Formal Concept Analysis applied to Fault Localization

Peggy Cellier
IRISA, Campus Bealieu,
35042 Rennes cedex, France
peggy.cellier@irisa.fr
http://www.irisa.fr/LIS/cellier/

ABSTRACT

One time-consuming task in the development of software is debugging. Recent work in fault localization crosschecks traces of correct and failing execution traces, it implicitly searches for *association rules* which indicate that executing a line will most probably cause the whole execution to fail. This technique has some limitations: it assumes that an error has a single faulty statement origin, and that lines are independent. Our research hypothesis is that using association rules with more expressive premises, some limitations can be alleviated. The solution that we propose combines *association rules* and *formal concept analysis*. Our technique is already usable when the size of the execution traces is not too large. We conjecture that the technique can be used to analyze large executions, thanks to the information contained in the Abstract Syntax Tree.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Measurement, Reliability, Experimentation, Theory

Keywords

Formal Concept Analysis, Association Rules, Data Mining, Fault Localization, Debugging

1. INTRODUCTION

When a program fails, i.e. when it does not produce the expected results, a debugging process begins. It corresponds to the detection and the correction of the responsible faults. Debugging is a time-consuming task in the development of software. Our research focuses on the first part of the debugging process, i.e. fault localization. In order to locate faults in programs, several approaches to crosscheck traces exist. Some are based on the differences between a passed execution and a failed execution [11, 4]. Others use statistical indicators in order to rank lines of the program [8, 9, 10]. As it has been demonstrated, recent work in fault localization that crosschecks traces of correct and failing execution traces [8], implicitly searches for *association rules* [1] which indicate that executing a particular source line will most probably cause the whole execution to fail [5]. Altougth the first experiments give good results, Denmat *et al.* showed that this technique has some limitations. For example, it assumes that an error has a single faulty statement origin, and lines are independent.

Our work is based on the same intuition than other methods, i.e. a line in traces of failed executions has more chances to be faulty than a line in traces of passed executions. Our research hypothesis is that using association rules with more expressive premises, the limitations highlighted by Denmat *et al.* can be alleviated. The solution that we propose combine *association rules* [1] and *Formal Concept Analysis* (FCA) [7] in order to improve the fault localization. Formal Concept Analysis (FCA) has already been used for several software engineering tasks [12]. FCA finds interesting clusters, called *concepts*, in data sets. The input of FCA is the same as association rules, a *formal context*, i.e. a binary relation describing elements of a set of *objects* by subsets of *attributes*.

In the sequel, Section 2 introduces the methods that are used to carry out our research. In addition, our first results are presented [3, 2]. Section 3 discusses the differences between our approach and existing methods. Section 4 proposes further work with the Abstract Syntaxic Tree (AST) to improve fault localization, it gives a evaluation plan.

2. FAULT LOCALIZATION WITH DATA MINING

In this section, our method is briefly described, for more details see [2]. We use the Trityp program given in Figure 1 to illustrate our approach. Program Trityp classifies three segment lengths into four categories: *scalene*, *isosceles*, *equilateral*, *not a triangle*. One fault has been introduced at Line 84[1]. The condition `(trityp == 2)` is replaced by `(trityp == 3)`. That fault implies a failure in two cases. The first case is when `trityp` is equal to 2. That case is not taken into account as a particular case and thus it is treated as a default case, at Lines 89 and 90. The second case is

