Data Mining for Fault Localization: towards a Global Debugging Process

Peggy Cellier, Mireille Ducassé, Sébastien Ferré, Olivier Ridoux

Data Mining for Fault Localization: towards a Global Debugging Process

Peggy Cellier & Mireille Ducassé - IRISA/INSA Rennes
Sébastien Ferré & Olivier Ridoux - IRISA/Université de Rennes

2018

1 Introduction

The IEEE Standard Glossary of Software Engineering Terminology [13] defines three terms, *mistake*, *fault*, and *failure*, that bear in themselves the idea of a *bugging process*. In this process, a programmer, through incompetency, distraction, mental strain, etc., makes one or several mental *mistakes*, which lead to one or several *faults* in the program, which remain undetected until an execution *fails*. Observing a failure yields an anomaly report, which triggers the *debugging process*, that consists in going upstream from the failure, to some fault, and ideally to a mistake. The whole thing is made even more tricky if one considers that there does not generally exist a single programmer, but several programmers, that a programmer may commit several mistakes, that a mistake may cause several faults (eg. a systematic programming fault), and that a fault is not necessarily something that causes a failure when executed, but may also be something lacking, whose non-execution causes a failure (eg. a missing initialization). In the latter case, the fault will appear under the form of reading a non-initialized variable.

The debugging process is really a difficult one, with a lot of trials and errors. The *debugging person* (as opposed to the *debugging tool*) will try to reproduce the signaled failure (if the failure is not reproducible, things are getting even worse), most probably to simplify the circumstances under which it happens, then try to localize the fault, and try to understand the mistake at its origin, and finally try to correct the fault. Having understood the mistake, the debugging person may also try to correct still undetected faults, that are consequences of the same mistake. All this represents a complex network of causes and effects, of test inputs and outputs, of hypotheses and refutations or confirmations. The complexity is made worse by the size of the problem, e.g. dealing with programs of ever increasing size. Thus, it is tempting to assist this process with automated tools.

The automated tool envisioned in this chapter is fairly rustic; it analyzes traces of the execution of passed and failed test cases and returns trace elements as fault hints. The events are then checked by a *debugging oracle* that

deduces fault locations from the hints. We assume that the debugging oracle is the debugging person. We also assume that the debugging oracle is *competent*, namely that presented with a set of hints that indicate a fault, she will correctly deduce the fault; we call this the *competent debugger hypothesis*. This hypothesis parallels the *competent programmer hypothesis*, which is familiar in testing theory [9].

Software engineering processes generate a lot of data, and several authors advocate the use of *data mining* methods to deal with it (eg. in *Int. Conf. on Software Engineering and Knowledge Discovery* and *Int. Conf. on Software Engineering and Knowledge Engineering*). There exist many data mining methods, with different merits, but very often the very first progress is to simply consider as data what was previously considered as mere by-product of a process. As one of the first historical example of uncovering new knowledge from pre-existing data is Florence Nightingale's (1820-1910) demonstration that soldiers died more often from bad sanitary conditions in military hospitals than from battle wounds. For her demonstration, she gathered previously ignored data and presented it in revealing graphics. This example demonstrates that data mining is itself a process with important questions, from the selection and gathering of data, to the presentation of the results. In the fault localization context the questions are: Which existing data can be leveraged to improve the localization? Which presentation of the results is the best suited to the debugging process?

Among the data mining approaches which one can oppose numeric methods and symbolic methods. As software engineering data are symbolic by nature, we propose to use symbolic methods. Furthermore, symbolic methods tend to lend themselves naturally to give explanations, and this is exactly what we are looking for in fault localization. Indeed, we prefer a system with the capacity of saying "The failure has to do with the initialization of variable `x`" to a system limited to saying "The fault is in this million lines with probability 0.527". Therefore, we propose to use *Association Rules* (AR) and *Formal Concept Analysis* (FCA) as data mining techniques (see a survey on software engineering applications of FCA in [28]). Formal concept analysis and association rules deal with collections of objects and their features. The former extracts contextual truth, like "In this assembly, all white-haired female wear glasses", while the latter extracts relativized truth, like "In this assembly, carrying an attaché-case increases the chance of wearing a tie". In a fault localization context, the former could say that "all failed tests call method `m`", and the latter could discover that "most failed tests call method `m`, which is very seldom called in passed tests".

In the sequel, we will explain our running example for the *Fault Localization Problem* in Section 2, give a brief introduction to *Formal Concept Analysis* and *Association Rules* in Section 3, present our proposition in Section 4, and refine it to the case of multiple faults in Section 5. Experiments are presented in Section 6, and discussion and further works in Section 7.

```
public int Trityp(){
[57] int trityp ;
[58] if ((i==0) || (j==0) ||
        (k == 0))
[59]   trityp = 4 ;
[60] else
[61]  {
[62]    trityp = 0 ;
[63]    if ( i == j)
[64]      trityp = trityp + 1 ;
[65]    if ( i == k)
[66]      trityp = trityp + 2 ;
[67]    if ( j == k )
[68]      trityp = trityp + 3 ;
[69]    if (trityp == 0)
[70]    {
[71]      if ((i+j <= k) ||
              (j+k <= i) ||
              (i+k <= j))
[72]          trityp = 4 ;
[73]      else
[74]          trityp = 1 ;
[75]    }
[76]    else
[77]    {
[78]      if (trityp > 3)
[79]          trityp = 3 ;
[80]      else
[81]          if ((trityp == 1)
                   && (i+j > k))
[82]            trityp = 2 ;
[83]          else
[84]            if ((trityp == 2)
                     && (i+k > j))
[85]              trityp = 2 ;
[86]            else
[87]              if((trityp == 3)
                     && (j+k > i))
[88]                 trityp  = 2 ;
[89]              else
[90]                 trityp = 4 ;
[91]    }
[92]   }
[93]   return(trityp) ;}
static public
string conversiontrityp(int i){
[97]  switch (i){
[98]     case 1:
[99]         return "scalen";
[100]    case 2:
[101]        return "isosceles";
[102]    case 3:
[103]        return "equilateral";
[104]    default:
[105]        return "not a ";}}
```

Figure 1: Source code of the Trityp program

2 Running Example

2.1 The program

Throughout this chapter, we use the Trityp program (partly given in Figure 1) to illustrate our method. It is a classical benchmark for test generation methods. Its specification is to classify sets of three segment lengths into four categories: *scalene*, *isosceles*, *equilateral*, *not a triangle*, according to whether a given kind of triangle can be formed with these dimensions, or no triangle at all. The program contains one class with 130 lines of code.

We use this benchmark to explain the ability of data mining process for localizing faults (for more advanced experiments see Section 6). We do so by introducing faults in the program, in order to form slight variants, called *mutants*, and by testing them through a test suite [9]. The data mining process starts with the output of the tests, i.e., execution traces and pass/fail verdicts. The mutants can be found on the web[1], and we use them to illustrate our localization method.

Table 1 presents the eight mutants of the Trityp program that are used in Section 4. The first mutant is used to explain in details the method. For mu-

[1] http://www.irisa.fr/lis/cellier/Trityp/Trityp.zip

Mutant	Faulty line
1	`[84] if ((trityp == 3) && (i+k > j))`
2	`[79] trityp = 0 ;`
3	`[64] trityp = i+1 ;`
4	`[87] if ((trityp != 3) && (j+k > i))`
5	`[65] if (i >= k)`
6	`[74] trityp = 0 ;`
7	`[90] trityp == 3 ;`
8	`[66] trityp = trityp+20 ;`

Table 1: Mutants of the Trityp program

tant 1, one fault has been introduced at Line 84. The condition `(trityp == 2)` is replaced by `(trityp == 3)`. That fault causes a failure in two cases:

1. The first case is when `trityp` is equal to 2; execution does not enter this branch and goes to the default case, at Lines 89 and 90.

2. The second case is when `trityp` is equal to 3; execution should go to Line 87, but due to the fault it goes to Line 84. Indeed, if the condition `(i+k>j)` holds, `trityp` is assigned to 2. However, `(i+k>j)` does not always imply `(j+k>i)`, which is the real condition to test when `trityp` is equal to 3. Therefore, `trityp` is assigned to 2 whereas 4 is expected.

The faults of mutants 2, 3, 6 and 8 are on assignments. The faults of mutants 4, 5 and 7 are on conditions. We will also develop our method for multiple faults situations in Section 5. In this case, we simply combine several mutations to form new mutants.

2.2 The testing process

We assume the program is passed through a test suite. For the Trityp program, 400 test cases have been generated with the *Uniform Selection of Feasible Paths* method of Petit and Gotlieb [23]. With that method, all feasible execution paths are uniformly covered.

Other testing strategies, like non-regression tests [20] or test-driven development [3] are possible. However, for the sake of illustration we simply assume we have a program and a test suite, without knowing how they have been produced.

3 Formal Concept Analysis and Association Rules

Formal Concept Analysis (FCA [12]) and Association Rules (AR [1]) are two well-known methods for symbolic data mining. In their original inception, they both consider data in the form of an *object-attribute* table. In the FCA world, the table is called a *formal context*. In the AR world, objects are called *transactions*,

	size			sun distance		moons	
	small	medium	large	near	far	with	without
Mercury	×			×			×
Venus	×			×			×
Earth	×			×		×	
Mars	×			×		×	
Jupiter			×		×	×	
Saturn			×		×	×	
Uranus		×			×	×	
Neptune		×			×	×	

Table 2: The Solar system context

attributes are called *items*, so that a line represents the items present in a given transaction. This comes from one of the first application of AR, namely the *basket analysis* of retail sales. We will use both vocabularies interchangeably according to context.

Definition 1 (formal context and transactions) *A formal* context*, $\mathcal{K}$, is a triple $(\mathcal{O}, \mathcal{A}, d)$ where $\mathcal{O}$ is a set of* objects*, $\mathcal{A}$ is a set of* attributes*, and d is a relation in $\mathcal{O} \times \mathcal{A}$. We write $(o, a) \in d$ or $o\,d\,a$ equivalently.*

In the AR world, $\mathcal{A}$ is called a set of items*, or* itemset*, and each $\{i \in \mathcal{A} \mid o\,d\,i\}$ is the o-th* transaction*.*

For visualization sake, we will consider objects as labeling lines, and attributes as labelling columns of a table. A cross sign at the intersection of line o and column a indicates that object o has attribute a.

Table 2 is an example of context. The objects are the planets of the Solar system, and the attributes are discretized properties of these planets: size, distance to sun, and presence of moons. One can observe that all planets without moons are small, but that all planets with moons except two are far from sun. The difficulty is to make similar observations in large data sets.

