White on Black: A White-Box-Oriented Approach for Selecting Black-Box-Generated Test Cases*

T. Y. Chen
School of Information Technology
Swinburne University of Technology
Hawthorn 3122, Australia
tychen@it.swin.edu.au

P. L. Poon
Department of Accountancy
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
acplpoon@inet.polyu.edu.hk

S. F. Tang†
Department of Accountancy
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
acsharon@inet.polyu.edu.hk

Y. T. Yu
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Kowloon Tong, Hong Kong
csytyu@cityu.edu.hk

Abstract

Many useful test case construction methods that are based on important aspects of the specification have been proposed in the literature. A comprehensive test suite thus obtained is often very large and yet is non-redundant with respect to the aspects identified from the specification. This paper addresses the problem of selecting a subset of test cases from such a test suite. We propose the use of white box criteria to select test cases from the initial black-box-generated test suite. We illustrate our ideas with examples and demonstrate the viability and benefits of our approach by means of a case study.

Keywords Category-partition method, classification-tree method, partition testing, specification-based testing, test case selection

1. Introduction

It would be ideal to test a program with its entire *input domain* (that is, with all the possible inputs). With this approach, in theory, any program fault that exists in the program is guaranteed to be detected. In reality, however, this "exhaustive" approach is difficult or almost impossible to apply because of the huge number of test cases involved and the various resource constraints imposed on the software tester. Thus, a more practical approach is to construct a *test suite* (a subset of the input domain) for testing [4, 11].

Generally speaking, test-suite construction methods belong to either the *white box* (or *code-based*) or the *black box* (or *specification-based*) approach. The former refers to the testing that is based on the information derived from the source code of the program, whereas the latter makes use of information from the specification. The two methods have their own merits and limitations; they are generally considered complementary to each other [2, 3, 14].

Regardless of how the test suite is constructed, it has to satisfy certain requirements, some of which are frequently conflicting. For example, the test suite constructed should be as comprehensive as possible so that it is effective in detecting all possible faults in the software, and it should be as small as possible in order to control the cost of the testing. A very comprehensive test suite containing too many test cases can be too costly to be practical, whereas a small but ineffective test suite may lead to many undetected faults that severely compromise the quality of the software.

In the black box approach, the *category-partition method* (CPM) [1, 13] and the *classification-tree method* (CTM) [5, 6, 10, 15] are particularly useful, as they are easy to

*This work is supported in part by a Strategic Research Grant (project no. 7000958) from City University of Hong Kong, and in part by a grant (project no. CityU1118/99E) from the Research Grants Council of the Hong Kong Special Administrative Region, China.

†**Contact author.** Also with the School of Information Technology, Swinburne University of Technology, Australia.

understand and use, and can be applied to both formal and informal specifications [7, 12]. However, experience shows that in many situations the size of a comprehensive test suite derived from these methods is very large. There is a need to select a subset from this test suite for use, particularly when it is impractical to execute all the test cases in the original comprehensive test suite.

In this paper, we address the problem of selecting test cases from a comprehensive test suite *TS* that is derived from a specification-based criterion C_B (we shall use C_B to denote a ***B***lack box or specification-based criterion, and C_W to denote a ***W***hite box or code-based criterion). We note that, unless *TS* contains test cases that are redundant with respect to C_B, it cannot be reduced without jeopardising its completeness [4]. Therefore, a theoretically sound methodology of selecting a subset from *TS* has to make use of information from a source different from the original criterion C_B. We propose that the specification-based criterion should be supplemented by white box information in the selection of test cases from *TS* for use. We shall illustrate our ideas and demonstrate the viability and benefits of our approach by means of examples and a case study. In our examples, the test suite *TS* is constructed by using the classification-tree method (CTM) [5, 6, 10, 15], but it should be clear in the context that our approach applies equally well to test suites constructed by using other specification-based methods such as the category-partition method (CPM) [1, 13].

The rest of this paper is structured as follows. Section 2 provides, by means of an example, an overview of CTM for constructing a test suite from the specification. Section 3 describes our approach. Section 4 presents a case study showing the viability and benefits of our approach. Section 5 summarises and concludes this paper.

