I. INTRODUCTION

Executing a test suite can be very costly, especially when the test oracle, that tells whether a run is correct, is not fully automated. In the multiple fault localization context, being able to explain as many as possible of the failures detected by the test suite execution is therefore an important property. However, it is in general impossible to tell how many faults are present in a buggy program. In order to deal with multiple faults, Liblit et al. [11] propose a ranking based on a statistic treatment of inserted predicates. Zheng et al. [16] propose a method based on bi-clustering in order to group failed executions and to identify one feature that characterizes each cluster. Jones et al. [9], with their parallel debugging, propose to cluster failed executions in order to distribute each cluster of executions to different debuggers. Ideally, one cluster should represent one fault, and instead of searching the faults one by one, what the authors call sequential debugging, the faults can be searched in parallel by different persons. In all those methods, there is, however, no guaranty that each cluster represents only one fault. Furthermore, the dimension used to measure suspicion tends to ignore the actual structure of the program. On the other hand, lines with close suspicion scores may be uncorrelated; on the other hand, correlated lines may have very different suspicion scores (e.g. a condition and the branches it controls).

In a previous work [4], we have proposed a new data-structure that can organize program elements in a multi-dimensional space: the failure lattice. The failure lattice is a partial ordering of the elements of the traces that are most likely to lead to failures. The traces can contain different kinds of information called events, for instance, executed lines or variable values. We assume that each execution trace contains, in addition to the events, the verdict of the execution, PASS if the execution produces the expected results, and FAIL otherwise. In a previous work [5], we have compared the exploration of the failure lattice with existing approaches that handle only a single fault. We have shown, in particular, that the method gives a comparable number of lines to analyze while providing a richer environment for the analysis.

In this paper, we propose an interactive fault localization algorithm that explores the failure lattice and can deal with multiple faults. The contribution of this paper is twofold. Firstly, the exploration stops when all the detected failures are explained. That stopping criterion is independent of the number of faults. Secondly, a programmer can exploit all the results of a single execution of a test suite, thus reducing the number of executions of the test suite.

In the remaining of the paper, Section II presents the running example. Section III presents the failure lattice introduced in [4]. Section IV introduces an extended labelling of the failure lattice for the multiple fault problem. Section V defines the proposed algorithm to explore the data structure and locate multiple faults in a program. Section VI and Section VII discuss the characteristics of the algorithm with respect to fault dependencies and statistical indicators. Finally Section VIII presents the related work.

II. RUNNING EXAMPLE

To illustrate our method, we use a classical benchmark for test generation methods, the Trityp Java program shown in Figure 1. It classifies sets of three segment lengths into four categories: not a triangle, scalene, isosceles, equilateral.

The program contains one class with 130 lines of code. The
faulty programs used in this article are a courtesy of Petit and Gotlieb [14] and they can be found on the web. Table I presents six faults for the Trityp program that are used in different blends in the following.

### III. Data Mining and Failure Lattice

In this section, we give background knowledge about two data mining techniques in order to explain how to read a failure lattice. The construction is explained in [4].

Data mining is a process to extract relevant information from a huge amount of data. Two data mining techniques are used to compute the failure lattice: Association Rules and Formal Concept Analysis (FCA). The input of those techniques is a formal context, i.e., a binary relation describing elements of a set of objects by subsets of a set of attributes. Table II is an example of context. The objects are the executions. The attributes are the program lines and the verdicts. Each execution is described by its execution trace.

1) **Association Rules:** Searching for association rules [1] allows interesting regularities to be found. An association rule has the form: \( P \rightarrow C \), where \( P \) and \( C \) are sets of attributes. \( P \) is called the premise of the rule and \( C \) the conclusion. Any pair of premise and conclusion forms an association rule, but some of them are more relevant than others. In order to measure the relevance of computed rules, statistical indicators are used. In the following, we use two indicators, support and lift, that will be defined when they are actually needed.

2) **Formal Concept Analysis (FCA):** Formal Concept Analysis [7] allows relevant clusters to be computed. In FCA, the set of all objects that share a set of attributes is called the extent of the set of attributes. For example, in the context of Table II, the extent of \{line 57, line 58\} is the set of executions that execute line 57 and line 58. The extent of \{FAIL\} denotes all failed executions. The set of all attributes shared by all elements of a set of objects is called the intent of the set of objects. For instance, the intent of a set of executions, \( S \), is the set of lines that appear in the traces of all executions of \( S \) and the verdict if it is the same for all executions of \( S \). In FCA, a formal concept is defined as a pair \((O, A)\), where the set of objects, \( O \), is the extent of the set of attributes, \( A \), and \( A \) is the intent of \( O \). The set of concepts of a context can be represented by a concept lattice where each attribute and each object labels only one concept. Namely, each concept is labelled by the attributes and the objects that are characteristic to its intent and extent.

