Using Machine Learning to Refine Black-Box Test Specifications and Test Suites

Lionel C. Briand (1) Yvan Labiche (2) Zaheer Bawar (2)
(1) Simula Research Laboratory
& University of Oslo
P.O. Box 134, Lysaker, Norway
(2) Carleton University, Squall Lab
1125 Colonel By Drive
Ottawa, ON K1S5B6, Canada
briand@simula.no labiche@sce.carleton.ca zbawar3@scs.carleton.ca

Abstract

In the context of open source development or software evolution, developers often face test suites which have been developed with no apparent rationale and which may need to be augmented or refined to ensure sufficient dependability, or even reduced to meet tight deadlines. We refer to this process as the re-engineering of test suites. It is important to provide both methodological and tool support to help people understand the limitations of test suites and their possible redundancies, so as to be able to refine them in a cost effective manner. To address this problem in the case of black-box testing, we propose a methodology based on machine learning that has shown promising results on a case study.

1 Introduction

In the context of open source development, it is often the case that one is confronted with existing test suites that are based on no explicit rationale or specifications, other than general guidelines for exercising the main user functionalities for example (e.g., based on check lists [25]). For instance, open source software development projects have been shown to lack “attention to basic, accepted, and mature testing techniques [32].” In practice, software developers who intend to reuse open source code are commonly confronted with such ad hoc test suites. It is therefore important to evaluate them and possibly reduce or augment them, depending on whether they are deemed redundant or too weak to achieve a sufficient level of confidence. For instance, Zhao and Elbaum report that in a large proportion of open source software development projects, test suites achieve low source code coverage (e.g., 30%) [32]. Developers hence have an alternative: either build new test suites or reuse existing ones; and they often go for the latter as it (hopefully) reduces effort and therefore costs. This will inevitably lead them to understanding, evaluating and possibly improving these test suites.

Similarly, in an evolution context, because of personnel turnover, the originator of the test suite may not be available and whoever is in charge of modifying and re-testing the software is confronted with understanding and evaluating existing test suites. Even in the context of regression testing, where one needs to select a subset and prioritize existing test cases, it is important to ensure that the original test suite is sufficiently complete and not redundant before selecting or prioritizing.

We propose a partially automated methodology to help software engineers analyze the weaknesses of test suites and iteratively improve them. We refer to this process as the re-engineering of test suites as it is similar to what can be seen in re-engineering source code where code information is extracted, abstracted from a design standpoint, and then used to decide about design changes [11]. Similarly, our methodology is based on abstracting test suite information and then deciding about changes to the test suite. To transform test cases into test case specifications at a higher level of abstraction, we rely on a black-box test specification technique: Category-Partition [26]. Test cases are abstracted under the form of category and choice combinations, as defined in Category-Partition. These choice combinations characterize a test case in terms of input and execution environment properties. A machine-learning algorithm is then used to learn about relationships between inputs/environment conditions and outputs as they are exercised by the test suite. This allows the tester to precisely understand the capabilities and weaknesses of the test suite. Based on a series of systematic heuristics to guide the analysis of those relationships, our methodology then facilitates the improvement of the test suite specification (Category-Partition) and test cases.

Section 2 provides some background. Our approach is described in Sections 3 and 4. A case study is discussed in Section 5. Related work is described in Section 6. Conclusions are drawn in Section 7.

2 Background

2.1 Using Category Partition

To illustrate how the Category Partition (CP) [26] black-box testing method works, let us take the wellknown and simple Triangle program example [21], which we will use as a working example to illustrate the concepts of our methodology. The test input values characterize the length of triangle sides `(a,b,c)` and its output determines whether these sides correspond to an equilateral, isosceles, or irregular triangle. In addition, the program may determine that the sides cannot correspond to a triangle (based on checking certain inequalities) or that the side values are illegal (below or equal to zero). CP requires that we identify properties of the triangle sides that will affect its behavior and possibly its output. The motivation is to ensure that the behavior of the software under test is fully exercised. In our Triangle example, these properties may correspond to Boolean expressions stating relationships between sides, e.g., how `a` compares to `b` and `c`. These properties are called "categories" and are associated with "choices". For example, taking the "`a` compares to `b` and `c`" category, choices could correspond to the two mutually exclusive situations where `a<=b+c` and `a>b+c`. In addition, though we do not make use of them in our approach, CP requires that "properties" and "selectors" be defined to model interdependencies between choices and thus be used to automatically identify impossible combinations of choices across categories [26]. The complete application of CP to the Triangle program is available in [6].

In addition to categories and choices describing input parameters of the program, CP requires the identification of categories and choices for environment conditions, i.e., conditions of the environment of the program that may affect its behavior (e.g., contents of a database, state of external systems, load of the processor or network). CP can therefore help characterize functional as well as non-functional behavior, targeting functional testing, performance testing, and robustness testing.

In our context, once categories and choices are defined, we use them to automatically transform test cases into "abstract" test cases. These can be seen as tuples of choices associated with an output equivalence class. In our example, test case `(a=2,b=3,c=3)` could be abstracted into a tuple such as `(a<=b+c, b=c, isosceles)`: the first choice is the one discussed earlier, for category "how `a` compares to `b` and `c`", the second choice belongs to another category, and the expected output value is `isosceles`. Note that tuples would in reality contain pairs of the form (category, choice) and output equivalence classes instead of simply choices and output values. In this paper, we only show choices for the sake of brevity. Furthermore, simply using output values is usually only applicable in simple cases such as the Triangle example. Even in this case, examples of output equivalent classes could be: (IsTriangle, NotTriangle), the first equivalence class including the following values: Isosceles, Equilateral, Irregular. The selection of an appropriate level of granularity for output equivalence classes will be the tester's decision and will depend on the behavioral complexity of the program and the number of test cases that can be run as, the finer the granularity, the larger the number of test cases generated by our approach.