[1] http://www.irisa.fr/lande/gotlieb/resources/Javaexp/trityp/

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

```
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
// FAULTY LINE
[84]            if ((trityp == 3)
                     && (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.

	66	68	81	84	85	87	90	$\cdots$	P	F
e_1	×	×							×	
e_4	×		×	×		×	×		×	
e_{108}		×	×	×	×					×
$\cdots$										
e_{400}									×	

Table 1: ***Example of the trace context for the faulty version of program Trityp.***

when `trityp` is equal to 3. That case should lead to the test Line 87, but due to the fault it is first tested at line 84. Indeed, if the condition `(i+k>j)` holds, `trityp` is assigned to 2. However, `(i+k>j)` does not always entail `(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 Trace Context and Association Rules.

In order to reason about program executions we use traces of these executions. Let us assume that each trace contains at least the executed lines and the verdict of the execution, $PASS$ if the execution produces the expected results and $FAIL$ otherwise. This forms the *trace context*. The *objects* of the trace context are the execution traces. The *attributes* are all the lines of the program and the two verdicts. Each trace is described by the executed lines and the verdict of the execution. Table 1 gives a part of the resulting trace context for the program of Figure 1. For instance, during the first execution (e_1), the program executes lines 66, 68, ... and passes[2].

In order to understand the causes of the failed executions, we use our data mining algorithm [3] which searches for association rules. We search for particular association rules where the premise are sets of lines which are mostly contained in failed execution traces and few passed execution traces (according to the statistical indicators) and the conclusion is the attribute $FAIL$, i.e. rules that have the form: $line\_i, line\_j, \cdots \;\rightarrow FAIL$. The support is the first filtering statistical indicator for the extraction of association rules. It measures the frequency of a rule. For the fault localization problem, the value of the threshold of the support, *minsup*, indicates the minimum number of failed executions that should be covered by a rule to be selected. Choosing a very high threshold, only the most frequent execution paths are represented in the set of association rules. Choosing a very low threshold, *minsup* equals to one object, all execution paths that are stressed by the test cases are represented in the set of association rules. The other well-known indicator that we use is lift. In our approach, the lift indicates how the execution of a set of lines improves the probability to have a failed execution.

	81	84	87	90	105	66	78	112	$\cdots$	93
r_1	×	×	×	×	×	×	×	×		×
r_2	×	×	×	×			×	×		×
$\cdots$										
r_9							×	×		×

Table 2: ***Example of the rule context for the faulty program Trityp with $minlift = 1.25$ and $minsup = 1$.***

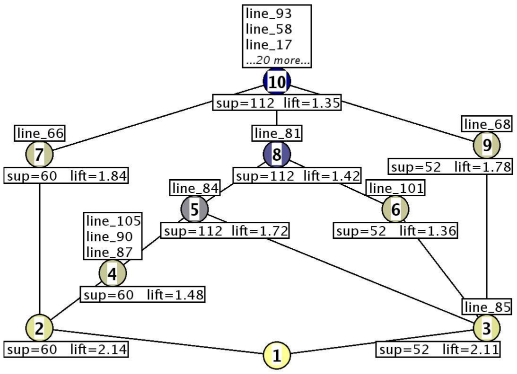

Figure 2: Rule lattice associated to the rule context of Table 2.

The Rule Context and The Rule Lattice.

The computation of association rules generates a lot of rules with large premises. Understanding the links that exist between the rules, for example if a rule is more specific than another, is difficult to do by hand. The computed association rules, however, are partially ordered according to their premises; indeed $L_1 \;\rightarrow\; FAIL$ is more specific than $L_2 \;\rightarrow\; FAIL$ when L_1 and L_2 are sets of lines such that $L_2 \subset L_1$. Therefore, in order to help analyze the rules, we propose to build a new context, the *rule context*. The objects are the association rules; the attributes are lines. Each association rules is described by the lines of its premise. Table 2 shows a part of the rule context for the faulty version of the Trityp program with the support threshold, *minsup*, equal to 1 object and the lift threshold $minlift$ equal to 1.25. The premise of $rule_1$ contains line 81, line 84, line 87, line 90, ...[3] .

The *rule lattice* is the *concept lattice* associated with the rule context. In FCA a *concept* is defined by a pair *(extent, intent)*, where *extent* is the maximal set of objects that have in their description all attributes of *intent*, and *intent* is the

[2] Complete context:
http://www.irisa.fr/LIS/cellier/icfca08/trace_context.txt

[3] Complete rule context:
http://www.irisa.fr/LIS/cellier/icfca08/rules_context.rl

maximal set of attributes common to the description of all objects of *extent*. The concepts of the context can be represented by a *concept lattice* where each concept is labelled by its intent and extent. The lattice allows association rules to be structured in a way that highlights the partial ordering which exists between them. Figure 2 displays the rule lattice associated with the rule context of Table 2[4]. The rule lattice is presented with a *reduced labelling*. In that representation, a node is a concept and, each attribute and each object is written only once. Namely, each concept is labelled by the attributes and the objects that are specific to it. As a consequence, the premise of a rule r can be computed by collecting the attributes labelling all the concepts above the concept that is labelled by r. For example, on Figure 2 the premise of the rule which labels Concept 3 is line 85, line 84, line 68, line 101, line 81, line 93, line 58, line 17, plus the other 20 attributes (lines) that label the top concept (Concept 10).

