

Using machine learning to refine Category-Partition test specifications and test suites[☆]

Lionel C. Briand^a, Yvan Labiche^{b,*}, Zaheer Bawar^b, Nadia Traldi Spido^b

^a Simula Research Laboratory and University of Oslo, P.O. Box 134, Lysaker, Norway

^b Carleton University, Squal Lab., 1125 Colonel By Drive, Ottawa, ON, Canada K1S5B6

ARTICLE INFO

Article history:

Available online 5 July 2009

Keywords:

Black box testing
Category-Partition
Machine learning
Test improvement

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, Category-Partition testing, we propose a methodology and a tool based on machine learning that has shown promising results on a case study involving students as testers.

© 2009 Elsevier B.V. All rights reserved.

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 [29]). For instance, open source software development projects have been shown to lack “attention to basic, accepted, and mature testing techniques [37].” 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%) [37]. Developers hence have a decision to make: either build new test suites using a black-box testing technique such as Category-Partition [30] or reuse existing ones; and they often go for the latter as it (hopefully) reduces effort and therefore costs (e.g., the tester can reuse test harness, including drivers, oracles ...). 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 [14]. 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 [30]. 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.

[☆] A preliminary version of this paper was presented at the 8th International Conference on Quality Software (QSIC 2008) [8].

* Corresponding author.

E-mail addresses: briand@simula.no (L.C. Briand), labiche@sce.carleton.ca (Y. Labiche), zbawar3@scs.carleton.ca (Z. Bawar), nspido@scs.carleton.ca (N.T. Spido).

The rest of the paper is structured as follows. Section 2 provides some background. Our approach is described in Sections 3 and 4. A case study is discussed in Section 5 and key aspects of our proposed tool support are described in Section 6. Related work is described in Section 7. Conclusions are drawn in Section 8.

2. Background

2.1. Using Category Partition

To illustrate how the Category Partition (CP) [30] black-box testing method works, let us take the well-known and simple Triangle program example [24], 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 (less than or equal to zero). CP requires that we identify properties of the triangle sides that will affect the execution behavior of the program 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 distinct “choices”. For example, taking the “ a compares to b and c ” category, choices could correspond to the two mutually exclusive situations where $a \leq b+c$ 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 [30]. The complete application of CP to the Triangle program is available in [7].

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 each test case into an “abstract” test case. (There are as many abstract test cases as 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 ($a \leq b+c, b=c, \text{isosceles}$): 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 ($a \leq b+c, b=c, a >=0, b >=0, c >=0, \text{isosceles}$), 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* (string contains only numbers) and *C2* (string contains only letters); and a category *Cat2* for strings containing letters with two choices: *C3* (string contains capital letters) and *C4* (string does not contain capital letters). 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 specification of the software under test. This is a necessary exercise, as discussed above, both in the 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 [36] and these 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 relationships between input and environment properties (categories and choices), and output equivalence classes.

A specific category of machine learning techniques generates classification rules [36] which are easily amenable to interpretation: e.g., the C4.5 decision tree algorithm [33,36] (where the paths from the root node of the tree to any leaf can be considered a rule), the Ripper rule induction algorithm [12]. 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

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.

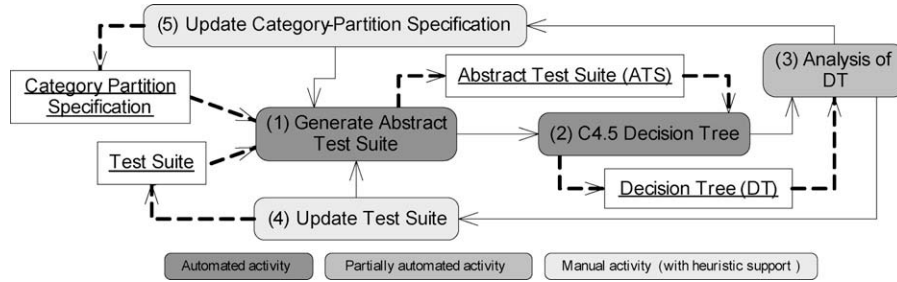


Fig. 1. The MELBA methodology.

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 [36] 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:

