ABSTRACT

In the context of open source development or software evolution, developers are often faced with 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 possibly reduced to meet tight deadlines. We will 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.

Keywords
Black-box testing, Category-Partition, Machine Learning.

1. INTRODUCTION

In the context of open source development or software evolution, it is often the case that one is confronted with existing test suites that are based on no explicit rationale or specifications. In practice, software developers 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 example, software developers who intend to reuse open source code will obviously test it to ensure its dependability and may want to reuse available test suites, which will inevitably lead them to evaluate and possibly improve these test suites. 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 an automated methodology, based on machine learning, to help software engineers analyze the weaknesses of test suites so as to be able to 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 [10].

The rest of the paper is structured as follows. Related work is described in Section 2. Section 3 provides some background on Category-Partition, the black-box testing technique we use, and machine learning. Our iterative approach is described in Sections 4 and 5. The results of a case study are discussed in Section 6. Conclusions are drawn in Section 7.

2. RELATED WORK

In [1], the authors introduce a technique to learn a specification from execution traces. The generated specification is a finite automaton and the authors focus on mining uses of APIs or ADTs (i.e., their specification): nodes of the automaton are calls to the API (or ADT). The approach can be the starting point of a testing procedure: The (automaton) specification of an API can be learnt from its use in program P1 and its use in program P2 can be checked against the learnt specification. As opposed to our work, their approach does not guide the definition (or refinement) of test cases.

Execution traces are also used in [5] to learn program behavior: the technique records the execution of branches in the program (inter-procedural) control flow graph. The classifier being learnt is a map between execution traces (i.e., branch execution profiles) and behavior classes (e.g., fail/pass), and can be used to guide the construction (or extension) of a test suite. A classifier is first built using execution traces and corresponding behavior classes. Any newly defined and executed test case is then either used to refine the classifier (when the test case provides a new mapping between a profile and behavior class) or discarded (if the mapping is already in the classifier). In the former case, the tester has to provide a behavior class, typically by building an oracle. Although the classifier helps the tester identify (somewhat) redundant test cases, thus avoiding the cost of building an oracle for them, the technique requires that the test case be defined by other means and executed before determining whether it is a useful addition or not. The strategy can only be practical if the tester can ensure that the new test data have high chances of triggering new behavior (otherwise, building test cases and more so executing them may be expensive): e.g., if an automatic test data generator is used, as suggested by the authors, there is a risk that many executed test cases be redundant and recognized by the classifier, therefore providing little help. Though the approach is very effective at determining whether a new test case is a useful addition, it provides little help for the definition of new, interesting test cases (the authors assume an automated test input and test case generation procedure is available).

Another execution trace based approach to test suite evaluation and extension is proposed in [16]. As long as the reverse-engineered specification, in the form of so-called ‘likely invariants’ [12], does not change when adding a test case to a test
suites, the test case is deemed redundant. This approach does not require the construction of oracles, as opposed to the previous ones. The quality of the result however depends on the program invariant patterns that are used. The program invariants being discovered from execution information must be instances of a set of pre-defined invariant patterns. An invariant that does not fall into this category cannot be recognized. Furthermore, the approach does not provide guidance regarding the definition of new test cases.

This approach is expanded upon in [24] where the authors use a specification (invariants), reverse-engineered from passing test cases, to determine whether new automatically-generated (mainly randomly) test inputs are illegal, legal, or fault revealing based on the run-time monitoring of invariants. Illegal test cases violate the reverse-engineered pre and post conditions of a method, whereas fault revealing test cases only violate post conditions. However, such a test case may not necessarily represent a failure since the pre and post conditions may not be complete (they have been learnt from a set of test cases that may not have exercised every possible behavior of the method). The technique is further expanded in [9] where reverse-engineered likely invariants [12] are analyzed by a constraint solver to automatically produce test inputs and test cases. Constraint solving is however limited; for example, integer variables are supported but not floating point variables. In both [9] and [24], only likely invariants matching pre-defined patterns are considered (recall the discussion above).

In [11] the authors investigate how profiling deployed software, i.e., collecting execution information in the field, can help improve a test suite. No learning mechanism is employed though: additions to the initial in-house test suite entail repeating scenarios observed in the field and new test cases derived from trace data. The use of learning algorithms to understand program executions is applied to the problem of profiling deployed software in [15]. The objective is to be able to accurately classify program executions as fail or pass executions based on low-cost, limited instrumentation of deployed programs, rather than to improve a test suite.

Improving diagnosability by pinpointing faulty statements with a high accuracy is the objective of the test suite improving technique presented in [4]. The approach relies on a new testing criterion, that evaluates the “fault locating power” of a test case (measured using Tarantula [19]), to evaluate the test suite to improve, and uses a bacterial algorithm (an adaptation of a genetic algorithm) to actually find improved test cases.

