Discriminative Pattern Mining in Software Fault Detection

Giuseppe Di Fatta
Department of Computer and Information Science
University of Konstanz
78457 Konstanz, Germany
fatta@inf.uni-konstanz.de

Stefan Leue
Department of Computer and Information Science
University of Konstanz
78457 Konstanz, Germany
Stefan.Leue@uni-konstanz.de

Evghenia Stegantova
Department of Computer and Information Science
University of Konstanz
78457 Konstanz, Germany
steganto@inf.uni-konstanz.de

ABSTRACT

We present a method to enhance fault localization for software systems based on a frequent pattern mining algorithm. Our method is based on a large set of test cases for a given set of programs in which faults can be detected. The test executions are recorded as function call trees. Based on test oracles the tests can be classified into successful and failing tests. A frequent pattern mining algorithm is used to identify frequent subtrees in successful and failing test executions. This information is used to rank functions according to their likelihood of containing a fault. The ranking suggests an order in which to examine the functions during fault analysis. We validate our approach experimentally using a subset of Siemens benchmark programs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Diagnostics, Tracing*

General Terms

Experimentation, Reliability

Keywords

Automated debugging, Fault isolation

1. INTRODUCTION

Insufficient software production and quality assurance technologies are a tremendous societal cost factor for software dependent societies [16]. Considering individual software development projects, software quality assurance amounts to more than 50% of the total cost in the software life cycle. This money is largely spent during the software testing and debugging phases [12].

The applicable software terminology standard [2] defines a software *error* as an inappropriate action committed by a programmer. The results of an error appear as *faults* during the coding of a program. The existence of *errors* and *faults* leads to unexpected results obtained while executing the program on a specific input. The occurrence of unexpected results during program execution is defined as a program *failure*. The objective of this paper is to contribute to the goal of locating software faults by proposing an automated data analysis method working on large sets of software testing data which can support and speedup the detection of software faults.

The most commonly used software quality assurance method is software testing. Customarily, testing a software system involves a large set of test cases. Our method is based on the assumption that for a given set of programs, we have a large number of test cases available. The set of test cases can be split in two categories, namely those corresponding to correct and those corresponding to failing test runs. After execution of the set of test cases every test run is documented in the form of a program execution trace. The large number of program execution traces that we thus obtain gives rise to a data mining problem that is aiming at localizing faults by comparing correct and faulty test runs.

A failure is caused by the presence of a fault. However, a fault does not always lead to an observable failure. The cause of a failure is a set of *circumstances* of the program execution. These are defined by, amongst others, the environment of the program, the program input and the program code [3]. A set of *circumstances* is called a *scenario*. We are not interested in the circumstances themselves but in the effect that they have on the test runs. The main idea is that a scenario that leads to a failing test run is necessarily executed in circumstances in which it contains faults in the program code. And at least one of these faults induces the failure. Hence the data flow created by this scenario will cause a specific control flow in the *neighborhood* of the faulty statement that induced the failure. By the *neighborhood* of a given statement we refer to a part of the control flow of the program run that includes the given statement itself plus some preceding and some succeeding program statements.

The method presented in this paper uses data mining techniques in the analysis of the data generated during program test runs. It reveals meaningful parts of the control flow in program tests that are useful in the discovery of software faults. As an input the method takes a database of traces of program test runs classified as failing and passing runs. The traces are represented in the form of *reduced function call trees*, a data structure that we will define in

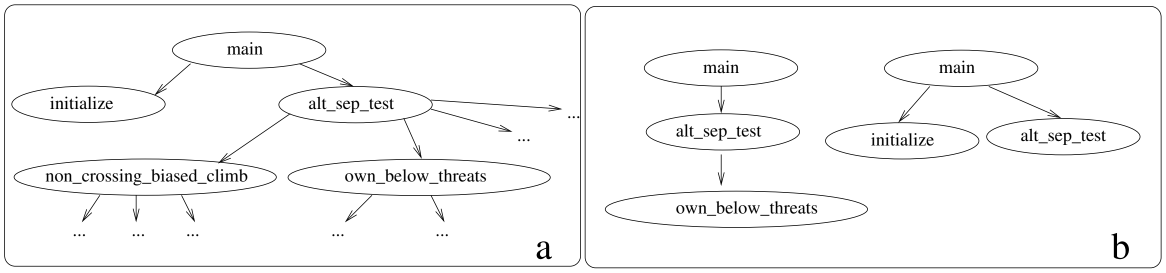

Figure 1: A fragment of the function call tree of the execution of the *tcas* program.

the paper. The method determines frequent patterns inside the reduced function call trees that carry discriminative information which distinguishes between passing and failing test runs. The discriminative trace patterns play the role of the above mentioned control flow *neighborhoods*. We claim that such patterns can potentially be the *neighborhoods* of the faulty statement. To identify these patterns we use a commonly known frequent subtree mining algorithm. We refer to these mining steps of our approach as *data filtering*. Based on the discriminative patterns that the preceding steps yield we finally develop quantitative metrics that rank the functions according to the probability of them containing a fault. The final analysis report contains a list of ranked functions suggesting the order in which the functions should be examined in order to quickly detect the actual fault.