2. Background

Basically, both CPM and CTM make use of the approach of *partition testing* [8, 9, 14, 16]. In this approach, the input domain of a program is divided into subsets, called *subdomains*, according to a partitioning scheme. In CTM, the tester identifies important relevant aspects of the specification for testing. Each aspect, called a *classification*, corresponds to a partitioning scheme, and the corresponding subdomains are called *classes* associated with the classification. The idea is that all elements within a class are essentially the same with respect to the relevant aspect for the purpose of testing. For a given specification, usually many aspects (corresponding to different partitioning schemes) may be identified. Test cases are then formed by combining classes associated with different aspects, so that each combination forms one single test case. A different terminology is used in CPM, and a comparison of CPM and CTM can be found in [7].

In this paper, we shall illustrate our approach by an example in which the test suite is constructed using CTM. However, we stress again that our approach is also applicable to test suites constructed using other specification-based methods such as CPM.

Example 1 (rewards)
Consider a specification **rewards** of a program which accepts the details of a credit card purchase transaction and, based on the credit balance information, determines the number of reward points if the transaction is approved. Cardholders can use the reward points to claim various benefits. Other details such as the way of computing the reward points and benefit claims need not concern us here.

Suppose that the *classifications* and their associated *classes* have been identified as in Table 1. For ease of reference, we abbreviate the classifications by letters in upper case and the classes by corresponding letters in lower case with numeric subscripts. For example, M and m_2 refer to the classification "Class of Ticket" and its associated class "Business", respectively.

Once all the relevant aspects and classes have been identified, test cases can be formed by selecting and combining classes from different classifications. For example, two test cases tc_1 and tc_2 formed by this approach are:

$$tc_1 = \{ a_4, b_4, c_1, d_1, e_1, f_1, g_1, h_2, i_2, j_1, k_2, n_1 \}$$
$$tc_2 = \{ a_4, b_4, c_2, d_1, e_1, f_2, g_2, h_3, i_2, j_1, k_1, l_1, m_1 \}$$

An exhaustive evaluation of all combinations of classes produces a total of 221184 ($= 4^2 \times 2^2 \times 3 \times 2^2 \times 3 \times 2^4 \times 3 \times 2$) test cases. However, many of these test cases are in fact not useful because they contain incompatible classes. These test cases are said to be *illegitimate* [5, 6]. For example, since holders of corporate cards are not further distinguished as principal or additional cardholders, the two classes "Corporate" (c_1) and "Principal" (d_1) are incompatible. Thus, the test case tc_1 above is illegitimate because it contains the incompatible classes c_1 and d_1. Test cases that are not illegitimate are said to be *legitimate*.

In CTM, a classification tree is constructed which organises classifications and classes at alternate levels. A combination table is then formed from which test cases are defined [5, 6, 10, 15]. In this example, only 1302 test cases will be defined, and all the remaining combinations are illegitimate. In this way, a significant amount ($221184 - 1302 = 219882$ or 99.4%) of illegitimate test cases is then eliminated right from the start of defining the test cases. For more details, readers may refer to [6, 10].

Even so, some of the test cases defined through the classification tree and the combination table may still be illegitimate. For instance, a cardholder's cumulative

balance can never exceed his/her credit limit. Thus, the test case $tc_3 = \{a_4, b_4, c_2, d_1, e_1, f_1, g_1, h_3, i_2, j_2, k_2, n_1\}$ is also illegitimate as it contains the incompatible classes "Credit Limit in HK$ = 40000" (f_1) and "40000.00 < Cumulative Balance in HK$ $\leq$ 80000.00" (h_3). Such illegitimate test cases have to be further identified and eliminated. In this example, 432 test cases are then removed, resulting in a test suite of 870 legitimate test cases. ■

3. Our approach

3.1. Motivation

Let us first recapitulate the essentials of Example 1. By considering the specification, the tester identifies important aspects (classifications) and their associated classes that are relevant for the purpose of testing as in Table 1. By organising the classifications and classes in the form of a tree and then further eliminating the illegitimate test cases, a test suite *TS* containing all the 870 legitimate test cases is constructed. We shall refer to *TS* as the *initial test suite*.

The initial test suite *TS* satisfies the criterion C_B of covering all compatible combinations of the classes identified from the specification. According to this criterion C_B, there is no redundant test case in *TS*. Thus, all test cases in *TS* should be selected for testing should resources allow.