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

3. Overview

Fig. 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 assumptions 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.² Though the test specification, used to characterize existing test cases, is assumed in this paper to follow the Category-Partition [30] (CP) strategy, future work will investigate how our methodology could be tailored to other black-box strategies. It is worth mentioning that, as with 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 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 Fig. 1). Recall that each test case in the test suite is transformed into an abstract test case, and that therefore the abstract test suite has the same number of elements (size) as the test suite. 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

```
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 “ $a \neq b$ ”—line 1), c is less than or equal to $a+b$ (category “ c vs. $a+b$ ” and choice “ $c \leq a+b$ ”), a is less than or equal to $b+c$ (line 3), b is less than 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 six 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.

² For automation purposes we simply assume a specific format for reading test case descriptions (see Section 6).

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 Fig. 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 obtaining 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 can be automated. 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 [9] and with a larger problem: the Space program [32]; 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 Fig. 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 such as the classification-tree method [18]. 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) and by providing adequate information in our MELBA tool (Section 6).

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 and 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. Note however that this effort (e.g., building the test harness that include drivers and oracles) will only involve the newly created test cases: the tester will be able to reuse drivers and oracles for existing test cases.

In other words, we provide partially automated support for test suite specification reconstruction and test suite improvement, activities that are usually entirely manual [26]. Though some level of manual effort is un-avoidable (e.g., discovering categories and choices, creating new test cases), we provide help, under the form of heuristics and tool support, 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. For instance, in our case study, the iterative process stopped after three iterations.

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 Fig. 1, on our Triangle working example (the CP specification is available in [7]).

4.1. Identifying problems in C4.5 trees

In our context, we can identify a number of *potential* problems when analyzing C4.5 decision tree:

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 as discussed in Section 6. However, as discussed next (Section 4.2), determining the exact cause of the problem can only be facilitated by providing relevant information but not entirely automated.

Cases 2 and 3 have been shown to be the main issues when practitioners apply Category-Partition [10]. The authors suggest that practitioners follow a checklist to systematically identify these

problems. We provide automated support and a set of heuristics that 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 Fig. 2. These problems are illustrated on the simple Triangle program.

Misclassifications in the decision tree (Case 1) can have two potential causes: Cases 1.1 and 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 Fig. 3, 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 (Fig. 3), showing two misclassified instances. Because the relational operators were changed (by moving the “=” operator from one to the other), 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 specification, 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: Cases 2.1, 2.2 and 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 coarse 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 [7]. 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 $a \leq 0$ 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.

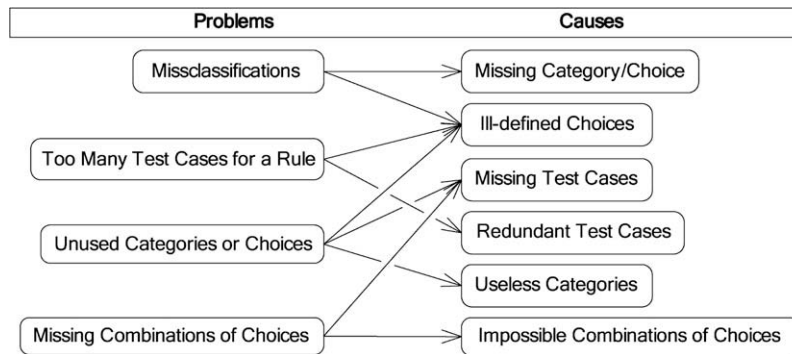


Fig. 2. Problems and potential causes.

<p>Example 1:</p> <pre>(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)</pre>	<p>Example 2:</p> <pre>(a vs. b) = a=b (c) = c>0 (b vs. c) = b!=c: Isosceles (24.0/2.0)</pre>
--	---

Fig. 3. Examples of detected problems using the triangle program.

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 result of several potential causes: Cases 3.1 and 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 combination.

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 equivalence classes.

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

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: Cases 4.1 and 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.

Assuming choices C17 and C18 of category 9 of the Triangle CP [7], which is to compare length c to lengths a and b , are incorrectly specified as follows:

C17: $c < a + b$ (should be \leq)
C18: $c \geq a + b$ (should be $>$)

C4.5 returns the tree in Fig. 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.

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

```
(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 (22.0)
| | (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
| | | | | (c vs. a) = c=a: Isosceles (22.0)
| | | | | (c vs. a) = c!=a: Irregular (60.0)
| | | (b vs. a+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)
```

Fig. 4. Triangle DT with combinations of choices removed.

