | | | | | • | | |
| 11 | $else\, if(x > z)$ | | | | • | | | |
| 12 | $m = x;$ | | | | | | | |
| 13 | $print(x, y, z);$ | | • | • | • | • | • | • |
| | } | | | | | | | |
| | Pass/Fail status | | P | P | P | P | P | F |

Figure 1: Visualization of Test information

of the failed tests and few of the passed tests, then $JHS(i)$ tends towards 0 and the color is almost red.

Thus, this definition looks reasonable. Moreover, its general form is familiar: a ratio of a measure of events of interest on a measure of all events. However, the denominator seems odd; it sums up two values expressed in two different units, namely, number of passed tests and number of failed tests.

Figure 1, copied from [12], shows how program *mid* is colored after the execution of six test cases (dark gray corresponds to red whereas light gray and white correspond to green). On this figure, a bullet represents the fact that a statement is executed by the corresponding test case. The last line of the table indicates the status of the different tests: passed (P) or failed (F). This program contains an error: the statement on line 7 should be $m = x$ instead of $m = y$. This statement is executed by one out of the five passed test cases and the failed test case. The approach works well on this example and colors the faulty statement in red.

To evaluate their technique of fault localization, Jones et al. used a C program named *space*. It consists of 9564 lines among which 6218 are executable. They used a test pool of 13585 test cases covering every node and edge of the control-flow graph at least thirty times. Using this pool, they made 1000 test sets randomly sized. Then, they used 20 mutants and they computed the $JHS$ indicator for each of the 1000 test sets. In a first experimentation, the mutants had just one faulty statement and in a second experimentation they had from 2 to 5 faults. They colored the program and evaluated the results using two criteria:

- The number of times that a *faulty* statement is colored in a red or reddish color
- The number of times that a *non-faulty* statement is colored in a red or reddish color

Two observations were made:

- In the one-fault experimentation, 90 % of the faulty statements were colored in red. With many faults, at least one of the faulty statements was systematically colored in red.
- Nearly 10 % of the non-faulty statements were colored in red.

### 2.1 Informal Discussion

Even if the previous result is encouraging, 10 % of the faulty statements were not colored in red. There are what is called "false negatives"; i.e., statements that should be presented as suspicious, and are not presented at all. Furthermore, 10 % of the non-faulty statements colored in red means that more than 600 statements were false positives; i.e., statements that should not be presented as suspicious, but are presented as such.

We made experiments on small programs of the *Siemens test suite* [11] which confirmed that in most cases a statement can have a bad $JHS$ value and be correct and that a faulty statement can have a value of $JHS$ that makes it be considered as non faulty. Hence, the approach produces both false negatives and false positives.

Let us consider again the program of Figure 1. If the faulty statement was not the only statement of a basic block, all the statements of its block would have had the same $JHS$, whether there were faulty or not. Hence all statements of that block but the faulty one would be false positives. In the example, the faulty statement is in an innermost block. If it was not the case, for example if statement 3 was faulty, the $JHS$ of the blocks depending of this statement would be close to 0. Therefore their color would be closer to red than to green, whether there were faulty or not. This again would produce false positives.

## 3 Mining execution traces

The indicator computed by Jones et al. automatically extracts information from a large amount of data. In this section, we re-interpret the $JHS$ approach as a data mining procedure. This gives a framework to formally identify circumstances which induce false positives and false negatives.

### 3.1 Association Rules

A family of data mining procedures consists of mining large amounts of data to find *association rules* hidden in this information. Agrawal et al. point out the interest of association rules when analyzing a supermarket sales [1]. Starting from the customers' tickets, they were able to discover rules such as *"if a customer buys fish and lemon then he will probably also buy rice"*. Such rules can be used when the data can be turned into a large set of objects, each object being described by a set of *items*. These objects are also named *transactions*. Table 1 is an example of the kind of information that is provided to the algorithm of Agrawal et al. It is sufficient to know whether an item is present or not, this algorithm does not take into account how many times an item appears on a transaction.

| Transaction | Items |
|---|---|
| A | $\{fish, milk\}$ |
| B | $\{fish, lemon, rice, bread\}$ |
| C | $\{bread, fish\}$ |
| D | $\{meat, bread\}$ |
| E | $\{fish, lemon, rice\}$ |
| F | $\{lemon, milk, meat\}$ |

Table 1: Example of data for the mining of association rules

The output of the algorithm is a set of association rules showing possible links between items. A rule is represented by a formula $X \rightarrow Y$, where $X \subseteq I$ and $Y \subseteq I$ with $X \cap Y = \emptyset$, $I$ being the set of all items.