Both FCA and AR try to answer questions such has "Which attributes entails these attributes?", or "Which attributes are entailed by these attributes?". The main difference between FCA and AR is that FCA answers these questions to the letter, i.e., the mere exception to a candidate rule kills the rule, though association rules are accompanied by statistical indicators. In short, association rules can be *almost true*. As a consequence, in FCA rare events are represented as well as frequent event, whereas in AR, frequent events are distinguished.

3.1 Formal Concept Analysis

FCA searches for sets of objects and sets of attributes with equal significance, like {Mercury, Venus} and {without moons}, and then order the significances by their specificity.

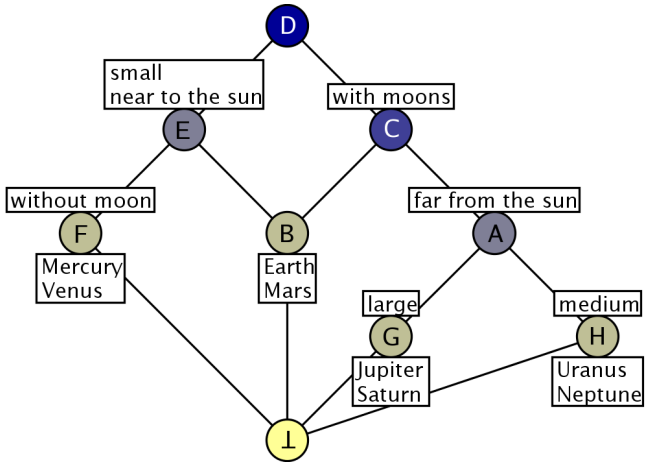

Figure 2: Concept lattice of the Solar system context (see Table 2)

Definition 2 (extent/intent/formal concept) *Let $\mathcal{K} = (\mathcal{O}, \mathcal{A}, d)$ be a formal context.*

$\{o \in \mathcal{O} \mid \forall a \in A. o\, d\, a\}$ is the extent *of a set of attributes $A \subseteq \mathcal{A}$. It is written $extent(A)$.*

$\{a \in \mathcal{A} \mid \forall o \in O. o\, d\, a\}$ is the intent *of a set of objects $O \subseteq \mathcal{O}$. It is written $intent(O)$.*

A formal concept *is a pair (O, A) such that $A \subseteq \mathcal{A}$, $O \subseteq \mathcal{O}$, $intent(O) = A$ and $extent(A) = O$. A is called the intent of the formal concept, and O is called its extent.*

Formal concepts are partially ordered by set inclusion of their intent or extent. $(O_1, A_1) < (O_2, A_2)$ iff $O_1 \subset O_2$. We say that (O_2, A_2) contains *(O_1, A_1).*

In other words, (O, A) forms a formal concept iff O and A are mutually optimal for describing each others; i.e., they have same significance.

Lemma 1 (basic FCA results) *It is worth remembering the following results:*

$extent(\emptyset) = \mathcal{O}$ and $intent(\emptyset) = \mathcal{A}$.

$extent(intent(extent(A))) = extent(A)$ and $intent(extent(intent(O))) = intent(O)$. Hence, $extent \circ intent$ and $intent \circ extent$ are closure *operators.*

$(O_1, A_1) < (O_2, A_2)$ iff $A_1 \supset A_2$.

$(extent(intent(O)), intent(O))$ is always a formal concept, it is written $concept(O)$. In the same way, $(extent(A), intent(extent(A)))$ is always a formal concept, which is written $concept(A)$. All formal concepts can be constructed this way.

Theorem 1 (fundamental theorem of FCA, [12]) *Given a formal context, the set of all its partially ordered formal concepts forms a lattice called the* concept lattice.

Given a concept lattice, the original formal context can be reconstructed.

Figure 2 shows the concept lattice deduced from the Solar system context. It is an example of the *standard representation* of a concept lattice. In this representation, concepts are drawn as colored circles with an optional inner label that serves as a concept identifier, and 0, 1 or 2 outer labels in square boxes. Lines represent non-transitive containment; therefore, the standard representation displays a *Hasse diagram* of the lattice [25]. The figure is oriented such that higher concepts (higher in the diagram) contain lower concepts.

The upper outer label of a concept (e.g. large for concept G), when present, represents the attributes that are new to this concept intent compared with higher concepts; we call it an *attribute label*. It can be proven that if A is the attribute label of concept c, then A is the smallest set of attributes such that $c = concept(A)$. Symmetrically, the lower outer label of a concept (e.g. Jupiter, Saturn for concept G), when present, represents the objects that are new to this concept extent compared with lower concepts; we call it an *object label*. It can be proven that if O is the object label of concept c, then O is the smallest set of objects such that $c = concept(O)$. As a consequence, the intent of a concept is the set of all attribute labels of this concept and higher concepts, and the extent of a concept is the set of all object labels of this concept and lower concepts. E.g., the extent of concept A is {Jupiter, Saturn, Uranus, Neptune}, and its intent is {far from sun, with moons}. In other words, an attribute labels the highest concept to which intent it belongs, and an object labels the lowest concept to which extent it belongs.

It is proven [12] that such a labelling where all attributes and objects are used exactly once is always possible. As a consequence, some formal concepts can be named by an attribute and/or an object, eg. concept G can be called either concept large, Jupiter, or Saturn, but some others like concepts D and $\perp$ have no such names. They are merely unions or intersections of other concepts.

In the standard representation of concept lattice, "a_1 entails a_2" reads as an upward path from $concept(a_1)$ to $concept(a_2)$. Attributes that do not entail each others label incomparable concepts, e.g., attributes small and with moons. Note that there is no purely graphical way to detect that "a_1 nearly entails a_2".

The bottom concept, $\perp$, has all attributes, and usually 0 objects unless some objects have all attributes. The top concept, $\top$, has all objects, and usually 0 attributes, unless some attributes are shared by all objects.

The worst-case time complexity of the construction of a concept lattice is exponential, but we have shown that if the size of the problem can only grow with the number of objects, i.e. the number of attributes per object is bounded, then the complexity is linear [11]. Moreover, though the mainstream interpretation of FCA is to compute the concept lattice at once and use it as a means for presenting graphically the structure of a dataset, we have shown [11, 21] that the concept lattice can be built and explored gradually and efficiently.

3.2 Association rules

FCA is a crisp methodology that is sensitive to every details of the dataset. Sometimes one may wish for a method that is more tolerant to exceptions.

Definition 3 (association rules) *Let $\mathcal{K}$ be a set of transactions, i.e., a formal context seen as a set of lines seen as itemsets. An association rule is a pair (P, C) of itemsets. It is usually written as $P \longrightarrow C$.*

The P part is called the premise*, and the C part the* conclusion*.*

Note that any $P \longrightarrow C$ forms an association rule. It does not mean it is a relevant one. Statistical indicators give hints at the relevance of a rule.

Definition 4 (support/confidence/lift) *The* support *of a rule $P \longrightarrow C$, written $sup(P \longrightarrow C)$, is defined as*[2]

$$\|extent(P \cup C)\| \ .$$

The normalized support *of a rule $P \longrightarrow C$ is defined as*

$$\frac{\|extent(P \cup C)\|}{\|extent(\emptyset)\|} \ .$$

The confidence *of a rule $P \longrightarrow C$, written $conf(P \longrightarrow C)$, is defined as*

$$\frac{\|sup(P \longrightarrow C)\|}{\|sup(P \longrightarrow \emptyset)\|} = \frac{\|extent(P \cup C)\|}{\|extent(P)\|} \ .$$

The lift *of a rule $P \longrightarrow C$, written $lift(P \longrightarrow C)$, is defined as*

$$\begin{aligned} \frac{\|conf(P \longrightarrow C)\|}{\|conf(\emptyset \longrightarrow C)\|} &= \frac{\|sup(P \longrightarrow C)\|}{\|sup(P \longrightarrow \emptyset)\|} \Big/ \frac{\|sup(\emptyset \longrightarrow C)\|}{\|sup(\emptyset \longrightarrow \emptyset)\|} \\ &= \frac{\|extent(P \cup C)\| \times \|extent(\emptyset)\|}{\|extent(P)\| \times \|extent(C)\|} \ . \end{aligned}$$

Support measures the prevalence of an association rule in a data set. E.g., the support of near sun $\longrightarrow$ with moon is 2. Normalized support measures its prevalence as a value in $[0, 1]$, i.e. as a probability of occurrence. E.g., the normalized support of near sun $\longrightarrow$ with moon is $2/8 = 0.25$. It can be read as the probability of observing the rule in a random transaction of the context. It would seem that the greater the support the better it is, but very often one must be happy with a very small support. This is because in large contexts with many transactions and items, any given co-occurrence of several items is a rare event. Efficient algorithms exist for calculating all ARs with a minimal support (e.g. [2, 4, 22, 27]).

Confidence measures the "truthness" of an association rule as the ratio of the prevalence of its premise and conclusion together on the prevalence of its premise

[2] where $\|.\|$ is the cardinal of a set; how many elements it contains.

alone. Its value is in $[0, 1]$, and for a given premise the bigger is the better; in other words, the less exceptions to the rule considered as a logical implication is the better. E.g., the confidence of near sun $\longrightarrow$ with moon is $2/4 = 0.5$. This can be read as the conditional probability of observing the conclusion knowing that the premise holds. However, there is no way to tell whether a confidence value is good in itself. In other words, there is no absolute threshold above which a confidence value is good.

Lift also measures "truthness" of an association rule, but it does so as the increase of the probability of observing the conclusion when the premise holds wrt. when it does not hold. In other words, it measures how the premise of a rule increases the chance of observing the conclusion. A lift value of 1 indicates that the premise and conclusion are independent. A lower value indicates that the premise *repels* the conclusion, and an higher value indicates that the premise *attracts* the conclusion. E.g., the lift of near sun $\longrightarrow$ with moon is $0.5/0.75$, which shows that attribute near sun repels attribute with moon; to be near the sun diminishes the probability of having a moon. On the opposite, rule near sun $\longrightarrow$ without moon has support 0.25, confidence 0.5, but lift $0.5/0.25$, which indicates an attraction; to be near the sun augments the probability of not having a moon. The two rules have identical supports and confidences, but opposed lifts. In the sequel, we will use support as an indicator of the prevalence of a rule, and lift as an indicator of its "truthness".