In practice, software testers have to take into account of the need to control testing costs. Resource limitations often dictate that only a subset TS' of the entire test suite *TS* can be used when the latter is large. In such situations, there is a need for some methodical guidelines as to how test cases should be selected from *TS* for use. Doing so in an ad hoc manner is obviously undesirable since the effect of such a process is unknown and therefore the resulting subset TS' may be of unknown quality.

We note that conventional test reduction methods such as those proposed in [4, 11] are not applicable here, simply because there is no redundancy in *TS* with respect to the original criterion C_B. A theoretically sound methodology of selecting test cases from *TS* must bring in a different source of information for this purpose.

We propose to use white box information to select test cases from the black-box-generated initial test suite *TS*, for the following reasons. Firstly, if any other black box criterion is considered appropriate, it could have already been taken into account during the construction of *TS*. Secondly, it is well known that white box testing should be complementary to black box testing in providing valuable additional information that the latter lacks [2, 3, 14].

In what follows, we shall illustrate how white box information can be used for selecting test cases from a black-box-generated test suite. Although we use the information of the paths executed by the test cases for illustration in our example, it should be clear that other white box information may also be used in our approach.

3.2. Rationale

Consider the three test cases from Example 1 as follows:

$tc_4 = \{a_4, b_4, c_1, e_1, f_1, g_1, h_1, i_2, j_1, k_1, l_1, m_1\}$

$tc_5 = \{a_4, b_4, c_1, e_1, f_1, g_1, h_1, i_2, j_1, k_1, l_1, m_2\}$

$tc_6 = \{a_4, b_4, c_1, e_1, f_1, g_1, h_1, i_2, j_1, k_1, l_1, m_3\}$

These test cases contain different classes "First" (m_1), "Business" (m_2) and "Economy" (m_3), respectively, for the classification "Class of Ticket" (M), but are otherwise identical. That is, they contain exactly the same classes for all other classifications. We refer to a pair of test cases (such as tc_4 and tc_5) with this property as a *matching pair*. The two classes in which a matching pair of test cases differ form a pair of *differentiating classes*, or a *differentiating class pair*. By definition, a pair of differentiating classes must be associated with the same classification. For the matching pair tc_4 and tc_5, the differentiating classes are m_1 and m_2. Similarly, the test cases tc_5 and tc_6 also form a matching pair with m_2 and m_3 as the differentiating classes.

With only the specification, there is no choice but to regard a matching pair as different since they differ in one aspect. However, the program may process a matching pair similarly or differently, depending on the way of implementation.

Consider Figure 1(a) which shows part of one possible implementation of the specification **rewards** in Example 1. Clearly, if this code segment is the only part in the implementation that handles the aspect "Class of Ticket" (M), the two test cases tc_4 and tc_5 will execute the same path which contains the line ⟨55⟩, while the paths corresponding to tc_5 and tc_6 will differ. On the other hand, if the relevant part of the implementation is as shown in Figure 1(b), then the paths executed by tc_4, tc_5 and tc_6 will all differ.

We note that in the second case (Figure 1(b)), all the three test cases tc_4, tc_5 and tc_6 have to be selected from the initial test suite *TS*, or else its effectiveness will be compromised. This is apparent if we consider the possibility that a fault might be present in line ⟨78⟩ but not in line ⟨76⟩. In this situation, omitting tc_5 (which includes the class "Business") might leave this fault undetected. Similar reasons show that neither tc_4 nor tc_6 should be omitted.

In contrast, in the first case (Figure 1(a)), the differentiating classes m_1 and m_2 have been processed in a similar way according to the implementation. Thus, any fault revealed by inputs from class m_1 is likely to be revealed by inputs from m_2 as well. Therefore, selecting only tc_4 (or tc_5) from the initial test suite *TS* should not affect its effectiveness. In other words, although the