In [3] the authors present an adaptive sampling mechanism to identify feasible paths in a control flow graph with high traversing probability. Instead of uniformly sampling the set of paths in the control flow graph of a program (which usually contains a very large number of unfeasible paths), the authors devise an adaptive (learning) approach to ease the identification of feasible paths. The adaptive aspect of the approach is to learn, from the structure of known feasible paths, which branches to traverse in order to obtain a new feasible path.

In [27], the authors present a technique to improve a test suite based on a Z specification of the program/function under test and the Classification Tree method¹. They show how a classification tree of the program input parameters can be created semi-automatically from its Z specification. The classification can be enriched with any category/choice if this is deemed necessary by the tester. Test cases can then be defined using the classification: they are predicates on the program input parameters, i.e., conjunctions of the categories/choices in the tree. These predicates are then combined with the predicates of the Z specification (see [27] for details): (1) determine the expected output of a test case, (2) refine the user defined test cases, completing their definitions with categories/choices that have been omitted when applying the Classification Tree method, and (3) identify problems in the Z specification, such as incompleteness (though no guidance is provided to solve this issue).

Many other applications of machine learning techniques to software engineering exist in literature (e.g., [6, 13]) 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 (and not only the improvement of existing ones), (3) It helps refine the test specifications from which test cases can then be derived following a clear rationale.

3. BACKGROUND

Our approach, detailed in Sections 4 and 5, relies on a well-known black box testing technique, namely Category Partition [23], as well as machine learning. In this section we discuss these technologies in our context.

3.1 Using Category Partition

To illustrate how the Category Partition (CP) [23] black-box testing method works, let us take the well-known and simple Triangle program example [20], 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., \(a \geq b\) and \(c\). These properties are called “categories” and are associated with “choices”. For example, taking the “\(a \geq b\) 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

¹ The Classification Tree method [14] is a black-box partition testing technique, supported by a tool, similar to Category-Partition [23].
categories [23]. The complete application of CP to the Triangle program is available in the Appendix.

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 the 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 (Section 5).

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 case2. 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 \(C1\) with two choices \((C1\) and \(C2\), respectively) for the two different types of strings, and a category \(C3\) for strings containing letters with two choices \((C3\) and \(C4\), respectively) specifying whether the string contains capital letters or not. Suppose now 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 \(C2\) are applicable. We then define a N/A choice for \(C2\) 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 in the next section, 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.

### 3.2 C4.5 Decision Trees

There are a large number of machine learning and data mining techniques [28]. 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 learn about the relationship between input properties (categories and choices) and output equivalence classes.

A specific category of machine learning techniques focuses on generating classification rules [28] which are easily amenable to interpretation. Examples of such techniques include the C4.5 decision tree algorithm [26] (where the paths from the root node of the tree to any leaf can be considered a rule) or the Ripper rule induction algorithm [8]. 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 [28] 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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>(a vs. b) = a!=b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>(c vs. a+b) = c&lt;=a+b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>(a vs. a+c) = a=b+c</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>(b vs. a+c) = b&lt;=a+c</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>(b vs. c) = b=c</td>
<td></td>
</tr>
</tbody>
</table>
| 5 |   |   |   | (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 “\(c<=a+b\)”—line 2), \(a\) is below or equal to \(b+c\) (line 3), \(b\) is below or equal to \(a+c\) (line 4), \(b=c\) (line 5), and \(a>0\) (line 6), 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.
4. OVERVIEW

Figure 1 provides an overview of the steps involved in the MELBA (MachinE Learning based refinement of BIack-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.

4.1 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 that the feasibility of transforming test cases into abstract test cases given pre-defined categories and choices. Though the test specification is assumed in this paper to follow the category-partition [23] (CP) strategy, we could probably tailor our methodology to any black-box strategy that would allow us to abstract test cases in terms of interesting properties. In the context of CP, the SUT is typically decomposed into user functionalities (e.g., use case) which are then independently tested. However, CP can be applied to the test of subsystems or even methods, and as other black-box test techniques, the complexity of testing depends on the behavioral specification of the SUT, not 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 to be the most common situation, the test specification has to be either reverse-engineered or created from high level (likely plain language) specification. Furthermore, the output domain has to be divided into equivalence classes. The Triangle program is so simple that its output is already under the form of equivalence classes: equilateral, isosceles, irregular triangles, and not a triangle. However, even this could be simplified, for example, into two equivalence classes: Triangle, no Triangle. 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 (as illustrated in Section 5) but will characterize the SUT behavior in a more precise way. As the input domain is modeled using CP categories and choices (Section 3.1) 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, instead of raw inputs.

