Classification-Hierarchy Table: A Methodology for Constructing the Classification Tree

T. Y. Chen and P. L. Poon
Department of Computer Science
University of Melbourne
Parkville 3052, Australia

Abstract

The Classification-Tree Method developed by Grochtmann and Grimm [4] provides a systematic approach to produce test cases based on the functional specification. This paper supplements the Classification-Tree Method by proposing a methodology to construct a classification-tree from a given set of classifications and classes based on the notion of Classification-Hierarchy Table. The Classification-Hierarchy Table is used to capture the hierarchy of a classification tree. From this table, useful information such as the relative level of the classifications in the tree, their parent classifications and parent classes could be obtained to guide the construction of the classification tree.

Keywords Category-Partition Method, Classification-Hierarchy Table, Selection of Test Data, Software Testing

1. Introduction

The primary objective of software testing is to reveal the presence (not the absence) of errors in a program. Although the level of confidence on the correctness of a program is increased if the results of testing are found to be consistent with the functional specification [2], it is important to know that undetected errors could still exist even after the most comprehensive and exhaustive testing. To maximize the chance of uncovering errors, therefore, software testing has to be well-planned, organized and performed.

One important activity during the planning stage of testing is the determination of *test cases* as this affects the kind and scope of testing and thus the quality of the test [3, 4]. To assist the test engineers to identify and organize the test cases effectively, Ostrand and Balcer [5] have developed the *Category-Partition Method* (later refined by Ammann and Offutt [1]) which provides a systematic approach to derive test cases from specifications. Based on the Category-Partition Method, Grochtmann and Grimm [4] have proposed the *Classification-Tree Method* so as to overcome some identified pitfalls of the former method. However, Grochtmann and Grimm did not provide a systematic approach to construct the classification trees from a given set classifications and corresponding classes. As a result, the classification trees constructed may vary from individual test engineers according to their own expertise and experience. This paper addresses this problem by introducing the *Classification-Hierarchy Table.*

The rest of this paper is structured as follows. Section 2 gives an overview of the Classification-Tree Method. Section 3 describes the methodology of the Classification-Hierarchy Table and Section 4 illustrates its application using an example. Finally, Section 5 concludes the whole paper.

2. The Classification-Tree Method

The basic idea of the Classification-Tree Method is to divide the input domain of the test object under different aspects and then to recombine the different partitions to form test cases. Its major procedures are as follow:

- According to the functional specification, the test engineer identifies various aspects which are relevant to the test for detailed consideration. These aspects should give a clear differentiation of all possible inputs of the test object. Each aspect identified is referred to as a *classification.*

0-8186-7635-3/96 $05.00

- For each classification, its input domain is then divided into disjoint subsets (referred to as *classes*).

- All the classifications and classes are then combined to form a hierarchical tree that is used as the head of a *combination table* from which all feasible and logical combinations of classes are marked. This marking procedure forms the basis for selecting test cases to be used for subsequent testing.

The original example used by Grochtmann and Grimm in their paper [4] is re-used here to illustrate the construction of a classification tree. Consider the following `COUNT` function:

```
COUNT (IN list: LIST OF ELEM, IN element:
       ELEM, OUT number: INT)
```

The input parameters of the `COUNT` function are the list of elements (`LIST OF ELEM`) and the element to be searched (`ELEM`), whereas the output parameter is the number of occurrence (`OUT`) of `ELEM` in `LIST OF ELEM`. Table 1 contains the classifications and classes used by Grochtmann and Grimm, and the corresponding classification tree and combination table are depicted in Figure 1.

It can be seen from Figure 1 that the classification tree has classifications and classes on alternate levels. In Figure 1, a column of the combination table represents an individual class which cannot be further subdivided and a row corresponds to a test case. For each row of the table, those classes which constitute a test case is marked. For example, row 1 represents a test case for an empty list, whereas row 4 refers to a test case for a list with more than one element sorted in an ascending order, and the element to be searched only occurs once in the list. During the marking process, the test engineers have to use their own judgement to ensure that the combination of classes is legitimate. For example, if the class "Once" under the classification "Element Occurs" is marked, then the class "All Elements Identical" under the classification "Sorting" should not be marked at the same time.

In this simple example, it is relatively straightforward to construct a classification tree. However, it would become difficult to construct a classification tree in a realistic testing which may possibly contain a large number of classifications and classes. Therefore, a methodology is required to construct the classification tree from a set of classifications and classes. This paper addresses this issue via the introduction of the Classification-Hierarchy Table.