Interpretation of the Rule Lattice.

Navigating in the rule lattice bottom up first displays rules that are in general too specific to explain the error. It then displays rules that are more general and maybe more informative, and finally displays the top of the lattice which is labelled by the attributes (line numbers) that are common to all failed executions.

The bottom concept of the rule lattice in Figure 2 has no attribute in its labelling. During the debugging session two paths are proposed to follow. The leftmost path from the bottom concept, Concept 2, corresponds to the case where variable `trityp` is equal to 3 and condition `(i+k>j)` holds whereas the condition `(j+k>i)` does not hold. It leads to two concepts. The first concept is Concept 7 labelled by line 66, it is the statement which initializes `trityp` to 2. The second concept is Concept 4 labelled by three line numbers: 105, 90, 87. These lines correspond to the case when the variable `trityp` is equal to 2 and `trityp` is assigned to 4 when 2 is expected, i.e. the triangle is labelled as not a triangle instead of isosceles. Those two concepts are too specific but by looking at the rule of the concept upwards (Concept 5), the faulty line is localized. Concept 5 covers the greatest number of failed executions (support=112) and has the greatest lift among rules which have support equal to 112. The same reasoning can be done with the rightmost concept, Concept 3. It also leads to line 85. It corresponds to the *then branch* of the faulty conditional, i.e. the line where variable `trityp` is assigned to 2 when 4 is expected. This example shows that the rule lattice gives relevant clues for exploring the program. The faulty line is not immediately highlighted but exploring the lattice bottom up guides the user in its task to understand the fault.

In addition, the concepts of the rule lattice have two properties thanks to the statistical indicators related to the trace lattice. The first property states that the support of rules that label the concepts of the rule lattice decreases when exploring the lattice top down. The second property is about the lift value. If two ordered concepts in the rule lattice are labelled by rules with the same support value, the lift value of the rule which labels the more specific concept is greater. That is why the rule lattice is explored bottom up.

3. DISCUSSION ABOUT PRIOR WORK

This section gives some elements of comparison with related work (see [2] for more details).

There exists several fault localization methods based on the differences of execution traces: the *union model*, the *intersection model*, the *nearest neighbor* [11], *Delta debugging* [4]. They all assume a single failed execution and several passed executions. The first context that we have introduced, the trace context, contains the whole information about execution traces (see Section 2). In particular, the associated lattice, the trace lattice, allows programmers to see in one pass all the differences between traces of the above mentioned approaches.

The *union model* is based on trace differences between the failed execution f and a set of passed executions S: $f - \bigcup_{s \in S} s$. The *intersection model* is the complementary. It computes the features whose absence is discriminant of the failed execution: $\bigcap_{s \in S} s - f$. The information computed by these two methods can be found in the trace lattice. In addition, these methods often compute 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 little chance to be localized.

The *nearest neighbor* approach and the *Delta debugging* approach are based on the difference between the failed execution trace, f, and only one passed execution trace, the nearest one, p: $f - p$. The traces of the first method contain the executed lines, whereas the second method reasons on the values of variables of the program during executions. One of the purposes of those methods is to find a passed execution relatively similar to the failed execution. 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. All the nearest neighbors are naturally in the trace lattice. In addition, our method is not restricted to traces that contain line numbers. Indeed, it can be applied when the traces contain different kind of information as the variable values.

Jones *et al.* [8] compute association rules with only one line in the premise. As already mentioned, Denmat *et al.* showed that the method has limitations, because it implicitly assumes that an error has a single faulty statement origin, and that lines are independent. By using association rules with more expressive premises, namely with several lines, the limitations mentioned above are alleviated. Firstly, the faults that can be located are not restricted to faults which have a single faulty statement origin. Secondly, the dependency between lines is taken into account. Indeed, dependent lines are clustered or ordered together.

Jones *et al.*'s present the result of their 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 [9, 10] also try to rank lines in a total ordering. With our approach, reading the lattice gives a context of the fault and not just a sequence of independent lines to be examined. In addition, the lattice allows the number of lines to be examined at each step (concept) to be reduced by structuring the lines. The user who wants to localize a

[4] The lattice was generated with the ToscanaJ tool (http://toscanaj.sourceforge.net/)

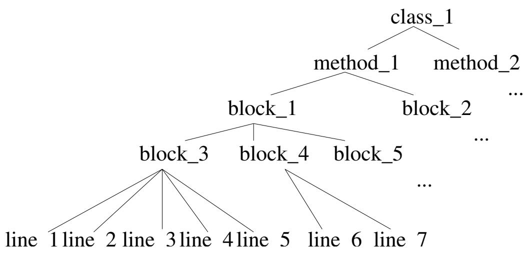