The rest of the paper is structured as follows. In Section 2 we describe the general architecture of our fault localization approach and give a detailed description of every phase. Experiments that we performed as well as results and discussions are presented in Section 3. In Section 4 we give an overview of related work and previous research in this area. Section 5 finally summarizes the approach, presents conclusions and proposes topics for further research.

2. SYSTEM DESCRIPTION

In this section we outline the general architecture of our method.

Our analysis follows the steps that we summarize below

1. *Collection, classification and abstraction phase.* During the executions of the program test suites the execution traces are stored in a suitable format. An oracle classifies the test execution traces as failing or successful. In order to reduce the size of the traces an abstraction procedure is performed. This step is useful to reduce the overall memory and computational requirements.

2. *Filtering phase.* Here we employ a frequent pattern mining algorithm to extract patterns from the database of execution traces that carry discriminative information between correct executions and executions that lead to a failure.

3. *Analysis phase.* We rank the functions according to the probability of them containing a fault and present the final report to the programmer.

2.1 Representation of program traces

The objective of our method is to extract potential *neighborhoods* of the faulty statement in the program code, to analyze them, and finally to draw a conclusion regarding the location of the fault. We use the term *neighborhood* to denote a section of the control flow of a program that is carried out during a part of a particular test run on this program. As a consequence, the execution traces of the test runs that our method gets as an input will contain suitably selected and represented control flow information.

We define a program P as a set of functions $\mathcal{F}$ associated with a given function *main()* which is always executed first. In our approach execution traces gather the information on the function calls in the given program P. We represent a trace of function calls that are executed during a test run of the program as a rooted, ordered, labeled tree $\mathbf{T} = (V, E, \mathcal{F}, L, v_0, \preccurlyeq)$. The set of vertices V represents a set of function calls that occurred during the execution of the program. The set $E \subseteq V \times V$ represents the set of edges such that $(f, g) \in E$ if there is a call to a function g during the executions of the function f. The labeling function $L : V \rightarrow \mathcal{F}$ assigns to each function call v a name of the called function $L(v)$ from the alphabet $\mathcal{F}$. The vertex v_0 represents the root of the tree T. It corresponds to the first function call in the program execution and it is labeled with $main()$. The binary relation $\preccurlyeq \subseteq V^2$ represents a sibling relation for the ordered tree T such that f and g are children of the same parent and $f \preccurlyeq g$ iff the function call f happened before the function call g. A test execution trace represented in the above described way is called a *function call tree*.

As an example, Figure 1a illustrates a fragment of a function call tree corresponding to some execution of the *tcas* program from the set of the Siemens Programs. Function *main()* was executed first. During the execution function *main()* calls consecutively functions *initialize()* and *alt_sep_test()* etc.

Any subtree G that occurs in a function call tree G' is a *neighborhood* in G‘.Figure 1b gives some examples of the neighborhoods that occur in the function call tree illustrated in Figure 1a. We say that a subtree G is of size l iff it has l nodes, i.e. $|V_G| = l$. In this case we can also say that a neighborhood G has a size l.

2.2 Abstractions in program traces

Due to the iterations and recursions that commonly occur in program code, an average function call tree contains a lot of repeated information, in particular identical, repetitive function calls. Figure 2a illustrates an example of a toy program execution tree, where function D is called repeatedly during the execution of a loop in function C a number of times. Such contiguous repeating function calls increase the number of neighborhoods in the execution traces. This

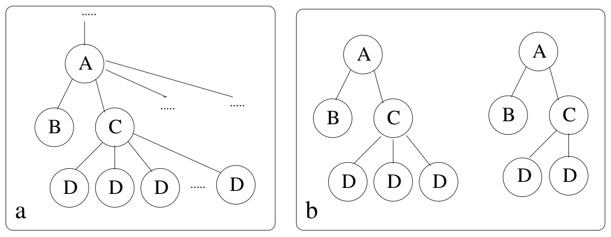

Figure 2: Repetitions.

enlarges the exploration space of the frequent pattern mining algorithm because more candidates in the computation of frequent patterns will be generated. As a consequences the memory requirements of the frequent pattern mining algorithm will increase significantly. On the other hand, as we argue in the following paragraphs we maintain that repeated calls to the same function do not commonly cause errors, although in some situations ignoring repeated function invocations may bring some imprecision into our analysis.