The failure lattice is computed thanks to combining association rules and formal concept analysis. The set of execution traces is a context where the objects are the executions and where each execution is described by the events occurring during the execution. From that context, association rules are computed: \( e_1, e_2, \cdots \rightarrow FAIL \). Those rules can be read as "when events \( e_i, e_j, \cdots \) appear in a trace most of the time the execution fails". Those rules, extracted from the execution trace context, form the context used to compute the failure lattice. In that second context, the objects are the association rules and each rule is described by the events that appear in its premise. Some rules are more specific than others and that partial order is highlighted in the failure lattice. Figure 2 displays an example of failure lattice for the faulty Trityp program which is obtained by combining faults 1, 2 and 3. Note that, in the sequel, for the examples, the events in execution traces are the executed lines and verdicts.

The premise of a rule \( r \), in the failure lattice, can be obtained by collecting the attributes labelling all the concepts above the concept that is labelled by \( r \). In our particular case, there is only one rule with a given premise, so a concept can only be labelled by one rule in the failure lattice. We will therefore give the same number to a concept and the rule that labels it. In Figure 2, \( r_3 \) is line 105, line 97, line 93, line 58, ... → FAIL. Two pieces of information are added to the rule: support and

---


2. There are some tools that automatically compute concept lattices and their labelling. For instance, Toscana (http://toscanaj.sourceforge.net/) is used for the lattices that appear in this paper.
lift values. The support of a rule is the number of failed executions that have in their trace all events of the premise of the rule. The support of \( r_3 \) means that 116 failed executions execute lines 105, 97, 93, 58, \ldots. The lift of a rule indicates if the occurrence of the premise in a trace increases the probability to fail. A lift value greater than 1 means that the observation of the premise in a trace increases the probability to fail.

In Figure 2, we see that \( r_2 \) is more specific than \( r_3 \), indeed its premise contains the premise of \( r_3 \). It is denoted by \( r_2 < r_3 \). Therefore, \( c_3 \) is a superconcept of \( c_2 \).

Note that in the illustrations, the red ellipses point to the concepts labelled by faulty lines. Note also that we illustrate the method with pictures of concept lattices, but it is only for the sake of exposing the logic of the process. In fact, failure lattices are in general very large and are not legible. They however need not to be computed in totality, neither need to be globally exposed to the user.

IV. THE FAILURE LATTICE FOR MULTIPLE FAULTS

This section presents how the labelling of the lattice is extended by failure concepts and support clusters. Properties for the multiple fault problem are given.

A concept \( c \) is a failure concept of the failure lattice if there exists some failed execution that contains all events of the intent of \( c \) in their trace but not all the events of the intent of subconcepts of \( c \). For instance in Figure 2, concept 5 is a failure concept, indeed it covers 40 failed executions that are not covered by a more specific concept. We also say that those 40 failed executions are covered by rule 5. Concept 4 is not a failure concept because the 40 failed executions that it covers are already covered by concept 5 which is a more specific concept. In the same way, concept 2 is not a failure concept because among the 116 failed executions that are covered by it, 40 are already covered by concepts 4 and 5 and the remaining 76 are covered by concept 10, which are all more specific concepts than concept 2. In the example, there are four failure concepts: 5, 13, 12 and 9. We note FAILURES the set of failure concepts.

We call support cluster a maximal set of connected concepts labelled by rules which have the same support value. We note \( \text{cluster}(c) \) the support cluster that contains \( c \). For instance, in Figure 2, concepts 4 and 5 belong to the same support cluster. It means that lines 71 and 74 are executed by exactly the same 40 failed executions. Note that there is at most one failure concept by support cluster.

The failure lattice has two properties which are essential for the design of the algorithm of next section.

Property 1 (global monotony of support): Let \( c_i \) and \( c_j \) be two concepts of a failure lattice. If \( c_j < c_i \) then \( \text{sup}(r_j) \leq \text{sup}(r_i) \). For instance, in Figure 2, concepts 11 and 13 are ordered, \( c_{13} < c_{11} \) and \( \text{sup}(r_{13}) = 16 < \text{sup}(r_{11}) = 76 \).

