Postprint of article in *Proceedings of the 20th IASTED International Multi-Conference on Applied Informatics* (*AI 2002*), ACTA Press, Calgary, Canada, pp. 176–181 (2002)

A Study on Input Domain Partitioning *†

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

M. Y. Cheng
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
mycheng@cs.hku.hk

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

T. H. Tse
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
thtse@cs.hku.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

Partition testing is one of the widely used approaches in software testing. The efficacy of partition testing methods, such as the category-partition method and the classification-tree method, depends highly on the partitioning scheme, which groups together inputs that share some common characteristics. This paper reports case studies on the partitioning of the input domain from considerations of both the input and output perspectives, and discusses the relationships among the subdomains so formed.

Key Words

Software Testing, Partition Testing, Test Case Selection, Category-Partition Method, Classification-Tree Method

* © 2002 ACTA Press. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from ACTA Press.

† This research was supported in part by grants of the Research Grants Council of Hong Kong (Project nos. CityU 1118/99E and HKU 7029/01E), a grant of the City University of Hong Kong (Project no. 7000958), and a Research and Conference Grant of The University of Hong Kong.

‡ **Contact author.** Tel: +852 2788-9831. Fax: +852 2788-8614.

1. Introduction

Nowadays, software is not only used in simplifying and automating much of our daily routine work, but also used to perform increasingly complex and critical tasks such as in missile control systems, banking systems, and life-monitoring equipment. As the size and complexity of software systems grow, it is all the more important to ensure their quality and our confidence on these systems. To achieve this goal, one of the most widely used methods is software testing [1].

Typically, the number of possible inputs is enormous, and it is usually infeasible to test the program by feeding it with all possible inputs. The normal practice is to select only a limited number of elements from the input domain for testing. A central issue in software testing is how these elements are to be selected. One obvious choice is to do random testing, which selects test cases at random from the entire input domain [2]. However, it is often believed that a more "intelligent" way of searching for inputs that are likely to detect failures is more desirable [3].

Partition testing is a common approach for the selection of test cases [4]. The idea is to divide the input domain into subdomains using a partitioning scheme, and then select test cases from the subdomains according to a test allocation scheme [5]. The partitioning scheme is determined by the important aspects that are identified for testing, whereas the test allocation scheme governs the number of "representatives" to be chosen from the various subdomains.

The fault-detecting ability of partition testing depends mainly on the partitioning scheme; but given such a scheme, the allocation of test cases among the subdomains also affects the efficacy of partition testing. Interestingly,

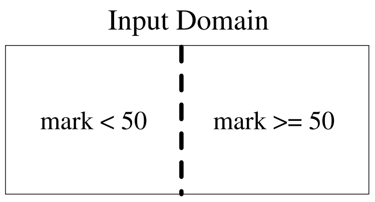

Figure 1. Input Domain for Example 1

although the effect of the partitioning scheme is usually considered more significant [3], recently there has been more work on the analysis of the effect of test allocation schemes [2–5].

This paper reports several case studies on the partitioning of the input domain. We observe that partitioning schemes are often based on the considerations of both input and output characteristics, and that there are some interesting relationships among the subdomains formed from the different partitioning schemes. We shall suggest some guidelines for testers on the selection of test cases by making use of these relationships.

Section 2 briefly reviews some essential concepts and terminology on partition testing. Section 3 reports case studies on the partitioning of the input domain from considerations of both the input and output perspectives. Section 4 discusses the relationships of the subdomains so formed and concludes this paper.

2. Partition Testing

2.1 Equivalence Partitioning

Partition testing is founded on the notion of equivalence partitioning. From the specification and based on his/her experience, the tester identifies important aspects for the purpose of testing as criteria for partitioning the input domain into subdomains. For each such criterion, as far as the aspect is concerned, test cases within the same subdomain are considered "equivalent". Let us consider a simple example.