We will now analyze the information value of contiguous repeating function calls with respect to locating a failure. As it was mentioned earlier a scenario that leads to a failing test run is necessarily executed in circumstances in which it contains faults in the program code. At least one of these faults can be assumed to induce the failure. This gives rise to the hypothesis that the data flow created by this scenario will entail a specific control flow in the neighborhood of a faulty statement. Assume that in the program execution illustrated in Figure 2a the fact that C calls the function D in a loop does not become an inevitable consequence of the circumstances that lead to the failure of the program run. However the existence of this loop in the execution trace leads to the appearance of the neighborhoods that differ only by a number of contiguously repeating leaves, for example neighborhoods illustrated in Figure 2b differ by the number of calls to D. Since the loop doesn't become the cause of the failure, these neighborhoods do not carry distinct information in relation to the fault. Hence, we may abstract from contiguously repeating leaves by deleting the repetitions without any loss of meaningful information.

Consider again Figure 2a. It represents a part of the execution trace of a program induced by some faulty scenario. We now analyze the case when executing the faulty statement causes at some point the initiation of function calls to the function D in a loop a particular number of times. Our subsequent reasoning applies equally to the scenario in which a repeated execution of the function D in a loop later becomes a cause for the execution of a faulty statement. In both cases, the loop that initiates calls to the function D is highly related to the fault in the program. Thus it becomes a part of a specific control flow in the neighborhood of the faulty statement. In these situations abstracting a function call tree by reducing the number of repetitions will lead to the loss of a subset out of the set of neighborhoods of the faulty statement. For example, assume that in Figure 2a the execution of the faulty statement leads to a program failure only if function D was called more that 2 times in the loop. Than the occurrence of the neighborhood (Figure 2b) that calls D three times becomes a cause of the failure of the program. On the other hand, the neighborhood that contains only two calls to D can represent the common behavior of the program that not necessarily fails. Reducing the number of repetitions of D in this case will cause the loss of the neighborhood in Figure 2b and thus to the loss of relevant information for our method. We assume that in such cases other neighborhoods specific for this particular failure will also be generated and by deleting the contiguous repeating function calls we do not eliminate all specific neighborhoods, hence we do not predispose our method to the failure.

As a consequence, we perform a *zero-one-many* abstraction on the function call trees by reducing the number of contiguously repeating leaves to two, i.e., the "many" case will be abstracted to the value two. A trace represented as a function call tree with the sequences of contiguously repeating leaves of length reduced to two is called *reduced function call tree*. When in the sequel we speak about execution traces we refer simply to thus *reduced function call trees*.

2.3 Filtering Procedure

In the previous sections we often referred to the specific neighborhoods of a faulty statement that can hint us at the place where a fault is located. To define formally the notion of the *specific neighborhood of a faulty statement* we need to classify the neighborhoods of an average execution trace of a program.

For a given program, a set of potential neighborhoods is defined by the structure of the program code, in particular by the static function call graph. Every time the program is executed on a given input a subset of the set of potential neighborhoods is executed.

Given a dataset of program execution traces D and a stability threshold $s \in [0, 1]$ we adapt the definition of frequent subtrees in a tree dataset to define a set of *neighborhoods* $\mathcal{N}$ *s-stable to the input* in the following way:

$$\mathcal{N}(D, s) = \{n \mid supp(n, D) \geq s\}, \tag{1}$$

where n is a subtree and $supp(n, D)$ is the percentage of traces in D which contain a neighborhood n, i.e. the support of n in D. We also call this set *s-stable neighborhoods* in D.

The term stability appears from the intuition behind the threshold s. Thus the case when threshold $s = 1$ means that neighborhoods from the set $\mathcal{N}(D, 1)$ occurred in all program executions in the database D independently on the input, i.e the set $\mathcal{N}(D, 1)$ is stable in relation to the variations of input data. On the other hand the lower the threshold s is the less is the percentage of the program runs that execute the neighborhoods $\mathcal{N}(D, s)$, i.e. the set $\mathcal{N}(D, s)$ depends on the input data and the stability in relation to the variation of input data decreases.

A simple example of a 1-stable neighborhood could be the function call to the function *main()* that happens at the beginning of every execution. The threshold $s = 0$ defines the set of all neighborhoods that occurred during the execution of the program. Note that any subtree of an s-stable neighborhood is also an s-stable neighborhood.

