Using Machine Learning to Support Debugging with Tarantula

Lionel C. Briand Yvan Labiche Xuetao Liu
Software Quality Engineering Laboratory
Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
(613) 520 2600 {2471, 5583}
{briand, labiche, xtliu}@sce.carleton.ca

Abstract

Using a specific machine learning technique, this paper proposes a way to identify suspicious statements during debugging which is based on principles similar to Tarantula but addresses its main flaw: Its difficulty to deal with the presence of multiple faults as it assumes that failing test cases execute the same fault(s). The improvement we present in this paper results from the use of C4.5 decision trees to identify various failure conditions based on information regarding the test cases' inputs and outputs. Failing test cases executing under similar conditions are then assumed to fail due to the same fault(s). Statements are then considered suspicious if they are covered by a large proportion of failing test cases that execute under similar conditions. We report on a case study that demonstrates improvement over the original Tarantula technique in terms of statement ranking. Another contribution of the paper is to show that failure conditions as modeled by a C4.5 decision tree accurately predict failures and can therefore be used as well to help debugging.

1. Introduction

One of the most time consuming activities during software testing is debugging. Locating faults causing failures is a very complex endeavor [22]. Though many techniques exist to support this activity [22], this paper is re-visiting the technique that has shown to perform best on existing empirical studies (Tarantula). Based on a careful analysis of its cost-effectiveness we identify a significant problem affecting its applicability and propose a solution to improve it based on the application of a machine learning technique (C4.5). At a high level, Tarantula uses the proportion of test cases that fail when executing a specific statement to determine the ranking of statements in terms of their likelihood to contain a fault. One important issue is that this assumes that test cases fail due to the same fault(s), a situation which nearly never occurs in the presence of multiple faults.

We use C4.5 (decision tree algorithm) [16] to analyze test executions and identify distinct conditions of failures in terms of properties on inputs and outputs. Note that, as further discussed below, a test specification is required (e.g., category partition [12]) as raw test case values are not usable for machine learning algorithms to identify meaningful conditions of failures. Following a strategy similar to Tarantula, within each specific failure condition identified by C4.5, we analyze which test cases matching that condition cover which statements and obtain a ranking. All the rankings of all failure conditions are then combined into a single statement ranking. The most important difference with Tarantula is that because the statement coverage of test cases is analyzed within each distinct failure condition, these test cases are then more likely to fail due to the same faults, an assumption that does not hold when analyzing all test cases at once.

Note that, the failure conditions identified are of interest in their own right, independently of how they help statement ranking. We expect that debuggers would be helped a great deal if we are able to retrieve, from test case definitions and executions, precise conditions on inputs that are almost certain to trigger failures.

Through a case study, we show that our solution has a practically significant, positive impact when compared to Tarantula. We also demonstrate that C4.5, following our procedure, can retrieve accurate rules predicting failures, thus implying that conditions triggering failures can be automatically and precisely characterized in terms of input and output properties.

The above approach can be applied in a number of contexts. For example, this might suit very well the context of test-driven development processes where large test suites are developed up-front and are available during development. It however requires that equivalence classes or categories/choices be defined to specify the test cases or at least—but this is less efficient—to re-express test cases at a high level of abstraction (test specification) in order to feed the machine learning algorithm. Our approach is probably more cost-effective when test suites are specified using a systematic, black-box technique.

The structure of this paper is as follows. We present an extensive overview of related work in Section 2. Section 3 presents background information on the machine learning and testing techniques used in the paper. Section

4 presents the principles and rationale of our proposed approach. Section 5 presents an in-depth case study and we conclude in Section 6.

2. Related work

A number of techniques exist to support the systematic debugging of software [22]. One family of techniques directly relevant to our approach are used to detect anomalies, using Zeller's terminology [22], i.e., differences (e.g., in terms of coverage) between passing and failing executions of the program. These anomalies are good candidates for fault sites, hopefully narrowing down the amount of source code to investigate, and therefore reducing the time (and cost) required to locate faults.

Five such techniques have been so far reported in literature. The simplest ones, referred to as Set Union and Set Intersection in [17], are based on the idea that failing and passing executions (likely) involve different statements. One can therefore look at the statements that are executed in a failing execution but not in passing executions (Set Union—set of statements executed in a failing execution minus the union of the sets of statements executed in passing executions). Alternatively, one can look at the statements common to passing executions but absent from a failing execution (Set Intersection—intersection of the sets of statements executed in passing executions minus the set of statements executed in a failing execution).

The Nearest Neighbor technique is also defined in [17]. It consists in finding a single passing execution that is as similar as possible (e.g., in terms of code covered) to a failing execution (i.e., the nearest neighbor). The difference between the two executions is then worth investigating when looking for faulty statements.

In [3], the authors recognize that some failures are due to specific sequences of method calls, rather than simply the coverage of some statements. Their approach is specific to OO systems (they work at the class level), and in particular Java. The approach is to collect, for each class of the system, the sequences of calls the instances of the class execute at runtime. The idea is then to compare, for each class, the sequences triggered by a failing execution and the sequences triggered by the passing executions, and then rank the classes in such a way that classes whose sequence sets differ the most get the higher priority. (See [3] for the precise ranking mechanism.)