Example 1 Suppose that a program receives an examination mark as input, and that the passing mark for the examination is 50. The tester may consider that "pass or fail" is an important aspect, and therefore handles the two situations separately: (1) mark < 50 and (2) mark ≥ 50. Accordingly, the input domain is divided into two parts (Figure 1). With respect to the aspect "pass or fail", any two inputs from the left subdomain (that is, any two marks that are less than 50) are "equivalent", and similarly for any two inputs from the right subdomain.

The division of the input domain into subdomains may be formalized into the notion of a partition. A *partition* of a set D is mathematically defined as a set of disjoint subsets of D, called *equivalence classes*, whose union is equal to D. Thus, each important aspect relevant for the purpose of testing induces a partition of the input domain, where an equivalence class corresponds to a subdomain consisting of "equivalent" elements (with respect to the important aspect).

2.2 Merging or Refining Subdomains

Suppose that the input domain is partitioned into k subdomains, and that the tester plans to select n test cases. If $n = k$, it is a common practice to select exactly one test case from each subdomain so that every subdomain will be tested once.

If $n < k$, then the tester may choose either to omit some subdomains, hence taking the risk that any failures within the omitted subdomains will have no chance of being detected, or to merge some of the subdomains into a larger subdomain so as to decrease the value of k. A third alternative is to increase the value of n to allow for more thorough testing if testing resources permit.

If $n > k$, then there are many possible schemes for allocating the n test cases into the k subdomains. Recently, there have been many studies on the fault-detecting ability of partition testing that uses different test allocation schemes [4, 5]. Regardless of the test allocation scheme chosen, there must be at least a subdomain from which more than one test case is selected. For such a subdomain, the tester may choose to refine the subdomain into several smaller subdomains, say, by incorporating more details in the partitioning criterion or using a different or additional criterion.

For instance, in Example 1, the original partitioning criterion is "pass or fail", giving two subdomains as shown in Figure 1. For the program under test, the tester may regard this aspect as one of primary concern. If only two test cases are allowed, he/she naturally selects one test case from each of these two subdomains. However, if the tester plans to select more test cases, he/she may like to incorporate some secondary criteria for guiding the selection of test cases. Suppose that a grade of A, B, or C is awarded, respectively, to a mark "≥ 80", "< 80 but ≥ 65", and "< 65 but ≥ 50". Then he/she may choose to refine the subdomain "mark ≥ 50" into smaller subdomains based on the "Grade" criterion. In such a situation, the smaller subdomains are said to be *refinement* of the original subdomain, and the new partition is said to be a *refinement* of the original partition.

In general, however, refining subdomains may or may not improve the fault-detecting ability of the testing [4]. Intuitively, refinement is useful insofar as it helps to focus more narrowly on the likely faults by incorporating extra information relevant to testing.

2.3 Combination of Equivalence Classes

In practice, a tester can often identify many important and relevant aspects for the purpose of testing. Correspondingly, each of these aspects may induce a different partition which divides the input domain in a different way. When selecting test cases, it is often necessary to consider the *combination of equivalence classes* from different partitions.

Ostrand and Balcer [1] proposed a systematic method, called the *category-partition method*, for combining equivalence classes (which they called "choices") from different partitions (which they called "categories"). Following up the ideas of Ostrand and Balcer, Grochtmann and Grimm [6] used a hierarchical structure called *classification tree* to represent the relations and constraints among different partitions (which they called "classifications") and equivalence classes (which they simply called "classes"). Chen *et al*. [7] have recently extended the method of Grochtmann and Grimm into an integrated classification-tree methodology. Let us consider the example used in [6] for illustrating the main ideas of the classification-tree method.

Example 2 Consider the program `COUNT` which inputs a list and an item, and then outputs a count of the occurrences of the item in the list. The important aspects identified from the specification include: (1) length of list, (2) whether the item is in the list when the list is a singleton, (3) how many times the item occurs in the list when the list has more than one element, and (4) in what way the list is ordered. Using the terminology of the classification-tree method, each of the aspects (1)–(4) is called a *classification*. For each classification, the input domain is divided into subdomains called *classes*. For instance, there are three classes associated with classification (1) "length of list", namely, "$= 0$", "$= 1$", or "> 1".