## 3.2 Support and Confidence of Association Rules

In order to characterize such rules, metrics can be used. Two common metrics are the *support* and the *confidence* of a rule. They are defined as follows [1]. $T$ denotes the set of transactions.

- *Support* of a rule :

  $$sup(X \rightarrow Y) = \frac{\text{card}(\{t \in T \quad | \quad X \cup Y \subseteq t\})}{\text{card}(T)}$$

  Remember that $T$ is a set of transactions, and that transactions are sets of items; hence, $T$ is a set of set of items, and $t$, $X$ and $Y$ are sets of items.

  The $sup$ metrics represents the proportion of the transactions described by all items from $X \cup Y$ with respect to the whole set of transactions. It can be seen as the frequency where the rule is observed. This metrics is important to distinguish the rules corresponding to exceptions from the rules reflecting a trend in the data set. The support is also defined for a set of items:

  $$sup(Y) = \frac{\text{card}(\{t \in T \quad | \quad Y \subseteq t\})}{\text{card}(T)}$$

  This is a particular case of the previous definition where the left-hand side set of items is empty.

- *Confidence* of a rule :

  $$conf(X \rightarrow Y) = \frac{\text{card}(\{t \in T \quad | \quad X \cup Y \subseteq t\})}{\text{card}(\{t \in T \quad | \quad X \subseteq t\})}$$

  The confidence represents the proportion of the transactions where items from $X \cup Y$ appear together with respect to the set of transactions where the items from $X$ appear

together. If the items from $Y$ are present almost each time items from $X$ are, then $\text{card}(\{t \in T \mid X \cup Y \subseteq t\}) \simeq \text{card}(\{t \in T \mid X \subseteq t\})$ and the confidence is close to 1. Reciprocally, if the items from $X$ are often present without those from $Y$, then the confidence tends towards 0. The confidence measures the validity of $X \rightarrow Y$ as a rule; the lesser the exceptions to the rule, the greater its validity.

If we consider a set $T$ of transactions, there is a very large number of association rules: the rules involving two items, three items, ..., $k$ items, where $k$ is the maximum number of items in a transaction from $T$. Consequently, it is often unrealistic to observe all the existing association rules in a set of transactions. To face this problem, the algorithm *A priori*, introduced by Agrawal and Srikant [2], asks the user to specify two thresholds, one for the support and the other for the confidence. These thresholds correspond to the minimum values of metrics for which the user considers that the rules are valid. The algorithm is able to compute all the rules that satisfy these thresholds. We do not detail the exact behavior of the algorithm but the key idea is that the sets of items having a support greater than the minimum threshold are computed, starting from sets of 2 elements until convergence. The confidence of the rules extracted from these different sets are computed and only the rules with a confidence greater than the threshold are selected.

## 3.3 Relevance of Support and Confidence

An important problem of the algorithm *A priori* is that the extracted rules are not necessarily the most interesting. Indeed, the two metrics *sup* and *conf* are not totally appropriate to evaluate the relevance of association rules. This is because validity, i.e., a constraint on exceptions to rules, does not mean relevance, i.e., a gain in information. For instance, assume the university restaurant serves fried potatoes every day. One can say *"every time I am ill, there have been fried potatoes the day before"* with confidence 1. However, this is clearly not relevant as a rule. Now assume the restaurant serves fried potatoes every two days. If the above statement has a confidence greater than 0.5, then it may be relevant.

More formally, let us interpret the *conf* metrics with a probabilistic approach, as presented by Hipp et al. [9]. We call $x$ and $y$ the events corresponding to the fact that a transaction contains respectively the items of $X$ and $Y$. $P(x, y)$ denotes the probability of $x$ and $y$. Consequently:

$$\begin{aligned} conf(X \rightarrow Y) &= \frac{\text{card}(\{t \in T \mid X \cup Y \subseteq t\})}{\text{card}(\{t \in T \mid X \subseteq t\})} \\ &= \frac{\text{card}(\{t \in T \mid X \cup Y \subseteq t\})}{\text{card}(T)} * \frac{\text{card}(T)}{\text{card}(\{t \in T \mid X \subseteq t\})} \\ &= \frac{P(x,y)}{P(x)} \end{aligned} \tag{5}$$