Notice that tuples typically involve many choices as every existing choice condition that applies to a test case is used when creating the corresponding abstract test case. For example, test case `(a=2,b=3,c=3)` could be abstracted into a tuple such as `(a<=b+c, b=c, a>=0, b>=0, c>=0, isosceles)`, where the last three choices belong to three different categories, each one defining the property of a triangle side as being either strictly negative or not. Last, it may happen that none of the choices defined for a specific category can be used when creating an abstract test case[1]. In such a situation, we add a "not applicable" (or N/A) choice to the category and use this pseudo choice in the tuple. For example, assume a program manipulates a string of characters and its behavior depends on whether the string contains only numbers or only letters (the behavior would furthermore depend on whether the string contains capital letters or not). Then one would define (at least) a category `Cat1` with two choices (`C1` and `C2`, respectively) for the two different types of strings, and a category `Cat2` for strings containing letters with two choices (`C3` and `C4`, respectively) specifying whether the string contains capital letters or not. Suppose that we want to create the abstract test case for a test case where the input parameter contains only numbers. Choice `C1` would be used in the tuple but none of the choices of `Cat2` are applicable. We then define a N/A choice for `Cat2` and use it in the tuple.

Our main reason to transform the test suite into an abstract test suite is that it will be much easier, as described next, for the machine learning algorithm to learn relationships between input properties and output equivalence classes. Devising such categories and choices is anyway necessary to understand the rationale behind test cases and is a way for the tester to formalize her understanding of the functional specifications of the software under test. This is a necessary exercise, as discussed above, both in a context of software evolution or reuse of open source software: if one needs to evolve a test suite one has to first make the effort to understand the system (possibly its source code) and the test suite. Note that the initial categories and choices defined by the tester do not have to be perfect as our methodology will help identify problems in their definitions.

2.2 C4.5 Decision Trees (DT)

There are a large number of machine learning and data mining techniques [31]. They differ widely in terms of

---

[1] This is typically the case when choices cannot be combined across categories, or when categories are not applicable. Such a situation would be specified with "properties" and "selectors" if we were applying CP for the purpose of generating test cases, instead of using CP to characterize existing test cases.

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 learn about the relationship between input and environment properties (categories and choices), and output equivalence classes.

A specific category of machine learning techniques generates classification rules [31] which are easily amenable to interpretation: e.g., the C4.5 decision tree algorithm [29] (where the paths from the root node of the tree to any leaf can be considered a rule), the Ripper rule induction algorithm [9]. In our context, the rules would look like properties on test inputs, i.e., combinations of pairs (category, choice), being associated with output equivalence classes. The main advantage of these techniques is the interpretability of their models: certain conditions imply a certain output equivalence class.

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 overfitting 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 [31] to build and assess the trees.

For the Triangle problem, and based on an abstract test suite, a rule generated by the C4.5 algorithm in the context of the WEKA tool could look like:

```
1  (a vs. b) = a!=b
2  |  (c vs. a+b) = c<=a+b
3  |  |  (a vs. b+c) = a<=b+c
4  |  |  |  (b vs. a+c) = b<=a+c
5  |  |  |  |  (b vs. c) = b=c
6  |  |  |  |  |  (a) = a>0: Isosceles (22.0)
```

This should be read as follows: if `a` is different from `b` (category "`a vs. b`" and choice "`a!=b`"—line 1), `c` is below or equal to `a+b` (category "`c vs. a+b`" and choice "`c<=a+b`"), `a` is below or equal to `b+c` (line 3), `b` is below or equal to `a+c`, `b=c`, and `a>0`, then the triangle is Isosceles (line 6). This rule is based on 22 instances (line 6), that is in our context 22 abstract test cases.

We need to create abstract test cases from concrete (raw) test cases since using raw data will likely generate incorrect and possibly meaningless decision trees. Since the raw data does not contain any explicit information on high-level properties (e.g., categories and choices), it is impossible for an inductive machine learning algorithm to learn which properties are of interest, as it can only relate values of the parameters instead of relevant properties of those parameters. To facilitate the learning process, further guidance needs to be provided to the learning algorithm to generate a meaningful tree. As an example, let us use the Triangle program to illustrate this. Executing C4.5 on the raw test cases for the Triangle program, we obtain a decision tree containing the rule below (among a total of 6 rules): contrary to the rule discussed previously, this rule shows parameter names instead of categories and choices. The rule, which indicates that if `c>1`, `b>1`, and `a>3` then the triangle is `isosceles`, is incorrect since other conditions should hold to have an isosceles triangle. It simply happens that in this test suite, each time those conditions hold, the triangle is isosceles.

```
(c) > 1
|   (b) > 1
|   |   (a) > 3: Isosceles
```

3 OVERVIEW

Figure 1 provides an overview of the steps involved in the MELBA (MachinE Learning based refinement of BlAck-box test specification) methodology we will describe in detail in the next section. The inputs of the methodology are the test suite to be re-engineered and a test specification.

3.1 An Iterative Process

We do not make any specific assumption regarding the contents of a test case and the software unit under test (SUT), other than the feasibility of transforming test cases into abstract test cases given pre-defined categories and choices. In particular, we do not assume that the test suite has been *originally* derived according to the Category-Partition method. Though the test specification, used to characterize existing test cases, is assumed in this paper to follow the category-partition [26] (CP) strategy, future work will investigate how our methodology could be tailored to other black-box strategies. It is worth mentioning that, as other black-box test techniques, the complexity of testing depends on the behavioral specification of the SUT, not necessarily its source code size.

