DeLLIS: a Data Mining Process for Fault Localization

Peggy Cellier, Mireille Ducassé, Sébastien Ferré, Olivier Ridoux
University of Rennes 1/INSA of Rennes/IRISA *

Abstract

Most fault localization methods aim at totally ordering program elements from highly suspicious to innocent. This ignores the structure of the program and creates clusters of program elements where the relations between the elements are lost. We propose a data mining process that computes program element clusters and that also shows dependencies between program elements. Experimentations show that our process gives a comparable number of lines to analyze than the best related methods while providing a richer environment for the analysis. We also show that the method scales up by tuning the statistical indicators of the data mining process.

1. Introduction

A large amount of information can be collected from program executions, for instance the executed lines, the values of variables or the values of predicates. We call that information the events of the execution, they constitute the execution trace. Fault localization methods that compare execution traces, distinguish two sorts of executions, passed and failed executions. Passed executions behave according to their specification, failed executions do not. These methods are based on the intuition that, firstly, the events that appear only in the traces of failed executions are more likely to be erroneous, and, secondly, the events that appear only in the traces of passed executions are more likely to be innocent. For instance, the nearest neighbor method, by Renieris and Reiss [12], takes into account only one failed execution. A passed execution “near” to the failed execution is then selected. The difference between the trace of the failed execution and the trace of the passed execution is presented as the suspicious events. Events are the executed lines. For the nearest neighbor method, the lines that appear in the traces of both passed and failed executions are innocent. However, many trace events belong to the traces of both passed and failed executions. The worst case of that method appears when the closest passed execution executes the same lines as the failed execution; no line is suspicious.

In order to solve that problem, the Tarantula method, by Jones et al. [9], considers as suspicious the lines that are executed “often” by failed executions and “seldom” by passed executions. The notion of “often” and “seldom” is measured by a score value. A score, assigned to each line of the program, takes into account the frequencies at which the line appears in the trace of failed and passed executions. Lines which are executed together for all executions have the same score value, for example lines that belong to the same basic block. Unfortunately, lines can have the same score value whereas there is no relation between them. Conversely, let us assume a conditional statement in which a branch always fails and the other branch always succeeds. The condition, the fail branch and the success branch may have very different scores although they are closely related. Thus, completely ordering events by their score may aggregate unrelated events and may separate related events. Nevertheless, most of the fault localization methods [4, 10, 11] try to give a total ordering of the trace events based on their score.

In this paper we describe DeLLIS, a process that takes into account the dependencies between elements of the traces. Thanks to the combination of two data mining techniques, DeLLIS computes all differences between execution traces and, at the same time, gives a partial ordering of those differences. The data mining details are given in [2]. This article focuses on experimentations, from a software engineering perspective. We first show on the programs of the Siemens suite [5], provided by SIR (Software-artifact Infrastructure Repository1) that our method gives a comparable number of lines to analyze while providing a richer environment for the analysis. Then, we show that the method scales up for a program of several thousands of lines by tuning the statistical indicators of the data mining techniques.

In the sequel, Section 2 briefly presents two techniques of data mining: association rules and formal concept analysis. Section 3 describes how they are used for fault localization. Section 4 shows experiments on the Siemens suite and the Space program.

* firstname.lastname@irisa.fr

1http://sir.unl.edu/content/sir.html
2. Background Knowledge on Data Mining

Data mining is the process of extracting relevant information from large amounts of data. We briefly present the two data mining techniques used in the sequel: Association Rules (AR) and Formal Concept Analysis (FCA). AR allows interesting regularities to be found. FCA allows relevant clusters to be computed and partially ordered. The input of both techniques is a formal context, i.e. a binary relation describing elements of a set of objects by subsets of attributes (properties). Table 1 is an example of context where the objects are the solar system planets and the attributes are the properties of the planets: e.g. to be small or near to the sun. Each planet is described by its properties; e.g. Mercury is a small planet, near to the sun and without a moon.

Association Rules AR is a data mining task with a well-documented rationale [1]. An association rule has the form: \( P \rightarrow C \), where \( P \) and \( C \) are sets of attributes. \( P \) is called the premise of the rule and \( C \) the conclusion. In the context of Table 1, \( \text{far from the sun} \rightarrow \text{with moons} \) is an association rule, which means that to be far from the sun implies to have moons. An association rule is only an assertion; it may suffer exceptions. Statistical indicators measure the relevance of association rules (see Section 3).