If two events $a$ and $b$ are independent, $P(a, b) = P(a).P(b)$. Hence, if events $x$ and $y$ are independent Equation 5 becomes :

$$conf(X \rightarrow Y) = P(y) \tag{6}$$

| Trans. | Items |
|---|---|
| 1 | $\{e(1), e(2), e(3), e(4), e(6), e(7), e(13), P\}$ |
| 2 | $\{e(1), e(2), e(3), e(4), e(5), e(13), P\}$ |
| 3 | $\{e(1), e(2), e(3), e(8), e(9), e(10), e(13), P\}$ |
| 4 | $\{e(1), e(2), e(3), e(8), e(9), e(11), e(13), P\}$ |
| 5 | $\{e(1), e(2), e(3), e(4), e(6), e(13), P\}$ |
| 6 | $\{e(1), e(2), e(3), e(4), e(6), e(7), e(13), F\}$ |

Table 2: Transformation of the data on the figure 1 to use association rules

Equation 6 shows that if $P(y)$ is high, i.e., the items of $Y$ appears in a lot of transactions, then $conf(X \to Y)$ is high, even if $X$ and $Y$ are independent.

## 3.4 The Lift Metrics

To face this problem, another metrics has been introduced: the *lift* of a rule [4]. It measures the increase of the probability to have the items of $Y$ knowing that we have those of $X$, with respect to the probability to have the items of $Y$.

$$lift(X \to Y) = \frac{conf(X \to Y)}{sup(Y)}$$

If $lift(X \to Y) > 1$, the probability to have the items of $Y$ knowing that we have those of $X$ is greater than the probability to have the items of $Y$. There is an *attraction* between the fact to have the items of $X$ and the fact to have the items of $Y$. On the contrary, if $lift(X \to Y) < 1$, there is a repulsion between these two facts. If $lift(X \to Y) = 1$, then $X$ and $Y$ are formally independent. In this case, the confidence has no signification.

Note that if $Y$ is set, $lift(X \to Y)$ and $conf(X \to Y)$ only differ by the constant factor $\frac{1}{sup(Y)}$.

## 3.5 $JHS$ trace cross-checking: a data mining problem

In this subsection, we show the links between association rules and the $JHS$ indicator presented in Section 2.

The data used by Jones et al. are the executed statements and the verdict of a set of test cases. We can turn these data into the form shown on Table 1. Each test case is a transaction. It is associated with the set of statements that were executed and the verdict of the test. This is illustrated by Table 2 for the example of Figure 1. On this table, we note $e(i)$ the fact that the statement $i$ is executed during a test case and we note $F$ and $P$ the verdict of a test case, respectively $FAIL$ and $PASS$.

From this table we can extract association rules among the different attributes of a test case. Particularly, we can look for rules like $(e(i) \to F)$[1], that means looking for single statements that

[1] Strictly speaking, such a rule should be noted $(\{e(i)\} \to \{F\})$. In order to simplify notations, in the following, rules involving singletons will be noted without parentheses.

often make the program crash if they are executed. This is exactly the kind of statements that Jones et al. want to locate.

Note that the number of rules $(e(i) \rightarrow F)$ is small, it is the number of statements in the tested program. Thus, one does not have to use the *A priori* algorithm to restrict the search. However, having only singleton in left-hand parts of association rules prevents from observing the impact of several distant but non-independent statements on test verdicts (see discussions in the following).

Note also that, in the data mining general case, both $X$ and $Y$ are variables of the problem of finding relevant association rules. In Jones et al. bug finding problem, $Y$ is a constant, i.e., $F$ for $FAIL$, and the problem is to find relevant $X$'s. In this case, $lift(X \rightarrow F)$ and $conf(X \rightarrow F)$ are related by the constant factor $\frac{1}{sup(F)}$. In particular, if $FAIL$ verdicts are rare, $lift(X \rightarrow F)$ is much greater than $conf(X \rightarrow F)$. Finally, note that $sup(F)$ can be 0, but in this case $conf(X \rightarrow F)$ and $lift(X \rightarrow F)$ are also equal to 0.

In a classical data mining problem, the relevance of these rules would be evaluated by computing the $conf$ and $lift$ metrics. Instead of that, Jones et al. use the $JHS$ indicator presented in Section 2. The question that arises now is: *how is this indicator adapted to evaluate the relevance of association rules?* To answer this question, we first transform the $JHS$ expression into a formula depending on $lift$.