Once an abstract test suite is available, a machine learning algorithm (C4.5 [26]) is used to learn rules that relate pairs (category, choice), modeling input properties, to output equivalence classes (Activity 2): the machine learning algorithm takes abstract test cases as inputs under the form of a text file. An example of such rule was discussed in Section 3.2. 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 described above 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 (see Section 4.2). The improvement 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 the C4.5 learning algorithm. 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 improvement 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.

4.2 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 Category-Partition specification of 11 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 [6] and with a larger problem: the Space program [25], 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.

The definition of categories and choices, on the other hand, requires much thinking as one must identify categories and define choices so that they determine output equivalence classes. This requires an understanding of the system domain but is, on the other hand, what a tester would typically do when trying to re-engineer 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 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 that 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 described in Section 5.1. The tester then has to identify the causes of those problems, a process that we support with a set of guidelines (Section 5.2).

Activities 4 and 5 are not automated at this point, as this relies on the know-how and expertise of the tester. However, as discussed in the next section, 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 some effort but this is an effort that is anyway incurred if one is trying to improve a test suite.

5. METHODOLOGY
As discussed previously, our approach is to identify problems in results produced by C4.5 (Section 5.1) and relating them to potential causes in terms of test suite or CP specification deficiencies (Section 5.2). We illustrate these two steps on our Triangle working example (the CP specification is available in Appendix). We then discuss strategies to augment a test suite in Section 5.3.

5.1 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).

As mentioned previously, all of the above cases can be automatically detected by a dedicated tool. However, as discussed next (Section 5.2), determining the exact cause of the problem can only be facilitated but not entirely automated.

Note that cases 2 and 3 above have been shown to be the main issues when practitioners apply Category-Partition [7]. In [7] the authors report on three empirical studies during which they studied the case with which subjects apply the Category-Partition method. Their observation (conclusion) is that practitioners need guidance since missing categories, missing choices, and ill-defined choices (e.g., non disjoint choices) often occur when applying Category-Partition. 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 also address the improvement of the test suite.

5.2 Linking Problems to Causes
The problems discussed above all have one or more potential causes, as summarized in Figure 3.

Missclassifications in the decision tree (Case 1) can have two potential causes: missing category or choice (Case 1.1), ill-defined choices (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 category 7 when using category partition (Appendix): category 7 tests whether $c$ is strictly positive or not (two choices). This omission results in the rule of Example 1 to show the misclassification of 2 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 choices (C17 and C18) of category 9 (Appendix), which is to compare length $c$ to lengths $a$ and $b$, are incorrectly specified as follows:

\[
\begin{align*}
\text{C17:} & \quad c \not< a+b & \text{(should be $\leq$)} \\
\text{C18:} & \quad c \not>= a +b & \text{(should be $>$)} 
\end{align*}
\]

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: useless category (Case 2.1), missing test cases (Case 2.2), ill-defined...
choices (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.

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 (Appendix). If added when applying CP, this category would not be selected in the decision tree.

 CATEGORY 10 - $c$ compared to $a$
 C19: $c > a$
 C20: $c <= a$

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 category 1 (Appendix), 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 anymore.

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 any branch in the tree (Case 3). This may be the results of several potential causes: impossible combination (Case 3.1), missing test case (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 (Appendix), combination of choices C6 (i.e., $a>b+c$) and C18 (i.e., $c>a+b$) is not possible. Recall (Section 3.1) 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 combination of choices.

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. For instance, if we remove the two test cases that are covered by the following combinations of choices (rule): $a=b$, $b!=c$, and $c>a+b$ with the output equivalence class being “Not_Triangle”, we get the decision tree in Figure 4. The tree no longer shows a rule with the above mentioned combinations of choices due to missing test cases.

The last problem we cover is related to the redundancy of test cases (Case 4). It is in general important to minimize functional

\[
\begin{align*}
(a \text{ vs. } b) &= a=b \\
(b \text{ vs. } c) &= b=c \\
(a) &= a>0: \text{ Equilateral (54.0)} \\
(c) &= c>0: \text{ null (6.0)} \\
(a 	ext{ vs. } b) &= a-b \\
(c \text{ vs. } a+b) &= c<=a+b \\
(a \text{ vs. } b+c) &= a<=b+c \\
(b \text{ vs. } a+c) &= b<=a+c \\
(b \text{ vs. } c) &= b<=c: \text{ Isosceles (22.0)} \\
(b \text{ vs. } c) &= b!=c \\
(c \text{ vs. } a) &= c=a: \text{ Isosceles (22.0)} \\
(c \text{ vs. } a) &= c!=a: \text{ Irregular (60.0)} \\
(b \text{ vs. } a+c) &= b>a+c: \text{ Not_Triangle (20.0)} \\
(a \text{ vs. } b+c) &= a>b+c: \text{ Not_Triangle (20.0)} \\
(c \text{ vs. } a+b) &= c>a+b \\
(a) &= a>0 \\
(b) &= b>0: \text{ Not_Triangle (18.0)} \\
(b) &= b<=0: \text{ null (2.0)} \\
(a) &= a<=0: \text{ null (2.0)}
\end{align*}
\]

Figure 4 Triangle DT with combinations of choices removed

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: too many test cases for a rule (Case 4.1), ill-defined choices (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): Similarly, ill-defined choices can lead to misclassifications but also to the impossibility for C4.5 to split further leaves with large numbers of instances.