In practice, the test specification may or may not exist to start with, especially if no black-box strategy was used to identify the test cases. In the latter case, which is likely

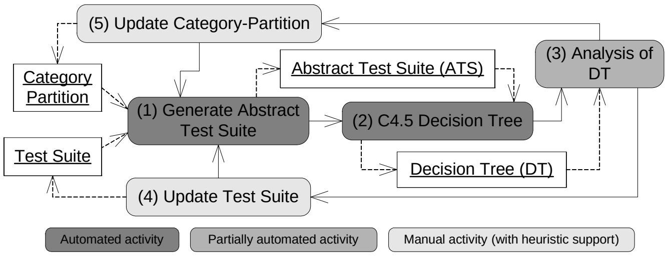


Figure 1 The MELBA Methodology

to be the most common situation, the test specification has to be either reverse-engineered or created from high-level specification (likely plain language). Furthermore, the output domain has to be divided into equivalence classes. The level of granularity of this partition of the output domain is a decision of the tester. Increased granularity will result into increased testing effort but will characterize the SUT behavior in a more precise way.

As the input domain is modeled using CP categories and choices (Section 2.1) the test suite is then transformed into an abstract test suite (Activity 1 in Figure 1). An abstract test case shows an output equivalence class and pairs (category, choice) that characterize its inputs and environment parameters (e.g., execution configuration), instead of raw inputs. Once an abstract test suite is available, a machine learning algorithm (C4.5) is used to learn rules that relate pairs (category, choice), modeling input properties, to output equivalence classes (Activity 2). An example of such rule was discussed in Section 2.2.

These rules are in turn analyzed (Activity 3) to determine potential problems that may indicate redundancy among test cases and the need for additional test cases (Activity 4). Those rules may also indicate that the CP specification needs to be improved, e.g., an important category is missing or certain choices are ill-defined (Activity 5). In the next section, we will detail a number of heuristics that can be used to automatically analyze the C4.5 rules and investigate ways to improve test suites and CP specifications.

The improvement process in Figure 1 is iterative as improvements to either the test suite or test specification can lead to the identification of new problems to be addressed. The learning algorithm will therefore be repeatedly executed (edges from Activities 4 and 5 to Activity 1, followed by Activity 2), which is not an issue as obtaining C4.5 decision trees for a few thousands of (abstract) test cases and a few dozen categories is quick (Section 3.2). The process stops when no more problems can be found in the rules learnt by the machine learning algorithm (Activity 3).

One issue is the presence of faults and its impact on MELBA and C4.5. MELBA assumes that the initial test suite has been run, failures have been detected and the corresponding faults corrected. In short, the starting point of the iterative process is a possibly incomplete but passing set of test cases. However, as the test suite is augmented with new test cases, failures can arise and new faults can be detected. These faults must then be corrected and the new test cases must pass before re-running C4.5 and obtain a new decision tree. Otherwise, since some of the outputs might be incorrect, this might lead to misclassifications in the tree which, though they would not necessarily prevent the use of MELBA, could make the decision tree analysis more complex.

In the context of software evolution, changes and additions to the software naturally lead to changes to the test specifications and corresponding test suites. In that case, the decision tree is automatically rebuilt and the MELBA process is run again to refine the new and changed parts of the test specifications and test suite.

3.2 Manual Effort and Automation

Once the CP specification is defined, the transformation of test cases into abstract test cases is easy to automate. For instance, in our case study, using a CP specification of nine categories and 33 choices and a test suite of 221 test cases, it took a couple of seconds to create 221 abstract test cases. We also used this technology for a different purpose in [7] and with a larger problem: the Space program [28], for which we defined 83 categories and 582 choices, and abstracted 13,585 test cases in less than a minute. In short, Activity 1 in Figure 1 is automated and fast.

Defining categories and choices, on the other hand, requires much thinking as one must identify them so that they determine the system behavior and therefore output equivalence classes. This requires an understanding of the system domain but is, on the other hand, what a tester would typically do anyway when trying to reengineer a test suite, for instance using CP or any other black-box test technique. Though this represents an up-front investment, there is no way one can reuse a test suite or modify a system with confidence without making an effort to understand the relationships between the inputs, environment conditions, and outputs of the system.

Activity 2 is another automated step, for which we use the WEKA tool, which implements C4.5. For our case study, it took less than a second for WEKA to generate a decision tree. In the case of the larger Space problem mentioned above, it took eight seconds to generate a tree based on 13,585 abstract test cases.

Activity 3 is partially automated. On the one hand, much information is automatically provided in the WEKA output: misclassifications, categories and choices used in learnt rules, number of instances (i.e., abstract test cases) involved in rules. This information is the source of our heuristics for problem identification (Section 4.1). The tester then has to identify the causes of those problems, a process that we support with guidelines (Section 4.2).

Activities 4 and 5 are not automated at this point, as they rely on the know-how and expertise of the tester. However, as discussed next, we provide guidance to help identify which categories/choices need to be refined, which abstract test cases need to be defined. Test suite amendment (Activity 4) requires an effort that is anyway incurred if one is improving a test suite.