To simplify the formulae, we note $\|i_1, ..., i_k\|$ the cardinal of the set of transactions that contain the set of items $\{i_1, ..., i_k\}$. With this notation, the metrics for the $(e(i) \rightarrow F)$ association rules that will be used in the following are:

$$\begin{aligned} sup(F) &= \frac{\|F\|}{card(T)} \\ conf(e(i) \rightarrow F) &= \frac{\|e(i), F\|}{\|e(i)\|} \\ lift(e(i) \rightarrow F) &= \frac{conf(e(i) \rightarrow F)}{sup(F)} \end{aligned}$$

The $JHS$ indicator actually characterizes the "innocence" of a statement. We therefore rather consider its complementary

$$\text{JHS'} = 1 - \text{JHS},$$

that characterizes the suspicion that we have in a statement.

$$\begin{aligned} \text{JHS'}(i) &= 1 - \frac{\%P(i)}{\%P(i) + \%F(i)} \\ &= \frac{\%F(i)}{\%P(i) + \%F(i)} \end{aligned}$$

Let us recall that $\%P(i)$, respectively $\%F(i)$, is the number of passed, respectively failed, test cases that executed $i$ divided by the total number of passed, respectively failed, test cases. With the

above notations, they become:

$$\begin{aligned} \%P(i) &= \frac{\|\text{e}(i), P\|}{\|P\|} \\ \%F(i) &= \frac{\|\text{e}(i), F\|}{\|F\|} \end{aligned}$$

Consequently,

$$\begin{aligned} \text{JHS'}(i) &= \frac{\frac{\|\text{e}(i), F\|}{\|F\|}}{\frac{\|\text{e}(i), P\|}{\|P\|} + \frac{\|\text{e}(i), F\|}{\|F\|}} \\ &= \frac{\frac{\|\text{e}(i), F\|}{\|F\|} * \frac{\text{card}(T)}{\|\text{e}(i)\|}}{\frac{\|\text{e}(i), P\|}{\|P\|} * \frac{\text{card}(T)}{\|\text{e}(i)\|} + \frac{\|\text{e}(i), F\|}{\|F\|} * \frac{\text{card}(T)}{\|\text{e}(i)\|}} \end{aligned}$$

With the definitions of confidence, support and lift:

$$\text{JHS'}(i) = \frac{\frac{\text{conf}(\text{e}(i) \rightarrow F)}{\text{sup}(F)}}{\frac{\text{conf}(\text{e}(i) \rightarrow P)}{\text{sup}(P)} + \frac{\text{conf}(\text{e}(i) \rightarrow F)}{\text{sup}(F)}}$$

We therefore have

$$\text{JHS'}(i) = \frac{\text{lift}(\text{e}(i) \rightarrow F)}{\text{lift}(\text{e}(i) \rightarrow P) + \text{lift}(\text{e}(i) \rightarrow F)} \tag{7}$$

In order to analyze this indicator, we introduce two parameters, $k_i$ and $K$, defined as follows:

$$\begin{aligned} k_i &= \frac{\|\text{e}(i), F\|}{\|\text{e}(i), P\|} \\ K &= \frac{\|F\|}{\|P\|} \end{aligned}$$

$K$ is the ratio of the number of $FAIL$ tests on the number of $PASS$ tests, and $k_i$ is the ratio of the number of $FAIL$ tests that execute statement $i$, on the number of $PASS$ tests that execute statement $i$. It is important to note that there is no a priori correlation between these two parameters. In fact, the very goal of $JHS$ bug finding is to study their correlation. A suspicious statement is a statement that increases the probability of observing a failure when it is executed.

Note that $K$ and $k_i$ live in range $[0\ \infty[$. However, the $lift(X \rightarrow F)$ metrics indicates that the association rules are more relevant if $FAIL$ is rare (i.e., $sup(F)$ is small). Hence, it is better if $K$ stays rather small. Note that if $K$ is equal to 0, then $k_i$ is also equal to 0.

With these two parameters, $conf$ and $lift$ can be reformulated as follows.

$$\begin{aligned} conf(\text{e}(i) \to F) &= \frac{\|\text{e}(i), F\|}{\|\text{e}(i)\|} \\ &= \frac{\|\text{e}(i), F\|}{\|\text{e}(i), F\| + \|\text{e}(i), P\|} \end{aligned}$$

We therefore have

$$conf(\text{e}(i) \to F) = \frac{k_i}{k_i + 1}$$

In the same way,

$$conf(\text{e}(i) \to P) = \frac{\|\text{e}(i), P\|}{\|\text{e}(i), F\| + \|\text{e}(i), P\|} = \frac{1}{k_i + 1}$$

$$\begin{aligned} lift(\text{e}(i) \to F) &= \frac{conf(\text{e}(i) \to F)}{sup(F)} \\ &= \frac{\|\text{e}(i), F\|}{\|\text{e}(i)\|} * \frac{card(T)}{\|F\|} \\ &= \frac{\|\text{e}(i), F\|}{\|\text{e}(i), F\| + \|\text{e}(i), P\|} * \frac{\|F\| + \|P\|}{\|F\|} \end{aligned}$$