Proof: The fact \( c_j < c_i \) implies that the intent of \( c_j \) strictly contains the intent of \( c_i \). The intent of concept \( c_i \) (resp. \( c_j \)) is the premise, \( p_i \) (resp. \( p_j \)), of rule \( r_i \) (resp. \( r_j \)), thus \( p_i \subset p_j \). Conversely extent(\( p_j \)) \subset extent(\( p_i \)). Thanks to the definition of the support \( \text{sup}(r_j) \leq \text{sup}(r_i) \) holds. \( \square \)

Property 2 (local monotony of lift): Let \( c_i \) and \( c_j \) be two concepts of a failure lattice. If \( c_j < c_i \) and \( \text{sup}(r_j) = \text{sup}(r_i) \) then \( \text{lift}(r_j) > \text{lift}(r_i) \). For instance, concept 4 and 5 are ordered, \( c_5 < c_4 \), \( \text{sup}(r_5) = \text{sup}(r_4) = 40 \) and \( \text{lift}(r_5) = 2.38 > \text{lift}(r_4) = 1.19 \). It means that \( r_5 \) and \( r_4 \) cover the same failed executions but \( r_5 \) covers less passed executions than \( r_4 \).

Proof: In the previous proof we have seen that \( c_j < c_i \) implies that extent(\( p_j \)) \subset extent(\( p_i \)). The definition of the lift of a rule \( r \rightarrow \text{FAIL} \) is \( \text{lift}(r) = \frac{\text{extent}(p_j)}{\text{sup}(r)} - \frac{\text{extent}(\text{FAIL})}{\text{sup}(r)} \times |O| \). As \( \text{sup}(r_j) = \text{sup}(r_i) \) and \( \|\text{extent}(p_j)\| < \|\text{extent}(p_i)\| \), \( \text{lift}(r_j) > \text{lift}(r_i) \) holds. \( \square \)

V. EXPLORATION OF THE FAILURE LATTICE FOR MULTIPLE FAULTS

In this section, we propose an algorithm to explore the failure lattice in order to find clues to understand faults. The algorithm presents events to a debugging oracle, currently most likely a human person. We assume the competent debugger hypothesis, namely when a set of events that indicates a fault is presented to the debugging oracle he will detect the fault. The same kind of hypothesis is assumed in [2].

When locating faults in a program, relevant parts of the program are the ones specific to failed executions. In the failure lattice, that information belongs to the most specific rules which label concepts at the bottom of the failure lattice. Furthermore, as stated by Property 1, the bottom concepts are the ones with the lowest support and, as stated by Property 2, the bottom concepts inside a support cluster are the ones with the highest lift. As a consequence, the rule lattice is explored bottom-up, starting from the more specific rules with the lowest support and highest lift.

We define the fault context of a concept \( c \) as the set of events that label strict subconcepts of \( c \). For example, in Figure 2 the fault context of concept 11 is lines 64, 79, 87 and 90. For the localization task, as the lattice is traversed bottom-up, when exploring concept \( c \), the events of its fault context have already been explored. Each time a concept \( c \) is explored, the debugging oracle has to say if the lines that label \( c \), in addition to the fault context of \( c \), indicates a fault. In that case we say that the failed executions that label \( c \) (and thus subconcepts of \( c \)) are explained by that fault. By definition, the concepts belonging to the support cluster of \( c \) are also explained, indeed, the concepts of a same support cluster cover exactly the same failed executions.

The exploration of the failure lattice stops when all failure concepts in FAILURES are explained by at least one fault. Namely, when all failed executions that are covered by a concept in the failure lattice are explained.

The exploration strategy is specified by Algorithm 1. The failure lattice is traversed bottom-up, starting with the failure
Algorithm 1 Failure lattice traversal

1: \( C_{next} := \text{FAILURES} \)
2: \( C_{failure} := \text{FAILURES} \)
3: while \( C_{failure} \neq \emptyset \land C_{next} \neq \emptyset \) do
4: \( \text{let } c \in C_{next} \text{ in} \)
5: \( C_{next} := C_{next} \setminus \{c\} \)
6: if the debugging oracle locates no fault in the label of \( c \) given the fault context of \( c \) then
7: \( C_{next} := C_{next} \cup \{\text{upper neighbours of } c\} \)
8: else
9: \( \text{let } \text{Coverage} = \text{subconcepts}(c) \cup \text{cluster}(c) \text{ in} \)
10: \( C_{next} := C_{next} \setminus \text{Coverage} \)
11: \( C_{failure} := C_{failure} \setminus \text{Coverage} \)
12: end if
13: end while