Formal Concept Analysis In FCA [7], the set of all objects that share a set of attributes is called the extent of the set of attributes. The set of all attributes shared by all elements of a set of objects is called the intent of the set of objects. A formal concept is defined as a pair \( \langle \text{set of objects}, \text{set of attributes} \rangle \), where the set of objects is the extent of the set of attributes and the set of attributes is the intent of the set of objects.

The set of all concepts of a context can be represented by a concept lattice. The concepts are partially ordered in the concept lattice. \( X \) is a subconcept of \( Y \) if there is a path in the lattice between \( X \) and \( Y \) and \( X \) is below \( Y \), it is denoted by \( X \leq Y \). In the concept lattice, each attribute and each object labels only one concept. Namely, each object labels the most specific concept (i.e. with the smallest extent) to which it belongs. Each attribute labels the most general concept to which it belongs. Figure 1 shows the concept lattice associated to the context of Table 1. For example, the extent of concept A is \{Jupiter, Saturn, Uranus, Neptune\} and its intent is \{far from the sun, with moons\}. Regarding A’s extent, Jupiter and Saturn label the more specific concept G and Uranus and Neptune label the more specific concept H. The attribute with moons labels the more general concept C. Thus only the attribute far from sun labels concept A.

3. Data Mining and Fault Localization

A Lattice of Failures The whole process is summarized in Figure 2. It aims at building what we call a failure lattice. The first step is the execution of the test cases. A summary of each execution is stored in a trace composed of events. In addition, we assume that each execution trace contains the verdict of the execution, \( \text{PASS} (p) \) if the execution produces the expected results and \( \text{FAIL} (f) \) otherwise. The multiset of all traces forms the trace context. The objects of the trace context are the test cases. The attributes are all
the events and the two verdicts, $p$ and $f$. Each test case is described in the trace context by the events that belong to its trace and the verdict of the execution. Note that the extent of a set of trace events $E$, $\text{extent}(E)$, denotes all the test cases that contain all elements of $E$ in their trace. The extent of $\text{FAIL}$, $\text{extent}([f])$, denotes all the test cases that fail. The second step of the process aims at selecting events that appear often in failed executions and seldom in passed executions. From the trace context, specific association rules, called fault localization rules, are computed. The premise of a rule is a set of events and the conclusion is $\text{FAIL}; E \rightarrow f$.

We use two statistical indicators to select events that are involved in failed tests: support and lift. The support of a fault localization rule $E \rightarrow f$ is the number of failed executions that have in their trace all events of $E$. It is defined as:

$$\text{sup}(E \rightarrow f) = \left| \text{extent}(E) \right| \cap \left| \text{extent}([f]) \right|.$$  

For the fault localization problem, a threshold of the support, $\text{minsup}$, indicates the minimum number of failed executions that should be covered by a rule in order to that rule be selected.

The lift of rule $E \rightarrow f$ indicates if the occurrence of the premise increases the probability to observe a failed execution. It is defined as:

$$\text{lift}(E \rightarrow f) = \frac{\text{sup}(E \rightarrow f) \| \text{All exec} \|}{\| \text{extent}(E) \| \| \text{extent}([f]) \|}.$$  

There are three possibilities. In the first case, $\text{lift}(E \rightarrow f) < 1$, executing $E$ decreases the probability to fail (the premise repels the conclusion). In the second case, $\text{lift}(E \rightarrow f) = 1$, executing $E$ does not impact the probability to fail ($E$ and $f$ are independent). In the last case, $\text{lift}(E \rightarrow f) > 1$, executing $E$ increases the probability to fail (the premise attracts the conclusion).

The computed association rules are partially ordered by set inclusion of their premise. Let $r_1 = E_1 \rightarrow f$ and $r_2 = E_2 \rightarrow f$ be two rules; rule $r_1$ is more specific than $r_2$ if and only if $E_1 \supset E_2$. It is denoted by $r_1 < r_2$. In order to display the relations between the events of the program that have been filtered, the last step of the process builds the failure context. In that context, the objects are the association rules; the attributes are events. Each association rule is described by the events of its premise. The failure lattice is the concept lattice associated with the failure context. The failure lattice highlights the partial ordering of rules according to their premises.

**Statistical Indicators** The concepts in the failure lattice obey several interesting properties with respect to the statistical indicators. Firstly, the support of rules that label the concepts of the failure lattice decreases when exploring the lattice top-down. Setting $\text{minsup}$ equal to one object is equivalent to searching for all rules that cover at least one failed execution. Setting $\text{minsup}$ equal to the number of failed executions is equivalent to searching for a common explanation for all failures. We call support cluster a maximal set of connected concepts labelled by rules which have the same support value.