4 Data Mining for Fault Localization

We consider a debugging process in which a program is tested against different test cases. Each test case yields a transaction in the AR sense, in which attributes correspond to properties observed during the execution of the test case, say executed line numbers, called functions, etc. (see Section 7.2 on Future works for more on this), and two attributes, *PASS* and *FAIL* represent the issue of the test case (again see Future works for variants on this). Thus, the set of all test cases yields a set of transactions that form a formal context, which we call a *trace context*. The main idea of our data mining approach is to look for a formal explanation of the failures.

4.1 Failure rules

Formally, we are looking for association rules following pattern $P \longrightarrow FAIL$. We call these rules *failure rules*. A failure rule propose an explanation to a failure, and this explanation can be evaluated according to its support and lift.

Note that failure rules have a variable premise P and a constant conclusion *FAIL*. This simplifies a little bit the management of rules. For instance, relevance indicators can be specialized as follows:

Definition 5 (relevance indicators for failure rules)

$$\begin{aligned} sup(P \longrightarrow FAIL) &= \|extent(P \cup \{FAIL\})\| \ , \\ conf(P \longrightarrow FAIL) &= \frac{\|extent(P \cup \{FAIL\})\|}{\|extent(P)\|} \ , \\ lift(P \longrightarrow FAIL) &= \frac{\|extent(P \cup \{FAIL\})\| \times \|extent(\emptyset)\|}{\|extent(P)\| \times \|extent(\{FAIL\})\|} \ . \end{aligned}$$

Observe that $\|extent(\emptyset)\|$ and $\|extent(\{FAIL\})\|$ are constant for a given test suite. Only $\|extent(P)\|$ and $\|extent(P \cup \{FAIL\})\|$ depend on the failure rule.

It is interesting to understand the dynamics of these indicators when new test cases are added to the trace context.

Lemma 2 (dynamics of relevance indicators wrt. test suite) *Consider a failure rule $P \longrightarrow FAIL$:*

A new passed test case that executes P will leave its support unchanged (normalized support will decrease slightly[3]), will decrease its confidence, and will decrease its lift slightly if P is not executed by all test cases.

A new passed test case that does not execute P will leave its support and confidence unchanged (normalized support will decrease slightly), and will increase its lift.

A new failed test case that executes P will increase its support and confidence (normalized support will increase slightly), and will increase its lift slightly if P is not executed by all test cases.

A new failed test case that does not execute P will leave its support and confidence unchanged (normalized support will decrease slightly), and will decrease its lift.

In summary, support and confidence grow with new failed test cases that execute P, and lift grows with failed test cases that execute P, or passed test cases that do not execute P. Failed test cases that execute P increase all the indicators, but passed test cases that do not execute P only increase lift[4].

Another interesting dynamics is what happens when P increases.

Lemma 3 (dynamics of relevance indicators wrt. premise) *Consider a failure rule $P \longrightarrow FAIL$, and replacing P with P' such that $P' \supsetneq P$:*

Support will decrease (except if all test cases fail, which should not persist). One says $P' \longrightarrow FAIL$ is more specific *than $P \longrightarrow FAIL$.*

Confidence and lift can go either ways, but both in the same way because $\frac{\|extent(\emptyset)\|}{\|extent(\{FAIL\})\|}$ is a constant.

[3] Slightly: if most test cases pass, which they should do eventually.

[4] Observing more white swans increases the belief that swans are white, but observing non-white non-swans increases the interest of the white swan observations. Observing a non-white swan does not change the support of the white swan observations, but it decreases its confidence and interest. But still the interest can be great if there are more white swans and non-white non-swans that non-white swans.

Test case	Executed lines				Verdict	
	57	58	...	105	*PASS*	*FAIL*
t_1	×	×		×	×	
t_2	×	×		×		×
...	...	...	...	...	...	...

Table 3: A trace context

For the sequel of the description or our proposal, we assume that the attributes recorded in the trace context are line numbers of executed statements. Since the order of the attributes in a formal context does not matter, neither their multiplicities, this forms an abstraction of a standard trace (see a fragment of such a trace context in Table 3). Thus, explanations for failures will consist in line numbers; lines that increase the risk of failure when executed. Had other trace observations be used, and the explanations would have been different (see Section 7.2). For faults that materialize in faulty instructions, it is expected that they will show up as explanation to failed test cases. For other faults that materialize in missing instructions, they will still be visible in actual lines that would have been correct if the missing lines where present. For instance, a missing initialization will be seen as the faulty consultation of a non initialized variable[5]. It is up to the competent debugger to conclude from faulty consultations that an initialization is missing. Note finally that the relationships between faults and failures are complex:

- executing a faulty line does not necessarily cause a failure- e.g. a fault in a line may not be stressed by a case test (e.g. faulty condition `i > 1` instead of the expected `i > 0`, tested with `i` equals to 10), or a faulty line that is "corrected" by another one.
- absolutely correct lines can apparently cause failure- e.g. lines of the same basic block [29] as a faulty line (they will have *exactly* the same distribution as the faulty line), or lines whose precondition a distant faulty part fails to establish.

Failure rules are selected according to a minimal support criteria. However, there are too many such rules, and it would be inconvenient to list them all. We have observed in Lemma 3 that more specific rules have less support. However, it does not mean that less specific rules must be preferred. For instance, if the program has a mandatory initialization part, which always executes a set of lines I, rule $I \longrightarrow FAIL$ is a failure rule with maximal support, but it is also the less informative. On the contrary, if all failures are caused by executing a set of lines $F \supset I$, rule[6] $F \backslash I \longrightarrow FAIL$ will have same support as $F \longrightarrow FAIL$, but it will be the most informative. In summary, maximizing support is good, but it is not the definitive criteria for selecting informative rules.

[5] Sir, it's not my fault! It was your responsibility to check for initialization!

[6] where .\. is set subtraction; the elements of a first set that do not belong to a second set.

Rule id	Executed lines										
	17	58	66	81	84	87	90	105	93	···	113
r_1	×	×	×	×	×	×	×	×	×		×
r_2	×	×		×	×	×	×		×		×
···										···	
r_8	×	×		×					×		×
r_9	×	×							×		×

Table 4: Failure context for mutant 1 of the Trityp program with $min\_lift = 1.25$ and $min\_sup = 1$ (fault for mutant 1 at Line 84, see Table 1)

Another idea is to use the lift indicator instead of support. However, lift does not grow monotonically with premise inclusion. So finding rules with a minimal lift cannot be done more efficiently than by enumerating all rules, and then filtering them.

4.2 Failure lattice

We propose to use FCA to help navigating in the set of explanations. The idea is as follows:

Definition 6 (failure lattice) *Form a formal context with the premises of failure rules. The rules identifiers are the objects, and their premises are the attributes (in our example line numbers) (see an example in Table 4). Call it the* failure context*.*

Observe that the failure context is special in that all premises of failure rules are different from each others[7]*. Thus, they are uniquely determined by their premises (or itemsets). Thus, it is not necessary to identify them by objects identifiers.*

Apply FCA on this formal context to form the corresponding concept lattice. Call it the failure lattice*. Its concepts and labelling display most specific explanations to groups of failed tests.*

Since object identifiers are useless, replace object labels by the support and lift of the unique rule that labels each concept. This forms the failure lattice *(see Figure 3). The overall trace mining process is summarized in Figure 4.*

Observe the following:

Lemma 4 (properties of the failure lattice) *The most specific explanations (i.e. the largest premises) are at the bottom of the lattice. On the contrary, the least specific failure rules are near the top. For instance, line numbers of a prelude sequence executed by every test cases will label topmost concepts.*

[7]This is a novel property with respect to standard FCA where nothing prevents two different objects to have the same attributes.

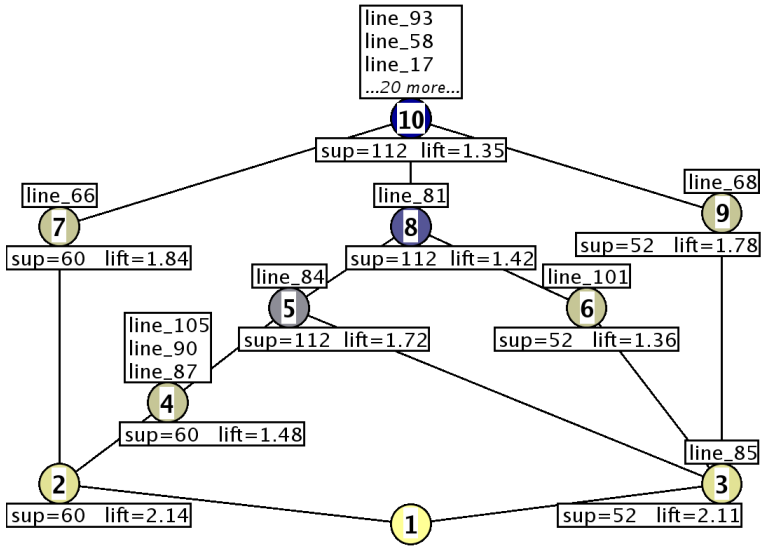

Figure 3: Failure lattice associated to the failure context of Table 4 (for mutant 1, the fault is at line 84)

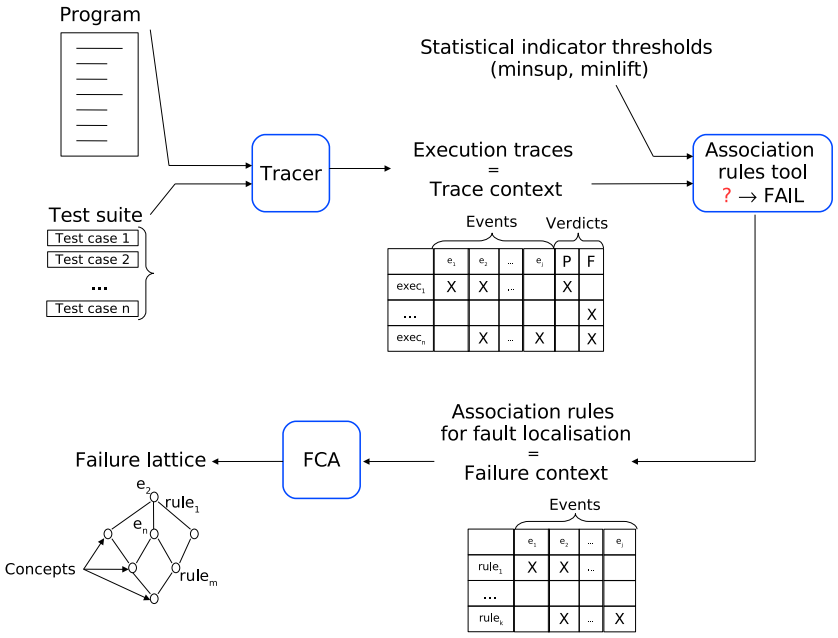

Figure 4: The trace mining process

The explanations with the smallest support are at the bottom of the lattice. E.g. line numbers executed only by specific failed test cases will label concepts near bottom.