We therefore have

$$lift(\text{e}(i) \to F) = \frac{k_i}{k_i + 1} * \frac{K + 1}{K} \tag{8}$$

In the same way:

$$lift(\text{e}(i) \to P) = \frac{K + 1}{k_i + 1} \tag{9}$$

Using Equations 7, 8 and 9, the $JHS'$ indicator can then be reformulated as:

$$\begin{aligned} \text{JHS'}(i) &= \frac{\frac{k_i}{k_i + 1} * \frac{K + 1}{K} * \frac{k_i + 1}{K + 1}}{1 + \frac{k_i}{k_i + 1} * \frac{K + 1}{K} * \frac{k_i + 1}{K + 1}} \\ &= \frac{\frac{k_i}{K}}{1 + \frac{k_i}{K}} \end{aligned}$$

We therefore have

$$\text{JHS'}(i) = \frac{k_i}{K + k_i} \tag{10}$$

The interesting thing with this reformulation is that

$$lift(\text{e}(i) \rightarrow F) = 1 \text{ if and only if } K = k_i.$$

and

$$lift(\text{e}(i) \rightarrow F) > 1 \text{ if and only if } K < k_i.$$

It means that there is an attraction between e$(i)$ and $F$ if the density of tests that fail is smaller than the density of tests that execute statement $i$ and fail.

Remember that 1 is the boundary between attraction and repulsion. A value of $lift(\text{e}(i) \rightarrow F)$ that is greater than 1 indicates an attraction between failed tests and tests that execute e$(i)$. A value smaller than 1 indicates repulsion. However, Equation 10 shows that $JHS(i) = 1/2$ if and only if $K = k_i$. The consequence is that $JHS(i)$ changes from greenish to reddish at the same time as $lift(\text{e}(i) \rightarrow F)$ changes from repulsion to attraction. Thus, the $JHS$ indicator measures the attraction between executing a given line and failing, but it does so in a different unit than $lift$.

We have seen that, when $Y$ is set, $lift(X \rightarrow Y)$ and $conf(X \rightarrow Y)$ differ by $\frac{1}{sup(Y)}$, which is a constant factor for a given set of test cases. This shows that for the bug finding application, both $lift$ and $conf$ can be used in place of $JHS$ provided that $conf$ is compared with $sup(F)$. The change from greenish to reddish is made when $conf(X \rightarrow F)$ becomes greater than $sup(F)$.

Figure 2 presents the curves corresponding to $conf(\text{e}(i) \rightarrow F)$, $lift(\text{e}(i) \rightarrow F)$, and $JHS'(i)$ as functions of $k_i$ for the three values of $K$: 0.2 (i.e., few failures), 1 and 5 (i.e., many failures). In each plot, the point where the $lift$ curve crosses line 1 marks the boundary between the repulsion area, on its left side, and the attraction area, on its right side. The values of $k_i$ that make a statement suspicious belong to the attraction area. E.g., the third plot belongs entirely in the repulsion area, hence none of the presented $k_i$'s indicates suspicion. Of course for larger $k_i$'s the lift would eventually be larger than 1. The other plots show a more or less narrow band on the left where the set of tests is inconclusive.

In conclusion, the specific indicator is not necessary for the bug tracking problem; the classical indicators work well. An apparent advantage of $JHS$ is that it belongs to $[0\ 1]$, but this is also the case with $conf$. The second advantage of $JHS$ is that it specifies a priori a boundary value between repulsion and attraction, though confidence does not in the general case. However, the analysis of the definition of $lift$ shows that the boundary value for $conf$ is $sup(F)$; this value is also known a priori since it is a constant for a given set of test cases, though it is not the same for all bug tracking experiences.

Guillaume introduces a variant of $conf$ that integrates a $lift$ component [7]. It belongs to $[-1\ 1]$, and value 0 is a boundary value that plays exactly the same role as value 1 for $lift$ and value $1/2$ for $JHS$. The definition is as follows:

$$\text{if} \quad conf(X \rightarrow Y) \geq sup(Y)$$

$$\text{then} \quad conf'(X \rightarrow Y) = \frac{card(T) * \|X \cup Y\| - \|X\| * \|Y\|}{\|X\| * (card(T) - \|Y\|)} \tag{11}$$

$$\text{else} \quad conf'(X \rightarrow Y) = \frac{card(T) * \|X \cup Y\| - \|X\| * \|Y\|}{\|X\| * \|Y\|} \tag{12}$$

Definition 11 represents attraction, whereas Definition 12 represents repulsion.