Note that classification (2) is only applicable to inputs within the class "$= 1$" of classification (1), and both classifications (3) and (4) only apply to the class "> 1" of classification (1). In [6, 7], this kind of relation is captured by representing the classifications and classes in the form of a tree. Instead of reproducing the tree here, let us describe how the input domain is subdivided due to the interaction of these classifications and classes. A diagrammatic view is provided in Figure 2 to facilitate understanding.

In Figure 2, the input domain is represented by the outermost rectangle. It is first divided into three subdomains (shown as three "columns" separated by two thick vertical bars) according to classification (1) "length of list". The subdomain corresponding to the second column is refined into two smaller subdomains (shown by the two parts separated by a thin horizontal line within the second column) according to classification (2). The third subdomain is refined by classification (3) into 3 columns and by classification (4) into 4 rows, separated by thin lines, giving $3 \times 4 = 12$ subdomains. Note that the subdomain corresponding to "length of list > 1; all elements identical; item occurs once" is in fact empty.

We observe that, in Example 2, classifications (1) and (4) are concerned with the characteristics of the input list, while classifications (2) and (3) are concerned with the characteristics of the output (count of item occurrences in the list). For ease of reference, the former type of classifications will be called "input-driven classifications" and the latter "output-driven classifications". Similarly, partitions, classes, and subdomains will also be qualified by the words "input-driven" or "output-driven" to indicate whether they are formed from consideration of the input or output characteristics.

It is obvious that the output-driven subdomains in Example 2 are actually refinement of the input-driven subdomains corresponding to the classes "$=$ 1" and "$>$ 1" of classification (1) "length of list". We shall next present several possible relationships among input-driven and output-driven subdomains.

3. Case Studies

To partition the input domain, it is natural that the tester will try to group or classify the inputs into disjoint subdomains from the characteristics of the inputs. The formation of such subdomains is therefore driven by the input characteristics. For instance, when the input involves a list of elements, it is likely that the tester will consider characteristics of the input such as the length of the list and the ordering of the elements within the list. However, it turns out that apart from the input characteristics, often the tester also makes use of the characteristics of outputs in trying to classify the *outputs*. In turn this leads to ways of grouping *inputs* that produce the corresponding classes of outputs. In Example 2, the tester classifies the output (the number of occurrences of the item in the list) into two or three classes, depending on whether the length of the list is equal to 1 or greater than 1, respectively. The inputs corresponding to each of these classes of outputs will be grouped together to form a subdomain. In this way, the partitioning of the input domain is driven by the characteristics of the output rather than the input.

In trying to understand the possible relationships among input-driven and output-driven classifications and classes, we have performed several case studies, which are to be presented as follows.