Support increases when going upstream, from bottom to top. This we call the global monotony of support ordering. *This is a theorem [5].*

Lift does not follow any global monotony behaviour.

Concepts form clusters of comparable concepts with same support; eg. concepts 2, 4, and 7 in Figure 3 form a cluster of rules with support 60. We call them support clusters. *This means that explanations of increasing size represent the same group of failures.*

In a support cluster a unique concept has the largest extent. We call it the head concept *of the support cluster. It corresponds to the explanation with the highest lift value in the support cluster. More generally, lift decreases when going bottom-up in a support cluster. We call this behaviour the* local monotony of lift ordering*, and it is also a theorem [5].*

It is useless to investigate other explanations than the head concepts. This can be done by a bottom-up exploration of the failure lattice.

In the lattice of Figure 3, only concepts 2 (head of support cluster with value 60), 3 (head of support cluster with value 52), and 5 (head of support cluster with value 112) need be presented to the debugging oracle. Concept 5 has Line 84 in its attribute label, which is the location of the fault in this mutant. The local monotony of lift ordering shows that the lift indicator can be used as a metric, but only inside support clusters.

The process that we have presented is dominated by the choice of a minimal value for the support indicator. Recall that the support of an explanation is simply the number of simultaneous realizations of its items in the failure context, and that normalized support is the ratio of this number on the total number of realizations. In this application of ARs, it is more meaningful to use the non normalized variant because it directly represents the number of failed test cases covered by an explanation. So, what is a good value for the minimal support? First, it cannot be larger than the number of failed test cases ($= \|extent(FAIL)\|$), otherwise no $P \longrightarrow FAIL$ rule will show up. Second, it cannot be less that 1. The choice in between 1 and $\|extent(FAIL)\|$ depends on the nature of the fault, but in any case, experiments show that acceptable minimum support are quite low, a few percents of the total number of test cases.

A high minimal support will filter out all faults that are the causes of less failures than this threshold. Very singular faults will require a very small support, eventually 1, to be visible in the failure lattice. This suggests to start with a high support to localize the most visible faults, and then decrease the support to localize less frequently executed faults. So doing, the minimal support acts as a *resolution* cursor; a coarse resolution will show the largest features at low cost, and a finer resolution will be required to zoom in smaller features, at higher cost.

We have insisted using lift instead of confidence as a "truthness" indicator, because it lends itself more easily to an interpretation (recall Definition 4 and

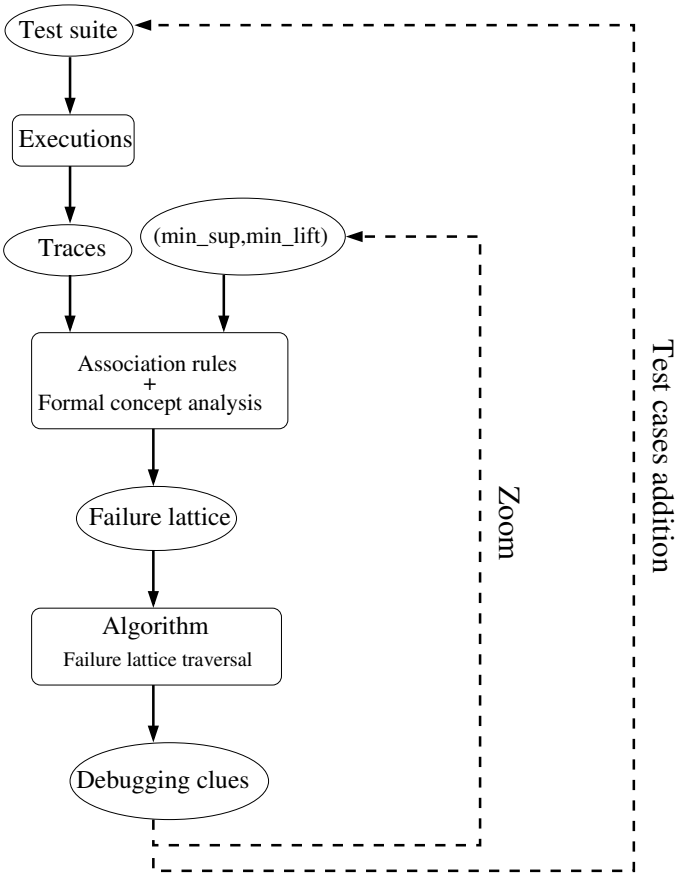

Figure 5: The global debugging process

subsequent comments). However, in the case of failure rules the conclusion is fixed (= *FAIL*), and both indicators increase and decrease in the same way when the premise changes (recall Lemma 3). The only difference is that the lift indicator yields a normalized value (1 is independence, bellow 1 is repulsion, over 1 is attraction). So, what is the effect of a minimum lift value? Firstly, if it is chosen larger or equal to 1, it will eliminate all failure rules that show a repulsion between the premise and conclusion. Secondly, if it is chosen strictly greater than 1, it will eliminate failure rules that have a lower lift, thus compressing the representation of support clusters, and eventually eliminating some support clusters. So doing, the minimal lift also acts as a zoom.

This suggests a *global debugging process* in which the results of an increasingly large test suite are examined with increasing acuity (see Figure 5). Given a test suite, an inner loop computes failure rules, i.e. explanations, with decreasing support, from a fraction of $\|extent(FAIL)\|$ to 1, and build the corresponding failure lattice. In the outer loop, test cases are added progressively, to cope with

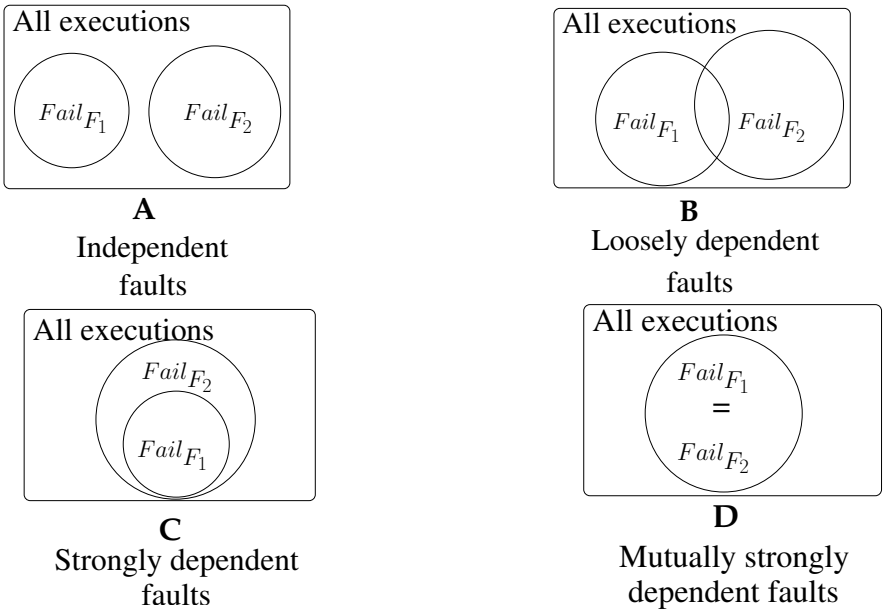

Figure 6: The four Venn diagrams of 2-fault dependency

added functionality (e.g. test driven development), or to cope with new failure reports. Thus, the global debugging process zooms into the failed test cases to find explanations for more and more specific failures.

5 The failure lattice for multiple faults

This section extends the analysis of data mining for fault localization for the multiple fault situation. From the debugging process point of view there is nothing special with multiple faults. Some software engineering life cycle like test-driven development tend to limit the number of fault observed simultaneously, but one can never assume a priori that there is a single fault. Thus, we assume there are one or several faults.

5.1 Dependencies between faults

In the multiple fault case, each failure trace accounts for one or several faults. Conversely, faulty lines are suspected in one or several failure trace. Thus, the inner loop of the global debugging process cannot just stop because a fault is found. The process must go on until all failures are explained. How can this been done without exploring the entire failure lattice?

Consider any pair of two faults F_1 and F_2, and $Fail_{F_1}$ and $Fail_{F_2}$ the sets of failed test cases that detect F_1 and F_2, respectively. We identify 4 types of

possible dependency between the two faults.

Definition 7 (dependencies between faults) *If* $Fail_{F_1} = Fail_{F_2}$ *we say that they are* mutually strongly dependent *(MSD).*

Otherwise, if $Fail_{F_1} \subsetneq Fail_{F_2}$ *we say* F_1 *is* strongly dependent *(SD) from* F_2 *(and vice-versa).*

Otherwise, if $Fail_{F_1} \cap Fail_{F_2} \neq \emptyset$*, we say that they are* loosely dependent *(LD).*

Otherwise, $Fail_{F_1} \cap Fail_{F_2} = \emptyset$*, we say that they are* independent *(ID).*

Note that this classification is not intrinsic to a pair of faults; it depends on the test suite. However, it does not depend arbitrarily from the test suite.

Lemma 5 (how failure dependencies depend on growing test suites) *Assume that the test suite can only grow, then an ID or SD pair can only become LD, and an MSD pair can only become SD or LD.*

This can be summarized as follows:

$$ID \longrightarrow LD \longleftarrow SD \longleftarrow MSD\ .$$

Note also that this knowledge, there being several faults, and the dependencies between them, is what the debugging person is looking for, whereas the trace context only gives hints at this knowledge. The question is: How does it give hints at this knowledge?

The main idea is to distinguish special concepts in the failure lattice that we call *failure concepts.*

Definition 8 (failure concept) *A* failure concept *is a maximally specific concept of the failure lattice whose intent (a set of lines) is contained in a failed execution.*

Recall that the failure rules are an abstraction of the failed execution. For instance, choosing minimal support and lift values eliminates lines that are seldom executed or that do not attract failure. Thus the failure lattice describes exactly the selected failure rules, but only approximately the failed executions. That is why it is interesting; it compresses information, though with loss. The failure concepts in the failure lattice are those concepts that best approximate failed executions. All other concepts contain less precise information. For the same reasons, there are much less failure concepts than failed executions; each failure concept accounts for a group of failures that detects some fault.