```
(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
| | | | | (c vs. a) = c=a: Isosceles (22.0)
| | | | | (c vs. a) = c!=a: Irregular (60.0)
| | | (b vs. a+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)
```

Fig. 5. Misclassification due to Ill-defined choices.

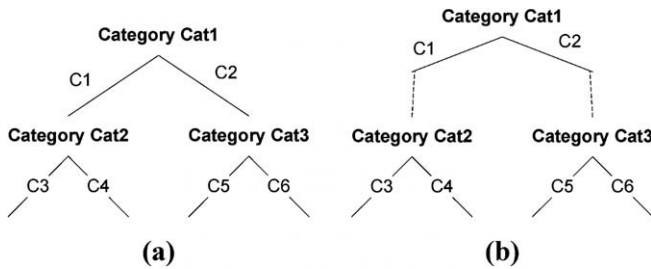


Fig. 6. Adding test cases from the tree.

the category.³ 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 Fig. 6a. 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 added.⁴ 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): Fig. 6b.

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 this 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 (Sections 5.3 and 5.4). We discuss the main observations in Section 5.5.

5.1. The PackHexChar program

PackHexChar is a Java adaptation of the sreadhex procedure, used in the GhostScript program and described in [27], to manipu-

late 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 3 h.

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 a much larger system, for instance by testing entire use cases [21,22,30], the number of categories and choices may not be very different from what we present below.

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. In the first situation we have the test suite but no CP specification (e.g., a typical situation for Open Source Software). Testers must then build the CP specification based on their understanding of the software functional behavior and then iteratively refine it and the test suite. The second distinct situation 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. Twenty one 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. Instead we used the students CP specifications and test suites to emulate the two application contexts discussed above. To emulate the Evolution Context, the MELBA process was applied by a Masters student, starting from the CP

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.

specifications and test suites provided by the students. Results are reported in Section 5.3 for one representative student's CP specification and test suite. To emulate the Open Source Context, we developed our own ("Expert") CP specification (see below) and a Masters student applied it to students' test suites. These results are reported in Section 5.4 for one representative student's 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 [25]) and then computing the mutation scores of test suites to assess their effectiveness. All non-equivalent mutants (see below) generated by MuJava were retained for 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) [7]. 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 three-fold. First it is intended to be a reference for assessing the student's CP specifications and understanding 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 [3], these live mutants were considered to be equivalent though for some of them this is probably not the case. By following this procedure we can ensure most mutants used for the case study are not equivalent. Third, the expert CP specification was required for the Open Source Context.

5.3. Evolution context—representative results

Though, due to size constraints, only the results of one student could be reported here, this student (labeled student B for convenience) was selected as representative of the cases that we have investigated [7]. 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 obtained the decision tree of Fig. 7a. The decision tree shows eight misclassified test cases (Case 1): number of instances underlined. 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 test suite (Case 3). Some of them are actually identified in the tree: they have a number of instances equal to 0 (number of instances in bold face). 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 added the missing category to the student's CP (regarding the first `rLen` number of characters to compact). Once the abstract test cases are (automatically) re-created from the updated CP, the execution of C4.5 produces a new decision tree: Fig. 7b. 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 created eight test cases, (automatically) produced the corresponding abstract test cases and re-ran C4.5, which re-

<pre> 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) </pre>	<pre> ODD_DIGIT = odd_digit=-1 #RLENCHARS = Zero: -1.0 (8.0) #RLENCHARS = Odd: S[rlen-1] (9.0) #RLENCHARS = Even RLEN = rlen=0: -1.0 (0.0) RLEN = 0<rlen<=sLength: -1.0 (7.0) RLEN = rlen>sLength: -2.0 (1.0) RLEN = rlen<0: -1.0 (0.0) ODD_DIGIT = odd_digit=[0-15] #RLENCHARS = Zero RLEN = rlen=0: odd_digit (3.0) RLEN = 0<rlen<=sLength: odd_digit (2.0) RLEN = rlen>sLength: odd_digit (0.0) RLEN = rlen<0: -2.0 (1.0) #RLENCHARS = Odd: -1.0 (3.0) #RLENCHARS = Even: S[rlen-1] (2.0) ODD_DIGIT = odd_digit>15: -3.0 (2.0) ODD_DIGIT = odd_digit<-1: -3.0 (1.0) </pre>
(a) First iteration	(c) Third iteration