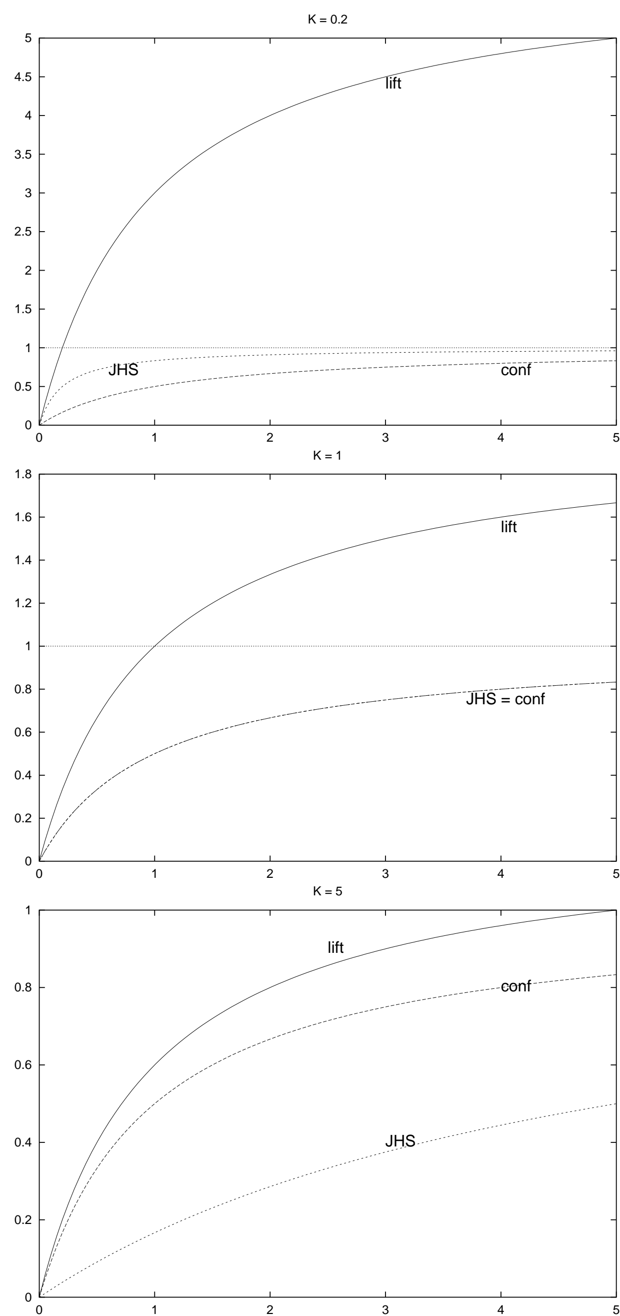



Figure 2: The different metrics for different values of $K$

```
...
if (...)
{// BLOCK 1
   ...
   r = 0;
   ...
}
else
{// BLOCK 2
   ...
}             //   fix : r = 1;
r = 1;        //   fix :      }
...
```

Figure 3: Faulty Program with no faulty statement

The advantage of $lift$ and $conf'$ is that they will work well in every variant of the bug finding problem (see conclusion), whereas $JHS$ is specialized to the e$(i) \rightarrow F$ association rule.

# 4 Implicit hypotheses

In the previous section, we have shown that the method presented by Jones et al. corresponds to finding association rules. We have underlined that the ratio between failed and passed runs had much impact on the result.

In this section we exhibit three hypotheses that have to be verified in order for the approach to be meaningful. They were not explicit in the original article and are hard to fulfill in practice. They explain the existence of false positives and false negatives.

## 4.1 The competent programmer hypothesis

The first hypothesis is that there exists at least one statement that can be considered as faulty. If not, looking for some statements which are associated to $FAIL$ is meaningless. What should be the response of a fault localization technique that underlines suspect statements when there is no faulty statement? The problem is especially blatant when the fault is a hole in the program; e.g., a missing case in a switch statement, or more simply, a missing statement. As a consequence, the technique will lack precision when the code fault corresponds to a deep error such as a wrong control-flow.

Figure 3 shows a simple case where the control flow is faulty but no instruction is to blame. In the program example the fault is that the statement "$r = 1$;" at the last line should be in the second block.

```
...
if (...)
{// BLOCK 1
   ...
   r = 0;
   ...
}
else
{// BLOCK 2
   ...
   r = 1;
   ...
}
if (r == 0)
   // BLOCK 3
   ...
```

Figure 4: Execution dependences in a program

The hypothesis concerning this type of fault is commonly used in the field of program testing. It is called the competent programmer hypothesis. Note that Jones et al. used program mutations for validating their method; this makes this hypothesis true by construction.