The main use for failure concepts is to give a criteria for stopping the exploration of the failure lattice. In a few words,

- the bottom-up exploration of the failure lattice goes from support clusters to support clusters as above,
- the line labels of the traversed concepts are accumulated in a *fault context* sent to the competent debugger,

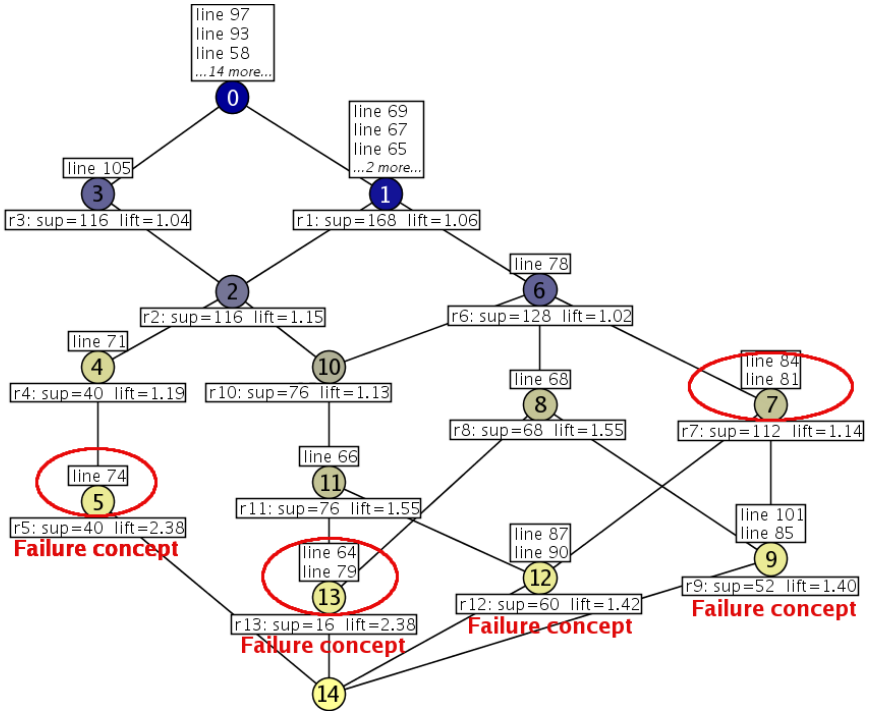

Figure 7: Failure lattice associated to program Trityp with ID faults of mutants 1, 2 and 6.

- any time the competent debugger finds a hint at an actual fault, all the failure concepts under the concept that gave the hint are deemed *explained*.
- the process continues until all failure concepts are explained.

The fault context is the part of the program that the debugging person is supposed to check. We consider its size as a measure of the effort imposed on the debugging person (see also Section 6 on comparative experiments).

Dependencies between faults has an impact on the way failure concepts are presented in the failure lattice.

Lemma 6 (ID faults wrt. failure concepts) *If two faults are ID their lines can never occur in the same failed trace, then no rule contains the two faults, the no concept in the failure lattice contains the two faults. Thus, the two faults will label failure concepts in two different support clusters that have no subconcepts in common except* $\perp$ *(e.g. see Figure 7).*

Concretely, when exploring the failure lattice bottom up, finding a fault in the label of a concept explains both the concept and the concepts underneath, but the faults in the other upper branches remain to be explained. Moreover, the order with which the different branches are explored does not matter.

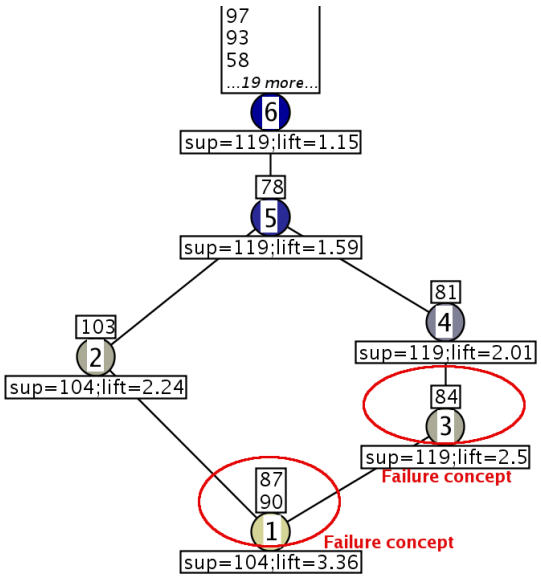

Figure 8: Failure lattice associated to program Trityp with SD faults 1 and 7

Lemma 7 (LD faults wrt. failure concepts) *If two faults are LD some failed traces contain both faults, while other failed traces contain either one or the other fault. They may label concepts in two different support clusters that share common subconcepts.*

Concretely, when exploring the failure lattice bottom-up, finding a fault for a failure concept does not explain the other LD failure concept. Once a fault is found, shared concepts must be re-explored in direction of other superconcepts.

Lemma 8 (SD faults wrt. failure concepts) *If two faults are SD, say F_1 depends on F_2, a failure concept whose intent contains $Line_{F_1}$ will appear as a subconcept of a failure concept whose concept contains $Line_{F_2}$ in a different support cluster (e.g. see Figure 8).*

Therefore, fault F_1 will be found before F_2, but the debugging process must continue because there is a failure concept above.

Lemma 9 (MSD faults wrt. failure concepts) *Finally, if two faults are MSD, they cannot be distinguished by failed executions, and their failure concepts belong to the same support cluster. However, they can sometimes be distinguished by passed executions (e.g., one having more passed execution than the other), and this can be seen in the failure lattice through the lift value.*

All this can be formalized in an algorithm that searches for multiple faults in an efficient traversal of the failure lattice (see Algorithm 1). The failure lattice

Algorithm 1 Failure lattice traversal

1: $C_{toExplore} := FAILURE\ CONCEPTS$
2: $C_{failure\_toExplain} := FAILURE\ CONCEPTS$
3: **while** $C_{failure\_toExplain} \neq \emptyset \wedge C_{toExplore} \neq \emptyset$ **do**
4: **let** $c \in C_{toExplore}$ **in**
5: $C_{toExplore} := C_{toExplore} \setminus \{c\}$
6: **if** the debugging_oracle($label(c)$, $fault\_context(c)$) locates no fault **then**
7: $C_{toExplore} := C_{toExplore} \cup \{$upper neighbours of $c\}$
8: **else**
9: **let** $Explained =$ subconcepts(c) $\cup$ cluster(c) **in**
10: $C_{toExplore} := C_{toExplore} \setminus Explained$
11: $C_{failure\_toExplain} := C_{failure\_toExplain} \setminus Explained$
12: **end if**
13: **end while**

is traversed bottom-up, starting with the failure concepts (step 1). At the end of the failure lattice traversal, $C_{failure\_toExplain}$, the set of failure concepts not *explained* by a fault (step 2), must be empty or all concepts must be already explored (step 3). When a concept, c (step 4), is chosen among the concepts to explore, $C_{toExplore}$, the events that label the concept are explored. Note that the selection of that concept is not determinist. If no fault is located, then the upper neighbours of c are added to the set of concepts to explore (step 7). If, thanks to those new clues, the debugging oracle understands mistakes and locates one or several faults then all subconcepts of c and all concepts that are in the same support cluster are "explained". Those concepts do not have to be explored again (step 10). It means that the failure concepts that are subconcepts of c are *explained* (step 11). The exploration goes on until all failed executions in the failure lattice are *explained* by at least one fault or all concepts have been explored.

Note that at each iteration, $C_{failure\_toExplain}$ can only decrease or remain untouched. It is the competent debugger hypothesis that makes sure that $C_{failure\_toExplain}$ ends at empty when *min_sup* is equal to 1. In case of an incompetent debugging oracle or a too high *min_sup*, the process would end when $C_{toExplore}$ becomes empty, namely when all concepts have been explored.

5.2 Example

For the example of Figure 7, the *min_sup* value is equal to 4 failed executions (out of 400 executions, of which 168 failed executions) and the min_lift value is equal to 1. There are four failure concepts: 5, 13, 12 and 9. Table 5 presents the values of $C_{toExplore}$ and $C_{failure\_toExplain}$ at each iteration of the exploration. We choose to explore the lattice with a queue strategy, it means first in $C_{toExplore}$, first out of $C_{toExplore}$. However, the algorithm does not specify one strategy.

At the begining, $C_{toExplore}$ and $C_{failure\_toExplain}$ are initialized as the set

Iteration	$C_{toExplore}$	$C_{failure\_toExplain}$
0	$\{c_5, c_{13}, c_{12}, c_9\}$	$\{c_5, c_{13}, c_{12}, c_9\}$
1	$\{c_{13}, c_{12}, c_9\}$	$\{c_{13}, c_{12}, c_9\}$
2	$\{c_{12}, c_9\}$	$\{c_{12}, c_9\}$
3	$\{c_9, c_7, c_{11}\}$	$\{c_{12}, c_9\}$
4	$\{c_7, c_{11}, c_8\}$	$\{c_{12}, c_9\}$
5	$\{c_{11}, c_8\}$	$\{\}$

Table 5: Exploration of the failure lattice of Fig. 7.

of all failure concepts (Iteration 0 in Table 5). At the first iteration of the while loop, concept 5 is selected ($c = c_5$). That concept is labelled by line 74. Line 74 actually corresponds to fault 6. Thanks to the competent debugging hypothesis, fault 6 is located. Concept 5, 4 and 14 are thus tagged as *explained*. The new values of $C_{toExplore}$ and $C_{failure\_toExplain}$ are presented at iteration 1 in Table 5.

At the second iteration, concept 13 is selected ($c = c_{13}$). That concept is labelled by lines 64 and 79. Line 79 actually corresponds to fault 2; the competent debugging oracle locates fault 2. Concept 13 is tagged as *explained*.

At the third iteration, concept 12 is selected. That concept is labelled by lines 87 and 90. No fault is found. The upper neighbours, concepts 7 and 11, are added to $C_{toExplore}$ and $C_{failure\_toExplain}$ is unchanged.

At the next iteration, concept 9 is selected. As in the previous iteration no fault is found. The upper neighbour, concept 8, is added to $C_{toExplore}$.