These researchers tend to compare one failing execution (instead of several failing ones) against a number of passing ones, assuming different failing executions may be caused by different faults, which is something unknown beforehand. On the contrary, Jones *et al.* consider all executions together, recognizing that the more a statement is executed during failing executions, the more likely is the statement faulty [7]. The Tarantula technique associates a “color” (ranking) to each statement, accounting for all executions: the redder the statement, the higher the percentage of failing executions that execute this statement. The statement color can be used to rank the statements, as suggested in [6], in order to support the search for faults during debugging. A similar technique is discussed in [9]. It is limited to the observation of certain predicates during program execution (e.g., the value of a predicate is an if statement) as the authors collect execution data from deployed programs, and instrumentation is therefore limited to ensure a reduced impact on the user. The result is a rank of monitored predicates indicating the (statistical) likelihood of the predicate being a failing condition as well as being the fault location. This is expanded upon in [11] where the authors define a similarity function between predicate rankings to group executions that (likely) fail due to the same fault.

As opposed to Tarantula, note that the Set Union, Set Intersection, and Nearest Neighbor techniques simply identify an initial set of suspicious statements to start the search from, not a ranking of statements. As suggested in [6], these three techniques can however be augmented to produce a ranking of statements, as originally suggested in [17] for the Nearest Neighbor technique. The ranking is based on a breadth first search (backward and forward directions) in the system dependence graph from the initial statements. Nodes at the same distance from initial statements are given the same rank.

A number of experimental results discuss how these techniques compare to one another. First, since the technique based on method sequences ranks classes, instead of statements, it cannot be directly compared with the other three techniques [3]. The Nearest Neighbor technique has been shown to outperform the Set Union and Set Intersection techniques [17]. Tarantula has been compared to Set Union, Set Intersection and Nearest Neighbor [6]. For comparison purposes, the authors used the so-called Siemens suite of seven different programs [5]: 122 faulty versions (one fault per version) were used. The ranking of each technique is used to identify the rank of the (known) faulty statement of each faulty version. This rank is used to compute a score for each faulty version, corresponding to the percentage of the program (code) that does not need to be examined to find the faulty statement. They then compare the cumulative number of faulty versions, on the Y-axis, as the score goes from 99% (the technique pinpoints the faulty statement) to 0% (all the code has to be verified to find the faulty statement), on the X-axis. Results show that Tarantula outperforms the other techniques. In particular, in 55.7% of the faulty versions, the fault was found by examining less than 10% of the code. These results were the main motivation for this paper to use Tarantula as a basis of comparison to improve upon.

The effectiveness of Tarantula was originally reported in [7] on 20 faulty versions (one fault per version) of the Space program [19]. No ranking was used though, and the authors only studied the coloring of faulty and non-faulty statements, showing that the former were always colored in red (i.e., suspicious) whereas the latter were most of the time colored in green. (A large number of the non-faulty statements received a reddish or yellowish color though.) There were two notable exceptions: for two faulty versions, the faulty statements received an average color (yellow)—they would therefore not necessarily appear suspicious to the tester—because they were executed by all or most of the test cases and roughly half of the test cases were failing on those versions. These statements were initializing global variables used by all test case executions. The authors also studied Tarantula on multiple-faults versions. They observed that the effectiveness of Tarantula declines (less faulty statements are colored red) as the number of faults increases, though no precise trend can be established.

There are other debugging techniques that are not discussed here, as they are less relevant to our context. They include deduction techniques (e.g., program slicing, delta debugging), and observation techniques (e.g., states, invariants). See [22] for further details.

Another, more recent, body of work related to our problem, attempts to use data mining techniques to help identify suspicious code areas. In [10], the authors record passing and failing program executions, monitoring true and false evaluations of predicates (e.g., if statements). They developed a data mining technique that relies on distributions of true and false evaluations in failing and passing executions to rank (suspicious) functions. They limit their analysis to statements containing predicates, report good results (buggy functions are ranked first) on the Siemens program suite (each faulty version contains only one fault, though), but do not report on any comparison with other techniques.

Similarly, Podgurski *et al.* [14] use passing and failing execution profiles to help debugging. Profiles entail recorded information that is not limited to predicates but can contain any data that can be collected about program executions (e.g., a program variable value, a call stack). The program execution features in the profile that can best distinguish passing from failing executions are first identified using logistic regression. Because the level of abstraction of the collected data needs to be increased to help predict failure, failing profiles are then grouped using either a cluster analysis technique or a multivariate visualization technique. Those groups are then analyzed by the tester to understand failing execution conditions. The type of properties that can be uncovered using this approach is limited and remains to be investigated.

A machine learning technique is used in [1] to identify general program properties (e.g., variables not initialized) that likely indicate the presence of faults in programs. As opposed to abovementioned approaches, it does not require test case executions as it only relies on program analysis (though, for evaluation purposes, the authors use "likely invariants" [4] as program properties, which discovery requires program executions). For a scalar variable x, examples of properties are: equality to a constant ($x=a$), lying in a range (e.g., $a<x\leq b$). In the learning step, properties of correct and incorrect programs are used to build a learning model (e.g., Support Vector Machine) to distinguish faulty and non-faulty programs). In the classification step, a new program's properties are used as inputs and those properties are ranked according to the strength of their association with faulty programs. Note that the approach above requires that abstract properties be defined beforehand, which is similar to our use of categories and choices.