Assuming choices C17 and C18 of category 9 of the Triangle CP (See Appendix), which is to compare length $c$ to lengths $a$ and $b$, are incorrectly specified as follows:

C17: $c < a+b$ (should be $<=$)
C18: $c >= a+b$ (should be $>$)

C4.5 returns the tree in Figure 5, 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 $c=a+b$.

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.

5.3 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
Figure 5 Misclassification due to ill-defined choices

(a vs. b) = a-b
| (b vs. c) = b-c
| | (a) = a>0: Equilateral (54.0)
| | (a) = a<0: null (6.0)
| (b vs. c) = b>c
| | (c) = c>0: Isosceles (24.0/2.0)
| | (c) = c<0: null (2.0)
(a vs. b) = a=b
| (c vs. a+b) = c+a+b
| | (a vs. b+c) = a<b+c
| | | (b vs. a+c) = b<+a+c
| | | | | (b vs. c) = b=c: Isosceles (22.0)
| | | | | (b vs. c) = b+c: Isosceles (22.0)
| | | | | (c vs. a) = c=a: Irregular (60.0)
| | | | (c vs. a) = c>a: Isosceles (20.0)
| | | | (b vs. c) = b>a+c: Not_Triangle (20.0)
| | | | (a vs. b+c) = a+b+c: Not_Triangle (20.0)
| | | (c vs. a+b) = c>a+b
| | | (a) = a>0
| | | (b) = b>0: Not_Triangle (18.0)
| | | (b) = b<0: null (2.0)
| | (a) = a<0: null (2.0)

Figure 6 (a). Such a tree excerpt indicates that combining
be relevant to determine the output class and could be missing
choices. Similarly, the tree sugge sts that the combinations of
C4.5 shows categories
providing by the decision tree. Furthermore, this is an
discussed that those properties and selectors were not required for
applying our methodology (Section 3.1). 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 6 (a).

Such a 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 added. 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 6(b).

6. CASE STUDY
In this section, we first describe the system used for the case study
and the application of CP on this program (Section 6.1). We then
present the design of the case study (Section 6.2) and describe the
results of applying C4.5 decision trees to drive the improvement
of the test specifications and test suites (Section 6.3).

6.1 The PackHexChar Program
PackHexChar is a Java adaptation of the sreadhex procedure, used
in the GhostScript program and described in [22], 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
read. If ODD_DIGIT has an illegal
value (strictly below -1 or not a hexadecimal value), the program
returns -3. The user can ask

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. However, as described above, its
behavior is not simple.

---

3 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.

4 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.
We asked an expert, well versed into black-box testing and the use of Category-Partition to use this technique on the PackHexChars program. When applying Category-Partition to the PackHexChars program, the expert identified eleven categories and 23 choices (see Appendix) (referred to as the expert CP specification). This led to the generation of 221 test cases by identifying all compatible choice combinations (referred to as the expert test suite).

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 [17, 18, 23], the number of categories and choices may not be very different from what we have here.

6.2 Design of the Case Study

Our case study took place in the context of a specialized 4th year course on software testing. The students were properly trained regarding white and black-box testing techniques, including Category-Partitioning. Their task, as further described below, was to devise a test specification from the source code using CP. Then, using this specification, they were expected to devise a test suite. Due to time constraints, we did not ask the students to go through the iterative MELBA process themselves. The process was however applied by a Master student, starting from the CP specification and test suites provided by the students.

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 (Section 6.3) and (2) software evolution (Section 6.4). This leads to two distinct situations that require two slightly different types of case studies. In the first situation we have the test suite but no CP specification (e.g., a typical situation for Open Source Software—OSS). Testers must then build the CP specification based on their understanding of the software functional behavior and then iteratively refine it. To emulate this situation we developed our own CP (“Expert” CP) and applied it to three other students’ test suites. These results are reported in section 6.4.

A second distinct situation (Section 6.3) 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). The students had three hours to understand the program and apply the CP methodology. The limited time available to browse through the code explains why we had to work with a small though functionally complex program. After students devised a CP specification and derived a test suite from it, we then generated the abstract test suites using each student’s CP specification and test suite.