Finally, concept 7 is selected. That concept is labelled by lines 81 and 84. By exploring those lines (new clues) in addition with the *fault context*, i.e. lines that have already been explored: 87, 90, 101 and 85, the competent debugging oracle locates fault 1 at line 84. The fault is the substitution of the test of `trityp = 2` by `trityp = 3`. Concepts 12 and 9 exhibit two concrete realizations (failures) of the fault at line 84 (Concept 7). Concepts 7, 12, 9 are tagged as *explained*. The set of failure concepts to explain is empty, thus the exploration stops. All four faults (for failures above support and lift threshold) are found after the debugging oracle has inspected nine lines.

6 Experiments

We have implemented our approach in a system called DeLLIS, which we compare with existing methods on the Siemens suite. Then, we show that the method scales up for the Space program. DeLLIS combines a set of tools developed independently: e.g., the programs are traced with gcov[8], and the association rules are computed with the algorithm proposed in [6].

[8] http://gcc.gnu.org/onlinedocs/gcc/Gcov57.html

Program	Description	‖Mutants‖	LOC	‖Tests‖
print_tokens	lexical analyzer	7	564	4130
print_tokens2	lexical analyzer	10	510	4115
replace	pattern replacement	32	563	5542
schedule	priority scheduler	9	412	2650
schedule2	priority scheduler	10	307	2710
tcas	altitude separation	41	173	1608
tot_info	information measure	23	406	1052

Table 6: Siemens suite programs

6.1 Total localization effort

In this section, we quantitatively compare the effort required for localizing faults using DeLLIS and other methods for which effort measures are available regarding the Siemens suite. These methods are Tarantula [16], Intersection Model (Inter Model), Union Model, Nearest Neighbor (NN) [24], Delta Debugging (DD) [7] and χDebug [30]. There is a total of 132 mutants of 7 programs (Table 6), each containing a single fault on a single line. Let 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 or fail with a segmentation fault. They are not considered by other methods, thus we do not consider them. 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.

6.1.1 Experimental Settings

We evaluate two strategies. The first strategy consists in starting from the bottom and traversing the lattice to go straightforwardly to the right fault concept. This corresponds to the best case of our approach. This strategy assumes a super competent debugging oracle, who knows at each step the best way to go and 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 her. Using a "Monte Carlo" approach and thanks to the law of large numbers, we compute an average estimation of the cost of this strategy.

Definition 9 (Jones *et al.* metric [15])
$Expense(F_m) = \frac{\|fault\_context(F_m)\|}{size\ of\ program} * 100$
where $fault\_context(F_m)$ *is the set of lines explored before finding* F_m.

The *Expense* metric measures the percentage of lines that are explored to find the fault.

For both strategies, the best strategy and the random strategy, *Expense* is thus as follows:
$Expense_B(F_m) = \frac{\|fault\_context_{Best}(F_m)\|}{size\ of\ program} * 100.$
$Expense_R(N, F_m) = \frac{1}{N} * \sum_{i=1}^{N} \frac{\|fault\_context_i(F_m)\|*100}{size\ of\ program}.$
$Expense_R$ 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, so we have chosen 20. Number N is chosen so that the confidence on $Expense_R$ is about 1%.

For any method M, $Expense_M$ allows to compute $Freq_M(cost)$ which measures how many failures are explained by M for a given cost:

$$Freq_M(cost) = \frac{\|\{m|Expense_M(F_m) \leq cost\}\|}{total\ number\ of\ mutants} * 100.$$

6.1.2 Results

$Freq_M(cost)$ can be plotted on a graph, so that the area under the curve indicates the global efficiency of method M. Figure 9 shows the curves for all the methods [9]. The DeLLIS strategies are represented by the two thick lines. For DeLLIS Best Strategy 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 methods, Tarantula and χDebug, 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.

Note finally that this comparison ignores the multiple fault capacity of DeLLIS.

6.2 The impact of relevance indicators

In this section, we study the impact of the choice of minimal values for the lift indicator on a program of several thousands lines, the Space program. In

[9] The detailed results of the experiments can be found on: http://www.irisa.fr/LIS/cellier/publis/these.pdf

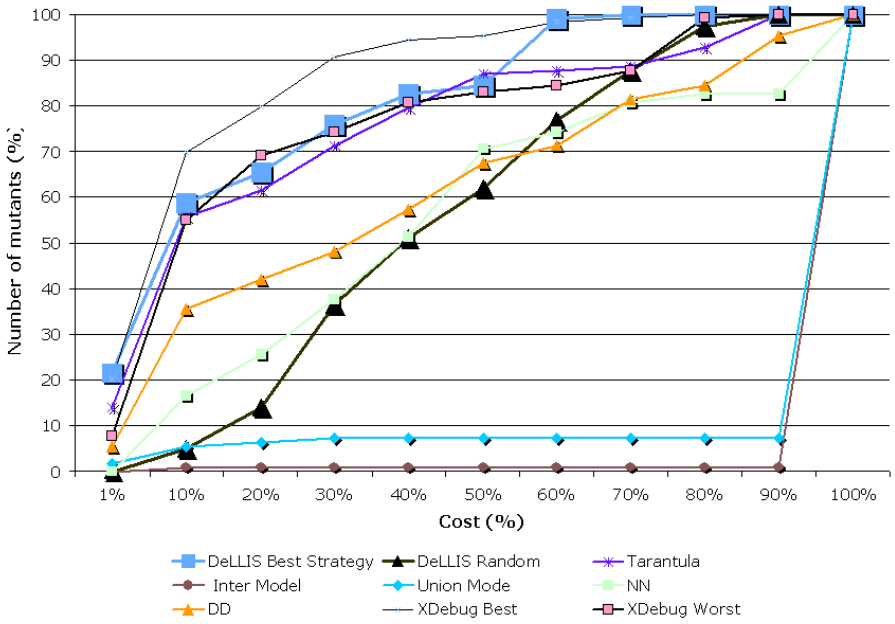

Figure 9: Frequence values of the methods

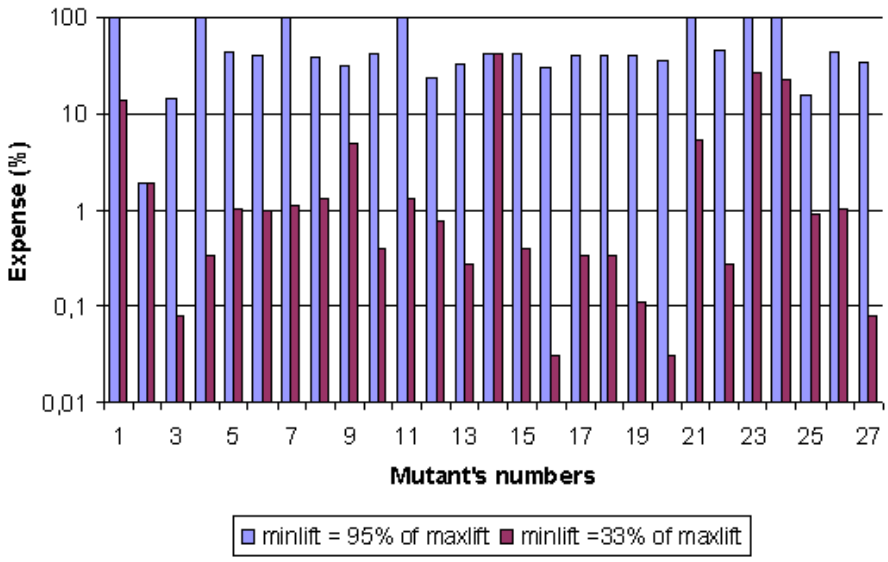

Figure 10: Expense values

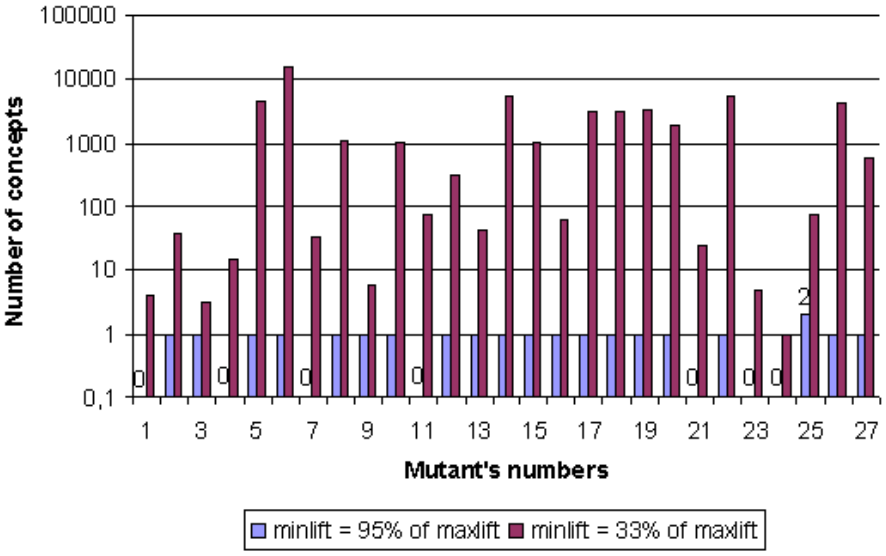

Figure 11: Size of failure lattice

particular, we present how the Expense value and the number of concepts of the best strategy vary with respect to the min_lift value.

6.2.1 Experimental Settings

Space has 38 associated mutants, of which 27 are usable, 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.

The support threshold is set to the maximum value of the support. The mutants contain a single 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 (8 values from 1 to $(maxlift - 1) * 0.95 + 1$). We discovered two representative threshold values among the studied ones. The first lift threshold is a value close to the max value: $(maxlift - 1) * 0.95 + 1$. The second lift threshold is $(maxlift - 1)/3 + 1$.

6.2.2 Results

Figure 10 shows the Expense values for each mutant when min_lift is set to 95% of max_lift (light blue) and 33% of max_lift (dark red). The Expense value is presented in a logarithmic scale. The expenses are much higher with the larger min_lift. When $min\_lift = 95\%\ of\ max\_lift$, some mutants, for example mutant 1, have an expense value equal to 100%, representing 3638 lines, namely the whole program. When $min\_lift = 33\%\ of\ max\_lift$, 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 min_lift confim that the lower min_lift, the lower the expense value is, and the fewer lines have to be examined by a competent debugger.

When min_lift = 33% of max_lift, DeLLIS, like Tarantula [16], is much better at detecting the fault than on the much smaller programs of the Siemens suite. For 51% of the versions, less than 1% of the code needs to be explored to find the fault. For 85% of the versions, less than 10% of the code needs to be explored to find the fault.