What we propose in this paper is different from the work presented above in at least one of the following ways: (1) We reuse black-box testing specifications (under the form of categories and choices) as inputs to identify failure conditions; (2) Failure conditions are formalized as logical rules, so as to provide interpretable inputs to support debugging; (3) These rules a generated using a machine learning algorithm automatically generating interpretable rules from transformed test cases; (4) Those rules are then used to also rank statements according to their likelihood of containing a fault.

One motivation is to rely on inputs that should anyway be part of any testing plan and is of general usefulness to the tester beyond debugging. A second motivation is to provide feedback to testers in a form that can be used to better understand the reasons for failure. Third, we want our technique to handle the common case where multiple faults are present in the program and different executions fail due to different faults.

3. Background

In this section, we provide overviews of techniques involved in our proposed statement ranking strategy. We describe the Category-Partition (CP) black box testing technique and explain why we need it (Section 3.1). We then discuss rule induction algorithms and explain our choice of C4.5 decision trees (Section 3.2).

3.1. Category-Partition for Rule Induction

In order to obtain meaningful and accurate machine learning rules, we cannot simply use the test case input and output values. Our experience is that it typically leads to meaningless and inaccurate rules because the machine learning algorithm cannot learn what input or output properties are potentially of interest but only which ones matter once they are defined. In other words, without some additional guidance, the learning algorithm is unlikely to find the precise conditions under which test cases fail. This guidance, in our context, comes under the form of categories and choices, as required by CP [12].

The CP method seeks to generate test cases that cover function. To apply the CP method, one identifies the parameters of each function, the characteristics (categories) of each parameter and the choices of each category. Categories are properties of parameters that can have an influence on the behavior of the software under test (e.g., size of an array in a sorting algorithm). Choices (e.g., whether an array is empty) are the potential value of a category which stands for a certain character of the category. Test frames and test data are generated according to the categories and choices defined.

Let's take a simple but hopefully illustrative example: for the well-known Triangle program [8], the input values characterize the length of triangle sides. We can use CP to define categories on the relationships among the length of sides, e.g., whether they are equal or otherwise. In this way, the input of the Triangle program can be described as [compare(side1, side2), compare(side2, side3), compare(side3, side1)] instead of simply [side1, side2, side3]. Then the test cases can be expressed as tuples in terms of these properties and we refer to these tuples as abstract test cases: e.g., raw data (1, 1, 2) becomes (side1=side2, side2<side3, side3>side1). If we obtain failures when the triangle sides are of equal length, based on a set of abstract test cases, the learning algorithm will be able to determine that when all sides are of equal length the program fails to recognize this is an equilateral triangle. Therefore, if no black box testing technique has been used beforehand to obtain a test plan or specification, test suites will need to be transformed into abstract test suites following this above procedure before a rule induction using machine learning algorithms can be applied. An alternative would be to consider higher-order learning algorithms (e.g., Foil [15]) but this is not a practical option at this stage of maturity of the technology.

The reason to use CP here is that it is more general and encompassing than equivalence class partitioning (which only partitions input value domains) and it is one of the most well-known black-box techniques. Though applying CP clearly represents an overhead, in many environments one would be expected to use a systematic functional testing approach, and reverse engineering a test specification (i.e., categories and choices) is in any case a useful investment in the context of legacy systems with existing test suites.

When using the CP methodology, one is also supposed to define constraints and inter-dependencies between choices across categories. Because this is not required in our context, we will not refer to this aspect of the CP methodology.

3.2. Rule Induction Algorithms

There is a large number of machine learning and data mining techniques [20]. They differ widely in terms of their basic principles, their working assumptions, and their weaknesses and strengths. None of the techniques is inherently better than the other and which one is most appropriate tends to be context dependent. Some of these techniques focus on classification, which is the problem at hand in this paper as we want to explain and predict when test cases fail.

A specific category of machine learning techniques focuses on generating classification rules [20]. Examples of such techniques include the C4.5 decision tree algorithm [16] or the Ripper rule induction algorithm [2]. In our context, the rules would look like conditions on test inputs and outputs being associated with probabilities of failures (Fail/Pass classification). The main advantage of these techniques is the interpretability of their models: certain conditions imply a certain probability of failure.

Some techniques, like C4.5, partition the data set (e.g., the set of test cases) in a stepwise manner using complex algorithms and heuristics to avoid over-fitting the data with the goal of generating models that are as simple as possible. Others, like Ripper, are so-called covering algorithms that generate rules in a stepwise manner, removing observations that are "covered" by the rule at each step so that the next step works on a reduced set of observations. With coverage algorithms, rules are interdependent in the sense that they form a "decision list" where rules are supposed to be applicable in the order they were generated. Because this makes their interpretation more difficult, we will use a classification tree algorithm, namely C4.5, and use the WEKA tool [20] to build and assess the trees.