<pre> ODD_DIGIT = odd_digit=-1 #RLENCHARS = Zero: -1.0 (6.0) #RLENCHARS = Odd: S[rlen-1] (8.0) #RLENCHARS = Even RLEN = rlen=0: -1.0 (0.0) RLEN = 0<rlen<=sLength: -1.0 (7.0) RLEN = rlen>sLength: -2.0 (1.0) ODD_DIGIT = odd_digit=[0-15] #RLENCHARS = Zero: odd_digit (3.0) #RLENCHARS = Odd: -1.0 (2.0) #RLENCHARS = Even: S[rlen-1] (1.0) ODD_DIGIT = odd_digit>15: -3.0 (2.0) ODD_DIGIT = odd_digit<-1: -3.0 (1.0) </pre>	
(b) Second iteration	

Fig. 7. Decision trees for student B's TS and CP.

turned a third decision tree: Fig. 7c. The tree shows potentially missing choice combinations (bold face): the first two rules are unfeasible combinations of choices; the last rule is feasible but involves an error condition ($r_{len} > s_{length}$) which already appears in another rule. According to our heuristic for adding test cases, the tree 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 $\#RLENCHARS = \text{Odd}$ and $r_{len} = 0$ 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 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-ran C4.5 and obtained 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. Future work will investigate refinements of our test suite reduction heuristic.

5.4. Open source context—representative results

Though, due to size constraints, only the results of one student could be reported here, this student (labeled student X for convenience) was selected as representative of the cases that we have investigated [7]. Student X's test suite contains 15 test cases. We (automatically) created 15 abstract test cases using the Expert CP specification. Executing C4.5 on these abstract test cases, we obtained the decision tree of Fig. 8a. The tree shows no misclassified test cases. This is to be expected as we used 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) six (out of nine) categories of the Expert CP do not appear in the tree (Case 2); (2) five choices (out of 13) of the remaining, used categories are missing (Case 2); (3) Missing choices and combinations of choices are due to missing test cases (Case 2.2): Using the heuristic described in Section 4.3 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.

We therefore created 14 new test cases and (automatically) transformed them into abstract test cases, and re-executed C4.5 with a total of 29 abstract test cases (15 initial ones and 14 new ones), resulting in a new decision tree: Fig. 8b. The tree shows potentially missing choice combinations (zero instances in a leaf): Some of them are feasible and cover no test case (labeled (a) in

<pre> RLEN = rlen>sLength: -2.0 (8.0) RLEN = rlen=[1...sLength] SCHARTYPE IN [0...RLEN] = AllHexadecimal ODD_DIGIT = odd_digit=-1: -1.0 (3.0) ODD_DIGIT = odd_digit>15: -3.0 (1.0) ODD_DIGIT = odd_digit=[0...9]: -1.0 (1.0) SCHARTYPE IN [0...RLEN] = N/A: -1.0 (0.0) SCHARTYPE IN [0...RLEN] = MixedChars:S[rLen-1] (1.0) RLEN = rlen=0: odd_digit (1.0) </pre>	<pre> RLEN = rlen>sLength: -2.0 (8.0) RLEN = rlen=[1...sLength] HEX CHARS IN [0...RLEN] = Even ODD_DIGIT = odd_digit=-1: -1.0 (3.0) ODD_DIGIT = odd_digit>15: S[rLen-1] (0.0) (a) ODD_DIGIT = odd_digit=[0...9]: S[rLen-1] (1.0) ODD_DIGIT = odd_digit<-1: S[rLen-1] (0.0) (a) ODD_DIGIT = odd_digit=[a...f]: S[rLen-1] (2.0) ODD_DIGIT = odd_digit=[A...F]: S[rLen-1] (1.0) HEX CHARS IN [0...RLEN] = Odd NHC POSITION IN [0...RLEN] = N/A ODD_DIGIT = odd_digit=-1: -3.0 (0.0) (b) ODD_DIGIT = odd_digit>15: -3.0 (1.0) ODD_DIGIT = odd_digit=[0...9]: -1.0 (1.0) ODD_DIGIT = odd_digit<-1: -3.0 (1.0) ODD_DIGIT = odd_digit=[a...f]: -3.0 (0.0) (b) ODD_DIGIT = odd_digit=[A...F]: -3.0 (0.0) (b) NHC POSITION [0...RLEN] = First: S[rLen-1] (1.0) NHC POSITION [0...RLEN] = Middle: -1.0 (1.0) HEX CHARS IN [0...RLEN] = Zero ODD_DIGIT = odd_digit=-1: -1.0 (1.0) ODD_DIGIT = odd_digit>15: odd_digit (0.0) (a) ODD_DIGIT = odd_digit=[0...9]: odd_digit (1.0) ODD_DIGIT = odd_digit<-1: odd_digit (0.0) (a) ODD_DIGIT = odd_digit=[a...f]: odd_digit (1.0) ODD_DIGIT = odd_digit=[A...F]: odd_digit (1.0) RLEN = rlen=0 ODD_DIGIT = odd_digit=-1: -1.0 (1.0) ODD_DIGIT = odd_digit>15: odd_digit (0.0) (a) ODD_DIGIT = odd_digit=[0...9]: odd_digit (2.0) ODD_DIGIT = odd_digit<-1: odd_digit (0.0) (a) ODD_DIGIT = odd_digit=[a...f]: odd_digit (1.0) ODD_DIGIT = odd_digit=[A...F]: odd_digit (1.0) RLEN = rlen<0: -2.0 (1.0) </pre>
(a) First iteration	(b) Second iteration
<pre> RLEN = rlen>sLength: -2.0 (8.0) RLEN = rlen=[1...sLength] HEX CHARS IN [0...RLEN] = Even ODD_DIGIT = odd_digit=-1: -1.0 (3.0) ODD_DIGIT = odd_digit>15: S[rLen-1] (0.0) ODD_DIGIT = odd_digit=[0...9]: S[rLen-1] (2.0) ODD_DIGIT = odd_digit<-1: S[rLen-1] (0.0) ODD_DIGIT = odd_digit=[a...f]: S[rLen-1] (2.0) ODD_DIGIT = odd_digit=[A...F]: S[rLen-1] (1.0) HEX CHARS IN [0...RLEN] = Odd ODD_DIGIT = odd_digit=-1: S[rLen-1] (3.0) ODD_DIGIT = odd_digit>15: -3.0 (1.0) ODD_DIGIT = odd_digit=[0...9]: -1.0 (4.0) ODD_DIGIT = odd_digit<-1: -3.0 (1.0) ODD_DIGIT = odd_digit=[a...f]: -1.0 (3.0) ODD_DIGIT = odd_digit=[A...F]: -1.0 (3.0) HEX CHARS IN [0...RLEN] = Zero ODD_DIGIT = odd_digit=-1: -1.0 (1.0) ODD_DIGIT = odd_digit>15: odd_digit (0.0) ODD_DIGIT = odd_digit=[0...9]: odd_digit (1.0) ODD_DIGIT = odd_digit<-1: odd_digit (0.0) ODD_DIGIT = odd_digit=[a...f]: odd_digit (1.0) ODD_DIGIT = odd_digit=[A...F]: odd_digit (1.0) RLEN = rlen=0 ODD_DIGIT = odd_digit=-1: -1.0 (1.0) ODD_DIGIT = odd_digit>15: odd_digit (0.0) ODD_DIGIT = odd_digit=[0...9]: odd_digit (1.0) ODD_DIGIT = odd_digit<-1: odd_digit (0.0) ODD_DIGIT = odd_digit=[a...f]: odd_digit (1.0) ODD_DIGIT = odd_digit=[A...F]: odd_digit (1.0) RLEN = rlen<0: -2.0 (1.0) </pre>	