Table 1. Some possible classifications and classes for rewards

Classifications	Corresponding Classes
Status of Card File (A)	File Not Exist (a_1), File Empty (a_2), File Not Empty and Card Not Found (a_3), Card Number in File (a_4)
Card Status (B)	Loss (b_1), Expired (b_2), Suspended (b_3), Normal (b_4)
Type of Card (C)	Corporate (c_1), Personal (c_2)
Type of Cardholder (D)	Principal (d_1), Additional (d_2)
Class of Card (E)	Diamond (e_1), Gold (e_2), Classic (e_3)
Credit Limit in HK$ (F)	40000 (f_1), 80000 (f_2)
Country of Purchase (G)	Hong Kong (g_1), Overseas (g_2)
Cumulative Balance in HK$ (H)	$0.00 \leq H \leq 10000.00$ (h_1), $10000.00 < H \leq 40000.00$ (h_2), $40000.00 < H \leq 80000.00$ (h_3)
Current Purchase Amount in HK$ (I)	≤ 0 (i_1), > 0 (i_2)
Cumulative Balance + Current Purchase Amount − Credit Limit (J)	≤ 0 (j_1), > 0 (j_2)
Type of Goods (K)	Airline Tickets (k_1), Other Goods (k_2)
Airline Company (L)	City Airline (l_1), Other Airlines (l_2)
Class of Ticket (M)	First (m_1), Business (m_2), Economy (m_3)
Bonus Partner (N)	Yes (n_1), No (n_2)

```
<51>  if "City Airlines" then
<52>      caculate extra rewards
<53>  else
<54>      if ("first-class ticket" or "business-class ticket") then
<55>          calculate extra rewards
<56>      else
<57>          calculate normal rewards
<58>      endif
<59>  endif
```

(a)

```
<71>  if "City Airlines" then
<72>      caculate extra rewards
<73>  else
<74>      begin case
<75>          case "first-class ticket"
<76>               calculate extra rewards
<77>          case "business-class ticket"
<78>               calculate extra rewards
<79>          case "economy-class ticket"
<80>               calculate normal rewards
<81>      end case
<82>  endif
```

(b)

Figure 1. Two possible partial implementations of rewards

matching pair tc_4 and tc_5 are considered different from the specification perspective, one of them seems to be redundant with respect to the white box (more specifically, same-path) criterion.

Notice that whether we should select only tc_4 or tc_5 from *TS* cannot be judged from the specification alone, but can only be determined by considering white box information. Clearly, the above argument applies to all matching pairs with the same pair of differentiating classes. In Figure 1(*a*), for every matching pair with differentiating classes m_1 and m_2, we may safely select one of the two test cases and omit the other with little effect on the efficacy of *TS*.

In short, we argue that the implementation provides additional information that are supplementary to the specification-based criteria based on which the initial test suite *TS* is constructed. Such additional information helps us to decide which test cases should be selected and which could be omitted from the initial test suite *TS* that is constructed solely from some specification-based criteria.

3.3. Automation via partial dynamic analysis

In Section 3.2, we have illustrated the use of white box information by means of the "same-path" criterion, that is, test cases are considered to be processed similarly if they execute the same path. In principle, however, other white box criteria may be used. More generally, test cases that are considered to be processed similarly by a white box criterion C_W are said to be *C_W-equivalent*. Two classes x_1 and x_2 of the same classification X are also said to be *C_W-equivalent classes* if *all* matching pairs with differentiating classes x_1 and x_2 are themselves C_W-equivalent.

For ease of reference, when the "same-path" criterion is used, two test cases (respectively classes) that are C_W-equivalent will be simply called a *copath pair* of test cases (respectively classes).

Our approach, in general, involves the following essential steps:

(1) Obtain an initial test suite *TS* which is comprehensive and contains no redundant test case according to a black box (specification-based) criterion C_B.

(2) Choose a white box criterion C_W (such as the same-path criterion) to be used in step (3).

(3) Select a classification (called *candidate classification*) X and two of its classes (called *candidate classes*) x_1 and x_2 that are expected to be C_W-equivalent.

(4) Identify all matching pairs with differentiating classes x_1 and x_2 from the initial test suite *TS*.

(5) Determine from the implementation if all the identified matching pairs are C_W-equivalent. If so, select only one test case from every such matching pair. Otherwise, all of the test cases are retained.

(6) Repeat steps (3) to (5) if appropriate.