We then tried to identify occurrences of problems using the heuristics described in Section 5. For both types of case studies, we actually went through the iterative improvement process illustrated in Figure 1. Both test suites and CP specifications were iteratively improved using the patterns we specified in Section 5.2 for analyzing decision trees and their potential relationships to problems in the test suite or CP specification. 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 [21]) and then computing the mutation scores of test suites to assess their effectiveness. All non-equivalent mutants generated by MuJava were retained for the analysis.

The reason for devising the “expert” CP specification (see Appendix)) described above 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.

6.3 Results with Students’ CPs

6.3.1 Student A’s CP and TS

Student A’s test suite contains 20 test cases. We (automatically) created 20 abstract test cases using A’s CP specification. Executing C4.5 on these abstract test cases, we obtain the decision tree of Figure 7(a). The decision tree shows one misclassified test case (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 6.1), resulting in a missing category in student A’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 three rules with zero instances are unfeasible combinations of choices (Case 3.1).

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 the decision tree of Figure 7 (b). The tree shows one rule with no instance: this is an unfeasible combination of choices (Case 3.1). Using the heuristic described in Section 5.3 for adding combinations, the tree also suggests that there are fifteen combinations of choices potentially missing. Looking at the test suite shows that nine of them are already exercised.

We therefore create six test cases; (automatically) produce the corresponding abstract test cases and re-run C4.5, which returns the decision tree in Figure 7 (c). The tree shows potentially missing choice combinations (0, 0) in bold face: the first rule is an unfeasible combinations of choices; the last two rules are feasible but involve an error condition (rlen<0 and rlen>=Length) which already appears in another rule. The tree also suggests other missing combinations of choices. However, they are already exercised by the test suite, correspond to an error choice which does not need to be combined, or are not relevant (e.g., combining #RLENCHARS=Odd and rlen=0 does not make sense: we compact 0 characters in the string and therefore the number of hexadecimal characters in the string does not matter). The tree shows three rules with a number of instances larger than one (2, 3, 7 and 4 instances), possibly suggesting redundant test cases. We removed some of the test cases in those rules.
In terms of mutation scores, the test suites of the three iterations found 225, 226, and 222 faults, respectively. The sizes of the test suites were respectively 20, 26, and 11 test cases. Augmenting the test suite in the second iteration seems rather effective: Eight additional test cases killed seven additional mutants. However, our heuristic for removing redundant test cases leads to four mutants remaining undetected, though the reduction in size is relatively substantial. Future work will investigate refinements of our test suite reduction heuristic.

6.3.2 Student B’s CP and TS

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 8(a). 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 6.1), 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 input string

**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 the decision tree of Figure 8(b). The tree shows one rule with no instance: this is an unfeasible combination of choices (Case 3.1). Using the heuristic described in Section 5.3 for adding combinations, the tree also suggests that there are eight combinations of choices 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 the decision tree in Figure 8(c). The tree shows potentially missing choice combinations ((0,0) in bold face): the first two rules are unfeasible combinations of choices; the last rule is feasible but involves an error condition (rlen>sLength) which already appears in another rule. The tree also suggests other missing combinations of choices. However, they are already exercised by the test suite, correspond to an error choice which does not need to be combined, or are not relevant (e.g., combining rlen<0, which does not make sense: we compact the first rlen characters of input string s

```
RLEN = rlen-[1..s.length]  
| ODD_DIGIT = odd_digit=1  
| | #RLENCHARS = Even: -1.0 (3.0)  
| | | #RLENCHARS = Odd: S[rlen-1] (7.0)  
| | | #RLENCHARS = Zero: S[rlen-1] (0.0)  
| | #RLEN = rlen>0: -3.0 (1.0)  
| | | SLENGTH = EvenLength: S[rlen-1] (1.0)  
| | | #RLEN = rlen<0: -1.0 (0.0)  
| | | | SLENGTH = Empty: -1.0  
| | | | SLENGTH = OddLength: S[rlen-1] (0.0)  
| | | | SLENGTH = EvenLength: -1.0 (2.0)  
| | #RLEN = rlen=0: -1.0 (0.0)  
| RLEN = rlen<0: -2.0 (1.0)  
| RLEN = rlen>0: -2.0 (1.0)  
```

(c) Iteration 3  
Figure 7 A’s TS + A’s CP  

(randomly selected), keeping one test case for each one of them. We finally obtain the same tree as the one in Figure 7 (c), except that each feasible rule which had a large number of instances finally contains one instance.

In terms of mutation scores, the test suites of the three iterations found 225, 226, and 222 faults, respectively. The sizes of the test suites were respectively 20, 26, and 11 test cases. Augmenting the test suite in the second iteration seems rather effective: six additional test cases kill seven additional mutants. However, our heuristic for removing redundant test cases leads to four mutants remaining undetected, though the reduction in size is relatively substantial. Future work will investigate refinements of our test suite reduction heuristic.
Figure 9 (b). The tree shows two rules with no instances: these are impossible combinations of choices (Case 3.1). The decision tree also shows a missing choice (#RLENCHARS=Even) and is due to missing test cases (Case 2.2). Using the heuristic described in Section 5.3 for adding combinations, the tree also suggests that there are 4 combinations of choices potentially missing. Looking at the test suite shows that only two of them are already exercised.