Figure 11 sheds some light on the results of Figure 10 and also explains why it is not always possible to start with a small min_lift. The figure presents the size of the failure lattice (the number of concepts) for each mutant when min_lift is set to 95% of max_lift and 33% of max_lift. The number of concepts is also presented in a logarithmic scale. For min_lift = 95% max_lift, 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 min_lift, the size of the lattice increases and the labelling of the concepts decreases. Thus, fewer lines have to be examined at each step when traversing the failure lattice, hence the better results for the Expense values with a low min_lift.

However, for min_lift = 33% of max_lift, 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 min_lift increases, the size and cost of computing the failure lattices increase. 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 min_lift. If the lattice exceeds a given number of concepts, the computation is aborted and restarted with a higher value of min_lift following a divide and conquer approach.

7 Discussion and Future Works

7.1 Discussion

The contexts and lattices introduced in the previous sections allow programmers to see all the differences between execution traces as well as all the differences between association rules. There exist other methods which compute differences between execution traces. We first show that the information about trace differences provided by the failure context (and the corresponding lattice) is already more relevant than the information provided by four other methods proposed

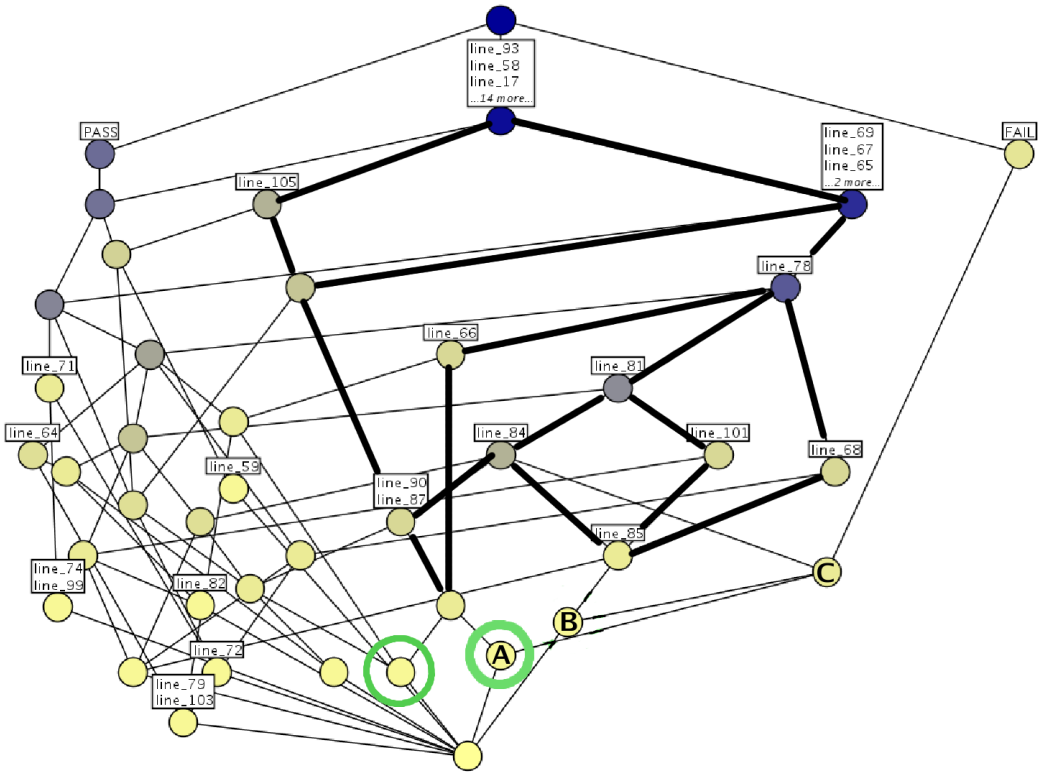

Figure 12: Lattice from the trace context of mutant 1 of the Trityp program

by Renieris and Reiss [24], and Cleve and Zeller [7]. Then we show that explicitly using association rules with several lines in the premise alleviate some limitations of Jones *et al.*'s method [17]. Finally we show that reasoning on the partial ordering given by the proposed failure lattice is more relevant than reasoning on total order rankings [17, 18, 8, 19, 31].

7.1.1 The structure of the execution traces

The trace context contains the whole information about execution traces. In particular, the associated lattice, the *trace lattice*, allows programmers to see in one pass all differences between traces. Figure 12 shows the trace lattice of mutant 1 (compare with the corresponding failure lattice in Figure 3).

There exist several fault localization methods based on the differences between execution traces. They all assume a single failed execution and several passed executions. We rephrase them in terms of search in a lattice to highlight their advantages, their hidden hypothesis and limitations.

Union model The *union model*, proposed by Renieris and Reiss [24], aims at finding features that are specific to the failed execution. The method is based on trace differences between the failed execution f and a set of passed executions S: $f - \bigcup_{s \in S} s$. The underlying intuition is that the failure is caused by lines that are executed only in the failed execution. Formalized in FCA terms, the concepts of interest are the subconcepts whose label contains *FAIL*, and the computed information is the lines contained in the labels of those subconcepts. For example, in Figure 12 this corresponds to concepts A, B, and C. They contain no line in their label, which means that the information provided by the union model is empty. If only one failed execution is taken into account as in the union model method, the concept of interest is the concept whose label contains *FAIL*, and the computed information is the lines contained in the label. The trace lattice presented in the figure is slightly different from the lattice that would be computed for the union model, because it represents more than one failed execution. Nevertheless, the union model often computes an empty information, namely each time the faulty line belongs to failed and passed execution traces. For example, a fault in a condition has a very slight chance to be localized. Our approach is based on the same intuition. However, the lattices that we propose do not lose information and help navigate in order to localize the faults, even when the faulty line belongs to both failed and passed execution traces.

The union model helps localize a bug when executing the faulty statement always implies an error, for example the bad assignment of a variable that is the result of the program. In that case, our lattice does also help, the faulty statement labels the same concept as *FAIL*.

Intersection model The *intersection model* [24] is the complementary of the previous model. It computes the features whose absence is discriminant of the

failed execution: $\bigcap_{s \in S} s - f$. Replacing *FAIL* by *PASS* in the above discussion is relevant to discuss the intersection model and leads to the same conclusions.

Nearest neighbor The *nearest neighbor* approach [24] computes a *distance* metrics between the failed execution trace and a set of passed execution traces. The computed trace difference involves the failed execution trace, f, and only one passed execution trace, the nearest one, p: $f - p$. That difference is meant to be the part of the code to explore. The approach can be formalized in FCA. Given a concept C_f whose intent contains *FAIL*, the nearest neighbor method search for a concept C_p whose intent contains *PASS*, such that the intent of C_p shares as many lines as possible with the intent of C_f. On Figure 12 for example, the two circled concepts are "near", they share all their line attributes except the attributes *FAIL* and *PASS*, therefore $f = p$ and $f - p = \emptyset$. The rightmost concept fails whereas the leftmost one passes. As for the previous methods, it is a good approach when the execution of the faulty statement always involves an error. But as we see on the example, when the faulty statement can lead to both a passed and a failed execution, the nearest neighbor method is not sufficient. In addition, we remark that there are possibly many concepts of interest, namely all the nearest neighbors of the concept which is labelled by *FAIL*. With a lattice that kind of behavior can be observed directly.

Note that in the trace lattice, the executions that execute the same lines are clustered in the label of a single concept. Executions that are near share a large part of their executed lines and label concepts that are neighbors in the lattice. There is therefore no reason to restrict the comparison to a single pass execution. Furthermore, all the nearest neighbors are naturally in the lattice.

Delta debugging *Delta debugging*, proposed by Zeller *et al.* [7], reasons on the values of variables during executions rather than on executed lines. The trace spectrum, and therefore the trace context, contains different types of attributes. Note that our approach does not depend on the type of attributes and would apply on spectra containing other attributes than executed lines.

Delta debugging computes in a memory graph the differences between the failed execution trace and a single passed execution trace. By injecting the values of variables of the failed execution into variables of the passed execution, the method tries to determine a small set of suspicious variables. One of the purpose of that method is to find a passed execution relatively similar to the failed execution. It has the same drawbacks as the nearest neighbor method.

7.1.2 From the trace context to the failure context

Tarantula Jones *et al.* [17] compute association rules with only one line in the premises. Denmat *et al.* [10] have shown that the limitations of this method, in particular due to three implicit hypothesis. The first hypothesis is that a failure has a single faulty statement origin. The second hypothesis is that lines are independent. The third hypothesis is that executing the faulty statement often causes a failure. That last hypothesis is a common assumption of fault

localization methods, including our method. Indeed, when the fault is executed in both passed and failed executions (e.g. in a prelude) it cannot be found so easily using these hypothesis. In addition, Denmat *et al.* demonstrate that the *ad hoc* indicator which is used by Jones *et al.* is equivalent to the lift indicator.

By using association rules with more expressive premises than in Jones *et al.* method (namely with several lines), the limitations mentioned above are alleviated. Firstly, the fault need not be a single line, but can be contain several lines together. Secondly, the dependency between lines is taken into account. Indeed, dependent lines are clustered or ordered together.

The part of the trace context which is important to search in order to localize a fault is the set of concepts that are related to the concept labelled by *FAIL*; i.e. those that have a non-empty intersection with the concept labelled by *FAIL*. Computing association rules with *FAIL* as a conclusion computes exactly those concepts, modulo the min_sup and min_lift filtering. In other words, the focus is done on the part of the lattice related to the concept labelled by *FAIL*. For example, in the trace lattice of the Trityp program presented in Figure 12, the failure lattice when min_lift is very low (yet still attractive, i.e. $min\_lift > 1$), is drawn in bold lines.

7.1.3 The structure of association rules

Jones *et al.*'s method presents the result of the analysis to the user as a coloring of the source code. A red-green gradient indicates the correlation with failure. Lines that are highly correlated with failure are colored in red, whereas lines that are highly not correlated are colored in green. Red lines typically represents more than 10% of the lines of the program, whithout identified links between them. Some other statistical methods [18, 8, 19, 31] also try to rank lines in a total ordering. It can be seen as ordering the concepts of the failure lattice by the lift value of the rule in their label. However, we have shown in Section 3 that the monotonicity of lift is only relevant locally to a support cluster.

For example, on the failure lattice of Figure 3, the obtained ranking would be: line 85, line 66, line 68, line 84, ... No link would be established between the execution of line 85 and line 68 for example.