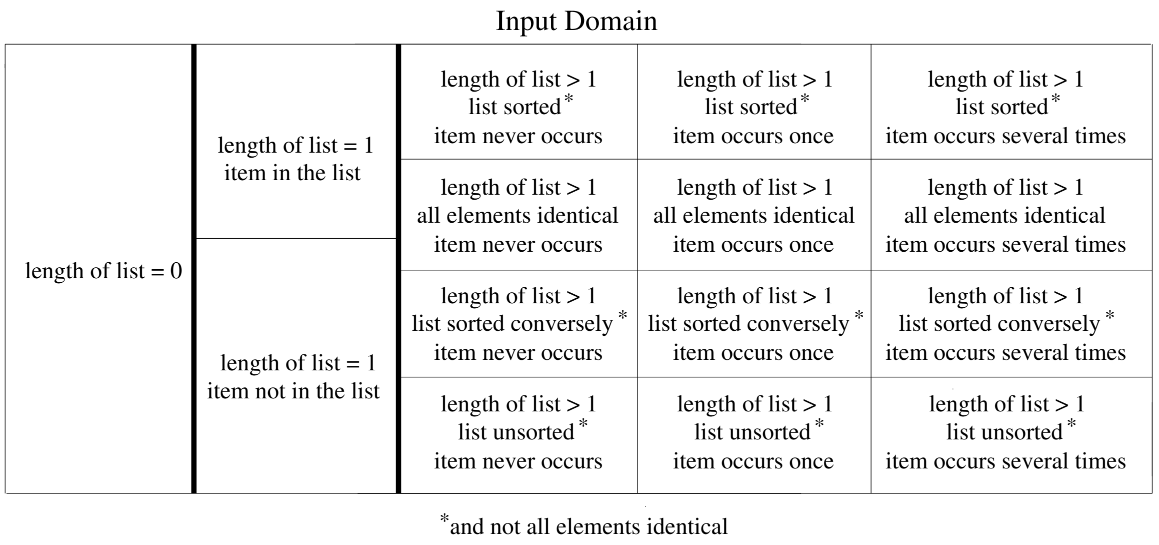

Figure 2. Input Domain for Example 2

Case 1: Postgraduate Recruitment

Consider a program that processes applications for admission to a postgraduate study programme. The inputs are the applicant's personal details, academic qualifications, occupation, and working experience. The program determines whether or not admission should be offered to the applicant. The rule is that an applicant will be given an offer if he/she possesses a relevant Masters degree, or if he/she possesses a relevant Bachelor degree, has a relevant occupation, and has 3 or more years of working experience.

For this program, the output is simply characterized by one classification with two classes: "application rejected" and "offer given". We shall label the subdomains corresponding to these two classes by o_1 and o_2, respectively. From the perspective of the inputs, however, one may identify the classifications and classes as shown below.

Classification	*Class*
Academic qualification	no relevant degree
	relevant Bachelor degree
	relevant Masters degree
Occupation	relevant
	irrelevant
Working experience	≥ 3 years
	< 3 years

Subdomains can be formed by selecting a class from each classification. Obviously, all such combinations of classes are feasible. Hence, there are $3 \times 2 \times 2 = 12$ input-driven subdomains as listed below.

(i_1) No relevant degree, relevant occupation, working experience ≥ 3 years

(i_2) No relevant degree, relevant occupation, working experience < 3 years

(i_3) No relevant degree, irrelevant occupation, working experience ≥ 3 years

(i_4) No relevant degree, irrelevant occupation, working experience < 3 years

(i_5) Relevant Bachelor degree, relevant occupation, working experience ≥ 3 years

(i_6) Relevant Bachelor degree, relevant occupation, working experience < 3 years

(i_7) Relevant Bachelor degree, irrelevant occupation, working experience ≥ 3 years

(i_8) Relevant Bachelor degree, irrelevant occupation, working experience < 3 years

(i_9) Relevant Masters degree, relevant occupation, working experience ≥ 3 years

(i_{10}) Relevant Masters degree, relevant occupation, working experience < 3 years

(i_{11}) Relevant Masters degree, irrelevant occupation, working experience ≥ 3 years

(i_{12}) Relevant Masters degree, irrelevant occupation, working experience < 3 years

According to the rule, if the inputs are from subdomains i_5, i_9, i_{10}, i_{11}, or i_{12}, the applicant will be given an offer. Hence, $o_2 = i_5 \cup i_9 \cup i_{10} \cup i_{11} \cup i_{12}$. Similarly, $o_1 = i_1 \cup i_2 \cup i_3 \cup i_4 \cup i_6 \cup i_7 \cup i_8$. Clearly, in this case, the input-driven subdomains are refinement of the output-driven subdomains. Figure 3(a) shows diagrammatically the relationship among the input-driven and output-driven subdomains.