It is important to note that, at present and to our best knowledge, there are no techniques to detect that the competent programmer hypothesis is not fulfilled.

## 4.2 Independence of association rules

In data mining theory, an association rule $Y \to Z$ is called *fallacious* if there exists two other rules $X \to Y$ and $X \to Z$ with $P(Y|X,Z) = P(Y|X)$. In such a case, $Y \to Z$ is only due to the fact that $X$ implies both $Y$ and $Z$, and it is not an interesting information.

In our case, the association rules corresponding to the $JHS$ indicator are of the form e$(i) \to F$. However, in the analyzed programs there are other associations due to the dependences between statements. If $j$ is a statement which depends of statement $i$ then e$(i) \to$ e$(j)$. If, in addition, e$(i) \to F$, then e$(j) \to F$ is a fallacious association rule.

The $JHS$ indicator of a statement is computed and interpreted independently from the $JHS$ indicators of the rest of the program. When it corresponds to a fallacious rule, it is a false positive. There are numerous dependencies inside a program. Indeed, as already mentioned in the introduction, program dependence analysis [10] and slicing [19, 18] cover a wide area of research. Therefore, there are numerous fallacious association rules and the approach generates numerous false positives.

For example, all the statements in a given block have the same $JHS$. If the first one is executed, all the others are, if the execution of all of these statements terminate. A block is therefore the minimal granularity that can be expected with the current $JHS$.

```
int multiplication(int x, int y)
{
  int r;
  if((x == 0) || (y == 0))
  {
     r = 0;
  }
  else
  {
     r = x*y;
  }
  return 2*r;  // fix : return r;
}
```

Figure 5: Example where executing the faulty statement does not condition the verdict

Another example is the case where two statements in two distinct blocks depend on the same variables. As a consequence, these executions are not independent and the $JHS$ indicators are biased because of this link. Figure 4 shows such an example. Executing the third block depends on executing block 1. Hence, if the $JHS$ of block 1 marks it as suspect, so will the $JHS$ of block 3 even if block 3 is correct.

## 4.3 Fundamental hypothesis

A statement $i$ will be considered suspect if and only if the $conf$ of the rule $(e(i) \rightarrow F)$ is high. This value will be high if executing the statement $i$ often leads to a failure. The fundamental hypothesis of this technique is thus that executing the faulty statements leads most of the time to a failure

Yet, this is not necessarily the case. For example, consider the program on Figure 5. The faulty statement is the last one, $r$ should be returned instead of $2 * r$. Consequently, if $r$ is equal to $0$, the verdict of the execution will be $PASS$. A failure will appear only if the block corresponding to $x \neq 0$ and $y \neq 0$ is executed. This example shows that there exists some faults where executing the faulty statement does not cause observable failures.

Particularly, the method cannot be used to localize faulty statements executed by all the test runs. Such statements are often present in initialization functions or main functions. They will be false negatives.

Another case where this hypothesis is a problem is programs with multiple faults. If errors are independent, which means that each fault is responsible for a particular failure, then the technique can be used, with the limitations previously presented. In that case, each fault can be localized separately from the others. On the opposite, there can be tighter links between the different faults. A failure can appear if and only if a combination of many faulty statements is executed. It can also

```
...
if (...)
{
    ...
    r = 1;  \\ fix : r = 0;
    ...
}
...
if (...)
{
    ...
    r ++;  \\ fix : r = r + 2;
    ...
}
...
```

Figure 6: Example where faulty statements can mask other faults

disappear if another faulty statement is executed. In this case two faults are mutually masked. If such cases happen, the method does not provide good results because the fundamental hypothesis is not verified: executing a faulty statement does not condition the verdict of the test. Figure 6 shows a case where two faults can mask each others. If only one of the blocks containing a faulty statement is executed, a failure will appear. But, if both of them are executed the effects of the faulty statements disappear.

Note also that programs that implement very regular algorithms, e.g., matrix or graph operations, execute almost all their statements at all runs. So, there is no statement that leads more often than the others to a failure.

# 5 Conclusion

**Summary** In this article we have re-interpreted the work of Jones et al. as a data mining procedure. In so doing, we have shown that the $JHS$ indicator corresponds to classical data mining metrics. We have uncovered a number of requirements and hypotheses which are implicit in the original article and help to explain a number of the limitations of the approach.

1. The competent programmer hypothesis: if an execution leads to a failure, then there is at least one statement that can be said to contain a fault.