It is clear that, in principle, our approach can be used without any tool. However, there may be many classifications and classes, and the initial test suite *TS* may be very large. Thus, carrying out the above steps manually can be tedious and error-prone. This is particularly true for step (5), which determines if every identified matching pair is C_W-equivalent. Automation would therefore relieve the effort of the tester and render the approach more appealing in practice.

One way of automating step (5) is to perform a dynamic analysis. For example, suppose that we choose the same-path criterion. Then there are tools, usually based on instrumentation, that can be used to check if every matching pair is copath. However, this dynamic analysis method requires the execution of every matching pair of test cases. This seems to have defeated the purpose of trying *not* to execute all these test cases in the first place.

We propose to address this problem by using a *partial* dynamic analysis method. With this method, we *sample* some of the matching pairs and monitor their executions. Basically, our heuristics is to extrapolate the sampling result to judge whether or not every matching pair is indeed C_W-equivalent.

Let us use the specification **rewards** and the same-path criterion to illustrate our partial dynamic analysis method.

Refer to Table 1 and Figure 1. Firstly, based on the information derived from **rewards**, the tester selects a pair of candidate classes x_1 and x_2 of some candidate classification X. By doing so, the tester considers it likely that every matching pair with differentiating classes x_1 and x_2 is a copath pair. Examples of candidate classes are "First" and "Business" of the candidate classification "Class of Ticket". These two classes are selected as candidate classes because the specification **rewards** states that if the air ticket is purchased from City Airline, then the cardholder can earn extra rewards points (calculated in the same way) no matter whether it is a first-class or a business-class ticket. Note that the selection of candidate classes is based on the tester's own experience and expertise, and the tester's guess that the matching pairs are all copath may be right or wrong.

Secondly, we construct the set TS_1 containing all these matching pairs. Thirdly, we sample a certain proportion r of the matching pairs of test cases from TS_1 and monitor their executions. Then either one of the following situations occurs:

(*a*) every matching pair of test cases selected from TS_1 are a copath pair; or

(*b*) some matching pair of test cases selected from TS_1 are *not* copath pairs.

If situation (*a*) occurs, then we may judge that the remaining matching pairs (those that have not yet been executed) are also copath pairs. Therefore, one test case of each of these remaining matching pairs will be considered redundant (with respect to the same-path criterion). Note that since this is essentially a sampling process, there is a chance of making the wrong assertion that every matching pair is copath.

On the other hand, if situation (*b*) occurs, then none of the test cases in the remaining matching pairs could be safely omitted.

In the above, the value of r is determined by the tester based on the available testing resources and the confidence level required of not making the wrong assertion that every matching pair is copath. Obviously, with a larger value of r, the tester will be more confident of the judgment, but then more matching pairs have to be sampled, giving a larger resulting test suite. In other words, there is a trade-off between the level of confidence and the size of the final test suite.

4. Case study

In order to assess the practicality and gain more experience with the issues involved in our approach, we have performed a case study. In this case study, we would like to shed light to the answers of the following questions:

(*a*) How applicable is our approach?

(*b*) How much savings of test cases are possible with our approach?

In this section, we outline the way our case study is performed, followed by a discussion of the results of analysis. The above two questions will be discussed in Sections 4.2.2 and 4.2.3, respectively, in light of the results obtained.

4.1. The study

We use the specification **rewards** outlined in Example 1. One of us, hereafter referred to as Person-A, performs step (1) of our approach (Section 3.3). More specifically, Person-A, who is familiar with CTM, identifies relevant classifications and their associated classes as in Table 1, organises them into a classification tree, defines all the 1302 test cases and identifies all the 870 legitimate test cases as described in Example 1. This forms the initial test suite *TS* which is based only on the information of the specification.

Another researcher, whom we call Person-B, chooses the same-path criterion for selecting test cases from *TS*. Based on his expertise and experience, Person-B identifies three candidate classifications and corresponding candidate classes. These candidate classes are marked with an asterisk (*) in Table 2. For the purpose of control, Person-B also identifies two pairs of classes that are expected *not* to be C_W-equivalent. These classes are marked with a dagger (†) in Table 2. In addition, Person-B identifies all matching pairs from the initial test suite *TS* with the differentiating classes shown in Table 2. This completes steps (2) to (4) of our approach.