\[
\begin{array}{c|c|c}
\text{Iteration} & C_{next} & C_{failure} \\
\hline
0 & \{c_5, c_{13}, c_{12}, c_9\} & \{c_5, c_{11}, c_{12}, c_9\} \\
1 & \{c_{13}, c_{12}, c_9\} & \{c_{13}, c_{12}, c_9\} \\
2 & \{c_{12}, c_9\} & \{c_{12}, c_9\} \\
3 & \{c_9, c_7, c_{11}\} & \{c_{12}, c_9\} \\
4 & \{c_7, c_{11}, c_9\} & \{c_{12}, c_9\} \\
5 & \{c_{11}, c_9\} & \{\} \\
\end{array}
\]

TABLE III
Exploration of the failure lattice of Fig. 2.

At each iteration, \( C_{failure} \) can only decrease or remain untouched. The competent debugger hypothesis makes sure that \( C_{failure} \) ends at empty when \( \text{min} \sup \) is equal to 1.

For the example of Figure 2, the support threshold, \( \text{min} \sup \), is equal to 4 failed executions (out of 400 executions, of which 168 failed executions) and the lift threshold, \( \text{min} \lift \), is equal to 1. There are four failure concepts: 5, 13, 12 and 9. Table III presents the values of \( C_{next} \) and \( C_{failure} \) at each iteration of the exploration. We choose to explore the lattice with a queue strategy, namely first in \( C_{next} \), first out of \( C_{next} \). Note that the stopping criterion of the algorithm depends only on the failure concepts, it is therefore valid whatever the chosen strategy.

At the beginning, \( C_{next} \) and \( C_{failure} \) are initialized as the set of all failure concepts (Iteration 0 in Table III). 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 3. The debugger oracle locates fault 3. Concept 5, 4 and 14 are thus tagged as explained. The new values of \( C_{next} \) and \( C_{failure} \) are presented at iteration 1 in Table III. 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 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 \( C_{next} \) and \( C_{failure} \) 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 \( C_{next} \).

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 debugging oracle locates fault 1 at line 84. The fault is the substitution of the test of trityp equal to 2 by a test of trityp equal to 3. Concepts 12 and 9 exhibit two concrete realisations (failures) of the fault at line 84 (Conce 7). Concepts 7, 12, 9 are tagged as explained. The set of failure concepts to explain is empty, thus the exploration stops. All four failures are explained after the debugging oracle has inspected nine lines.

VI. FAULT DEPENDENCIES

This section discusses the behavior of the algorithm with respect to fault dependencies. In the following, a fault is identified by its faulty line. We call fault concept of a fault \( F \) the most specific concept, \( c_F \), such that it is labelled by the faulty line of \( F \).

When two faults are independent, the execution of one fault implies that the other fault cannot be executed. The fault concepts of both faults thus appear in different support
clustering. That case is illustrated in Figure 2 which shows an example of three independent faults. The previous section has shown how the algorithm helps a debugging oracle locate the faults.

Two faults can be partially dependent, it means that some failed executions execute both faults. For example, Figure 3 presents the failure lattice for the Trityp program with faults 1 and 5. Fault 1 is at line 84, fault 6 is at line 64. The failure lattice contains four failure concepts: 2, 3, 4 and 5. There are also 8 support clusters. Running Algorithm 1, the exploration of concepts 2 and 3 does not allow a fault to be located, but gives clues when exploring concept 7. Concept 7 is labelled by line 64. Fault 6 is located. In the same way, exploring concepts 3, 4, 5 and 6 is not sufficient to locate a fault, but gives clues when exploring concept 8. Concept 8 is labelled by lines 84 and 81. Fault 1 is located. The two faults are not independent but they are represented separately, and can be distinguished.

Another example of partially dependent faults is given in Figure 4 that presents the failure lattice for the Trityp program with faults 1 and 4. Fault 1 is at line 84 and fault 4 is at line 90. Executing faulty line 90 implies to have executed faulty line 84 before, but executing line 84 does not always imply executing line 90. Fault 4 strongly depends on fault 1. The failure lattice contains two failure concepts, concepts 1 and 3. There are two support clusters. One support cluster has its support value equal to 104 failed executions. The second support cluster has its support value equal to 119 failed executions (all executions). Running Algorithm 1, concept 1 is explored. It is labelled by two lines: 87 and 90. Line 90 is fault 4. Concept 3 is labelled by line 84 then fault 1 is located. All failure concepts are tagged as explained. The search is finished, the two faults are located. The two faults are dependent but they are represented in different support cluster, and can be distinguished.