Figure 1 depicts what C4.5 decision trees would look like in our application context, using the Triangle example. Each node represent a (sub)set of abstract test cases and is characterized by a number of conditions on input or output properties (categories and choices using CP terminology). Some nodes are terminal (e.g., 4) whereas others are further decomposed (e.g., 3). Edges represent specific choices for a selected category and a path from the root node of the tree to any leaf can be considered a rule characterizing failure or success, i.e., a conjunction of choices. In Figure 1, nodes are characterized by the relationships between triangle sides. Node (4), for example, captures all the (abstract) test cases where s1 = s2 and s3 = s1 (Equilateral triangle). It is a terminal node predicting failure: in this leaf, the number of failing test cases is higher than the number of passing test cases. Our goal when building the tree is to

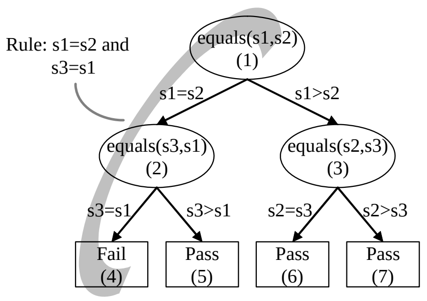

Figure 1 Structure of decision tree and rules

obtain leaves that are as consistent as possible, i.e., which contain test cases that either mostly passed or failed. This is one of the objectives of the C4.5 algorithm. Figure 1 depicts a rule for one of the paths where s1=s2 AND s3=s1 predicts failure (Fail).

4. Statement Ranking Strategies

In this section, we introduce several statement ranking strategies. We first revisit statement coloring according to Tarantula's approach (Section 4.1). We then describe how we adjust Tarantula's ranking based on a C4.5 decision tree, which is itself built on an abstract test suite or test specification (Section 4.2), in order to account for the diversity of execution conditions under which statements are involved in failing test cases.

4.1. Revisiting Tarantula

In order to facilitate the precise and concise definition of our methodology, we need to define and formalize a number of basic concepts.

The set of executable statements of the program *P* being tested and corrected is referred to as *S*. *T* is the test suite devised for testing program *P* (set of test cases). $T_F \cup T_P = T$ where T_F is the set of failed test cases ($T_F \subseteq T$) and T_P is the set of passed test cases ($T_P \subseteq T$): $T_F \cap T_P = \varnothing$. *S(t)* denotes the statements executed (covered) by test case *t*, where $t \in T$: $S(t) \subseteq S$.

R_i denotes a specific rule learned by the C4.5 algorithm. The set of rules is denoted by *R*, where $R_i \in R$. R_F (resp. R_P) is the subset of rules that classifies test cases as Fail (resp. Pass): $R_F \cap R_P = \varnothing$, $R_F \cup R_P = R$.

The set of test cases covered by rule R_i is referred to as $T(R_i)$: $T(R_i) \subseteq T$. If $i<>j$, $T(R_i) \cap T(R_j) = \varnothing$. This latter constraint is due to the partitioning nature of the C4.5 algorithm. Test cases in $T(R_i)$ can pass or fail. Following the same notation as above, $T(R_i) = T_F(R_i) \cup T_P(R_i)$ where $T_F(R_i)$ and $T_P(R_i)$ are the failing and passing test cases covered by rule R_i, respectively. The set of statements executed by a rule R_i, i.e., executed by the test cases covered by R_i, is referred to as $S(R_i)$: $S(R_i) = \bigcup_{t \in T(R_i)} S(t)$.

Following the above formalism, the Tarantula color of any statement *s* in the code is *Color(s) = passed(s) / (passed(s)+failed(s))* where *passed(s)* and *failed(s)* are the percentages of passing and failing test cases that execute statement *s* respectively ($Color(s) \in [0,1]$):

- $passed(s) = |\{ t \in T_P, s \in S(t) \}| / |T_P|$,
- $failed(s) = |\{ t \in T_F, s \in S(t) \}| / |T_F|$.

A small value for *Color(s)* suggests *s* is a suspicious statement. When statements are not covered by any test case, the authors in [6, 7] states that a value of zero should be assigned to *Color(s)*. (Note that this is what the description of Tarantula suggests [6, 7], although the running example used by the authors suggests that, on the contrary, uncovered statements are not ranked.) However, this makes no theoretical sense as there is no information to indicate that the statement is likely to be fault-prone. In fact, no fault causing the observed failures can possibly be located in uncovered statements. This situation is of practical importance as the presence of multiple faults often prevents certain statements from being executed. In our case study, we will discuss a stepwise debugging process to address this issue of uncovered statements (Section 5).

The original *Color(s)* formula presented above can be re-expressed to demonstrate that the statement ranking only depends on the ratio $|T_F(s)|/|T_P(s)|$, where $T_P(s)$ and $T_F(s)$ are the set of passing and failing test cases that execute statement *s* respectively.

$$\frac{1}{Color(s)} = 1 + \frac{|T_F(s)|}{|T_F|} * \frac{|T_P|}{|T_P(s)|} = 1 + \frac{|T_P|}{|T_F|} * \frac{|T_F(s)|}{|T_P(s)|}$$

Since T_P/T_F is constant for a given test set and program version, the ranking of statements, which is ultimately used to focus debugging, exclusively depends on the proportion of test cases that fail and pass when executing a specific statement *s*. Though this makes intuitive sense, there are two problems with this formula: (1) Test cases may fail due to different faults and therefore whether a statement s is covered by several failing test cases is not relevant, (2) redundant test cases that execute the system in identical or similar conditions may artificially affect the ranking.

Another issue is related to how Tarantula rankings were evaluated (Section 2). Ideally, we would like to assess the cost-effectiveness of rankings. In other words, we want to be able to compare some measure of cost and fault detection. To do so we use a scatterplot of percentages of statements verified (a surrogate measure for cost) versus percentages of faults contained in those statements (effectiveness) to compare various statement ranking methods in terms of cost-effectiveness. Though this was not done in the original studies, this will be the basis for our case study analysis.