In other words, we provide partially automated support for test suite specification reconstruction and test suite improvement, activities that are usually entirely manual [23]. Though some level of manual effort is un-avoidable (e.g., discovering categories and choices, creating new

Example 1:

```
(a vs. b) = a=b
| (b vs. c) = b!=c
| | (a vs. b+c) = a<=b+c
| | | (c vs. a+b) = c<=a+b: Isosceles (24.0/2.0)
```

Example 2:

```
(a vs. b) = a=b
| (c) = c>0
| | (b vs. c) = b!=c: Isosceles (24.0/2.0)
```

Figure 2 Examples of detected problems using the Triangle program

test cases), we provide help, under the form of heuristics, to facilitate the tester's work. Also, recall that the initial CP specification does not need to be perfect, and can be improved iteratively through the MELBA process.

4 METHODOLOGY

Our approach is to identify problems in C4.5 decision trees (Section 4.1) and relate them to potential test suite or CP specification deficiencies (Section 4.2). We then discuss strategies to augment a test suite in Section 4.3. We illustrate these steps, i.e., activities 3, 4, and 5 in Figure 1, on our Triangle working example (the CP specification is available in [6]).

4.1 Identifying Problems in C4.5 Trees

When analyzing a C4.5 decision tree in the context of our methodology, we can identify a number of *potential* problems:

Case 1—Instances (test cases) can be misclassified: the wrong output equivalence class is associated to a test case. In other words, a test case belongs to a tree leaf where the majority of instances belong to another output equivalence class.

Case 2—Certain categories or choices are not used in the tree (i.e., they are not selected as attributes to split a (sub)set of instances in the tree).

Case 3—Certain combinations of choices, across categories, are not present on any path, from the root node to any leaf of the tree.

Case 4—A leaf of a tree contains a large number of instances (test cases).

All of the above cases can be automatically detected by a dedicated tool. However, as discussed next (Section 4.2), determining the exact cause of the problem can only be facilitated but not entirely automated.

Cases 2 and 3 have been shown to be the main issues when practitioners apply Category-Partition [8]. The authors suggest that practitioners follow a checklist to systematically identify these problems. In some way, we provide automated support and a set of heuristics to help address these problems. Our work also goes beyond this as we address the improvement of the test suite.

4.2 Linking Problems to Causes

The problems discussed above all have one or more potential causes, as summarized in Figure 3. Specific examples of these problems on the Triangle program can be found in [6].

Misclassifications in the decision tree (Case 1) can have two potential causes: Case 1.1, Case 1.2.

Case 1.1 (Missing category/choice): A category or choice is missing, although it is necessary to determine the appropriate output equivalence class.

Example 1 in Figure 2, for the Triangle example, is produced by C4.5 if one omits the category that tests whether `c` is strictly positive or not (two choices) when using category partition. This results in two misclassified instances (abstract test cases) among 26 instances (24+2), classified as Isosceles triangles by the rule when they are in fact not triangles.

Case 1.2 (Ill-defined choices): Even though a category may be necessary and present in the characterization of test cases, the choices may be ill-defined, making the category a poor attribute to explain the output equivalence classes.

Assuming the two choices of category "`c compared to a and b`" are incorrectly specified as follows:

```
c < a+ b    (should be <=)
c >= a +b   (should be >)
```

C4.5 returns the rule in Example 2 (Figure 2), showing two misclassified instances. Because the relational operators were changed (by moving the "=" operator from `c17` to `c18`), these misclassifications are due to abstract test cases where `c=a+b`.

Both Cases 1.1 and 1.2 will lead to the refinement of the CP specifications, either by adding categories/choices or redefining choices for existing categories.

Some categories (or choices) can be defined in the CP specification but not end up being used in the decision tree (Case 2). This can also be explained by several potential causes: Case 2.1, Case 2.2, Case 2.3.

Case 2.1 (Useless categories): A category may be irrelevant if it turns out not to play a role in determining output equivalence classes. This may be due to the fact that the defined output classes are too rough for the category to play a role or simply that it is redundant (correlated) with other categories.

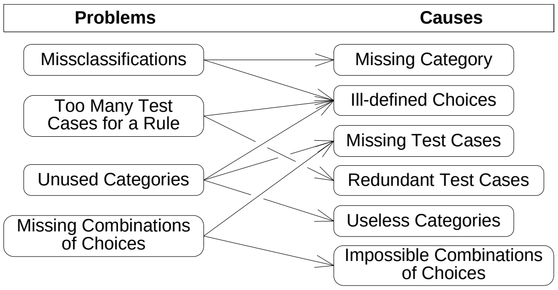


Figure 3 Problems and potential causes

For example, the following category obviously does not play a role in determining the type of a triangle formed by sides `a`, `b`, and `c`, since its choices are redundant with other choices of the CP specification [6]. If added when applying CP, this category would not be selected in the decision tree.

```
CATEGORY - c compared to a
  Choice:   c > a
  Choice:   c <= a
```

Case 2.2 (Missing test cases): Missing test cases can also lead to a category or choice not being selected. For example, there may not be test cases that exercise some or all of the choices of a category, thus resulting in that category being partly used (not all its choices are used) or not being relevant in the decision tree.

For example, to select an extreme case, if all test cases where `a<=0` are removed from the test suite (i.e., in all the test cases, `a>0`) then the category which tests whether `a` is strictly positive or not, will not be selected as this category does not differentiate test cases.

Case 2.3 (Ill-defined choices): Similar to Case 1.2, ill-defined choices may make a category irrelevant as it does not accurately determine the output classes.

Case 2.1 may lead to removing a category from the CP specification, thus leading to a smaller number of test frames and therefore fewer test cases. Case 2.3 would require the modification of choice definitions, possibly leading to an increased number of test cases. Case 2.2 requires the addition of test cases.