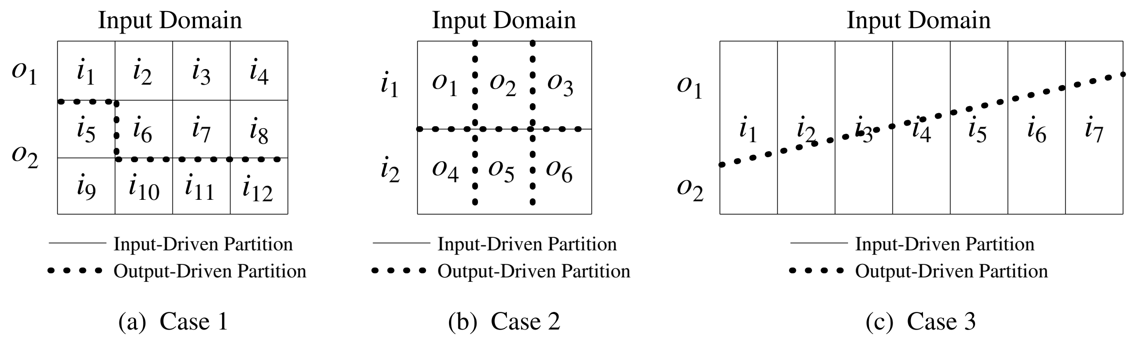

Figure 3. Three Cases of Input Domain Partitioning

Case 2: Search Engine

Search engine is one of the common web applications. Its simplest form involves the input of a keyword from the user, to be checked against an internal database for searching items that match the keyword. The output is a list of the matched items, if any.

From the consideration of the input alone, it is natural to group the inputs into two classes: keywords that are not found, and those that are found. We label the corresponding subdomains by i_1 and i_2.

Let N be the number of records in the database if it is present. From the consideration of the output, we may consider three classifications as follows.

Classification	*Class*
Database status	not present
	present
Content of database (if database present)	empty
	nonempty
Number of matched items found (if database nonempty)	$= 0$
	$= 1$
	> 1 and $< N$
	$= N$

Note that there are some inherent constraints among the classifications and classes, as indicated by the "if" clauses in the classifications. Taking these constraints into account, 6 output-driven subdomains can be formed:

(o_1) Database is not present.

(o_2) Database is present but empty.

(o_3) Database is present and nonempty, but no matched item is found.

(o_4) Database is present and nonempty, and one matched item is found.

(o_5) Database is present and nonempty, and m matched items are found, where $1 < m < N$.

(o_6) Database is present and nonempty, and N matched items are found.

Here the input-driven subdomain i_1 (corresponding to "keyword not found") can be refined into the three output-driven subdomains o_1, o_2, and o_3. In other words, $i_1 = o_1 \cup o_2 \cup o_3$. Similarly, $i_2 = o_4 \cup o_5 \cup o_6$. Hence, in this case, the output-driven subdomains are refinement of the input-driven subdomains. Figure 3(b) illustrates this scenario diagrammatically.

Case 3: Polynomial Equation

Consider a program that inputs the coefficients a, b, $\ldots$, g of the polynomial equation $ax^6 + bx^5 + cx^4 + dx^3 + ex^2 + fx + g = 0$ and outputs whether a real root exists or not.

From the consideration of the inputs alone, the input domain can be partitioned into 7 subdomains according to the degree of the equation as follows.

(i_1) $a \neq 0$ (degree 6)

(i_2) $a = 0$ and $b \neq 0$ (degree 5)

(i_3) $a = b = 0$ and $c \neq 0$ (degree 4)

(i_4) $a = b = c = 0$ and $d \neq 0$ (degree 3)

(i_5) $a = b = c = d = 0$ and $e \neq 0$ (degree 2)

(i_6) $a = b = c = d = e = 0$ and $f \neq 0$ (degree 1)

(i_7) $a = b = c = d = e = f = 0$ (degree 0)

From the consideration of the output, it is natural to consider the two classes: a real root exists, and no real root exists. The corresponding two output-driven subdomains are labeled by o_1 and o_2.

Note that a polynomial equation of even degree (say, 0, 2, 4, 6, and so on) may or may not have real roots. Thus, each of the input-driven subdomains i_1, i_3, i_5, and i_7 has non-empty intersection with both output-driven subdomains o_1 and o_2. Clearly, in this case, the input-driven subdomains are not refinements of the output-driven subdomains, nor vice versa.

