

# Data Mining and Cross-checking of Execution Traces

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

Tristan Denmat, Mireille Ducassé and Olivier Ridoux

IRISA/INSA and IRISA/Université de Rennes 1  
Campus Universitaire de Beaulieu, F - 35042 Rennes Cedex, France  
email: {Tristan.Denmat, Mireille.Ducasse, Olivier.Ridoux}@irisa.fr

## 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, they define an indicator which characterizes *association rules* between data. With this formal framework we are able to explain intrinsic limitations of the above indicator.

## Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging; I.2.6 [Artificial Intelligence]: Learning—*Knowledge acquisition*

## General Terms

Algorithms, Theory, Experimentation, Measurement

## 1. INTRODUCTION

The current trend in debugging and testing is to use several executions. Jones, Harrold and Stasko propose to cross-check the instruction coverage of passing and failing runs in order to visualize suspicious statements [5]. This seems promising but when we tried to replicate their results, they were disappointing. 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 an ad hoc customization 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE '05, Long beach, California USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

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. Firstly, there exists at least one statement that can be considered as faulty. I.e., an error has a single faulty statement origin. Secondly, *JHS* indicators for different statements are independent from each other. I.e., statements independently belong to fail traces. Thirdly, 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. 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 independance of indicators is impossible to achieve as many instructions of a given program depend from each other because of control dependencies and data dependencies. 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.

In the following, Section 2 first describes the work of Jones et al. and informally discusses its limitations. 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 [5]. 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. colorize 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 *i*. It is a function of an indicator, called *JHS* in the following. *JHS* is defined by Equation 2 as a function of  $\%P(i)$ , the number of passed tests executing *i* divided by the number of passed tests, and  $\%F(i)$ , the number of failed tests executing *i* divided by the number of failed tests

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

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

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

Figure 1: Visualization of Test information with JHS

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

Figure 1, copied from [5], shows how program `mid` is colored after the execution of six test cases (dark grey corresponds to red whereas light grey 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 case and the failed test case. The approach works well on this example and colours the faulty statement in red.

To evaluate this 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 of `space` 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: 1) the number of times that a *faulty* statement is colored in a red or reddish color ; 2) the number of times that a *non-faulty* statement is colored in a red or reddish color. Two observations were made. Firstly, in the first 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. Secondly, nearly 10 % of the non-faulty statements were colored in red.

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.

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, again 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.

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*. Association rules show 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 items.

#### 3.1 Support and Confidence of Association Rules

In order to characterize such rules, metrics can be used. Two common metrics are the *support* ( $sup(X \rightarrow Y)$ ) and the *confidence* ( $conf(X \rightarrow Y)$ ) of a rule. They are defined as follows [1].  $T$  denotes the set of transactions. Transactions are sets of items. Hence,  $T$  is a set of set of items, and  $t$ ,  $X$  and  $Y$  are sets of items.

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

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, e.g.  $sup(Y)$ , as a particular case of the previous definition where the left-hand side set of items,  $X$ , is empty.

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

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  $card(\{t \in T \mid X \cup Y \subseteq t\}) \simeq card(\{t \in T \mid X \subseteq t\})$  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.

If we consider a set  $T$  of transactions, there is a very large number of association rules: the rules involving two 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 interesting. 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.

### 3.2 The Lift Metrics

The two metrics *sup* and *conf* are not sufficient to evaluate the relevance of association rules. In particular, if the items of *Y* appears in a lot of transactions, then  $conf(X \rightarrow Y)$  is high, even if there is no link between *X* and *Y*. Another metrics has therefore been introduced: the *lift* of a rule [3]. 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*.

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

If  $lift(X \rightarrow Y) > 1$ , then 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  $lift(X \rightarrow Y) < 1$ , then there is a *repulsion* between these two facts. If  $lift(X \rightarrow Y) = 1$ , there is no link between the sets *X* and *Y*. In this case, the confidence has no signification.

### 3.3 JHS trace cross-checking

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 a form adequate for data mining. Each test case is a transaction. It is associated with the set of statements that were executed and the verdict of the test. 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 these data we can extract association rules among the different attributes of a test case. Particularly, we can look for rules like  $(e(i) \rightarrow F)$ , that means looking for single statements that often make the program crash if they are executed. This is exactly the kind of statements that Jones et al. want to locate.

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*. Due to space limitation the details of the transformation are skipped. They can be found in the technical report [].