Even if all expected categories show up in the tree, certain combinations of choices across categories may not be exercised by the tree (Case 3). This may be the results of several potential causes: Case 3.1, Case 3.2.

Case 3.1 (Impossible combinations): As it is often the case in the context of CP, some combinations of choices may not be possible.

For example, combination of choices `a>b+c` and `c>a+b` is not possible. Recall (Section 2.1) that when building an abstract test case from a concrete test case, we add a N/A choice when a category does not apply to a test case, therefore also suggesting an impossible combinations.

Case 3.2 (Missing test cases): Similar to Case 2.2, if test cases that exercise certain combinations of choices are missing from the test suite, then it is impossible for the tree to identify such combinations as relevant to determine output classes.

The last problem is related to the redundancy of test cases (Case 4). It is in general important to minimize functional test suites and ad hoc test suites often turn out to contain such redundancy. In our context, when many test cases end up in a decision tree leaf then the question arises whether they are all necessary. Indeed, this means that a number of test cases exercise the same choice combinations for a subset of categories and then, as a result, fall in the same output equivalence class. The tester may then consider whether all these test cases in the same tree leaf are necessary as they have similar properties, lead to similar outputs, and probably exercise the software in a similar fashion. Though this remains a subjective decision that only the tester can make, the decision tree points out potential redundancy. There may be, however, two reasons for redundancy: Case 4.1, Case 4.2.

Case 4.1 (Too many test cases for a rule): The most straightforward reason is of course the presence of redundant test cases, as described above.

Case 4.2 (Ill-defined choices): Ill-defined choices can lead to misclassifications but also to the impossibility for C4.5 to split further leaves with large numbers of instances.

It should be fairly easy to differentiate Case 4.1 from Case 4.2. The presence of misclassifications suggests that Case 4.2 is more plausible. No misclassification probably indicates the presence of redundant test cases.

4.3 Heuristics for Adding Test Cases

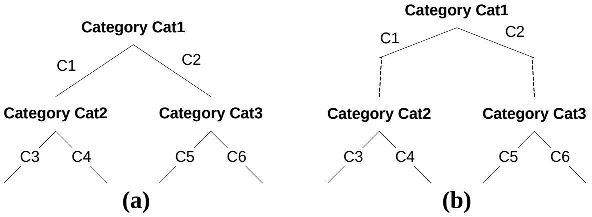
As discussed previously, different reasons can lead to the addition of test cases (Cases 2.2 and 3.2): a choice may be missing in the tree; a category may be missing in the tree; certain choice combinations may be missing.

If a category (or choice) is missing, and the category is useful, then the tester has to create test cases involving each choice of the category[2]. However, the question is which combinations with other choices to include in the test suite? The first solution is to follow the CP method and build all the feasible (according to properties and selectors) combinations of choices and select the ones that are missing in the abstract test suite. We have however discussed that those properties and selectors were not required for applying our methodology (Section 2.1). Furthermore, this is an expensive option that does not make use of the information provided by the decision tree.

An alternative is to identify which combinations of choices may be relevant to determine the output class and could be missing from the test suite. Assume that part of the tree obtained from C4.5 shows categories `Cat1`, `Cat2`, and `Cat3` with choices `C1` and `C2`, `C3` and `C4`, and `C5` and `C6`, respectively, as illustrated in Figure 4Figure 5 (a). The tree excerpt indicates that combining `C2` of category `Cat1` with `C5` or `C6` of category `Cat3` plays a role in determining output equivalence classes (the pairs of choices belong to different paths in the tree). Since `Cat1` has another choice than `C2`, namely `C1`, we may conjecture that `Cat3` might also be relevant to determine the output in the context of `C1` and that the tester should therefore combine choice `C1` with `Cat3`'s choices. Similarly, the tree suggests that the combinations of `C2` with `Cat2`'s choices may be missing in the test suite, thus resulting in four test cases being

---

[2] As a special case, we consider the situation where the tree shows a feasible rule (i.e., feasible choice combination) with no instance. The tester can then simply add a test case for that rule satisfying the corresponding choice combination.



(a) **(b)**

Figure 4 Adding Test Cases from the Tree

added[3]. This heuristic can be generalized to cases where category `Cat1` is not a *parent* of `Cat2` and `Cat3` in the tree but rather an *ancestor* of `Cat2` and `Cat3` (i.e., there are intermediate categories): Figure 4 (b).

One advantage of this heuristic is that by using the information provided by the tree, when intending to cover new choices, the tester does not have only the expensive option to exercise all the feasible combinations of choices, but can focus on combinations that are likely to affect the output.

5 CASE STUDY

In this section, we first describe the system used for the case study and the application of CP on this program (Section 5.1). We then present the design of the case study (Section 5.2) and describe the results of applying MELBA (Section 5.3).

5.1 The PackHexChar Program

PackHexChar is a Java adaptation of the sreadhex procedure, used in the GhostScript program and described in [24], to manipulate hexadecimal characters. PackHexChar takes a string of characters representing hexadecimal digits (parameter S) and compacts the representation of the string in binary format (output), specifically as an array of Bytes: e.g., string “34AB”, corresponding to binary values 0011, 0100, 1010, and 1011, is compacted into an array of two Byte values 00110100 and 10101011 (the binary representation of hexadecimal characters 3 and 4 are combined into the first binary value 00110100). In the input string, characters other than hexadecimal ones are ignored. In addition to the array of Bytes, the program returns an integer value. If the input string contains an even number of hexadecimal characters, pairs of hexadecimal characters are compacted, the program returns the array of Bytes and the returned integer value equals to -1. If the input string contains an odd number of hexadecimal characters, an even number of characters is compacted, and the program returns the remaining hexadecimal character. The user can decide to look at only a sub-string of the input string S, using the input parameter RLEN: the RLEN first characters of S are then analyzed. If RLEN is not a legal value (negative or greater than S’s length), the program returns value -2. The user can ask the program to append a hexadecimal character at the beginning of S. This is useful when a string is split and analyzed in pieces with repeated calls to PackHexChar: a call can return a trailing hexadecimal character, which has to be appended at the beginning of the string during the next call. This is done with input parameter ODD_DIGIT. An ODD_DIGIT value of -1 indicates that no character is to be appended. If ODD_DIGIT has an illegal value (strictly below -1 or not a hexadecimal value), the program returns -3.