In fact, even if a test case is selected from every input-driven subdomain, it is still possible that the output of "no real roots" will be left untested. Conversely, selecting a test case that produces the output "a real root exists" and another test case that produces "no real root exists" will leave some of the input-driven subdomains untested. The input-driven and output-driven partitions are therefore complementary to each other; it would be unsatisfactory to omit either of them. Figure 3(c) shows this case diagrammatically. Note that the boundary line that separates the two output-driven subdomains "cuts across" those that separate the input-driven subdomains. Here some intersections of the input-driven and output-driven subdomains are empty. For instance, $i_6 \cap o_2 = \emptyset$.

4. Discussions and Conclusion

The partition testing method requires some criteria for partitioning the input domain. Although the idea is to partition the input domain, the criteria may be determined from considerations of not only the input, but also the characteristics of the output. Our study has revealed three possible relationships among the input-driven and output-driven subdomains.

In Case 1, the input-driven subdomains are refinement of the output-driven subdomains. In such a situation, when testing resources permit, it is obvious that the tester should select test cases from the input-driven subdomains so that the testing can be as comprehensive as possible.

When the number of input-driven subdomains is too large, or when the testing resources are so tight that it is impossible to test all the subdomains, the tester may consider reducing the number of the subdomains by merging some of them, as explained in Section 2.2. If so, the tester may consider merging those input-driven subdomains that are refinement of the same output-driven subdomain, since from the output perspective these subdomains share the same characteristics. In other words, the output-driven subdomains may serve as useful guidelines for determining which subdomains to be merged.

Case 2 is an example of the output-driven subdomains being refinement of the input-driven subdomains. In such a situation, considering the characteristics of the inputs alone will probably render the testing inadequate, since inputs that have the same characteristics may actually produce different outputs, which intuitively should be covered as much as possible. Nevertheless, when the number of output-driven subdomains is so large that it is impossible to exhaustively cover them, the input-driven subdomains may also be used as guidelines for merging the output-driven subdomains.

More generally, there are cases such that neither the input-driven subdomains are refinement of the output-driven subdomains, nor vice versa, as exemplified in Case 3. In such a situation, the two partitions are complementary to each other. Omission of either partition is unsatisfactory as it risks the negligence of some important characteristics relevant for testing the program. Whenever possible, test cases should be selected from the intersections of the input-driven subdomains and output-driven subdomains to form a more comprehensive test suite.

Finally, even though when constructing an output-driven partition, the tester tries to group the *outputs* that are equivalent according to their characteristics, in the end he/she has to find the corresponding groups of *inputs* that will produce the desired class of outputs. This is because it is the inputs, not the outputs, that are eventually fed to the program for testing. Depending on the problem that the program purports to solve, the mapping to the inputs from the outputs is in general not a trivial task. Despite this, considering the important aspects from the perspective of the output is often a worthwhile pursuit that renders the testing more comprehensive and effective.

References

[1] T. J. Ostrand & M. J. Balcer, The category-partition method for specifying and generating functional tests, *Communications of the ACM*, *31*(6), 1988, 676–686.

[2] J. W. Duran & S. C. Ntafos, An evaluation of random testing, *IEEE Transactions on Software Engineering*, *10*(4), 1984, 438–444.

[3] D. Hamlet & R. Taylor, Partition testing does not inspire confidence, *IEEE Transactions on Software Engineering*, *16*(12), 1990, 1402–1411.

[4] E. J. Weyuker & B. Jeng, Analyzing partition testing strategies, *IEEE Transactions on Software Engineering*, *17*(7), 1991, 703–711.

[5] T. Y. Chen & Y. T. Yu, On the relationship between partition and random testing, *IEEE Transactions on Software Engineering*, *20*(12), 1994, 977–980.

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

[7] T. Y. Chen, P. L. Poon, & T. H. Tse, An integrated classification-tree methodology for test case generation, *International Journal of Software Engineering and Knowledge Engineering*, *10*(6), 2000, 647–679.