We stress that due care has been made to ensure that both Person-A and Person-B have performed their tasks independently of each other (except that Person-B has selected, from the classification tree constructed by Person-A, the five classifications and the corresponding class pairs as shown in Table 2), and that no implementation information is available to them.

Meanwhile, two groups of computer science undergraduate students were asked to write programs individually for implementing the specification **rewards**. Among them, the first group of students were studying full-time at their final year whereas the second group of students were studying part-time at the year before their final year. These students generally had one year working experience in the computer field. As such, they may be considered as novice programmers.

These students have been reminded of the need to well test their own programs, but they have not been taught CTM or CPM at the time when they wrote the programs. Nor have they been told what test cases we will use to test their programs.

To limit the scope of this case study, we picked 15 programs arbitrarily from the students' programs for analysis. These 15 programs were instrumented and tested with the entire initial test suite containing all the 870 legitimate test cases defined by Person-A. All executions were monitored and the executed program paths were recorded.

4.2. Results and analysis

4.2.1. Copath and quasi-copath pairs. In Section 3.3, we have defined a "copath pair" of *test cases* as those that execute the same path. We now extend the definition of the term "copath pair" to *classes* as follows. Two candidate classes x_1 and x_2 of the same classification X are said to form a *copath pair*, if *all* matching pairs with differentiating classes x_1 and x_2 are themselves copath.

We also define another term for classes that satisfy a slightly less restrictive condition as follows. Two candidate classes y_1 and y_2 of the same classification Y are said to form a *quasi-copath pair* of level $p\%$ (where $0 < p < 100$), if at least $p\%$ but not all of the matching pairs with

differentiating classes y_1 and y_2 are themselves copath.

Intuitively, every matching pair of test cases are C_W-equivalent if their differentiating classes form a copath pair. In Section 3.2 we have argued that one test case from each of these matching pairs may be safely omitted from the initial test suite without affecting its effectiveness.

If the differentiating classes form a quasi-copath pair of level $p\%$, then a matching pair picked randomly will have at least probability $p\%$ of being C_W-equivalent. Given a large value of p, a similar argument as in Section 3.2 leads to a slightly weaker conclusion: Omitting one of the two test cases of a matching pair whose differentiating classes form a quasi-copath pair of level $p\%$ will have a high chance (which depends on the level p) of preserving the effectiveness of the initial test suite.

4.2.2. Applicability. Obviously, whether two candidate classes form a copath or quasi-copath pair depends on the implementation. Table 3 shows the number of programs in which the selected candidate classes are copath or quasi-copath of level 85%. It shows that the candidate classes "First" (m_1) and "Business" (m_2) of the classification "Class of Ticket" are copath in 6 out of the 15 programs, that is, in about 40% of the programs. These two candidate classes are quasi-copath (of level 85%) in *all* the remaining 9 programs.

For the first two candidate class pairs, they are copath in 3 and 4 out of the 15 programs, respectively, that is, in about 20% and 27% of the programs. These two pairs are also quasi-copath (of level 85%) in 11 out of the remaining 12 programs and 9 out of the remaining 11 programs, respectively. Clearly, the candidate class pairs selected by Person-B without any knowledge of the implementations are, as expected, very likely to contain matching pairs of test cases that are copath.

Also, as expected by Person-B before looking at the implementations, the pair of classes "Hong Kong" (g_1) and "Overseas" (g_2) are neither copath nor quasi-copath in all the 15 programs under study. Moreover, the last pair of classes, "First" (m_1) and "Economy" (m_3), are *not* copath in any of the 15 programs, though they are quasi-copath of level 85% in 5 of the 15 programs.

As argued in Sections 3.2 and 3.3, once the candidate classes are judged to be copath, it is expected that one test case of each matching pair can be safely omitted without loss of effectiveness. If our partial dynamic analysis method is used, then even if the class pair is actually quasi-copath but not copath, chances are still high that the matching pairs sampled are all copath pairs. If so, then the tester will judge that the class pair is copath and therefore omit some test cases from the initial test suite. In the latter case, although the tester has made an incorrect judgment, the effectiveness of the initial test suite will probably be only slightly reduced. This is because the proportion of matching pairs being copath is high, so that only few, if any, of the paths might be missed due to the omission of some test cases.