Due to time constraints in the design of our case study (see below), we had to select a small program that could be reasonably understood within three hours.

Though the source code itself is small, we can see that the behavior of the PackHexChars program is from a functional standpoint far from being simple. Even when testing entire use cases [18, 19, 26], the number of categories and choices may not be very different from what we have here.

5.2 Design of the Case Study

Recall from the introduction that the MELBA methodology we propose can be applied in two broad application contexts: (1) The reuse, validation, and integration of open source software and (2) software evolution. This leads to two distinct situations that require two slightly different types of case studies. The first situation is not addressed in this paper but is discussed in [6]. The second distinct situation, that is the focus of this paper, is when the CP specification is used to generate the test suite and the test suite must evolve to account for changes in the system under test (Evolution context).

Our case study took place in the context of a specialized 4th year course on software testing. 21 students were properly trained regarding white and black-box testing techniques, including CP. They were asked, during a three hour lab period, to devise a test specification from the source code using CP, and devise a test suite from this specification. The limited time available to browse through the code explains why we had to work with a small though functionally complex program. Due to time constraints, we did not ask the students to go through the MELBA process themselves. The process was applied by a Master student, starting from the CP specification and test suites provided by the students. Results are reported in the next section for one representative student’s CP specification and test suite.

During the MELBA improvement process, the size of each augmented test suite was monitored as well as its capability to detect 231 seeded faults. Faults were seeded by using the usual method of creating mutant programs using a mutation system (Mujava [22]) and then computing the mutation scores of test suites to assess their

---

[3] There is one exception though: if `C1` is an error condition (e.g., an out of range input value), then `C1` is not combined with `C5` and `C6`. This is consistent with the CP strategy.

effectiveness. All non-equivalent mutants (see below) generated by MuJava were retained for the analysis.

We asked an expert, well versed into black-box testing (including CP), to use CP on the PackHexChars program. The expert identified nine categories and 23 choices (referred to as the expert CP specification) [6]. This led to the generation of 221 test cases by identifying all compatible choice combinations (referred to as the expert test suite). The reason for devising the "expert" CP specification was two-fold. First it is intended to be a reference for assessing the student's CP specifications and understand the cause of problems in the decision trees. Second, the resulting large, expert test suite can be used to weed out equivalent mutants. They were identified by running the complete test suite (221 test cases) and then by identifying the remaining live mutants. Following a common heuristic [2], these live mutants were considered to be equivalent though for some of them this is probably not the case. But following this procedure we can ensure all mutants used for the case study are not equivalent.

5.3 Results with one Student' CPs

Student B's test suite contains 31 test cases. We (automatically) created 31 abstract test cases using B's CP specification. Executing C4.5 on these abstract test cases, we obtain the decision tree of Figure 5. The decision tree shows eight misclassified test cases (Case 1). This is due to the student failing to recognize that the program compacts the first `RLEN` hexadecimal characters in the input string (Section 5.1), resulting in a missing category in student B's CP (Case 1.1). Some combinations of choices are also missing in the decision tree (Case 3). Some of them are actually identified in the tree: they have a number of instances equal to 0. The first two rules are feasible combinations and indicate missing test cases (Case 3.2). The subsequent two rules with zero instances are unfeasible combinations of choices (Case 3.1). The decision tree also shows a missing choice (`rlen<0`), which is simply due to missing test cases (Case 2.2).

We first add the missing category to the student's CP:

```
   Category: Number of hexadecimal characters in
the first rlen characters of input string s
   Choice 1: Odd
   Choice 2: Even
   Choice 3: Zero
```

Once the abstract test cases are (automatically) re-created from the updated CP, the execution of C4.5 produces a new decision tree [6]. The tree shows one rule with no instance, which is an unfeasible combination of choices (Case 3.1). Using the heuristic of Section 4.3, the tree also suggests that eight combinations of choices are potentially missing. Looking at the test suite shows that none of them is already exercised.

We therefore create eight test cases, (automatically) produce the corresponding abstract test cases and re-run C4.5, which returns a third decision tree [6]. The tree shows potentially missing choice combinations which are either unfeasible or already appear in some rules. The tree shows three rules with a number of instances larger than the other rules (8, 9, and 7 instances), possibly suggesting redundant test cases. We removed some of the test cases in those rules (randomly selected), keeping one test case for each one of them. We re-run C4.5 and obtain the same tree except that the three rules which had a large number of instances finally contain one instance.

In terms of mutation scores, the test suites of the three iterations found 200, 207, and 205 faults, respectively. The sizes of the test suites were respectively 31, 39, and 12 test cases. Augmenting the test suite in the second iteration seems rather effective: Eight additional test cases kill seven additional mutants. However, our heuristic for removing redundant test cases leads to two mutants remaining undetected, though the reduction in size is relatively substantial. Future work will investigate refinements of our test suite reduction heuristic.