Let a program P contain a fault and D_f be the set of traces of the failing test runs. We assume that in each of the test runs in the set D_f a data flow has caused the appearance of the neighborhoods of the faulty statement that are specific for the faulty scenarios and that these neighborhoods have a high stability to the input, i.e occur in most of the failing executions . At the same time due to the fact that these neighborhoods appear as a consequence of the

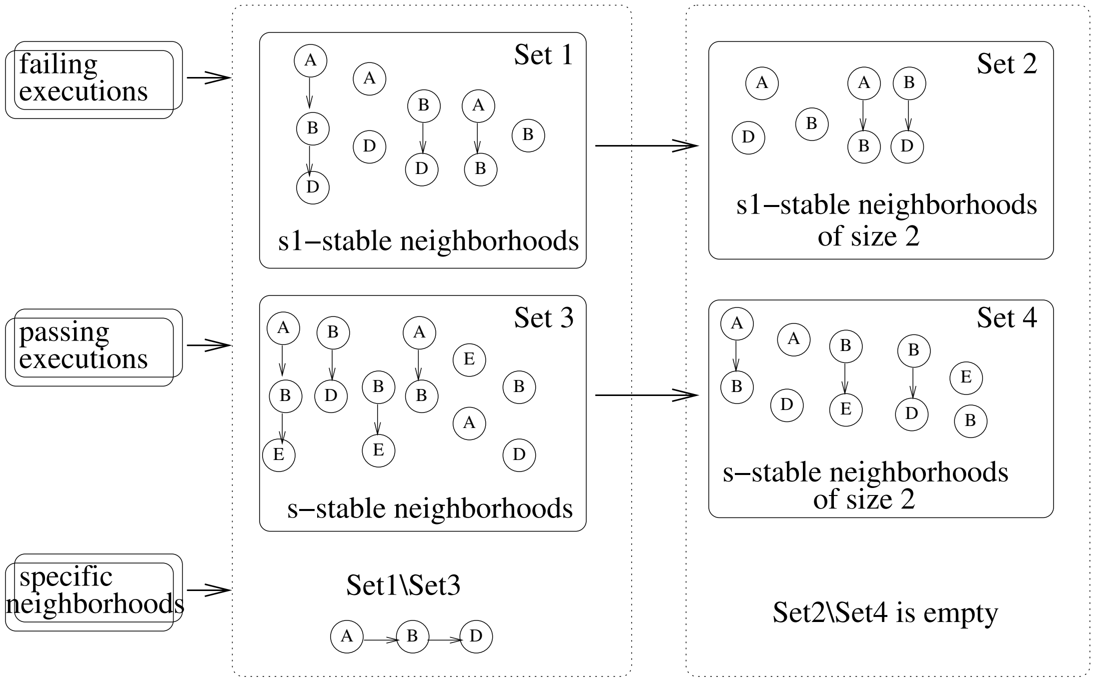

Figure 3: Restricted specific neighborhoods.

faulty scenario in the failing runs, they are likely to occur occasionally but not commonly in the database of traces of passing test runs. Thus we call a set of neighborhoods $\mathcal{SN}$ *specific neighborhoods of a faulty statement* or just *specific neighborhoods* if given a stability threshold s_p for the set of traces of passing test runs D_p and a stability threshold s_f for the set of traces of failing test runs D_f

$$\mathcal{SN}(D_p, s_p, D_f, s_f) = \mathcal{N}(D_f, s_f) \setminus \mathcal{N}(D_p, s_p). \tag{2}$$

The set $\mathcal{SN}$ represents the set of discriminative patterns, i.e. those pattern that are frequent in the set D_f and not frequent in the set D_p.

Our method locates the *s-stable neighborhoods* by means of a frequent pattern mining algorithm.

Realistic program traces are very large, which implies a large number of neighborhoods to be mined by the frequent pattern mining algorithm. This results in a high memory consumption [17, 1]. To control memory consumption in our analysis we restrict ourselves to mining neighborhoods of a particular size l, i.e., neighborhoods that include no more that l function calls. Thus the equation above changes form to

$$\mathcal{SN}(D_p, s_p, D_f, s_f) = \mathcal{N}(D_f, s_f, l) \setminus \mathcal{N}(D_p, s_p, l), \tag{3}$$

where l indicates the upper bound of the size of the neighborhoods to be mined.

However, as it is illustrated on the toy example in Figure 3, restricting the size of a neighborhood entails a loss of information. In Figure 3 the *Set1* is generated by the frequent pattern mining algorithm from the dataset of traces of the failing test runs, given a stability threshold $s1$. The *Set3* is generated from the dataset of traces of the passing test runs, given a stability threshold s. The specific neighborhood identified by our method is the set difference $Set1 \setminus Set3$ and contains one single neighborhood $A \rightarrow B \rightarrow D$. In the next step we restrict the size of neighborhoods that frequent pattern mining algorithms looks for to 2. In this case we obtain *Set2* of s1-stable neighborhoods in the set of failing executions and *Set4* of s-stable neighborhoods in the set of passing execution. As a result the set of specific neighborhoods becomes the empty set. Thus maximizing the parameter l improves the quality of our method.

2.4 Function Ranking

We consider a function an atomic unit of program code. The outcome of our analysis should guide the the programmer which functions to inspect first when searching for the fault. As final step we thererfore rank functions according to their probability of containing a fault.