The user who has to localize a fault in a program has a background knowledge about the program, and can use it to explore the failure lattice. Reading the lattice gives a context of the fault and not just a sequence of independent lines to be examined, and reduces the number of lines to be examined at each step (concept) by structuring them.

7.1.4 Multiple Faults

We have compared the failure lattice with existing single fault localization methods. In this section, we compare our navigation into the failure lattice with the strategies of the other methods to detect several faults.

Our approach has a flavour of algorithmic debugging [26]. The difference lays in the traversed data structure. Whereas Shapiro's algorithm helps traverse a

proof tree, our algorithm helps traverse the failure lattice, starting from the most suspicious places.

For multiple faults, Jiang *et al.* [14] criticize the ranking of statistical methods. They propose a method based on traces whose events are predicates. The predicates are clustered, and the path in the control flow graph associated to each cluster is computed. In the failure lattice, events are also clustered in concepts. The relations between concepts give information about the path in the control flow graph and highlight some parts of that path as relevant to debug without computing the control flow graph.

Zheng *et al.* [31] propose a method based on bi-clustering in order to group failed executions and to identify one feature (*bug predictor*) that characterizes each cluster. They propose to look at one bug predictor at a time. Several bug predictors can be in relation with the same fault but no link is drawn between them. Our approach gives a context to the fault, in order to help understand the mistakes of the programmer which have produced the fault.

Jones *et al.* [15] propose a method which first clusters executions and then finds a fault in each cluster in parallel. That method has the same aim as our method. Indeed, in both cases we want to separate the effects of the differents faults in order to treat the maximum of faults in one execution of the test suite, but in our approach, the clusters are partially ordered to take into account dependencies between faults.

Finally, SBI [18] introduces a stop criterion as we did in our algorithm. SBI tries to take advantage of one execution of the test suite. The events are predicates. SBI ranks those predicates. When a fault is found thanks to the ranking, all execution traces that contain the predicates used to find the fault are deleted and a new ranking on predicates with the reduced set of execution traces is computed. Deleting execution traces can be seen as equivalent to tagging concepts, and thus the events of their labelling, as explained in DeLLIS. The difference between SBI and DeLLIS is that DeLLIS does not need to compute the failure lattice several times.

7.2 Future works

We have presented a deliberately simplistic approach to using formal concept analysis and association rules for fault localization. In the current approach, a trace is a set of line numbers, and the trace issue is *PASS* or *FAIL*. However, the proposed approach lends itself easily to refinements like taking into account the structure of the tested software, the scheduling or the semantic of the events, or a classification of failures.

7.2.1 Using taxonomies to reflect structure

Formal concept analysis has been extended since its beginning to cope with structured contexts. The first approach has been to encode structures, called *scales*, into formal contexts in order to reflect such structures as hierarchy, dichotomy, etc. [12]. A more recent approach, called Logical Formal Analysis,

as shown how to use logical formulas instead of attributes in contexts, and use logical implication between sets of attributes, instead of set inclusion, as the ordering that defines formal concept [11]. In both cases, the extension is conservative and leads to the construction of a regular concept lattice.

As a consequence, our fault localization approach can directly benefit from these refinement of formal concept analysis.

The first benefit could be to use evidences of the software structure, eg. file, package, class, method, function, loop, block, to give a structure to line numbers. This could be used for refining the zoom effect of our global debugging process, and also to factorize the presentation of concepts. Note that these structure evidences obey to a reach logical structure: eg. a line of a block of a loop of a function of a method of a class of a package of a file...

A second benefit could be to recognize *basic blocks* syntactically when possible, eg. for goto-less languages, instead of recognizing them empirically as lines that always come together.

In both cases, this refinement will make the dialog between the debugging person and the fault localization tool more effective because it will use formal notions that are closer to the richness of the developer experience.

7.2.2 Using n-grams to reflect scheduling

Our primitive approach consider traces as *sets* of execution events, ie. an unordered collection. However, this completely ignores that execution events are ordered in time by the execution scheduler. In principle, it is possible to represent an order, even a total order, in a formal context, but at a considerable cost.

We propose to use a cheap but incomplete rendition of execution scheduling by *n-grams* of trace events. An n-gram is simply a n-tuple of trace events. In our case, it could have the semantics that in an n-gram the j-st component immediately precedes the $(j+1)$-st component in the execution scheduling.

It is not necessary to consider n-grams with a large n. For instance, 2-grams essentially reconstruct fragments of the *control flow graph* of the program [29]. The reconstructed fragments depend on the test coverage. This could be used for further analyses based on data flow [29]. For instance, a *liveness analysis* could discover that an assignment is *dead*, ie. the variable is not read after being assigned. The reason for this could be a program fault, but also a lack of test coverage, however in both cases it can guide the debugging person into further investigations.

7.2.3 Using valued attributes to reflect semantics

When explaining a fault, one does not say "It is a line 1729 fault", but rather one says "It is a badly initialized variable `v` in method `m`". We propose to use valued attributes to express the semantics of the trace events. For instance, *Def* and *Use* are two classical roles that are used in semantics analysis [29]. *Def* of an instruction is the set of memory locations (usually one variable) that are

assigned a content by the instruction. *Use* of an instruction is the set of memory locations that are read. This reflects the *dataflow* of a program. This can be represented as def and use attributes whose values are memory locations.

Note that dataflow analysis with pointers is very difficult especially because one cannot generally know the location that is actually written in `*x = y`, or the location that is read in `x = *y`. However, the *Def* and *Use* locations are all known at run-time. So, it is easier to analyze them from traces than from sources files. Since traces represent fragments of all possible executions one must be cautious when generalizing trace analyses, but for every universal property like "$\forall$ execution, ... is true", ie. an invariant, finding a counter-example in a trace suffices to prove the condition false. And it is all testing is about: finding plausible truth from partial evidences.

7.2.4 Using valued attributes to refined failure conditions

It is often the case that a failure is detected by different means:

- Confrontation with a test oracle for *functional failure*. In this case, the program executes normally but produces an unexpected result.
- Detection of a *failure condition* by the execution environment. In this case, the execution is halted before the normal termination of the program.

Failure conditions depend on what the execution environment is equipped to detect. This can go from an invalid memory address and out-of-bound index, to invalid database requests and invalid URLs.

These refined failure conditions are a key input for the debugging person, and they must not be blurred in a single *FAIL* verdict. So, a normal refinement of our approach is to actually represent different failure conditions as a fail attribute with a value indicating the actual condition.

There is more than a gain in semantic precision in this variant. Indeed, this also changes the lift value of a $P \longrightarrow \mathsf{fail}(cond)$ association rule with respect to $P \longrightarrow FAIL$. It is possible that the lift of the first rule shows an attraction whereas the second shows a repulsion. This is because $\|extent(FAIL)\|$ comes to the denominator of the lift formula; therefore, when it is replaced by $\|extent(\mathsf{fail}(cond))\|$, which is smaller, the lift increases. So, this variant yields both an increase in precision in the association rules, and an increase in precision in the evaluation of the rules because failure rules become better focused.

8 Conclusion

We have proposed an approach for software fault localization that uses formal concept analysis and association rules as a means for giving a structure to a set of trace events. The proposed approach articulates two levels of analysis. At a first level, a set of trace events produced by the execution of test cases is mined to evaluate their correlation with *PASS* and *FAIL* test outputs. This yields a set of association rules that is much too large for practical purposes, and a second

level of analysis is used for exploring the set. Both levels can be fine-tuned in terms of precision and sensibility, permitting a progressive approach in which the time-cost compromise can be adjusted.

This leads to a Global Debugging Process which gives a rational to managing the test and analysis effort. We do not pretend this is the definitive debugging process, but we advocate that Fault Localization, as well as Testing and Debugging, should be formalized as a process.

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining associations between sets of items in massive databases. In *Proceedings of the International Conference on Management of Data*, pages 207–216. ACM, 1993.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994.

[3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In Joan Peckham, editor, *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 255–264. ACM Press, 1997.

[5] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *Int. Conf. on Software Engineering & Knowledge Engineering*, pages 238–243. Knowledge Systems Institute Graduate School, 2011.

[6] Peggy Cellier, Sébastien Ferré, Olivier Ridoux, and Mireille Ducassé. A parameterized algorithm to explore formal contexts with a taxonomy. *International Journal of Foundations of Computer Science*, 19(2), 2008.

[7] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering*. ACM Press, 2005.

[8] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 3586, pages 528–550. Springer Berlin / Heidelberg, 2005.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[10] Tristan Denmat, Mireille Ducassé, and Olivier Ridoux. Data mining and cross-checking of execution traces: a re-interpretation of Jones, Harrold and Stasko test information. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2005.

[11] Sébastien Ferré and Olivier Ridoux. An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419, Elsevier, 2004.

[12] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[13] IEEE. *Standard Glossary of Software Engineering Terminology*, 1990.

[14] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the International Conference on Automated Software Engineering*, pages 184–193. ACM Press, 2007.

[15] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 16–26, July 2007.

[16] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282. ACM, 2005.

[17] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477. ACM Press, 2002.

[18] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the International Conference on Programming Language Design and Implementation*. ACM Press, 2005.

[19] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transaction Software Engineering.*, 32(10):831–848, 2006.

[20] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[21] Yoann Padioleau and Olivier Ridoux. A logic file system. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 2003.

[22] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, pages 398–416. Springer-Verlag, 1999.

[23] Mattieu Petit and Arnaud Gotlieb. Uniform selection of feasible paths as a stochastic constraint problem. In *Proceedings of the International Conference on Quality Software*, IEEE, October 2007.

[24] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2003.

[25] Kenneth A. Ross and Charles R. B. Wright. *Discrete mathematics (3. ed.)*. Prentice Hall, 1992.

[26] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983. ISBN 0-262-19218-7.

[27] L. Szathmary and A. Napoli. CORON: A Framework for Levelwise Itemset Mining Algorithms. In B. Ganter, R. Godin, and E. Mephu Nguifo, editors, *Proceedings of the International Conference on Formal Concept Analysis*, pages 110–113, 2005. (demo paper).

[28] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. A survey of formal concept analysis support for software engineering activities. In *Proceedings of the International Conference on Formal Concept Analysis*, volume 3626 of *LNCS*. Springer Berlin / Heidelberg, 2005.

[29] Reinhard Wilhelm and Dieter Maurer. *Compiler design*. Addison-Wesley, 1995.

[30] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. *Int. Computer Software and Applications Conf.*, 1:449–456, 2007.

[31] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the Twenty-Third International Conference on Machine learning*, pages 1105–1112. ACM, 2006.