(c) Third iteration	

Fig. 8. Decision trees for student X's TS and the expert CP.

Fig. 8b) others are already captured by some rules' instances (labeled (b) in Fig. 8b). The tree also suggests 19 missing combinations of choices. However, when looking at the test suite we found that eight of them had already been exercised.

We therefore created eleven new test cases; (automatically) produced the corresponding abstract test cases and re-ran C4.5, which returned the decision tree of Fig. 8c. 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 obtained the same tree, 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 iterations seems very effective: 14 additional test cases kill 15 additional mutants, and 11 additional test cases kill three additional mutants. However, our heuristic for removing redundant test cases leads to two mutants remaining undetected. Future work will investigate refinements of our test suite reduction heuristic.

5.5. Discussion

The previous section illustrated the MELBA iterative process in two application contexts: when CP specifications are used to generate an initial, possibly incomplete test suite, and when the test suite is available but there is no CP specification to start with. 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 problems 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 are complete with respect to the PackHexChar program (seven of the nine problems discussed in Section 4.1 were observed). 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 [7].

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 compacts 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 exer-

cised, 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 (eight for student B, 25 for student X), we found a significant number of additional faults (7 for student B, 53 for student X). 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.

6. Tool support

This section describes the tool that was designed to support the MELBA methodology. We first provide a functional overview and then define all the concepts manipulated by the tool through conceptual models depicted as class diagrams.

6.1. Functional description

The MELBA process is a semi-automated process involving significant human input and interaction. It is therefore important to provide comprehensive tool support supporting decision-making in terms of assessing the potential problems related to test specifications or test suites. Fig. 9 provides a Use Case diagram for the MELBA tool that shows four high-level use cases. The first one (UC1) is about loading the test suite in a TTCN-3 format⁵ [35] and then semi-automatically specifying the details of the category-partition (CP) specification (categories and choices) and output equivalence classes. UC2 then builds the abstract test suite, generates the tree, and analyses the tree to identify potential problems as described above. The last high-level use cases (UC3 and UC4) allow the tester to modify the CP specification and test suite, depending on the results of the tree analysis.

UC1 can be further decomposed into loading the test suite (UC1.1), defining the input parameters and environment variables⁶ specifications (UC1.2), and defining the output variables' specifications (UC1.3). In UC1.2, the tool automatically identifies parameters and their types from the TTCN-3 test suite. The tester then has to define categories and choices. For each parameter, choices are defined using either operations defined for the parameter type or Java expressions.⁷ One of the choices is selected as a base choice [1] (i.e., a nominal choice), which may be useful if a base choice coverage strategy is used in the context of CP test specifications. Similarly, in UC1.3, for output variables, equivalence classes are defined using either type operations or a Java expression.

UC2 includes generating and analyzing the abstract test suite using the decision tree algorithm. Potential problems and possible root causes are then shown to the user. This is illustrated by Fig. 10 that shows how the decision tree analysis results are presented to the tester, for the first iteration of the Student B example presented in the case study (Section 5.3). On the left pane, one can see the decision tree itself in the upper part showing potential misclassifications and empty leaves. Various misclassification statistics are also provided. In the lower part of the left pane, a con-