A naive way to rank the functions is to compare the frequency of calls to a particular function in the initial sets of traces of passing and failing test runs. This approach does not take into account information on the context in which a particular function call occurred, i.e., the neighborhood of that function call. It just formalizes the hypothesis that if some function was called in a bigger percentage of failing test runs than passing test runs, it is more probable that this function contains the faulty statement.

Let D_f be the dataset of traces of failing test runs and D_p the dataset of traces of passing test runs. We denote the probability of a function f to contain a fault by $\mathcal{P}(f)$ and a subtree that contains one single node with label f by t_f. The $\mathcal{P}(f)$ values used in the ranking are computed as follows:

$$\mathcal{P}(f) = \frac{supp(t_f, D_f)}{supp(t_f, D_f) + supp(t_f, D_p)}, \tag{4}$$

As we can see the probability $\mathcal{P}(f)$ becomes 1 if a particular function call occurred only in failing test runs. Functions that have been called approximately in the same percentage of both failing and passing test runs (i.e. function *main()* that is called in all test runs) obtain a probability of around 0.5.

We contend that the above described filtering procedure extracts the specific neighborhoods of the faulty statement. They represent portions of the traces of the failing test runs. Hence these neighborhoods can be viewed as small contexts extracted from a trace that have a direct relation to the fault. We introduce another method to rank functions that exploits the fact that a call to a function has occurred within the *specific neighborhoods*. We say that a function is more probable to contain the faulty statement if it is frequently called in the *specific neighborhoods* and at the same time is highly 'unpopular', i.e infrequently called in passing test runs. We call this ranking system the Frequent Pattern ranking and compute the values for $\mathcal{P}(f)$ in the following way:

$$\mathcal{P}(f) = \frac{supp(t_f, \mathcal{SN})}{supp(t_f, \mathcal{SN}) + supp(t_f, D_p)}, \tag{5}$$

where $\mathcal{SN}$ represents a set of *specific neighborhoods* extracted in the filtering phase.

It is easy to see that the Frequent Pattern ranking assigns the a probability of 0 to contain a fault to functions that weren't called within the specific neighborhoods. It is possible that the set of specific neighborhoods becomes an empty set. This situation is illustrated in the example in Figure 3 and was discussed in the Section 2.3. To avoid having empty results in cases when the calculated set of specific neighborhoods is empty we resort to ranking functions using the naive ranking system for the lack of more suitable information.

3. EVALUATION AND RESULTS

3.1 Experiment Setup

We run our technique on the Siemens programs test suite benchmark. This benchmark consists of 7 correct *C* programs and 132 variations of those programs with injected faults. Each program consists of a number of functions which lies in the range from 7 to 21, and the number of lines in each program varies from 173 to 565. The number of tests provided for each program varies from 1026 to 5542. In our experiment we had to rule out 4 programs, two because the test suites didn't reveal any failure, and two due to technical problems with trace collection.

The filtering phase of our method (c.f. Section 2.3) uses 3 parameters: the stability thresholds s_p and s_f for the datasets of execution traces of failing and passing test runs respectively and a neighborhood size restriction parameter l. Due to the fact that there is one fault injected in the program code it is preferable to use a comparatively high value for s_f. This choice means that we mine the neighborhoods that are highly independent on input variations. For our experiments we keep the value of the s_f of 100%. The value of s_p controls the stability of the *specific neighborhoods* to the input in the passing test runs. In our experiments we set it to 85%. The neighborhood size limit l is mainly influenced by the size of the RAM available on the PC. We perform the experiments for the size limitations of the neighborhoods of 4, 3 and 2, which allowed us to compute frequent patterns in the obtained forest databases on a PC with 512MB RAM available.

Nr. of functions	FP method		
to examine	l=2	l=3	l=4
1	28	27	29
2	19	24	24
3	17	22	17
4	16	10	13
more than 4	47	44	44

Table 1: Results: FP performance

3.2 Report Evaluation

The results of our analysis are presented to the programmer in a report that lists functions ranked according to their probability of containing a fault. The availability of correct versions of the Siemens programs allows us to measure the effectiveness of our method by assigning a score to the final report. The score measures the percentage of program code that the programmer will not have to examine while searching for the fault. For a given report R, its score is measured in the following way:

$$score(R) = \frac{|\{f \mid \mathcal{P}(f) < \mathcal{P}(f^\star)\}|}{|F|}, \tag{6}$$

where $f^\star$ is a function that contains the faulty statement and $|F|$ is the number of functions in the program.

In the evaluation of the report we don't take into account that the size of functions can differ and that some faults are easier to spot than others. The best report possible produced by our method will pinpoint the faulty function, which means that the faulty function is the only one in the highest rank. The score of this best possible report is computed as

$$score(R) = \frac{|F| - 1}{|F|}$$

and converges to 1 for software systems with big numbers of functions. Related to the Siemens programs it can show a rather low score even in case when the best result is achieved, i.e the method pinpoints the faulty function. For example the *tot_info* package has 7 functions, thus the best score that can be achieved is

$$\frac{|F| - 1}{|F|} = \frac{7 - 1}{7} = 0.86.$$

A report scored with 0 ranks the faulty function as one of the least probable of containing the fault. And as a consequence a programmer will examine the whole program before locating the fault.