We therefore create two test cases; (automatically) produce the corresponding abstract test cases and re-run C4.5, which returns the decision tree in Figure 9 (c). The tree suggests no redundancy but shows three rules with no instances ((0.0) in bold face): these are combinations of choices that are possible but involve an error condition (odd_digit=INVALID) which already appears in another rule.

---

6.3.3 Student C’s CP and TS

Student C’s test suite contains 19 test cases. Using C’s CP specifications, 19 abstract test cases are created. After having executed C4.5, the decision tree in Figure 9 (a) shows two misclassified test cases (Case 1). This again is due to the student failing to recognize that the program compacts the first RLEN hexadecimal characters in the input string, resulting in a missing category in student C’s CP (Case 1.1). Some combinations of choices are also missing in the decision tree (Case 3), including two impossible combinations of choices (Case 3.1).

We first add the missing category to the student’s CP (the same category as for student B), leading to a new augmented tree in Figure 8 B’s TS + B’s CP.
We therefore obtain a test suite of 21 test cases. No occurrence of the problems discussed in Section 5.1 can be found and the iterative process of Figure 1 stops. The decision tree shows potential redundant test cases. We randomly removed some of those test cases, leading to a test suite of 10 test cases.

In terms of mutation scores, the test suites of the three iterations found 185, 200, and 200 faults, respectively. The sizes of the test suites were respectively 19, 21, and 10 test cases. Augmenting the test suite in the second iteration seems very effective: Three additional test cases kill 15 additional mutants. This time our heuristic for removing redundant test cases leads to a substantial reduction in size without loss in terms of fault detection.

6.4 Results with Experts’ CP

6.4.1 Student X’s TS and Expert CP

Student X’s test suite contains 15 test cases. We (automatically) created 15 abstract test cases using Expert CP specification. Executing C4.5 on these abstract test cases, we obtain the decision tree of Figure 10 (a). The tree shows no misclassified instances (test cases). This is expected as we use the Expert CP: a complete set of categories and well-defined choices should result into correct classifications. However, the tree shows a number of issues: (1) not all the categories of the Expert CP appear in the tree (Case 2), specifically, categories 5, 7, 8, 9, 10, and 11 are missing; (2) some choices (in the remaining, used categories) are missing (Case 2), specifically choices C3, C7, C8, C9, and C14; (3) Missing choices and combinations of choices are due to missing test cases (Case 2.2): Using the heuristic described in section 5.3 for adding combinations, the tree suggests 16 combinations of choices are potentially missing. Looking at the test suite shows that two of them are already exercised.

This initial use of C4.5 therefore allows us to improve the test suite. Fourteen new test cases are transformed into abstract test cases, and C4.5 is re-executed with a total of 29 abstract test cases (15 initial ones and 14 new ones). The result is the decision tree of Figure 10 (b).

The tree in Figure 10 (b) shows potentially missing choice combinations (0.0 in bold face): rules labeled (b) are possible and cover no test case, rules labeled (a) are combinations of choices that are possible but involve an error condition which already appears in another rule. The tree also suggests 19 missing combinations of choices. However, looking at the test suite shows that 8 of these combinations have already been exercised. We therefore create eleven new test cases; (automatically) produce the corresponding abstract test cases and re-run C4.5, which returns a new decision tree in Figure 10 (c).

The tree shows rules with the number of instances larger than one, 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 the rules. We obtain the same tree as the one in Figure 10(c), except that the rules which had a large number of instances finally contain one instance.

In terms of mutation scores, the test suites of the four iterations found 174, 224,227, and 225 faults, respectively. The sizes of the test suites were respectively 15, 29, 40, and 20 test cases. Augmenting the test suite in the second and third iteration seems very effective: 14 additional test cases kill 15 additional mutants, and 11 additional test cases kill 3 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.

### 6.4.2 Student Y’s TS and Expert CP

Student Y’s test suite contains 20 test cases. We (automatically) created 20 abstract test cases using Expert CP specification. Executing C4.5 on these abstract test cases, we obtain the decision tree of Figure 11(a). The tree shows no misclassified instances (test cases). This is expected as previously discussed. However, the tree shows a number of issues: (1) not all the categories of the Expert CP appear in the tree (Case 2), specifically, categories 5, 6, 7, 8, 10, and 11 are missing; (2) some choices (in the remaining, used categories) are missing; (3) Missing choices and combinations of choices are due to missing test cases (Case 2.2): Using the heuristic described in section 5.3, the tree suggests 15 combinations of choices are potentially missing. Looking at the test suite shows that three of them are already exercised.