⁵ The TRex library (<http://www.trex.informatik.uni-goettingen.de/trac/wiki/Wiki-Start>) was applied to help parsing the structure of the test suite TTCN-3 file, automatically obtaining both parameters definition and test cases data.

⁶ We will refer to both parameters and environment variables as parameters in the remainder of the text.

⁷ Java expressions are evaluated using the Janino Open source embedded Java compiler (<http://www.janino.net/>).

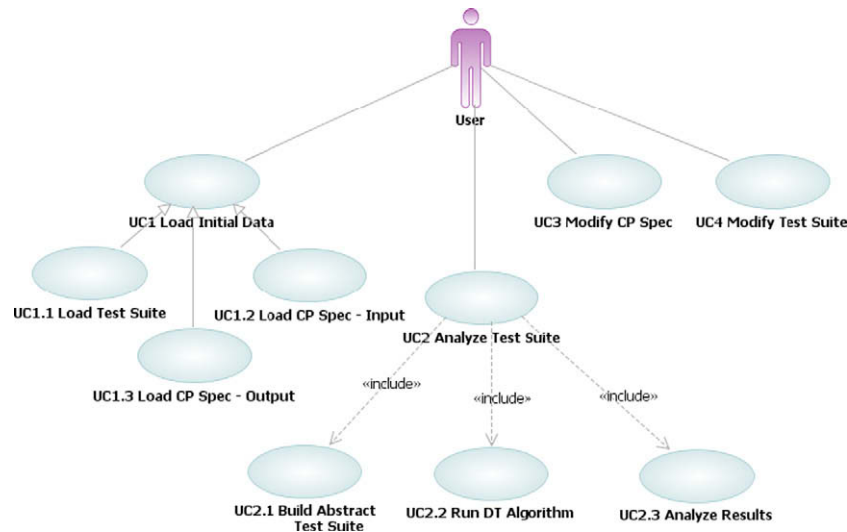


Fig. 9. Use case diagram for the MELBA tool.

fusion matrix is shown also summarizing the misclassifications taking place in the tree. In the right pane, potential problems are listed according to the problem classes defined in Section 4.1. For misclassifications, the leaf ID is provided as well as the number of test cases misclassified and possible causes to investigate. By clicking on the number of test cases the tester can obtain the exact list of test cases misclassified. For missing choices, which are obtained by comparing the decision tree and CP specification, missing choices and their categories are shown to the user as well as possible root causes.

The decision tree is also used to identify leaves with no instance as this suggests that a variable was found to be significant by the decision tree analysis at a certain position in the tree but that somehow one of its choices was not present in the test cases in this part of the tree. Since such a variable is partially determining the output, one might want to ensure all choices are exercised. The empty leaf IDs are listed, as well as possible causes. The last part in the right pane lists the combination of choices that are missing in the tree, following the heuristic that was presented in Section 4.3.

The remaining use cases (UC3 and UC4) are about providing support for the user to modify the CP specification and the test suite, according to the needs. The use cases presented above provide a complete environment for testers to refine their test specifications and test suites using the MELBA methodology.

6.2. Conceptual model

This section describes a model of all the concepts our tool is manipulating and their relationships. Our motivation is to describe how all the concepts relate to each other in a form that is rigorous, precise, and reusable by other tools as a starting point for analysis and design. We use two class diagrams (Figs. 11 and 12) for the sake of simplifying the presentation, though they correspond to one class diagram, as visible from the common classes in the two diagrams.

In Fig. 11, we model the inputs of the MELBA tool. Some inputs are imported from the TTCN-3 test suite (e.g., parameters) whereas others are provided by the tester (e.g., choice), as described in the use case model above. This is denoted with the stereotypes «TTCN-3» and «Tester». The class diagram can be briefly explained as follows: the system under test is associated with a test suite, a CP

specification, and eventually an abstract test suite. Test suites are composed of records, parameters, and their types, following the TTCN-3 standard. In addition test suites are composed of the test cases themselves, which provide parameter values. As defined in Section 2.1, abstract test cases are tuples of choices for input parameters and equivalence classes for output variables. Choices can be associated with properties and selectors, based on the Category-Partition method.

Fig. 12 shows that a system under test (e.g., the PackHexChar software) is also associated with one or more decision tree analyses (the several iterations discussed in Section 5.4 for instance). Each analysis may report nonconformities (or potential problems), which can be of different types as previously explained: A misclassification is a misclassified abstract test case; An unused category or choice is a choice of the CP specification (an unused category results in all its choices being unused); A missing choice combination is associated with a set of choices.

All the information in the conceptual models above, which is handled as objects in the MELBA tool, is serialized and deserialized to and from XML files using the Simple open source framework (<http://simple.sourceforge.net/>).

7. Related work

We see three different areas of work related to the MELBA technology. First, our work bears some similarities with techniques that learn program behavior [2,6,20,31]. 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 [2], control flow graph [6]); (2) They produce different kinds of (reverse-engineered) specification (e.g., ADTs [2], 'likely invariants' [20,31], as defined in [16]); (3) They provide no (or little [31]) guidance regarding the definition (or refinement) of test cases (some rely on an automatic test data generator [6]).

A second area of related works are those techniques that attempt to improve test suites [4,5,13,15,34]. 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' [13]) possibly from actual users in the field [15]; (2) A learning algorithm is not always used to help the user improve test suites (e.g., [15]) or no real guidance for the generation of new test cases