3.3 Time Complexity

In our experimental tests we use the frequent pattern mining algorithm FREQT, independently introduced by Asai et al. [1] and Zaki [17]. It efficiently extracts frequent ordered labeled subtrees from a database of rooted ordered labeled trees. In order to assess the time complexity we experimented with 30 versions of *replace* package from the Siemens Programs, which has 22 functions. The average time for FP with size restriction l=3 for the analysis of each version is 17.66 sec.

3.4 Results and Analysis

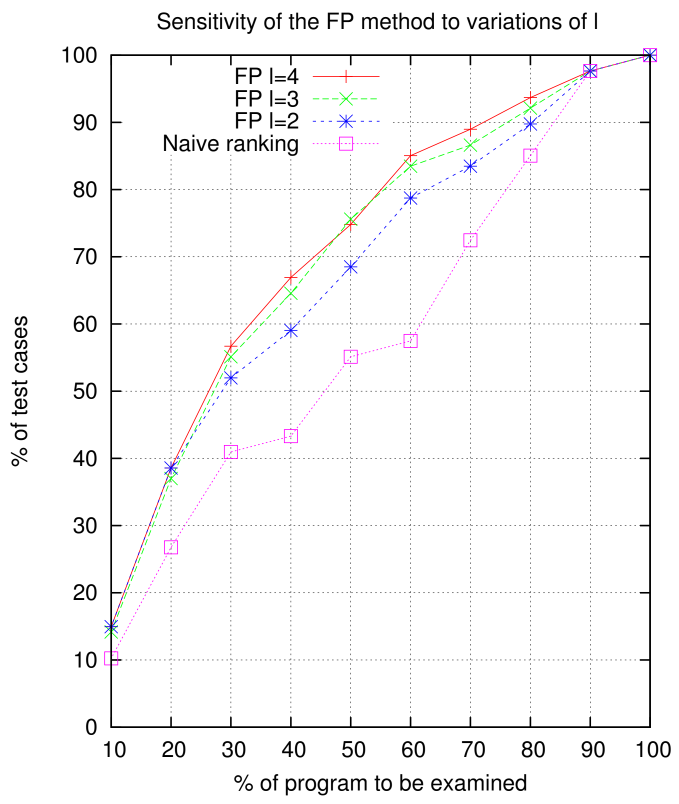

Figure 4: Performance of Naive ranking and Frequent Patterns (with variations of l)

We call our method *Frequent Pattern* and denote it by *FP*. As we can see from the Table 1 the FP technique with neighborhood size restriction $l = 2$, $l = 3$ and $l = 4$ achieve the best score, i.e pinpoint the faulty function in 28 , 27 and 29 cases respectively out of 127. This constitutes about $22-23\%$ of all test cases. And in 48, 51 and 53 test cases, which constitutes about $37-42\%$ of all test cases the programmer would need to walk through at most two functions in order to locate the fault in the program.

In Figure 4 we illustrate the cumulative distribution of effectiveness scores per analysis method where the x-axis indicates the maximum percentage of code that needs to be examined to locate a fault. The y-axis denotes a percentage of the total number of programs that we analyzed. The lines in the chart represent for what percentage of the total number of cases we applied our method to it is necessary to inspect at most which percentage of code in order to locate a fault. I.e., for the FP method with parameter $l = 4$, in 40% of the cases analyzed it is necessary to examine 20% or less of the code in order to locate the fault, and in 55% of the cases analyzed it is necessary to examine 30% or less of the code.

As it can be seen in Figure 4, in the whole range of the test runs Naive Ranking shows poorer performance than the Frequent Pattern method in the whole range of the test runs. This justifies the filtering procedure that we proposed for the FP method.

An important observation we made during the experiment is that the set of specific neighborhoods $\mathcal{SN}$ never occurred to be empty and we never applied Naive ranking to obtain a result.

As we can see from Figure 4 in about 70% of all test cases the programmer will examine 45% of program code to locate a fault, i.e about 30 % of all test cases are accumulated after the border of 45% of program code to be examined. Among these programs are 2 cases with the worst report obtained. This means that the faulty function was considered the least probable to contain the fault and obtained the lowest rank. The analysis of the failing test cases shows that the worst reports are the faulty versions of the *t_cas* program. In these cases in function *initialize()* the *Positive_RA_Alt_Thresh []* array was assigned wrong values. However in most of those cases *ALIM()* was calculated to be the most probable to contain the fault. Function *ALIM* contains just one statement: *return Positive_RA_Alt_Thresh[Alt_Layer_Value];*, it directs the programmer immediately to the values of the variable *Alt_Layer_Value* and array *Positive_RA_Alt_Thresh[]*. The latter was initialized in the function *initialize()*. The other 'bad' reports have diverse faults, for instance wrong type of the variable, wrong or omitted *if* condition or wrong precision of the constant. The cases are distributed over different programs with different static structure and number of functions. We have not yet found the precise classification of the these runs and the reason for the moderate result of the FP method in those cases. The possible explanation could be the choice of parameters and the loss of information due to the neighborhood size restriction.