Also related to evaluation is whether each fault should be considered in isolation or whether all faults should be considered in a single program. Tarantula has originally been used on faulty program versions with a single fault [6, 7] and found to be effective at pinpointing faulty statements on such faulty programs. When used on multiple-fault programs, Tarantula's effectiveness has shown to decrease [7], probably for the reasons we mentioned above. As acknowledged in [7], this deserves further analysis and is important as it is probably more realistic to perform such studies with multiple faults. This issue is the main focus of the current paper.

4.2. Ranking Statements Based on C4.5 Rules

In this section we present the rule-based statement ranking (RUBAR) method. We first explain the basic principles of using C4.5 rules for deriving a ranking and then explain how we select a subset of rules based on all the rules available in the decision tree.

4.2.1. Heuristic Assuming we have a large abstract test suite, we should be able to generate a C4.5 decision tree where each leaf corresponds to a rule predicting either Pass or Fail. The data set on which to build the tree would then be a set of instances, each corresponding to a test case that is known to pass or fail. The attributes to be selected for the tree construction are pairs (category, choice) characterizing the specification of each test case. Each path from the root of the tree to a leaf represents a rule characterized by conditions on (category, choice) pairs. Each leaf in the tree corresponds to a partition of the data set of test case instances. A rule leads to a Fail prediction when the instances in its leaf represent a majority of fail test cases, and Pass otherwise. Therefore, a C4.5 rule classifies a set of test cases with similar input conditions and same test results.

To understand the heuristic we are going to follow, let us first focus on Fail rules and take the example of a particular rule $R_i \in R_F$. If we assume the probability of failure associated with R_i is very high, then what this means is that (nearly) all the test cases in $T(R_i)$ fail under the same or similar conditions. What this suggests is that (nearly) all of the test cases fail due to similar reasons (faults). This in turn can be used to safely rank statements using a strategy similar to Tarantula as all test cases failing within a rule can be safely assumed, in most cases, to fail due to the same fault(s). We thus obtain a ranking per Fail rule that must then somehow be combined with the ranking of other rules to obtain a final ranking.

Within a rule, the higher the number of failing test cases covering a statement, the more suspicious the statement. Assuming that $T_F(R_i)=\{t_1, t_2, t_3\}$, the statements executed by these three test cases are depicted in Figure 2. According to the above hypothesis, the statements in D_1 (i.e., covered by all three test cases t_1, t_2, and t_3) should be more suspicious than those in D_2, D_3, or D_4. On the contrary, the statements in D_4 should be the safest of all.

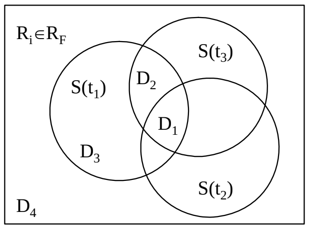

Figure 2 Statement divisions by test cases

If R_i were a Pass rule, we would obtain the opposite result: the statements in D_1 would be the safest and those in D_4 would be the most suspicious. This example illustrates how for each rule we can obtain a partial ordering of statements.

4.2.2. Computing a Statement Ranking To implement the above heuristic, we must define a mechanism to combine the ranking of all rules for a given statement. Proceeding with this goal in mind, a statement will be assigned a negative (resp. positive) weight if it is covered by a Fail (resp. Pass) rule. A high absolute weight value implies that a statement is executed by most of the test cases in the rule. Weight absolute values are normalized within [0, 1] in order to give equivalent weight to all perfect rules, i.e., rules which contain only test cases that either fail or pass. However, if a rule contains test cases with inconsistent Pass/Fail behavior, the maximum absolute weight a rule can contribute will therefore be below 1 and determined by the percentage of test cases it contains in the Fail/Pass category it predicts. The fact that rules should contribute to the extent of the consistency of their test cases' behavior should be intuitive: less consistent rules have less influence on the final ranking. We have also considered weighting the rules' contributions according to their number of test cases, but this would make the results sensitive to redundant test cases, as discussed above in the case of Tarantula, and would make rarely executed faults difficult to find. Assuming that $Weight(R_i,s)$ and $Weight(s)$ denote the weight of statement s for rule R_i and the overall weight of statement s, respectively, we obtain[1]:

$$Weight(R_i,s)=-|T(s)\cap T_F(R_i)|/|T(R_i)|, \text{ when } R_i \in R_F$$
$$Weight(R_i,s)=|T(s)\cap T_P(R_i)|/|T(R_i)|, \text{ when } R_i \in R_P$$
$$Weight(s)=\sum_{R_i \in R} Weight(R_i,s)$$

This can be visualized as a matrix (Table 1) where columns are statements and rows are rules. The matrix contains all the weights of each statement for each rule, all within [-1, 1] as defined above. The last row represents the sum of all rule weights for a given statement, which can then be used to rank statements as

[1] In case one needs to rank a statement that is not covered, we set its weight to 0: a negative weight is suspicious, a positive weight is safe, and for a weight of 0 we cannot conclude.

depicted in Figure 3: Statements with lower weight are ranked first, to follow Tarantula's convention for *Color(s)*.