The last case is when the faults are always executed together in failed executions. From the point of view of the failed executions, they cannot be distinguished. Figure 5 presents the failure context for the Trityp program with faults 2 and 5. Fault 5, at line 66, sets triptyp to a value above 20, trityp is tested at line 78, with the value set at line line 66 the execution necessarily goes to line 79 which contains fault 2. The failure lattice contains only one failure concept, concept 1. All concepts belong to the same support cluster. The support value of that cluster is equal to 81, which is the number of failed executions. Running Algorithm 1, concept 1 is explored. It is labelled by two lines: 79 and 66. No distinction is done between both faults in the failure lattice. They are seen as a unique fault. Therefore, if the debugging oracle stops diagnosing when he finds one fault without exploring all the lines presented at that step, only one fault is found. Note, that the lines related to the other fault, are nevertheless present at that step and that the oracle can still find both faults in this example. However, in the worst case, when faults are executed by the same failed executions but by some different passed executions, the fault concepts are two different concepts in the same support cluster. It implies that when one of the faults is found, the search stops. All concepts in the support cluster are explained. In that case the first found fault hides the second one and the program has to be executed again after the correction of the first fault.

VII. STATISTICAL INDICATORS

Computing association rules w.r.t. a support threshold \((\minsup)\) filters out some rules. For example, rules that stress very specific lines may have a support under the threshold because they are present in a small number of failed traces. Those rules do not appear in the failure lattice. The failed executions that are covered by only pruned rules may not label concept in the failure lattice. The proposed process helps drawing conclusions from failures that label concepts in the failure lattice. A run of the fault localization process draws every conclusion from the failures that have a sufficient support. A new run can safely considers only the low support failures. This suggests a progressive approach for fault localization as successive runs of rule exploration with decreasing supports.
When the program contains a single fault, a rule is relevant with a lift value greater than 1. When the program contains several faults the lift threshold can be lower. Indeed, when a program contains several faults, verdict $FAIL$ is used as an abstraction of several more specific verdicts. All failures are not caused by the same faults, nevertheless we consider only one attribute, $FAIL$, to characterize failed executions when searching for association rules ($\rightarrow FAIL$). If the failures were tagged by the faults that cause them, for example $FAIL_{F_i}$, the searched association rules would be $? \rightarrow FAIL_{F_i}$. However, we cannot assume that failures are tagged. When a rule $F_i \rightarrow FAIL_{F_i}$ would have a lift greater or equal to 1, the rule $F_i \rightarrow FAIL$ can have a lift lower than 1. The lift threshold can thus start equal to 1 at the beginning of the localization process and decrease below 1 if some faults remain hidden.

Note that, in both the above cases, the test suite does not need to be re-executed.

VIII. RELATED WORK

In [5], we have compared our data structure, the failure lattice, with existing fault localization methods. In this section, we briefly recall the main results and further our navigation into the failure lattice, with the strategies of other methods. Renieris and Reiss [15] as well as Cleve and Zeller [6] have proposed four methods based on the differences between executions traces. In those cases, navigating is the action to explore the whole set of events without guide whereas our approach is guided by the structure of the failure lattice. The statistical methods, Tarantula [10], parallel debugging [9], Falcon [13], SBI [11], the bi-clustering method [16] and SOBER [12] rank events and there is not necessarily a relation between an event and the following one in the ranking. On the contrary, our proposed approach gives a context to understand the faults thanks to a partial ordering of the events, which takes into account their dependencies. In [3], the authors take into account the program dependence graph in their scoring function thanks to the causal-inference theory, but as we previously seen with the other methods, no context is provided with each statement.

For multiple faults, Jiang et al. [8] 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 for debugging without computing the control flow graph. Other methods group together failed executions to separate the effects of different faults ([16], [9]). Those methods do not take into account the dependencies between faults whereas our method does. Finally, SBI [11] has a stopping criterion. SBI wants to take advantage of one execution of the test suite. SBI ranks 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. The difference between SBI and our approach is that our approach does not need to compute the failure lattice several times.

IX. CONCLUSION

In this paper, an algorithm to locate multiple faults in programs is proposed. It is based on a data structure, the failure lattice, that gives a partial ordering of the events of the traces. At each step of the proposed algorithm, a debugging oracle, for example the human debugger, has to say if a fault is found given a set of events. The advantage of our algorithm is that the debugging oracle knows when to stop the exploration of the program.

REFERENCES