Figure 3: Example of a part of an AST.

fault in a program has a background knowledge about the program, and can use it to explore the rule lattice.

4. FUTURE WORK AND EVALUATION PLAN

The method that is proposed for fault localization is already usable at the end of the debugging process. When the programmer has a rough idea of the location of the faults and that only a small part of the execution has to be traced, the current methods to visualize lattices can be directly used.

We conjecture that the technique, with some extensions, can be used to analyze large executions. At present, the information is described in terms of lines whereas it is not always the most relevant information granularity. For example, given a basic block all its lines always appear in the same label. Displaying the location of the basic block would be more relevant. This will help keeping concept labels to a readable size. We are currently working on the presentation of the results to reduce their size and to give more semantics to them so that they are more tractable by users. We are using the information that is contained in the Abstract Syntax Tree (AST) to reduce the size of the intent of the concept. For instance, Figure 3 shows a part of the AST of a program. In that AST, the granularity is: class, method, block and line. Let us assume that the computation of association rules generates a rule like: line_1, line_2, line_3, line_4, line_5 $\rightarrow FAIL$, thanks to the AST the rule can be simplified in: block$_C$_3 $\rightarrow FAIL$. It means that the complete block, i.e. all lines of the block, is involved in the failure of executions.

Another advantage of using the AST appears during the computation of association rules. Indeed, the AST can be seen as a *taxonomy*, i.e. a hierarchy of the elements of the program. The algorithm which is used to compute association rules [3], allows taxonomies to be taken into account. Introducing the AST as a taxonomy permits two improvements. Firstly, the computation time can be reduced. For instance, if the attribute method_1 is not sufficiently frequent, none of the attributes under method_1 (e.g. block_1, block_3, line_1, line_2) can be sufficiently frequent (see [3]) and thus that part of the taxonomy can be ignored. Secondly, thanks to the taxonomy, new rules can be generated. For example, let us assume that the attribute block_3 is a good candidate to be the premise of a rule, but neither block_4 nor block_5. If no taxonomy is taken into account, only one rule is generated: block_3 $\rightarrow FAIL$. However, if the AST is taken into account as a taxonomy, and if the rule block$_P$_1 $\rightarrow FAIL$ is relevant with respect to the statistical indicator thresholds, two rules are computed: block_3 $\rightarrow FAIL$ and block$_P$_1 $\rightarrow FAIL$. The notation block$_P$_1 $\rightarrow FAIL$ means that only a part of the block_1 is ivolved in the fact that executions fail. In order to understand the difference between block$_C$_1 $\rightarrow FAIL$ and block$_P$_1 $\rightarrow FAIL$, we can say that block$_C$_1 is a conjunction of sub-elements of block_1 in the taxonomy, i.e. block_3 and block_4 and block_5 can imply that the executions fail. The notation block$_P$_1 is a disjunction of sub-elements of block_1 in the taxonomy, i.e. execute block_3 or block_4 or block_5 can imply that the executions fail. That kind of *disjunctive rules* can be interesting when for example the fault is on a conditionnal statement and not all conditionnal branchs highlight the faulty behaviour. The interpretation is not the same when there is only one rule block_3 $\rightarrow FAIL$ and when there are two rules block_3 $\rightarrow FAIL$ and block_1 $\rightarrow FAIL$. In the first case, the fault is very likely in block_3 whereas in the second case the fault can be in block_1.

We have experimented on toy Java programs. We are currently setting up the environment to test the method on bigger programs, using JTransformer[5]. We envisage to evaluate our method on large programs of the SIR repository [6].

5. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Int. Conf. on Management of Data*. ACM Press, 1993.

[2] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Formal concept analysis enhances fault localization in software. In *Int. Conf. on Formal Concept Analysis*. Springer, 2008.

[3] P. Cellier, S. Ferré, O. Ridoux, and M. Ducassé. A parameterized algorithm to explore formal contexts with a taxonomy. *Int. Journal of Foundations of Computer Science*. To appear.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Int. Conf. on Soft. Eng.* ACM Press, 2005.

[5] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces. In *Int. Conf. on Automated Soft. Eng.* ACM, 2005.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Soft. Eng.: An Int. Journal*, 10(4):405–435, 2005.

[7] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[8] J. A. Jones, M. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *Int. Conf. on Soft. Eng.* ACM, 2002.

[9] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Conf. on Programming Language Design and Implementation*, pages 15–26. ACM Press, 2005.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Eur. Soft. Eng. Conf./FSE*. ACM Press, 2005.

[11] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Int. Conf. on Automated Software Engineering*. IEEE, 2003.

[12] G. Snelting. Concept lattices in software analysis. In *Formal Concept Analysis*. Springer, 2005.

[5] http://roots.iai.uni-bonn.de/research/jtransformer/