Secondly, the lift value strictly increases when exploring a support cluster top-down. The threshold of the lift, $\text{minlift}$, has a lower bound equal to 1, because when $\text{lift} < 1$ a rule is not relevant. It also has an upper bound, $\text{maxlift}$, equal to $\| \text{All exec} \|$. In our approach, the lift indicates how the presence of a set of events in an execution trace improves the probability to have a failed execution. The lift threshold can be seen as a resolution cursor inside the support clusters. On the one hand, a low $\text{minlift}$ implies a good resolution of the failure lattice, i.e. the events are separated, but it is costly because more rules are computed and thus more concepts. On the other hand, selecting a high $\text{minlift}$ is cheaper in terms of number of computed rules and concepts but many events are aggregated.

**Example** We illustrate our approach on a faulty version of the Space program from the SIR repository (Mutant 6). Space is written in C and contains 9126 lines of code (3638 of which are executable). For the example, events are the executed lines. An excerpt of the faulty code source is presented Figure 3. The fault, at line 3464, is related to a pointer. Instead of assigning the value of the memory pointed by $tp$ to the memory pointed by $curr_ptr$, both pointers point to the same memory space.

Figure 4 shows the related failure lattice. Each concept of that lattice is actually labelled by a rule but we have omitted to display some of them for readability reasons. The thresholds are set to 86.79% of the test cases for $\text{minsup}$ and 1 for $\text{minlift}$. The failure lattice contains one support cluster only. The support value of this support cluster is 86.79% which corresponds to the proportion of failed executions for this case; all failed executions are therefore covered by all computed rules.

As already mentioned, the rules are partially ordered in the failure lattice. For example, rule 1 is more spe-
specific than rule 2 (rule 1 < rule 2). Rule 1 contains in its premise all events that belong to the premise of rule 2, i.e., all lines that label concepts above concept 2 (3416, 3412, 3403, 4017,...). In addition, rule 1 contains in its premise 12 events that do not belong to the premise of rule 2: 3466, 3464, 3456, and nine others not shown: 3445, 3439, 3438, 3434, 3433, 3432, 3426, 3425, 3419. The most specific rules to explain the faults are at the bottom of the failure lattice. That’s why the failure lattice is explored bottom-up. The faulty line is in the most specific concept, at the bottom of the lattice. It is grouped together with eleven other lines. They are always executed together with line 3464 for this test suite. As can be seen on Figure 3, they all belong to the same basic block.

4. Experimental Study

We compare DeLLIS with existing methods on the Siemens suite. Then, we show that the method scales up for the Space program. DeLLIS uses a set of independent tools for each task of the process. The programs are traced with the C tracer gcov. The trace contains the executed lines and the verdict. The association rules are computed with the algorithm proposed in [3].

4.1. Siemens Suite Programs

In this section, we quantitatively compare DeLLIS to the methods for which results are available regarding the Siemens suite. These methods are Tarantula [9], Intersection Model (Inter Model), Union Model, Nearest Neighbor (NN) [12] and Delta Debugging (DD) [4]. The Siemens suite contains 7 programs described in Table 2. There are a total of 132 mutants that contain a single fault on a single line. $F_m$ denotes the fault of mutant $m$. Each program is accompanied by a test suite (a list of test cases). Some mutants do not fail for the test suites. They cannot be considered by any fault localization method. Some mutants fail with an exception (say, “segmentation fault”). They are not considered by other methods. Our method could consider those failures; it could treat them as special cases of failures. However, we do not consider them in these experiments for the sake of fair comparison. Thus, there remains 121 usable mutants.

For the experiments, we set statistical indicator values such that the lattices for all the debugged programs are of similar size. We have chosen, arbitrarily, to obtain about 150 concepts in the failure lattices. That number makes the failure lattices easy to display and check by hand. Nevertheless, in the process of debugging a program, it is not essential to display rule lattices in their globality.

Experimental Settings We evaluate two strategies. The first strategy consists in starting from the bottom and traversing the lattice to go straightforwardly to the fault concept. This corresponds to the best case of our approach. This strategy assumes a competent debugging oracle, who knows at each step the best way to find the fault with clues. The second strategy consists in choosing a random path from the bottom in the lattice until a fault is located. This strategy assumes a debugging oracle who has little knowledge about the program, but is still able to recognize the fault when presented to him. Using a “Monte Carlo” approach and thanks to the law of large numbers, we compute an average estimation of the cost of this strategy. The navigation strategies to explore the failure lattice are implemented inside Camelis [6].