4.2.3. Savings of test cases. Obviously, the actual amount of savings of test cases depends on the number of matching pairs for a given candidate class pair. For the candidate class pairs selected by Person-B, the number of matching pairs are shown in column 3 of Table 2. The last two columns of this table show the number of test cases saved by using the partial dynamic analysis method which samples a proportion of r of all matching pairs of the candidate classes. For the selected candidate classes and values of r shown in Table 2, the amount of savings ranges from 13% to 30%.

These preliminary results are rather encouraging, showing that a substantial amount of savings of test cases can be achieved using our approach. This case study demonstrates that our approach is indeed applicable, since the candidate classes selected by a person without any knowledge of the implementation are indeed copath in some of the programs, and are almost always quasi-copath in other programs.

5. Conclusion

Many black box testing methods have been developed to construct test suites systematically from the information in the specification. Typically, such a method generates a test suite that is comprehensive, in the sense that it covers all compatible combination of classes of inputs. This ensures that all aspects identified from the specification to be relevant for the purpose of testing will be sufficiently well tested.

Although considered comprehensive with respect to the black box criterion C_B, a test suite TS thus generated usually contains too many test cases to be practically tested in its entirety. Resource considerations often dictate the need of selecting only a subset of test cases from the initial test suite TS. This paper addresses the problem of how this should be done without jeopardising the effectiveness of TS.

We have argued that the black-box-generated test suite TS is non-redundant with respect to the specification-based criterion C_B. As such, conventional test reduction methods such as those proposed in [4, 11] are not applicable. A theoretically sound methodology must bring in a different source of information to guide the process of selecting test cases from TS. We have proposed that this new source of information should be a white-box-oriented one, as it is well known that the implementation often provides valuable additional information that supplements what the specification lacks.

Table 2. Number of matching pairs of selected classifications and classes

Classification	Class pair	No. of matching pairs	No. (percentage[#]) of test cases saved when	
			$r = 0.1$	$r = 0.2$
Type of Cardholder (D)	*Principal (d_1), *Additional (d_2)	288	259 (30%)	230 (26%)
Credit Limit in HK$ (F)	*40000 (f_1), *80000 (f_2)	216	194 (22%)	172 (20%)
Class of Ticket (M)	*First (m_1), *Business (m_2)	144	129 (15%)	115 (13%)
Country of Purchase (G)	†Hong Kong (g_1), †Overseas (g_2)	432	–	–
Class of Ticket (M)	†First (m_1), †Economy (m_3)	144	–	–

[#]*The total number of test cases is 870. All percentages are calculated with 870 as the denominator.*

**Those marked with an asterisk (*) are candidate classes.*

†*Those marked with a dagger (†) are pairs of classes that are expected not to be copath.*

Table 3. Number of programs in which the class pairs are copath or quasi-copath

Classification	Class pair	No. (percentage) of programs[#] in which the class pairs are copath	No. (percentage) of programs[#] in which the class pairs are quasi-copath of level 85%
Type of Cardholder (D)	*Principal (d_1), *Additional (d_2)	3 (20%)	11 (73%)
Credit Limit in HK$ (F)	*40000 (f_1), *80000 (f_2)	4 (27%)	9 (60%)
Class of Ticket (M)	*First (m_1), *Business (m_2)	6 (40%)	9 (60%)
Country of Purchase (G)	†Hong Kong (g_1), †Overseas (g_2)	0 (0%)	0 (0%)
Class of Ticket (M)	†First (m_1), †Economy (m_3)	0 (0%)	5 (33%)

[#]*The total number of programs is 15.*

**Those marked with an asterisk (*) are candidate classes.*

†*Those marked with a dagger (†) are pairs of classes that are expected not to be copath.*

In this paper, we have illustrated how this approach can be applied by means of an example. The example involves a specification **rewards** that processes the approval of credit card purchase transactions and the calculation of reward points. We use the classification-tree method (CTM) to construct the initial test suite *TS*, and then the "same-path" criterion to guide the selection of test cases from *TS*. We have demonstrated how the same-path criterion may help to decide which test cases should be selected and which could be safely omitted from *TS*.

Our approach involves six essential steps. One crucial step is to determine if the pairs of candidate classes are considered to be processed similarly according to the chosen white box criterion C_W. If so, these classes are called C_W-equivalent, and one test case from each of the corresponding matching pairs may be safely omitted with no loss of effectiveness.