The *JHS* indicator actually characterizes the “innocence” of a statement. We therefore rather consider its complementary that characterizes the suspicion that we have in a statement.

$$JHS' = 1 - JHS = \frac{lift(e(i) \rightarrow F)}{lift(e(i) \rightarrow P) + lift(e(i) \rightarrow F)}$$

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

$$k_i = \frac{\|e(i) \cup F\|}{\|e(i) \cup P\|} \quad K = \frac{\|F\|}{\|P\|}$$

```

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

```

Figure 2: Faulty Program with no faulty statement

*K* is the ratio of the number of Fail tests on the number of Success tests, and  $k_i$  is the ratio of the number of Fail tests that executes statement *i*, on the number of Success tests that executes 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.

With the two parameters, lift and *JHS'* can be reformulated:

$$lift(e(i) \rightarrow F) = \frac{k_i}{k_i + 1} * \frac{K + 1}{K} \quad (3)$$

$$JHS'(i) = \frac{k_i}{K + k_i} \quad (4)$$

The interesting thing with these reformulations is that

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

$$JHS(i) = 1/2 \text{ if and only if } K = k_i.$$

On the one side, as already mentioned, a value of  $lift(e(i) \rightarrow F)$  that is greater than 1 indicates an attraction between failed tests and tests that execute  $e(i)$ . A value smaller than 1 indicates repulsion. On the other side, *JHS*(*i*) indicates a change from greenish to reddish when it is equal to 1/2, namely at the same time as  $lift(e(i) \rightarrow F)$  changes from repulsion to attraction. Thus, the *JHS* indicator represents the attraction between executing a given line and failing.

We have seen that  $lift(X \rightarrow Y)$  and  $conf(X \rightarrow Y)$  differ by  $\frac{1}{sup(F)}$ , 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  $conf(X \rightarrow Y)$  becomes greater than *sup*(*F*).

## 4. FURTHER IMPLICIT HYPOTHESES

In this section we exhibit three hypotheses that have to be verified in order for the approach to be meaningful. None of them were explicit in the original article and none of them is fully achieved 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. 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 not work when the code fault corresponds to a deep error such as a wrong control-flow.

Figure 2 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.

```

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 3: Example where executing the faulty statement does not condition the verdict**

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 implicitly made this hidden hypothesis true.

## 4.2 Independence of association rules

In data mining theory, an association rule  $Y \rightarrow Z$  is called *fallacious* if there exists two other rules  $X \rightarrow Y$  and  $X \rightarrow Z$  with  $P(Y|X, Z) = P(Y|X)$ . In such a case,  $Y \rightarrow Z$  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  $e(i) \rightarrow F$ . 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  $e(i) \rightarrow e(j)$ . If, in addition,  $e(i) \rightarrow F$ , then  $e(j) \rightarrow F$  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. 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*.

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.

## 4.3 Fundamental hypothesis

A statement  $i$  will be considered as a 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 3. 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 passed. A failure will appear only if the block corresponding to  $x \neq 0$  or  $y \neq 0$  is executed. This example shows that there exists some faults where executing the faulty statement does not cause 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 may 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.

## 5. CONCLUSION

In this article we have re-interpreted the work of Jones et al. as a data mining procedure. In so doing we have uncovered a number of requirements and hypotheses which are implicit in the original article but help to explain a number of the limitations of the approach. 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. Independence of the indicators hypothesis: suspicion ratio of a statement can be interpreted separately from the rest of the program. Fundamental hypothesis: executing a faulty statement leads most of the time to a failure.

As a consequence, the bug in the program cannot be an error in the control-flow (Hyp. 1). As a consequence of , The most precise element that can be localized is a basic block (Hyp. 2). Faulty statements whose execution does not necessarily cause a failure cannot be localized by this method (Hyp. 3). Particularly, faults in statements that are always executed will never be localized. Programs containing many faults interacting with each others cannot be debugged with this method (Hypothesis 3).

Even if the proposed method is not fully satisfactory, the idea of association rules is nevertheless interesting. The *JHS* approach looks for association rules of the form  $e(i) \rightarrow F$ . A perspective would be to exploit the association rule formalism and the corresponding tools to the full. Indeed, left and right parts of association rules can be *sets of items*. For example, the error would no longer be supposed to reside in one statement only. Statement would no longer need be considered as independent, and the notion of a statement causing a failure would be replaced by the notion of a combination of statements causing a failure.

## 6. REFERENCES

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 207–216. ACM Press, 1993.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.
- [3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. ACM SIGMOD Int. conf. on Management of Data*, pages 255–264. ACM Press, 1997.
- [4] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. Int. Conf. on Software Engineering*, pages 467–477. ACM Press, 2002.