Table 1 Matrix of statement weight

	s_1	s_2	s_3	…	s_o
R_{P1}	1	0.01	1	…	0.5
…	…	…	…	…	
R_{Pm}	1	1	0.8	…	0.1
R_{F1}	-1	-1	-1	…	-1
…	…	…	…	…	…
R_{Fn}	-1	-1	-0.8	…	0
sum	-22.1	-18.7	0.06	…	-3.36

4.2.3. Rule Selection. A last practical issue is to determine which rules to consider among all the ones identified in a C4.5 decision tree. In order to only account for accurate rules, it seems logical to select rules that are above a certain probability of correct classification (Fail/Pass). We should also consider rules that are based on a large enough number of instances (abstract test cases) in order to avoid classifications that are due to chance. Deciding about such selection thresholds is of course subjective, but it can be done so as to obtain a reasonable number of rules to base the ranking on and will be dependent on how accurate the rules are overall for the specific test suite under consideration. In our case study we select rules with a probability of correct classification above 0.8 and a number of instances above 10, resulting in the selection of 35 Fail rules and 57 Pass rules.

5. Case Study

We first describe the system used as a subject and the settings and design of the case study (Sections 5.1 and 5.2). We then describe the results of applying C4.5 decision trees on our test suite (Section 5.4). Last we report on the relative cost-effectiveness of Tarantula and the method we proposed in Section 4 (Section 5.5).

5.1. The Space program

The Space program was originally developed by the European Space Agency, and first used in a software engineering study by Pasquini *et al.* [13]. It has subsequently been used in other experiments, and in particular in the initial evaluation of Tarantula [7]. Space allows the user to describe the configuration of an array of antennas using a specific array definition language (ADL). It reads a text file containing ADL statements, checks its conformance to the ADL grammar as well as specific consistency rules, and performs other computations. It is a 5905-NLOC C program. (See [21] for further details.) During "testing and operational use" of the program, 33 logical faults were identified and eliminated, and the details of the fault-fixing changes

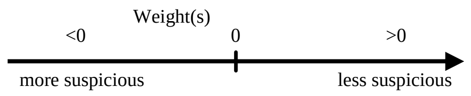

Figure 3 Statement Ranking using *Weight(s)*

were preserved so that the faults could be selectively re-introduced. Vokolos and Frankl [19] used the program for a study in which they compared the effectiveness of regression test selection strategies. For that study, they generated 10,000 test cases using a randomized input generation tool. Rothermel *et al.* [18] later added enough test cases to ensure that each executable Decision was covered by at least 30 test cases in each direction; this procedure added 3,585 test cases to the pool. The resulting test pool covers 90, 85, 85, and 80 percent of all Blocks, Decisions, C-Uses, and P-Uses present in the program, respectively. The total number of blocks, decisions, c-uses, and p-uses are 2995, 1191, 3733, and 1707, respectively.

During the course of their research, Rothermel *et al.* [18] identified and eliminated five more faults, bringing the total number of versions of the program to 38; however, they found that three of the original versions did not exhibit faulty behaviour, reducing the number of non-equivalent versions to 35. We obtained the program, faulty versions and test pool from the Galileo Research Group Subject Infrastructure Repository at the University of Nebraska - Lincoln.

5.2. A multi-fault version of Space

We randomly selected 15 of the 35 original logical faults for our study, avoiding the versions not exhibiting any faulty behavior. We also left out logical faults that involve problems in the initialization statements. Tarantula has been shown to not perform very well with such faults and, though such faults will need to be investigated in the future, we wanted to compare RUBAR and Tarantula on types of faults targeted by the latter. We then built one faulty version of Space by introducing, in the correct version, the 15 logical faults, i.e., 33 faulty statements. The rationale was to simulate a realistic situation where a program contains more than one fault. At the same time, the rationale for not including all the 35 original faults was to obtain a balanced number of failing and passing test cases. The failing rate of test cases on a faulty version including the 35 faults is 95.5%, whereas the rate is 47% on the faulty program containing the 15 selected faults. Future work will investigate ways to deal with high failing rates and initialization faults.

When executing the 13,585 test cases on our multi-fault version of Space, we achieve 69.72% statement coverage (as reported by Coverage Validator[2]). The

[2] http://www.softwareverify.com/cpp/coverage/index.html

executed statements contain 15 of the 33 faulty statements. Not reaching 100% statement coverage and not executing the 33 faulty statements is not surprising since we have several faults in the program. Some faults result in the program exiting prematurely, and therefore in some parts of the code (containing other faults) not being executed. This, however, can be considered a realistic situation under which to experiment.

5.3. Case Study Process

The case study followed an iterative process that aimed at emulating the actual debugging process as realistically as possible. In the first step only 69%, as discussed above, of the statements were reachable. Since the purpose was to identify the location of faults generating the observed failures, we only ranked the covered statements as other statements could not possibly contain the faults causing these failures. In the subsequent debugging step, we then assumed that the faults in the first set of ranked statements were corrected and went on to rank the remaining statements that were left out in the first step. Though our case study only shows two steps, there can be as many steps as necessary. We then append the rankings of all steps and compare the resulting final rankings of all program statements as determined by Tarantula and RUBAR. There are, of course, other ways we could have approached the comparison of these techniques. We could have assumed, for example, that only the faults in the first 20% most fault-prone statements were inspected and corrected. (Investigating the first 20% of the ranked statements has been shown to be a reasonable heuristic with Tarantula [7].) This would have led to two separate iterative debugging processes for each technique and more iteration steps. This will be addressed by future work.

Another important aspect of the case study is whether the decision tree and rules we generate are accurate. This is of course a prerequisite for such rules to be used for statement ranking purposes but also to assess whether we can characterize failure conditions (Fail rules) in an accurate manner.