5.4 Discussion

We showed that using MELBA we were able to identify instances of problems in the decision trees and use this information to improve both test suites and CP specifications. The iterative process stopped when no problem could be identified in the trees, at which point the test suites and CP specifications were reaching a quality level that would likely have been achieved by an expert and that was in any case equivalent to the best CP specifications we could derive: when considering only the categories and choices that are selected by C4.5 decision trees—as they determine the output equivalence classes—we found that one or more choices (C') in the expert CP specification correspond to one choice (C) in the students' CP specifications in such a way that the output equivalence class would be predicted the same using C or C'.

From the case study, we can also conclude that our taxonomies of decision tree problems and their possible

```
ODD_DIGIT = odd_digit=-1
| RLEN = rlen=0: -1.0 (2.0)
| RLEN = 0<rlen<=sLength
| | SLENGTH VS. RLEN = sLength>rlen
| | | SCHARTYPE = allValid: S[rlen-1] (5.0/2.0)
| | | SCHARTYPE = N/A: -1.0 (0.0)
| | | SCHARTYPE = allInvalid: -1.0 (0.0)
| | | SCHARTYPE = MixedChars: -1.0 (12.0/5.0)
| | SLENGTH VS. RLEN = sLength=rlen: -1.0 (2.0)
| | SLENGTH VS. RLEN = sLength<rlen: -1.0 (0.0)
| | SLENGTH VS. RLEN = sLength=0: -1.0 (0.0)
| RLEN = rlen>sLength: -2.0 (1.0)
ODD_DIGIT = odd_digit=[0-15]
| SLENGTH = OddLength: odd_digit (1.0)
| SLENGTH = EvenLength: -1.0 (3.0/1.0)
| SLENGTH = Empty: odd_digit (2.0)
ODD_DIGIT = odd_digit>15: -3.0 (2.0)
ODD_DIGIT = odd_digit<-1: -3.0 (1.0)
```

Figure 5 First decision tree for B's TS + B's CP

root causes are complete with respect to the PackHexChar program (seven of the nine problems discussed in Section 4.1 where observed), though future work will need to investigate further whether those taxonomies need to be extended. Furthermore, we observed that based on our students' test suites, who can be considered competent testers in terms of training, we could achieve a final CP specification and test suite in two to three improvement steps [6].

If we step back to reflect on the role of machine learning in the MELBA process, the case study clearly illustrated the necessity to abstract out rules characterizing relationships between inputs, environment conditions, and outputs, from the test specifications and corresponding test suites. For the student for which we report results, we discovered that the student failed to recognize that the program compact the first `RLEN` hexadecimal characters in the input string, resulting in a missing category in student' CP and in missing test cases. Without the machine learning algorithm we would have had to look at the raw test cases (perhaps at the abstract test cases) and it would have been difficult to identify this deficiency of the CP specification and test suite. Indeed, analyzing the raw (or even abstract) test cases (i.e., deciphering the test case description or test case implementation), would show—albeit with difficulty given the number of test cases—what is actually exercised, not necessarily what is not exercised. On the other hand, the machine learning algorithm provides useful information (e.g., misclassifications), thus suggesting that something is wrong and should be investigated.

By analyzing the size and mutation scores associated with the test suites, we can conclude that with a reasonable increase in test cases (8), we found a significant number of additional faults (7). However, though our results also showed that our heuristic to remove redundant test cases leads to significant reduction in test suite size (~50%), a small reduction in the number of faults detected may also be observed. Future work will have to investigate refined heuristics. Though, due to size constraints, only the results of one student could be reported here, the student was selected as representative of the cases that we have investigated [6].

6 RELATED WORK

We see two different areas of work related to the MELBA technology. First, our work bears some similarities with techniques that learn program behavior [1, 5, 17, 27]. Our work differs from those in a number of ways: (1) They all involve the instrumentation of the source code to collect execution traces (e.g., calls to APIs [1], control flow graph [5]); (2) They produce different kinds of (reverse-engineered) specification (e.g., ADTs [1], 'likely invariants' [17, 27], as defined in [13]); (3) They provide no (or little [27]) guidance regarding the definition (or refinement) of test cases (some rely on an automatic test data generator [5]).

A second area of related works are those techniques that attempt to improve test suites [3, 4, 10, 12, 30]. Again, our work differs from those in a number of ways: (1) They all involve the instrumentation of the source code to collect execution traces (e.g., specific statements to reverse-engineer 'likely invariants' [10]) possibly from actual users in the field [12]; (2) A learning algorithm is not always used to help the user improve test suites (e.g., [12]) or no real guidance for the generation of new test cases is always provided (e.g., [30]); (3) When test data are automatically generated, only rudimentary algorithms are used (e.g., simple constraint solving algorithms [10]); (4) They rely on different kinds of (reverse-engineered) specification (e.g., 'likely invariants' [10], as defined in [13]); Z specification [30].

Other research activities are related, as they involve some form of learning mechanism, but have a different overall objective than improving test suites: To understand failure conditions by profiling deployed software [16]; To improve diagnosability by pinpointing faulty statements with a high accuracy [4] (using Tarantula [20]); To identify feasible paths in a control flow graph with high traversing probability using an adaptive sampling mechanism [3] Many other applications of machine learning techniques to software engineering exist in literature (e.g., [7, 14]) but are less related to our focus on test suite and test specification improvement.

To summarize, our approach differs from the above with respect to one or several of the following aspects: (1) It addresses black-box functional testing, (2) It provides guidance in terms of new functional test cases to consider, (3) It helps refine the test specifications from which test cases can then be derived following a clear rationale, (4) It does not require any program instrumentation. To conclude, no existing technique can be directly compared with MELBA.