This initial use of C4.5 therefore allows us to improve the test suite. 12 new test cases are transformed into abstract test cases, and C4.5 is re-executed with a total of 32 abstract test cases (20 initial ones and 12 new ones). The result is the decision tree of Figure 11 (b).

The tree in Figure 11 (b) shows potentially missing choice combinations ((0.0) in bold face): rule labeled (b) is possible and covers no test case, rules labeled (a) are combinations of choices that are possible but involve an error condition which already appears in another rule. The tree also suggests 1 missing combinations of choices that has not been exercised. We therefore create one new test case; (automatically) produce the corresponding abstract test cases and re-run C4.5, which returns a new decision tree in Figure 11 (c).

The tree shows rules with the number of instances larger than one, possibly suggesting redundant test cases. We removed some of the test cases in those rules (randomly selected), keeping one test case for each of the rules. We obtain the same tree as the one in Figure 11 (c), except that the rules which had a large number of instances finally contain one instance.

In terms of mutation scores, the test suites of the four iterations found 225, 227,227, and 226 faults, respectively. The sizes of the test suites were respectively 20, 32, 33, and 20 test cases. Augmenting the test suite in the second iteration seems effective: 12 additional test cases kill 2 additional mutants. However, our heuristic for removing redundant test cases leads to one mutant remaining undetected, though the reduction in size is relatively substantial. Future work will investigate refinements of our test suite reduction heuristic.

### 6.4.3 Student Z’s TS and Expert CP

Student Z’s test suite contains 11 test cases. We (automatically) created 11 abstract test cases using Expert CP specification. Executing C4.5 on these abstract test cases, we obtain the decision tree of Figure 12(a). Similar to X’s DT, the decision tree shows no misclassification. However, the tree shows a number of issues: (1) not all the categories of the Expert CP appear in the tree (Case 2), specifically, categories 5, 6, 7, 9, 10, and 11 are missing; (2) some choices (in the remaining, used categories) are missing (Case 2), specifically choices C7; (3) Missing choices and combinations of choices are due to missing test cases (Case

### Table: Decision Tree Rules

| RLEN = rlen<0: -2.0 (1.0) | RLEN = rlen>sLength: -2.0 (1.0) |
| RLEN = rlen=0: -1.0 (4.0) | RLEN = rlen>Length: -2.0 (1.0) |

### (a) Iteration 1

| RLEN = rlen<1: -1.0 (3.0) | RLEN = rlen=1: -1.0 (3.0) |
| RLEN = rlen=0: -1.0 (3.0) | RLEN = rlen>sLength: -2.0 (1.0) |

### (b) Iteration 2

| NUM HEX CHARS IN [0...RLEN] = Even | NUM HEX CHARS IN [0...RLEN] = Odd |
| RLEN = rlen<1: -1.0 (4.0) | RLEN = rlen=sLength: -2.0 (1.0) |

### (c) Iteration 3

**Figure 11 Y’s TS + Y’s CP**
2.2): Using the heuristic described in section 5.3, the tree suggests 36 combinations of choices are potentially missing. Looking at the test suite shows that four of them are already exercised.

This initial use of C4.5 therefore allows us to improve the test suite. 32 new test cases are transformed into abstract test cases, and C4.5 is re-executed with a total of 43 abstract test cases (11 initial ones and 32 new ones). The result is the decision tree of Figure 12 (b).

The tree in Figure 12 (b) shows potentially missing choice combinations (0.0) in bold face: these are combinations of choices that are possible but involve an error condition which already appears in another rule. The tree also suggests five missing combinations of choices that have not been exercised. We therefore create five new test cases; (automatically) produce the corresponding abstract test cases and re-run C4.5, which returns a new decision tree in Figure 12 (c).

The tree shows rules with the number of instances larger than one, 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 the rules. We obtain the same tree as the one in Figure 12 (c), except that the rules which had a large number of instances finally contain one instance.

The tree in Figure 10(c) shows eight rules (0.0) in bold face): these are combinations of choices that are possible but involve an error condition which already appears in another rule. No occurrence of the problems discussed in section 5.1 can be found and the iterative process of Figure 1 stops.

In terms of mutation scores, the test suites of the four iterations found 226, 230, 230, and 230 faults, respectively. The sizes of the test suites were respectively 11, 43, 48, and 21 test cases. Augmenting the test suite in the second iteration is somewhat effective: 32 additional test cases kill 4 additional mutants. This time our heuristic for removing redundant test cases leads to a substantial reduction in size without loss in terms of fault detection.