Figure 4 also illustrates the sensitivity of the FP method to variations of the parameter l. The lines *FP l=2*, *FP l=3* and *FP l=4* illustrate the accumulative scores of the FP method with neighborhood size restrictions 2, 3 and 4 respectively. As one can see all three FP methods yield comparable results in the range of 0-20% of test runs, however later on the performance difference increases and already for 30% of the code *FP l=4* outperforms *FP l=3* and *FP l=3* outperforms *FP l=2*. This relationship is maintained as one moves along the x-axis towards higher percentages. This justifies our hypothesis that one of the reasons of having bad reports is the restriction over the size of the neighborhoods mined in the filtering procedure. At the same time the similarity of the results in the range of 0-20% of test runs implies that the restriction of the size of the neighborhoods does not have a big influence over the best reports FP technique produces.

4. RELATED WORK

The problem of automated fault localization has been investigated in various frameworks. A common approach is to suggest locations in the software source code that are most likely to contain the fault which causes a given failure. We now discuss the main research directions in this area.

The initiators of the coverage based approach, Reps et al. [15] introduce the idea to represent a program execution by a *spectrum*, i.e., a set of features of a program's execution applicable to all programs independent of specific characteristics of a particular program such as input data, control structure, etc. They propose the idea that differences in the spectra of passing and failing test runs are related to defects in the program code. Jones et al. [8] visualize coverage information to assist programmers in fault localization. They introduce a ranking system that ranks the source code statements in the following way: statements more often executed

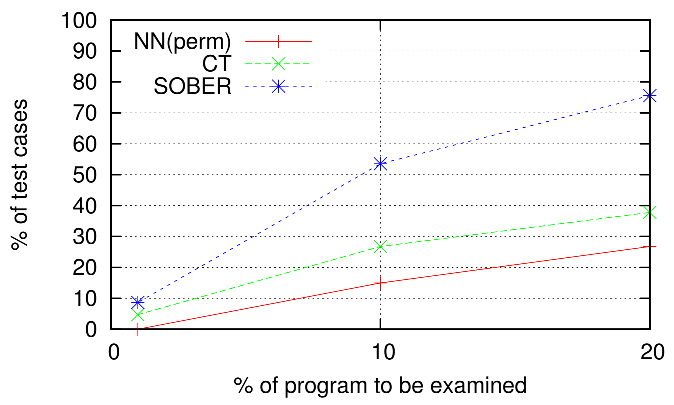

Figure 5: Cumulative score: Nearest Neighbor vs CT vs SOBER

in failing runs than in passing runs rank higher. We adapted this system over program functions as a Naive ranking. As it was illustrated in Section 3.4 FP outperforms Naive ranking, and hence it outperforms the Jones' analysis applied over function coverage of the program execution. Renieris et al. [14] propose a way to locate suspicious regions in the program code by analyzing three models built on the base of the spectra of failing and passing test runs. The most successful *nearest neighbor* model that they define shows encouraging performance. The central idea of the *nearest neighbor* approach is to select out of the set of passing runs the closest one (in terms of coverage, or some other criteria) to the failing run, and then to focus on the difference between these two runs alone. The results of the approach are illustrated in the Figure 5, denoted by NN(perm). The Figure 5 represents the cumulative effectiveness scores of three methods that performed their experiments over the Siemens Programs. It illustrates the percentage of test cases in which no more than a certain percent of code needs to be examined in order to locate a fault. The Plot has the same structure as plot in the Figure 4, however the results can not be compared directly due to the following reason. The methods illustrated in Figure 5 use as a program unit a program statement. Thus the x-axis represents the precise percentage of code. We use functions as program units and the "percentage of code" used for the representation of the results of the FP method in Figure 4 is an approximation (as explained in Section 3.2). Dallmeier et al. [5] propose a fault localization technique for Java based on the postmortem analysis of the method calling object-specific sequences showing up in passing and failing runs.