Determining whether two classes are C_W-equivalent can be a tedious and error-prone task. We have proposed a partial dynamic analysis method to aid the automation of this task. Basically, the method samples the matching pairs corresponding to the two candidate classes and the sampling result is used to judge whether the candidate classes are C_W-equivalent.

Finally, we have performed a case study using the specification **rewards**. Although the candidate classes have been identified without any knowledge of the implementation, they are found to be C_W-equivalent in several of the programs under study. Moreover, in most of the remaining programs, the candidate classes are "almost C_W-equivalent". That is, each such pair of classes have a large proportion of the corresponding matching pairs of test cases that are indeed C_W-equivalent.

Our case study also shows that a substantial amount of testing effort can be saved by using our approach. The proportion of test cases that are judged to be safely omitted from the initial test suite with little or no loss of effectiveness ranges from 13% to 30%, with respect to the sample programs in this study.

Our case study is exploratory in nature and is by no means a comprehensive one. Hence, over-generalisation of its results is inappropriate. Nevertheless, it does demonstrate the viability and potential benefits of our approach. In view of the very encouraging preliminary results, we are now performing more extensive case studies and experiments to find out to what extent these results may be generalised, and to investigate what other issues have to be addressed before the benefits of our approach can be fully realised.

Acknowledgement

The authors wish to thank Tina Fung and Candy Ng for helping to conduct the case study described in this paper.

References

[1] M. Balcer, W. Hasling and T. Ostrand, "Automatic generation of test scripts from formal test specifications", in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, West Florida, USA, pp. 210–218, December 1989.

[2] H. Y. Chen, T. H. Tse, F. T. Chan and T. Y. Chen, "In black and white: An integrated approach to class-level testing of object-oriented programs", *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 250–295, July 1998.

[3] H. Y. Chen, T. H. Tse and T. Y. Chen, "TACCLE: a methodology for object-oriented software Testing At the Class and Cluster LEvels", *ACM Transactions on Software Engineering and Methodology* (accepted for publication).

[4] T. Y. Chen and M. F. Lau, "On the completeness of a test suite reduction strategy", *The Computer Journal*, vol. 42, no. 5, pp. 430–440, 1999.

[5] T. Y. Chen and P. L. Poon, "On the effectiveness of classification trees for test case construction", *Information and Software Technology*, vol. 40, no. 13, pp. 765–775, November 1998.

[6] T. Y. Chen, P. L. Poon and T. H. Tse, "An integrated classification-tree methodology for test case generation", *International Journal of Software Engineering and Knowledge Engineering* (accepted for publication).

[7] T. Y. Chen, P. L. Poon and Y. T. Yu, "Analysing the category-partition method and the classification-tree method for software testing", in *Proceedings of the Fourth World Multiconference on Systemics, Cybernetics and Informatics (SCI '2000) and the Sixth International Conference on Information Systems Analysis and Synthesis (ISAS '2000)*, Orlando, USA, July 2000.

[8] T. Y. Chen and Y. T. Yu, "On the relationship between partition and random testing", *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 977–980, December 1994.

[9] T. Y. Chen and Y. T. Yu, "On the expected number of failures detected by subdomain testing and random testing", *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, February 1996.

[10] M. Grochtmann and K. Grimm, "Classification trees for partition testing", *Software Testing, Verification and Reliability*, vol. 3, pp. 63–82, 1993.

[11] M. J. Harrold, R. Gupta and M. L. Soffa, "A methodology for controlling the size of a test suite", *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, July 1993.

[12] A. J. Offutt and S. Liu, "Generating test data from SOFL specifications", *Journal of Systems and Software*, vol. 49, pp. 49–62, 1999.

[13] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests", *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, June 1988.

[14] D. J. Richardson and L. A. Clarke, "Partition analysis: A method combining testing and verification", *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1477–1490, December 1985.

[15] H. Singh, M. Conrad and S. Sadeghipour, "Test case design based on Z and the classification-tree method", in *Proceedings of the First IEEE International Conference on Formal Engineering Methods*, pp. 81–90, November 1997.

[16] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies", *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703–711, July 1991.