Metrics We use the Expense metrics of Jones et al. [8]:

$$\text{Expense}(F_m) = \frac{|\text{fault-context}(F_m)|}{\text{size of program}} \times 100$$

where $\text{fault-context}(F_m)$ is the set of lines explored before finding $F_m$. The Expense metrics measures the per-

| Program      | Description       | $|\text{Mutants}|$ | LOC  | $|\text{Tests}|$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>lexical analyzer</td>
<td>7</td>
<td>564</td>
<td>4130</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>pattern replacement</td>
<td>10</td>
<td>510</td>
<td>4115</td>
</tr>
<tr>
<td>schedule</td>
<td>priority scheduler</td>
<td>32</td>
<td>563</td>
<td>5542</td>
</tr>
<tr>
<td>schedule2</td>
<td>priority scheduler</td>
<td>9</td>
<td>412</td>
<td>2650</td>
</tr>
<tr>
<td>tcas</td>
<td>altitude separation</td>
<td>10</td>
<td>307</td>
<td>2710</td>
</tr>
<tr>
<td>tot_info</td>
<td>information measure</td>
<td>41</td>
<td>173</td>
<td>1608</td>
</tr>
</tbody>
</table>

Table 2. Siemens suite programs

4.2. Space Program

In this section, we show that the method scales up for the Space program. The Space program contains 16 mutants that contain a single fault on a single line. Each mutant is accompanied by a test suite (a list of test cases). Some mutants do not fail for the test suites. They cannot be considered by any fault localization method. Some mutants fail with an exception (say, “segmentation fault”). They are not considered by other methods. Our method could consider those failures; it could treat them as special cases of failures. However, we do not consider them in these experiments for the sake of fair comparison. Thus, there remains 15 usable mutants.

For the experiments, we set statistical indicator values such that the lattices for all the debugged programs are of similar size. We have chosen, arbitrarily, to obtain about 150 concepts in the failure lattices. That number makes the failure lattices easy to display and check by hand. Nevertheless, in the process of debugging a program, it is not essential to display rule lattices in their globality.

Experimental Settings We evaluate two strategies. The first strategy consists in starting from the bottom and traversing the lattice to go straightforwardly to the fault concept. This corresponds to the best case of our approach. This strategy assumes a competent debugging oracle, who knows at each step the best way to find the fault with clues. The second strategy consists in choosing a random path from the bottom in the lattice until a fault is located. This strategy assumes a debugging oracle who has little knowledge about the program, but is still able to recognize the fault when presented to him. Using a “Monte Carlo” approach and thanks to the law of large numbers, we compute an average estimation of the cost of this strategy. The navigation strategies to explore the failure lattice are implemented inside Camelis [6].

Metrics We use the Expense metrics of Jones et al. [8]:

$$\text{Expense}(F_m) = \frac{|\text{fault-context}(F_m)|}{\text{size of program}} \times 100$$

where $\text{fault-context}(F_m)$ is the set of lines explored before finding $F_m$. The Expense metrics measures the per-
For both strategies, the best strategy and the random strategy, Expense is thus as follows:

\[ \text{Expense}_B(F_m) = \frac{\text{fault-context}_{\text{best}}(F_m)}{\text{size of program}} \times 100. \]

\[ \text{Expense}_R(N, F_m) = \frac{1}{N} \times \sum_{i=1}^{N} \frac{\text{fault-context}(F_m)_i}{\text{size of program}} \times 100. \]

\text{Expense}_R is the arithmetic mean of the percentages of lines needed to find the fault during \( N \) random explorations of the failure lattice. A random exploration is a sequence of random paths in the rule lattice. A random path of the failure lattice is selected. If the fault is found on that path, the execution stops and returns the fault context. Otherwise a new path is randomly selected, the previous fault context is added to the new fault context and so on until the fault is found. In the experiments, if after 20 selections the fault stays unfound, the returned fault context consists of all the lines of the lattice. We have noted that between 10 and 50, the computed results are not significantly different. Number \( N \) is chosen so that the confidence on \text{Expense}_R is about 1%.