5.4. Rule Induction Based on Category Partition

When applying category-partition on the test inputs for Space, we identified 207 parameters, 83 categories, and 582 choices. (Note that we did not identify constraints, as required when applying category-partition [12], since we did not use the technique to devise test cases, but used it to characterize test cases.)

By applying the C4.5 algorithm on the initial version of Space containing all 15 faults, we obtained 285 rules (decision tree leaves). The maximum depth of the tree is 12. That is, the longest rules generated have 12 decisions (conditions) involved. Overall, when performing a cross-validation procedure [20], we obtain the confusion matrix in Figure 4. From the matrix, we learn that the C4.5 algorithm misclassified 335 Fail test cases and 550 Pass test cases as Pass test cases and Fail test cases respectively. For Fail test cases this corresponds to a 91.7% precision and a 94.7% recall, which is very good. Similar percentages are obtained for Pass test cases. What this means is that the rules learned by the decision tree model in a nearly complete and precise manner the conditions under which the Space program fails. This result partly reflects the completeness of our categories and choices.

The Fail probability does of course vary across rules as well as the number of test cases they cover. Table 2 shows examples of a few Fail rules (F) with their length (# of conditions), the number of test cases they covered (TC), and the Fail probability. The rules we show in Table 2 are the 33 Fail rules that involve more than 10 test cases and achieve a fail probability of at least 80%. Remember that we need enough instances associated with rules to obtain accurate fail probability estimates.

		Predicted	
		Fail	Pass
Actual	Fail	6045	335
	Pass	550	6655

Figure 4 Cross validation confusion matrix

As an example, let us look at rule F025 (Table 2). It has a length of five, i.e., it is made of five conditions. 667 test cases belong to this rule. The failing probability of a test case satisfying the conditions is 99.55%. Without going into too many details about the specification of the Space program, or a detailed description of our parameters, categories, and choices (i.e., our application of category-partition), rule F025 indicates that if a test case:

Rules	length	TC	Fail Probability
F109	1	30	100.00%
F025	5	667	99.55%
F104	2	240	98.75%
F106	1	304	98.36%
F105	2	158	97.47%
F107	1	114	97.37%
F009	12	147	97.28%
F010	12	132	96.97%
F003	12	37	94.59%
F004	11	200	94.50%
F012	11	97	93.81%
F005	11	548	93.80%
F070	10	271	93.73%
F069	9	116	92.24%
F087	9	199	91.96%
F016	7	12	91.67%
F086	8	700	91.57%

Table 2 Example Fail rules with Fail probabilities and test case coverage

1. defines a triangular grid of antennas (condition 1),
2. defines a uniform amplitude and phase of the antennas (conditions 2 and 3),
3. defines the triangular grid with angle coordinates or Cartesian coordinates, and a value is missing when providing the coordinates (conditions 4 and 5);

then, the test case has a failure probability of 99.55%.

As shown in Table 2, many other rules are highly accurate in predicting failure. Note that such information can potentially be useful to support debugging as it characterizes logically and precisely failure conditions. Since the confusion matrix in Figure 4 suggests the rules are very accurate, then they can be trusted by a tester trying to understand when and why the program fails.

5.5. Comparing Rankings

The cost-effectiveness of the ranking of statements based on Tarantula's coloring is shown in Figure 5 when combining the rankings of the two steps of the debugging process that are required to cover all statements, following the iterative process described in Section 5.3. The first step involves all 15 faults and 33 faulty statements whereas the second step only includes 8 faults and 18 faulty statements. The X-axis is the percentage of statements that need to be covered to detect the percentage of faulty statements on the Y-axis. Each observation represents a statement which is assigned a color value and the statements are ranked on the color value. For example (Figure 5), by visiting the first 10% (resp. 20%) of the covered statements, a tester will visit 10% (resp. 12%) of the faulty statements. Figure 6 shows the same curve when using RUBAR for ranking statements: the pattern is visually very different from Figure 5.

In general, it is very clear when comparing Figure 5 and Figure 6 that Tarantula does not perform well as many of the faulty statements seem to be assigned a higher color value and are not ranked among the first statements to be inspected. We believe, for the reasons discussed previously, that this is due to the incapacity of Tarantula to handle multiple faults in a program. (Note that the rankings stop before 100% as this is also the case for the original test suite.)

On the other hand, the technique we propose based on C4.5 rules performs much better (Figure 6). We can see a very sharp increase in the number of faulty statements covered in the higher 20% range of the ranked statements, which is roughly the range of practical interest in our situation, as discussed by Jones *et al.* [7].