The approach proposed by Cleve and Zeller [4] called Cause Transition Algorithm represents an enhancement of the Delta Debugging technique initially proposed by Zeller [18]. The original Delta debugging technique implements the "search in space" concept by comparing memory graphs [19] of failing and passing test runs, thus narrowing down the difference between these runs to a set of suspicious variables. In addition to "search in space" the Cause Transition Algorithm performs "search in time" by exploiting cause transitions [4]. The performance of the Cause Transition Algorithm is depicted in Figure 5, the method called CT. Subsequently Gupta et al. [7] combine the delta debugging approach with forward and backward dynamic program slicing. Both techniques use as a program unit a program statement and need only one pair of passing and failing runs for analysis. On the other hand they require the availability of automated scenario modification capabilities as well as means to observe and manipulate the program state. These means and capabilities may not always be available. Our analysis is more time efficient and requires less instrumentation effort.

Statistical analysis proves to be a very powerful approach towards automated fault localization. Dickinson et al. [6] cluster program execution profiles in order to find the program failures. Later on they first perform the feature selection using logistic regression and cluster failure reports within the space of selected features [13]. In their early work Liblit et al.[9] use logistic regression to select suspicious value predicates associated with the fault in the program. Later they propose a technque to isolate the faults in program code based on probabilistic correlations of value predicates and program crashes [10]. Liu et al. [11] use a technique called SOBER based on the hypothesis testing. SOBER uses predicates as program units and is evaluated using the Siemens benchmark programs. The experimental results of SOBER are summarized in Figure 5. SOBER and FP analyze two different types of information: control and data flow. We believe that combining this methods would improve the results in this domain.

5. CONCLUSIONS AND FURTHER DISCUSSIONS

We have present the Frequent Pattern method to enhance fault localization for software systems based on a frequent pattern mining algorithm. The method is applicable to software systems for which a large set of test cases as well as a test oracle are available. It uses a frequent pattern mining algorithm on the function call trees that are used to represent test executions. As a result of the mining, the functions in the program are ranked according to the probability of them containing an error, which greatly facilitates locating faults.

We have evaluated our method experimentally using the Siemens programs test suit as benchmark. The experiments showed that our method outperforms the coverage-based approaches. We also analyzed the sensitivity of our method to changes in the length parameter l which determines the maximum depth of considered subtrees. It turned out that by increasing l we improve the result in the worst cases and even the minimum $l = 2$ is already enough to pinpoint the faulty function in 22% of the test cases.

Threats to the validity of our results lie in the fact that we relied on a single benchmark to evaluate the performance of our method. Programs in the Siemens test suite have moderate size and faults are injected. It hence remains to be proven how the method scales to larger, real life scenarios.

Future research will address steps to alleviate some of the constraints of our method. For instance, it remains to be addressed how the necessary large set of test cases can be automatically synthesized.

To increase scalability, we will look into the question to which extent other program analysis and abstraction methods, such as program slicing, can be used in combination with our proposed method. Finally, it will also be of interest to us to investigate to what extent sequence mining algorithms that work on linear function or statement execution traces compare to the tree mining approach that we use in this paper.

Acknowledgments.. The first and third authors were supported by the DFG-funded PhD graduate program "Explorative Analysis and Visualization of Large Information Spaces" (GK 1042).

6. REFERENCES

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimuri, and S. Arikawa. Optimized substructure discovery for semi-structure data. In *LNAI 2431, Springer-Verlag*, August 2002.

[2] ANSI\IEEE. Ieee standard glossary of software engineering terminology. In *IEEE*, Std 729 - 1983, New York, 1983.

[3] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Proc. of the Fourth International Workshop on Automated Debugging*, Munich, Germany, August 2000.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proc. of the 19th European Conference on Object-Oriented Programming*, 2005.

[6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE'01: Proc. of the 23rd International Conference on Software Engineering*, pages 339–348, 2001.

[7] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code usinf failure-inducing chops. In *Proc. of the International Conference on Automated Software Engineering*, pages 263–272, Long Beach, California, November 2005.

[8] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. J. Jordan. Bug isolation via remote sampling. In J. James, B. Fernwick, and C. Norris, editors, *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of ACM SIGPLAN Notices, pages 141–154, 2003.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. J. Jordan. Scalable statistical bug isolation. In *Proc. of ACM SIGPLAN 2005 Int. Conf. on Programming Language Design and Implementation (PLDI-05)*, 2005.

[11] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. of the 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 05)*, 2005.

[12] G. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc, 1979.

[13] A. Podgurski, D. Leon, P. Francis, W. Marsi, M. Minch, J. Sun, and B. Wang. Automated support for clussifying software failure reports. In *Proc. of the 25th International Conference on Software Engineering*, pages 465–475, 2003.

[14] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *18th International Conference on Automated Software Engineering*, Montreal, Canada, 2003.

[15] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference*, pages 432–449, September 1997.

[16] RTI Health, Social and Economics Research. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2002.

[17] M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2002.

[18] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, Charleston, South Carolina, November 2002.

[19] T. Zimmermann and A. Zeller. Vizualizing memory graphs. In S. Diehl, editor, *Lecture Notes in Computer Science*, volume 2269, pages 191–204, Dagstuhl, Germany, May 2002. Springer-Verlag.