7 CONCLUSION

This paper proposed MELBA, a partially automated iterative methodology based on the C4.5 machine learning algorithm, to help software engineers analyze the weaknesses and redundancies of test specifications and test suites and iteratively improve them.

The MELBA methodology takes two inputs: (i) a predefined test suite, developed according to a possibly unknown testing method, (ii) a (possibly imperfect) test specification developed using the Category-Partition (CP) strategy. Based on the CP specification, test cases are transformed into abstract test cases which are tuples of pairs (category, choice) associated with an output equivalence class (instead of raw inputs/outputs). C4.5 is then used to learn rules that relate pairs (category, choice), modeling input properties, to output equivalence classes. These rules are in turn analyzed to determine potential improvements of the test suite (e.g., redundant test cases, need for additional test cases) as well as improvements of the CP specification (e.g., need to add a category or choices).

We have illustrated the main aspects of the MELBA methodology on a running example (the Triangle program), and evaluated its effectiveness on test suites and CP specifications created by fully trained 4th year students on a small size but logically complex program. The study showed that the iterative process can indeed improve the CP specification to a level that is equivalent to what an expert would likely produce within two to three improvement cycles. The resulting test suites were significantly more effective in terms of fault detection while only requiring a modest size increase.

Future work will include investigating other black-box specifications than CP, additional evaluations of MELBA on programs of varying sizes and complexities, as well as user-friendly, automated tool support.

8 References

[1] Ammons G., Bodik R. and Larus J. R., "Mining Specifications," *Proc. POPL*, pp. 4-16, 2002.

[2] Andrews J. H., Briand L. C., Labiche Y. and Namin A. S., "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *TSE*, 32 (8), pp. 608-624, 2006.

[3] Baskiotis N., Sebag M., Gaudel M.-C. and Gouraud S., "A Machine Learning Approach for Statistical Software Testing," *Proc. Int. Joint Conf. on Artificial Intelligence*, pp. 2274-2279, 2007.

[4] Baudry B., Fleurey F. and Le Traon Y., "Improving Test Suites for Efficient Fault Localization," *Proc. ICSE*, pp. 82-91, 2006.

[5] Bowring J. F., Rehg J. M. and Harrold M. J., "Active Learning for Automatic Classification of Software Behavior," *Proc. ISSTA*, pp. 195-205, 2004.

[6] Briand L. C., Labiche Y. and Bawar Z., "Using Machine Learning to Refine Black-box Test Specifications and Test Suites," Carleton Univ., Tech. Report SCE-07-05, 2007.

[7] Briand L. C., Labiche Y. and Liu X., "Using Machine Learning to Support Debugging with Tarantula," *Proc. ISSRE*, pp. 137-146, 2007.

[8] Chen T. Y., Poon P.-L., Tang S.-F. and Tse T. H., "On the Identification of Categories and Choices for Specification-Based Test Case Generation," *IST*, 46(13), pp. 887-898,2004.

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

[10] Csallner C. and Smaragdakis Y., "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," *Proc. ISSTA*, pp. 245-254, 2006.

[11] Demeyer S., Ducasse S. and Nierstrasz O., *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2003.

[12] Elbaum S. and Diep M., "Profiling Deployed Software: Assessing Strategies and Testing Opportunities," *TSE*, 31 (4), pp. 312-327, 2005.

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

[14] Francis P., Leon D., Minch M. and Podgurski A., "Tree-Based Methods for Classifying Software Failures," *Proc. ISSRE*, pp. 451-462, 2004.

[15] Grochtmann M. and Grimm K., "Classification Trees for Partition Testing," *STVR*, 3 (2), pp. 63-82, 1993.

[16] Haran M., Karr A., Last M., Orso A., Porter A., Sanil A. and Fouche S., "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks," *TSE*, 33 (5), pp. 1-18, 2007.

[17] Harder M., Mellen J. and Ernst M. D., "Improving Test Suites via Operational Abstraction," *Proc. ICSE*, pp. 60-71, 2003.

[18] Hartmann J., Vieira M., Foster H. and Ruder A., "A UML-Based Approach to System Testing," *Innovations in Systems and Software Eng.*, 1 (1), pp. 12-24, 2005.

[19] Hartmann J., Vieira M. and Ruder A., "UML based Test Generation and Execution," *Proc. Workshop on Software Test, Analyses and Verification*, 2004.

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

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

[22] Ma Y.-S., Offutt A. J. and Kwon Y.-R., "MuJava: A Mutation System for Java," *Proc. ICSE*, pp. 827-830, 2006.

[23] Maki-Asiala P. and Matinlassi M., "Quality Assurance of Open Source Components: Integrator Point of View," *Proc. COMPSAC*, pp. 189-194, 2006.

[24] Marick B., *The Craft of Software Testing*, Prentice Hall, 1995.

[25] Michlmayr M., Hunt F. and Probert D., "Quality practices and problems in free software projects," *Proc. Int. Conference on Open Source Systems*, pp. 24-28, 2005.

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

[27] Pacheco C. and Ernst M. D., "Eclat: Automatic generation and Classification of Test Inputs," *Proc. ECOOP*, pp. 504-527, 2005.

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

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

[30] Singh H., Conrad M. and Sadeghipour S., "Test Case Design Based on Z and the Classification-Tree Method," *Proc. ICFEM*, pp. 81-90, 1997.

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

[32] Zhao L. and Elbaum S., "Quality assurance under the open source development model," *JSS*, 66 (1), pp. 65-75, 2003.