6. Conclusions

This paper describes a new strategy (RUBAR) to support debugging by ranking statements according to their likelihood of containing a fault and by identify in a precise manner the input conditions/properties leading to failure. The general idea is to analyze test case executions using a decision tree learning algorithm that tries to model which conditions lead to test case failure. Potentially relevant conditions/properties are predefined using a black-box test technique (Category-Partition). Each path in the tree then represents a rule modeling distinct conditions of failures, possibly originating from different faults, and leads to a distinct failure probability prediction. Then, in a way similar to the well-known Tarantula technique, this is used to perform statement ranking by accounting for the statement coverage of fail

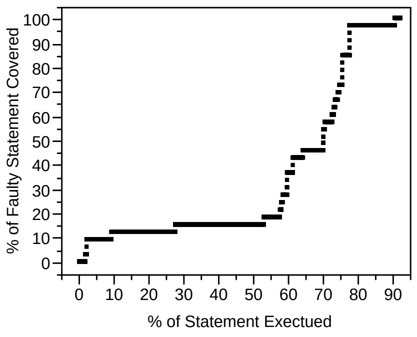

Figure 5 Tarantula ranking

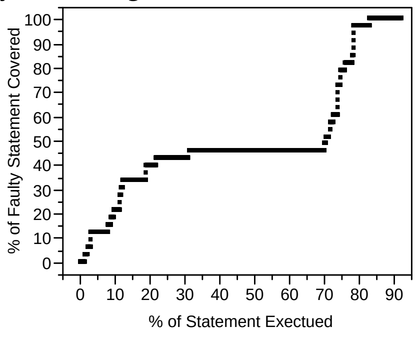

Figure 6 RUBAR ranking

and pass test cases within each rule, assuming that test cases are likely to fail due to the same faults when belonging to the same rule. The rankings of all Pass and Fail rules is then aggregated to form a final statement ranking to be used by the tester to prioritize her search. To summarize, our main objectives are two-fold: (1) deal with programs containing multiple faults and (2) provide the testers with precise information regarding the conditions of failure.

A case study was presented to demonstrate the improvements generated by our approach when compared to the original Tarantula technique. Results show that many faulty statements would be inspected much earlier when using this approach. The difference is clearly of practical significance.

In addition, though it is harder to demonstrate its practical value in quantitative terms, the C4.5 decision tree is also very accurate at modeling the various conditions of failures, thus helping testers understand what could be the causes of these failures. Such conditions are modeled as rules characterizing the input and output properties of test cases.

Future work includes additional case studies, and the application of similar ideas to regression testing and evolving software.

7. Acknowledges

Thanks to all the researchers who worked on and improved the Space program and artifacts over the years. Lionel Briand and Yvan Labiche were further supported by NSERC Discovery grants.

8. Reference

[1] Brun Y. and Ernst M. D., "Finding latent code errors via machine learning over program executions," *Proc. ICSE*, pp. 480-490, 2004.

[2] Cohen W. W. and Singer Y., "Simple, Fast, and Effective Rule Learner," *Proc. AAAI/IAAI*, pp. 335-342, 1999.

[3] Dallmeier V., Lindig C. and Zeller A., "Lightweight Defect Localization for Java," *Proc. ECOOP*, pp. 528-550, 2005.

[4] Ernst M. D., Cockrell J., Griswold W. G. and Notkin D., "Dynamically discovering likely program invariants to support program evolution," *IEEE TSE*, 27 (2), pp. 1-25, 2001.

[5] Hutchins M., Froster H., Goradia T. and Ostrand T., "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. ICSE*, pp. 191-200, 1994.

[6] Jones J. A. and Harrold M. J., "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. ASE*, pp. 273-282, 2005.

[7] Jones J. A., Harrold M. J. and Stasko J. T., "Visualization of Test Information to Assist Fault Localization," *Proc. ICSE*, pp. 467-477, 2002.

[8] Jorgensen P. C., *Software Testing: A Craftsman's Approach*, CRC Press, 2nd Edition, 1995.

[9] Liblit B., Naik M., Zheng A. X., Aiken A. and Jordan M. I., "Scalable Statistical Bug Isolation," *Proc. ACM SIGPLAN PLDI*, pp. 15-26, 2005.

[10] Liu C., "Mining Control Flow Abnormality for Logic Error Isolation," *Proc. SIAM Int. Conf. on data Mining*, 2006.

[11] Liu C. and Han J., "Failure Proximity: A Fault Localization-Based Approach," *Proc. ACM SIGSOFT FSE*, pp. 46-56, 2006.

[12] Ostrand T. J. and Balcer M. J., "The Category-Partition Method for Specifying and Generating Functional Test," *Com. of the ACM*, 31 (6), pp. 676-686, 1988.

[13] Pasquini A., Crespo A. and Matrelle P., "Sensitivity of reliability-growth models to operational profiles errors vs testing accuracy," *IEEE Transactions on Reliability*, 45 (4), pp. 531-540, 1996.

[14] Podgurski A., Leon D., Francis P., Masri W. and Minch M., "Automated Support for Classifying Software Failure Reports," *Proc. ICSE*, pp. 465-475, 2003.

[15] Quinlan J. R., "Learning logical definitions from relations," *Machine Learning*, 5, pp. 239-266, 1990.

[16] Quinlan J. R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

[17] Renieris M. and Reiss S. P., "Fault Localization with Nearest Neighbor Queries," *Proc. ASE*, pp. 30-39, 2003.

[18] Rothermel G., Untch R. H., Chu C. and Harrold M. J., "Prioritizing test cases for regression testing," *IEEE TSE*, 27 (10), pp. 929-948, 2001.

[19] Vokolos F. I. and Frankl P. G., "Empirical evaluation of the textual differencing regression testing technique," *Proc. ICSM*, pp. 44-53, 1998.

[20] Witten I. H. and Frank E., *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufman, 2005.

[21] Wong W. E., Horgan J. R., Mathur A. P. and Pasquini A., "Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application," Software Engineering Research Center, Report TR-173-P, 1997.

[22] Zeller A., *Why Programs Fail: A guide to Systematic Debugging*, Morgan Kaufman, 2005.