For any method \( M \), \text{Expense}_M allows to compute \( \text{Freq}_M(\text{cost}) \) which measures how many failures are explained by \( M \) for a given cost:

\[ \text{Freq}_M(\text{cost}) = \frac{\#(\text{m} \text{Exp} \leq \text{cost})}{\text{total number of versions}} \times 100. \]

**Results** \( \text{Freq}_M(\text{cost}) \) can be plotted on a graph, so that the area under the curve indicates the global efficiency of method \( M \). Figure 5 shows the curves for all the methods. The DeLLIS strategies are represented by the two thick lines. For DeLLIS Best about 21% of mutant faults are found when inspecting less than 1% of the source code, and 100% when inspecting less than 70%. The best strategy of DeLLIS is as good as the best method, Tarantula, and the random strategy of DeLLIS is not worse than the other methods. We conjecture that the strategy of a human debugger is between both strategies. A very competent programmer with a lot of knowledge will choose relevant concepts to explore, and will therefore be close to the best strategy measured here. A regular programmer will still have some knowledge and will be in any case much better than the random traversal of the random strategy. Our method is equivalent to the best methods dealing with fault on a single line, when comparing the number of lines to explore. Furthermore, it gives more than a set of independent lines, it highlights the existing relations between them.

**4.2. The Space Program**

In this section, we study the behavior of our method on a program of several thousands of lines, the Space program already mentioned in Section 3. In particular, we present how the expense value and the number of concepts of the best strategy vary with respect to the \text{minlift} value.

Space has 38 associated mutants. Each mutant contains a single fault. Some mutants of the program do not fail or fail with segmentation fault. SIR contains 27 usable mutants and 1000 test suites. For the experiment, we randomly choose one test suite such that, for each of the 27 mutants, at least one test case of the test suite fails.

In the experiments, the support threshold is set to the max value of the support. The mutants contain a single fault.
fault. The faulty line is thus executed by all failed executions. Different values of the lift threshold are set for each mutant in order to study the behavior of DeLLIS. The lift threshold is first set to a value close to the max value: \( (\text{maxlift} - 1) \times 0.95 + 1 \) (for \text{maxlift} see Section 3). The lift is then set to \( (\text{maxlift} - 1)/3 + 1 \).

**Results** Figure 6 shows the expense values for each mutant when \( \text{minlift} \) is set to 95% of \text{maxlift} (light blue) and 33% of \text{maxlift} (dark red). The Expense value is presented in a logarithmic scale. The expenses are much higher with the larger \( \text{minlift} \). For \( \text{minlift} = 95\% \) of \text{maxlift}, some mutants, for example mutant 1, have an expense value equal to 100%, representing 3638 lines, namely the whole program. For \( \text{minlift} = 33\% \) of \text{maxlift}, for all but 4 mutants, the percentage of investigated lines is below 10%. And for most of them, it has dropped below 1%. Note that 1 line corresponds to 0.03% of the program. Thus, 0.03% is the best Expense value that can be expected. Other experiments on intermediate values of \( \text{minlift} \) confirm that the lower \( \text{minlift} \), the lower the expense value is, and the fewer lines have to be examined by a competent debugger.

Figure 7 sheds some light on the results of Figure 6 and also explains why it is not always possible to start with a small \( \text{minlift} \). The figure presents the number of concepts for each mutant when \( \text{minlift} \) is set to 95% of \text{maxlift} and 33% of \text{maxlift}. The number of concepts is also presented in a logarithmic scale. For \( \text{minlift} = 95\% \) \text{maxlift}, for all but one mutant, either no rule or a single rule is computed. In the first case, the whole program has to be examined (Mutant 1). In the second case, the expense value is proportional to the number of events in the premise of the rule. For example, this represents 1571 lines for Mutant 5. When reducing \( \text{minlift} \), the number of concepts increases and the labelling of the concepts is reduced. Traversing the failure lattice, at each step, fewer lines have to be examined, hence the better results for the Expense values with a low \( \text{minlift} \).

However, for \( \text{minlift} = 33\% \) \text{maxlift}, for almost half of the mutants, the number of concepts is above a thousand and for one mutant it is even above 10000. Therefore, whereas Expense decreases when \( \text{minlift} \) increases, the cost of computing the failure lattices increases. Furthermore, when the number of concepts increases so does the number of possible paths in the lattice. For the best strategy this does not make a difference. However, in reality even a competent debugger is not guaranteed to always find the best path at once. Thus, a compromise must be found in practice between the number of concepts and the size of their labelling. At present, we start computing the rules with a relatively low \( \text{minlift} \). If the lattice exceeds a given number of concepts, the computation is aborted and restarted with a higher value of \( \text{minlift} \) following a divide and conquer approach.

5 Conclusion

The DeLLIS process combines two data mining techniques, association rules and formal concept analysis in order to compute a failure lattice that gives clues for fault localization. The partial ordering of the lattice shows the dependencies between the code instructions. Experiments with the Siemens suite simulate a highly competent debugger and a mildly competent one. They show that, in both cases, DeLLIS quantitatively compares well with other methods. Experiments with the Space program show that the method scales up for programs of several thousand lines by tuning the lift threshold.

**References**