2. Independence of the indicators hypothesis: suspicion ratio of a statement can be interpreted separately from the rest of the program.

3. Fundamental hypothesis: executing a faulty statement leads most of the time to a failure.

Thus, the method has intrinsic limitations:

- The bug in the program cannot be an error in the control-flow in order to verify Hypothesis 1.
- As a consequence of Hypothesis 2, the most precise element that can be localized is a basic block. Due to this hypothesis, many correct statements are considered as suspect statements: at least all the statements of a basic block that contains an actual erroneous statement.
- To verify Hypothesis 3, faulty statements whose execution does not necessarily cause a failure cannot be localized by this method. Particularly, faults in statements that are always executed will never be localized; e.g., in initialization.
- Programs containing many faults interacting together cannot be debugged with this method because if failures are due to more than one fault, Hypothesis 3 is not verified.

**Perspectives** Even if the proposed method relies on hypotheses that are hard to fulfill, the idea of association rules is nevertheless interesting.

The $JHS$ approach looks for association rules of the form

$$\mathrm{e}(i) \rightarrow F.$$

A very natural approach to enhance Jones et al. method is to exploit association rule formalism to the full. Indeed, the formalism permits that left and right parts of association rules are *sets of items*. The right part of $JHS$ association rules is $F$, but it could be extended to accommodate failure circumstances; e.g., name of failure event. The related association rules could then be, for example, of the form

$$\mathrm{e}(i) \rightarrow \{F,\ seg_fault\}.$$

Still, the details of such an extension requires some ingenuity. The left part is not limited to be a single statement number. One could allow several items in the left part. The related association rules could then be for example of the form

$$\{\mathrm{e}(i),\ ...,\ \mathrm{e}(n)\} \rightarrow F.$$

This would permit to alleviate Hypotheses 1, and 2, and relax Hypothesis 3. The error is no longer supposed to reside in one statement. Statement need no longer be considered as independent, and the notion of a statement causing a failure is replaced by the notion of a combination of statements causing a failure.

To address the limitations of Hypothesis 1, another avenue worth exploring is the use of alternative trace representations, or even alternative spectra. Namely, $\mathrm{e}(i)$, which is the trace that statement $i$ is executed, could be enhanced by other types of information about executions. In other contexts, other forms of traces have been used. For example, Denise et al. use execution paths [6], Dallmeyer et al. use method call subsequences of a given length [5], Souter et al. use contextual variable Def-Use information [17], Langevine et al. use a generic execution trace format [13]. The related association rules could then be for example of the form

$$\{\text{e}(i),\ \text{path}(p),\ \text{use}(x, p_i, p_j)\} \rightarrow \{F,\ seg_fault\}.$$

The search spaces would of course be much larger than for the $JHS$ indicator. There exists data mining algorithms, such as *A priori* already mentioned, that have proved their worth for very large data bases.

# References


[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 97–105, New York, NY, USA, 2003. ACM Press.

[4] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International conference on Management of Data*, pages 255–264. ACM Press, 1997.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proc. 19th European Conference on Object-Oriented Programming*, 2005.

[6] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *Proceedings of the 15th. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–34, 2004.

[7] S. Guillaume. *Traitement des donnés volumineuses, mesures et algorithmes d'extraction de règles d'association et règles ordinales*. PhD thesis, Université de Nantes, Décembre 2000.

[8] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90. ACM Press, 1998.

[9] J. Hipp, U. Guentzer, and G. Nakhaeizadeh. Data mining of association rules and the process of knowledge discovery in databases. In *Industrial Conference on Data Mining*, pages 15–36, London, UK, 2002. Springer-Verlag.

[10] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *14th International Conference on Sotware Engineering*, pages 392–411, Melbourne, May 1992.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE, 1994.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM Press, 2002.

[13] L. Langevine, P. Deransart, and M. Ducassé. A generic trace schema for the portability of cp(fd) debugging tools. In J. Vancza, K. Apt, F. Fages, F. Rossi, and P. Szeredi, editors, *Recent advances in Constraint Programming*. Springer-Verlag, Lecture Notes in Artificial Intelligence 3010, 2004.

[14] D. Leon, A. Podgurski, and L. J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 116–125. ACM Press, 2000.

[15] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475. IEEE Computer Society, 2003.

[16] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 30–39. IEEE Computer Society, 2003.

[17] A. L. Souter and L. L. Pollock. Contextual def-use associations for object aggregation. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, New York, NY, USA, 2001. ACM Press.

[18] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, September 1995.

[19] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[20] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.