6.5 Discussion

The previous section illustrated the MELBA iterative process in two application context: when CP specifications are used to generate an initial, possibly incomplete test suite, and when the test suite is available but no CP specification. 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 root causes (Sections 5.1 and 5.2) are complete with respect to the PackHexChar program5, though future work will need to investigate further whether those

---

5 Using the students’ test suites and CP specifications used in this paper, we were able to illustrate the four problems of Section 5.1 and five of the root causes discussed in Section 5.2.
taxonomies need to be extended. Furthermore, we have seen 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.

By analyzing the size and mutation scores associated with the test suites, we can conclude that with a reasonable increase in test cases: (6, 8 and 3) for students A, B and C and (25, 13, and 37) for students X, Y, and Z respectively, we found a significant number of additional faults: (1, 7 and 15) for students A, B, and C, and (53, 2, and 4) for students X, Y, and Z. 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.

7. CONCLUSION
This paper proposed the MELBA 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 so as to be able to iteratively improve them.

The MELBA methodology takes as inputs the test suite and test specifications 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 the iterative process on programs of varying sizes and complexities, as well as user-friendly automated tool support.

8. REFERENCES
APPENDIX

The Triangle program parameters: lengths a, b, and c.

<table>
<thead>
<tr>
<th>Parameter a</th>
<th>Category 1—values for a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Choice C1: a &gt; 0</td>
</tr>
<tr>
<td></td>
<td>Choice C2: a &lt;= 0</td>
</tr>
<tr>
<td>Category 2—a compared to b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Choice C3: a = b</td>
</tr>
<tr>
<td></td>
<td>Choice C4: a /= b</td>
</tr>
<tr>
<td>Category 3—a compared to b and c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Choice C5: a &lt;= b + c</td>
</tr>
<tr>
<td></td>
<td>Choice C6: a &gt; b + c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter b</th>
<th>Category 4—values for b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Choice C7: b &gt; 0</td>
</tr>
<tr>
<td></td>
<td>Choice C8: b &lt;= 0</td>
</tr>
<tr>
<td>Category 5—b compared to c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Choice C9: b = c</td>
</tr>
<tr>
<td></td>
<td>Choice C10: b /= c</td>
</tr>
<tr>
<td>Category 6—b compared to a and c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Choice C11: b &lt;= a + c</td>
</tr>
<tr>
<td></td>
<td>Choice C12: b &gt; a + c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter c</th>
<th>Category 7—values for c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Choice C13: c &gt; 0</td>
</tr>
<tr>
<td></td>
<td>Choice C14: c &lt;= 0</td>
</tr>
<tr>
<td>Category 8—a compared to c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Choice C15: c = a</td>
</tr>
<tr>
<td></td>
<td>Choice C16: c /= a</td>
</tr>
<tr>
<td>Category 9—a compared to a and b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Choice C17: c &lt;= a + b</td>
</tr>
<tr>
<td></td>
<td>Choice C18: c &gt; a + b</td>
</tr>
</tbody>
</table>

The following is the PackHexChar Category-Partition Specification.

| Parameter RLEN | CATEGORY 1—Valid values for rlen |
|               | C1: rlen = 0                     |
|               | C2: rlen = [1…sLength]           |
|                | CATEGOR 2—Invalid values for rlen |
|               | C3: rlen < 0                     |
|               | C4: rlen > sLength               |

| Parameter ODD_DIGIT | CATEGORY 3—Valid values for odd_digit |
|                     | C5: odd_digit = -1                |
|                     | C6: odd_digit = [0…9]             |
|                     | C7: odd_digit = [A…F]             |
|                     | C8: odd_digit = [a…f]             |

| Parameter String S | CATEGORY 6—Type of characters in the first rlen characters of S |
|                   | C13: AllHexadecimal               |
|                   | C14: AllNonHexadecimal            |
|                   | C15: MixedChars                   |

| CATEGORY 8—The first rlen characters of the string contains boundary Value [0, 9, a, f, A, F] |
| C20: Contains [0] |
| C21: Contains [9] |
| C22: Contains [a] |
| C23: Contains [f] |
| C24: Contains [A] |
| C25: Contains [F] |
| C26: ContainsMixed |
| C27: ContainsNone |

| CATEGORY 9—Number of hexadecimal characters in the first rlen characters of S |
| C28: Odd |
| C29: Even |
| C30: Zero |

| CATEGORY 10—Position of the first non-hexadecimal character in the first rlen characters of S |
| C31: First |
| C32: Middle |
| C33: Last |

| CATEGORY 11—Consecutive non-hexadecimal characters in the first rlen characters of S |
| C34: Yes |
| C35: No |

| CATEGORY 7—Case of chars in the first rlen chars of the string S |
| C16: allNumbers |
| C17: allLowerCase |
| C18: allUpperCase |
| C19: MixedCase |

15