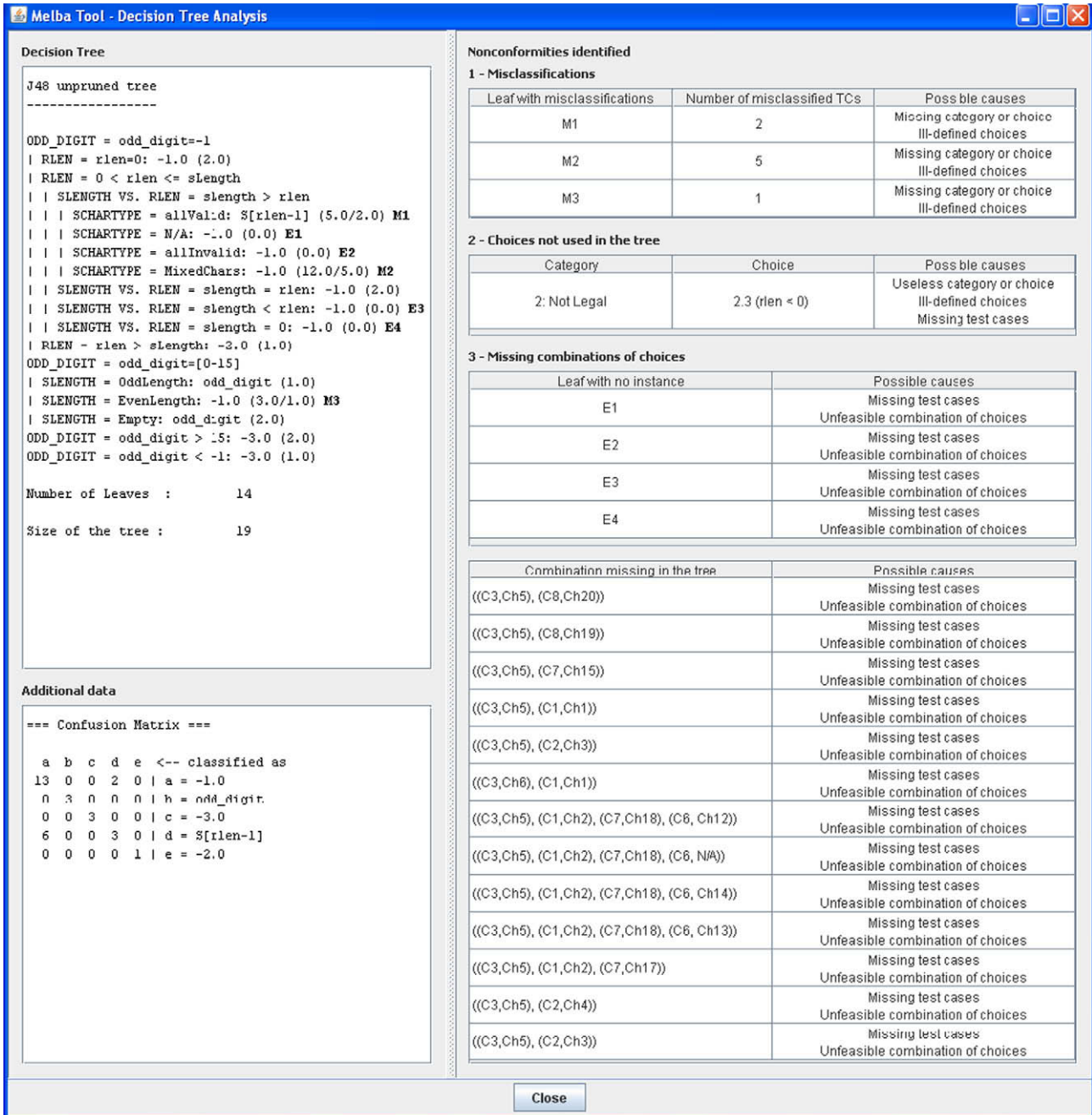


Fig. 10. Example decision tree analysis, student B, 1st iteration.

is always provided (e.g., [34]); (3) When test data are automatically generated, only rudimentary algorithms are used (e.g., simple constraint solving algorithms [13]); (4) They rely on different kinds of (reverse-engineered) specification (e.g., 'likely invariants' [13], as defined in [16], Z specification [34]).

A third area of related works are those techniques that attempt to reduce test suites (e.g., [28]). One of those is particularly related to our work since it works from a test suite generated according to the CP method. However, those methods rely on some kind of source code analysis to reduce a test suite. Instead, we use information learned about the test suite characteristics to do the reduction.

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 [19]; To improve diagnosability by pinpointing faulty statements with a high accuracy [5] (using Tarantula [23]); To identify feasible paths in a control flow graph with high traversing probability using an adaptive sampling mechanism [4]. Many other applications of machine learning techniques to software engineering exist in literature (e.g., [9,17]) but are less related to our focus on test suite and test specification improvement.

Last, there are a number of research activities conducted on the Category-Partition (CP) method. We already mentioned [10] where authors performed a case study on the use of CP. They identified that practitioners need to use a checklist when creating categories and choices. We see our MELBA method as a tool support for such heuristics (and others), though it goes beyond this as we provide guidance for the improvement of the test suite. The same authors

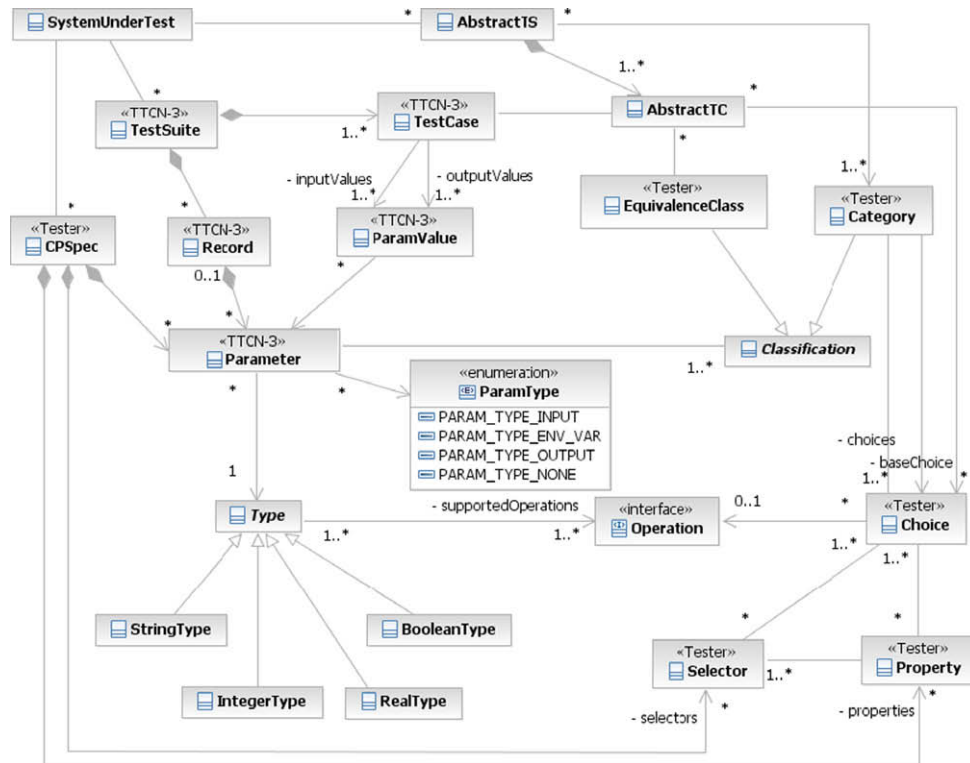


Fig. 11. MELBA Input conceptual model.

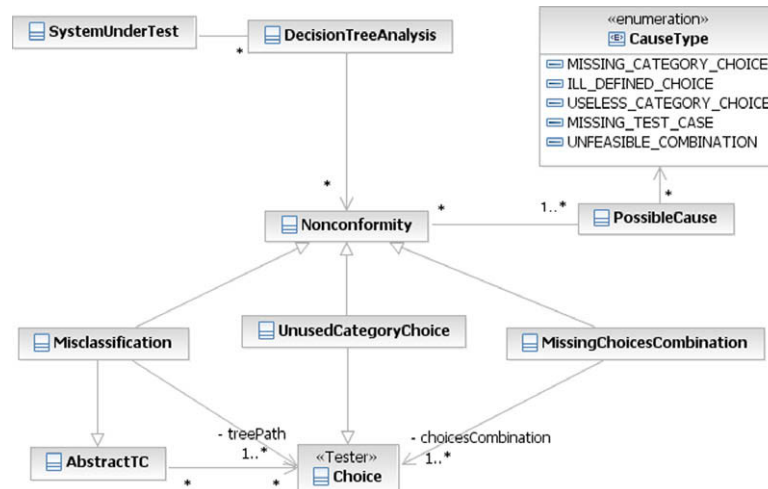


Fig. 12. MELBA decision tree analysis conceptual model.

also provided tool support [11] for the use of the CP method, specifically focusing on helping the user define constraints between choices (i.e., properties and selectors)—for instance by automatically detecting inconsistencies between constraints—and then on using those constraints to generate test frames (i.e., combinations of choices that satisfy constraints).

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 (non) functional test cases to consider, (3) it helps refine the test specifications from which test cases can then be derived following a clear rationale and (4) it does not require any program instru-

mentation. To conclude, no existing technique can be directly compared with MELBA.

8. 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 black-box 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 and environment 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). There are two possible application contexts for the above methodology. One scenario is when one is improving an existing test suite for which there is no explicit test specification and rationale (e.g., reusing open source software). A second scenario corresponds to a black-box testing process involving the definition of a test specification (e.g., using CP) and then the generation of test cases from it.

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 addressed the two abovementioned application contexts and showed that the iterative process can indeed improve both test specifications and test suites. The CP specifications were refined to a level that is equivalent to what an expert would likely produce within two to three improvement cycles. The augmented test suites were significantly more effective in terms of fault detection while only requiring a modest size increase.

A tool was presented in order to demonstrate how the MELBA methodology can be effectively supported. The tool's functionality, user interface, and underlying concepts were described.

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 improving the decision-making support provided by the MELBA tool.

References

- [1] P. Ammann, A.J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.
- [2] G. Ammons, R. Bodik, J.R. Larus, Mining specifications, in: *Proceedings of ACM Principles of Programming Languages*, 2002, pp. 4–16.
- [3] J.H. Andrews, L.C. Briand, Y. Labiche, A.S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, *IEEE Transactions on Software Engineering* 32 (8) (2006) 608–624.
- [4] N. Baskiotis, M. Sebag, M.-C. Gaudel, S. Gouraud, A machine learning approach for statistical software testing, in: *Proceedings of International Joint Conference on Artificial Intelligence*, 2007, pp. 2274–2279.
- [5] B. Baudry, F. Fleurey, Y. Le Traon, Improving test suites for efficient fault localization, in: *Proc. International Conference on Software Engineering*, 2006, pp. 82–91.
- [6] J.F. Bowring, J.M. Rehg, M.J. Harrold, Active learning for automatic classification of software behavior, in: *Proceedings of the ACM International Symposium on Software Testing and Analysis*, 2004, pp. 195–205.
- [7] L.C. Briand, Y. Labiche, Z. Bawar, Using Machine Learning to Refine Black-box Test Specifications and Test Suites, Carleton University, Technical Report SCE-07-05, 2007.
- [8] L.C. Briand, Y. Labiche, Z. Bawar, Using machine learning to refine black-box test specifications and test suites, in: *Proceedings of the International Conference on Quality Software*, 2008.
- [9] L.C. Briand, Y. Labiche, X. Liu, Using machine learning to support debugging with Tarantula, in: *Proceedings of International Symposium on Software Reliability Engineering*, 2007, pp. 137–146.
- [10] T.Y. Chen, P.-L. Poon, S.-F. Tang, T.H. Tse, On the identification of categories and choices for specification-based test case generation, *Information and Software Technology* 46 (13) (2004) 887–898.
- [11] T.Y. Chen, P.-L. Poon, T.H. Tse, A choice relation framework for supporting category-partition test case generation, *IEEE Transactions on Software Engineering* 29 (7) (2003) 577–593.
- [12] W.W. Cohen, Y. Singer, Simple, fast, and effective rule learner, in: *Proceedings of AAAI/IAAI*, 1999, pp. 335–342.
- [13] C. Csallner, Y. Smaragdakis, DSD-Crasher: a hybrid analysis tool for bug finding, in: *Proceedings of ACM International Symposium on Software Testing and Analysis*, 2006, pp. 245–254.
- [14] S. Demeyer, S. Ducasse, O. Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2003.
- [15] S. Elbaum, M. Diep, Profiling deployed software assessing strategies and testing opportunities, *Transactions on Software Engineering* 31 (4) (2005) 312–327.
- [16] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, *IEEE Transaction on Software Engineering* 27 (2) (2001) 1–25.
- [17] P. Francis, D. Leon, M. Minch, A. Podgurski, Tree-based methods for classifying software failures, in: *Proceedings of International Symposium on Software Reliability Engineering*, 2004, pp. 451–462.
- [18] M. Grochtmann, K. Grimm, Classification trees for partition testing, *software testing, Verification and Reliability* 3 (2) (1993) 63–82.
- [19] M. Haran, A. Karr, M. Last, A. Orso, A. Porter, A. Sanil, S. Fouché, Techniques for classifying executions of deployed software to support software engineering tasks, *Transactions on Software Engineering* 33 (5) (2007) 1–18.
- [20] M. Harder, J. Mellen, M.D. Ernst, Improving test suites via operational abstraction, in: *Proceedings of 25th International Conference on Software Engineering*, 2003, pp. 60–71.
- [21] J. Hartmann, M. Vieira, H. Foster, A. Ruder, A UML-Based approach to system testing, *Innovations in Systems and Software Engineering* 1 (1) (2005) 12–24.
- [22] J. Hartmann, M. Vieira, A. Ruder, UML based test generation and execution, in: *Proceedings of Workshop on Software Test, Analyses and Verification*, 2004.
- [23] J.A. Jones, M.J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: *Proceedings of International Conference on Automated Software Engineering*, 2005, pp. 273–282.
- [24] P.C. Jorgensen, *Software Testing: A Craftsman's Approach*, second ed., CRC Press, 1995.
- [25] Y.-S. Ma, A.J. Offutt, Y.-R. Kwon, MuJava: a mutation system for Java, in: *Proceedings of International Conference on Software Engineering*, 2006, pp. 827–830.
- [26] P. Maki-Asiala, M. Matinlassi, Quality assurance of open source components: integrator point of view, in: *Proceedings of Annual International Computer Software and Applications Conference*, 2006, pp. 189–194.
- [27] B. Marick, *The Craft of Software Testing*, Prentice Hall, 1995.
- [28] A.P. Mathur, *Foundations of Software Testing*, Pearson, 2008.
- [29] M. Michlmayr, F. Hunt, D. Probert, Quality practices and problems in free software projects, in: *Proceedings of First International Conference on Open Source Systems*, 2005, pp. 24–28.
- [30] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating functional test, *Communications of the ACM* 31 (6) (1988) 676–686.
- [31] C. Pacheco, M.D. Ernst, Eclat: automatic generation and classification of test inputs, in: *Proceedings of European Conference on Object-Oriented Programming, LNCS*, vol. 3586, 2005, pp. 504–527.
- [32] A. Pasquini, A. Crespo, P. Matrelle, Sensitivity of reliability-growth models to operational profiles errors vs testing accuracy, *IEEE Transactions on Reliability* 45 (4) (1996) 531–540.
- [33] J.R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.
- [34] H. Singh, M. Conrad, S. Sadeghipour, Test case design based on Z and the classification-tree method, in: *Proceedings of International Conference on Formal Engineering Methods*, 1997, pp. 81–90.
- [35] C. Willcock, D. Deiß, S. Tobies, S. Keil, F. Engler, S. Schulz, *An Introduction to TTCN-3*, Wiley, 2005.
- [36] I.H. Witten, E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, second ed., Morgan Kaufman, 2005.
- [37] L. Zhao, S. Elbaum, Quality assurance under the open source development model, *Journal of Systems and Software* 66 (1) (2003) 65–75.