Classifications	**Corresponding or Associated Classes**
Length of List	0, 1, > 1
Is Element Searched For#	Yes, No
Element Occurs@	Never, Once, Several Times
Sorting@	Sorted in Ascending Order, Sorted in Descending Order, Unsorted, All Elements Identical

The length of the list is assumed to be "1".
@ The length of the list is assumed to be "> 1".

Table 1. Classifications and Classes for the `COUNT` Function

3. Methodology of the Classification-Hierarchy Table

Prior to the discussion of the Classification-Hierarchy Table, let us formulate the problem first. Suppose there are m ($m \geq 1$) classifications denoted by X_i where $1 \leq i \leq m$. For each X_i, let Y_i denote its corresponding set of classes such that $Y_i = \{y_i^j : 1 \leq j \leq n_i\}$. The problem is how to construct a classification tree given X_i and Y_i, $1 \leq i \leq m$.

The problem of constructing the classification tree can be further decomposed into two subproblems:

- Given a pair of classifications (X_i and X_j), what are their relative positions in the classification tree?

- Given that X_i is the *parent classification* of X_j (a classification X_i is defined as the parent classification of X_j if the latter is "directly" underneath one of the classes of the former in the classification tree; and X_j is called a *child classification* of X_i), which class of X_i is X_j's *parent class* (a class y_i^k is

defined as the parent class of X_j if the latter is "directly" underneath the former in the classification tree)?

The fundamental idea of the Classification-Hierarchy Table is to capture the hierarchical relationships between every two classifications. For m classifications, the dimension of the table is obviously $m \times m$. Let us examine the Classification-Hierarchy Table in details to see how it can be used to construct the classification tree.

For a given pair of distinct classifications X_i and X_j, their hierarchical relationship, denoted as $X_i \rightarrow X_j$, has to be determined by the rules specified in Table 2. The symbols "$\Rightarrow$", "$\sim$" and "$\otimes$" in Table 2 are called *hierarchical operators*. Obviously, the conditions for "$\Rightarrow$", "$\sim$" and "$\otimes$" in Table 2 are mutually exclusive to each others. Thus, the hierarchical operator for a given pair of classifications is uniquely determined. For $X_i \rightarrow X_i$, the assigned operator is "$\otimes$". Intuitively, $X_i \Rightarrow X_j$ implies that X_i is an ancestor classification of X_j; and $X_i \sim X_j$ implies that X_i and X_j do not have a common ancestor class.

$X_i \rightarrow X_j$	**Rules**
$X_i \Rightarrow X_j$	$\exists\ y_i^k$ (where $1 \leq k \leq n_i$) such that when the class of X_i is y_i^k, all y_j^h's ($1 \leq h \leq n_j$) are feasible for X_j and $\exists\ y_i^k$ (where $1 \leq k \leq n_i$) such that when the class of X_i is y_i^k, none of y_j^h's ($1 \leq h \leq n_j$) are feasible for X_j
$X_i \sim X_j$	$\forall\ 1 \leq k \leq n_i$, when the class of X_i is y_i^k, none of y_j^h's ($1 \leq h \leq n_j$) are feasible for X_j
$X_i \otimes X_j$	Otherwise

Table 2. Types of Hierarchical Operators

In the process of determining the hierarchical relationship between two distinct classifications, all assumptions associated with these two classifications have to be considered. After the hierarchical relationship of every two distinct classifications is determined, then it is relatively straightforward to construct the classification tree from the Classification Hierarchy Table based on the following algorithm:

Tree Construction Algorithm

1. Construction of sub-trees for $X_i \Rightarrow X_j$

   All the table elements with the hierarchical operator "$\Rightarrow$" are identified from which one or more sub-trees can be formed. For example, suppose $X_i \Rightarrow X_j$. For every y_i^k such that when X_i's class is y_i^k, all y_j^h's are feasible for X_j, form a sub-tree with X_i as its root; y_i^k being X_i's child class; and X_j being y_i^k's child classification.

2. Consolidation of sub-trees

   (a) For every X_i, consolidate all sub-trees constructed in step (1) with X_i as its roots to form a new sub-tree so that its root is still X_i and its child classes are all the child classes of the sub-trees to be consolidated. If the same child class appears in more than one sub-tree before the consolidation process, this class should appear only once in the newly formed sub-tree. After the consolidation process, a set of sub-trees (with path $X_i - y_i^k - X_j$) is formed. Figure 2 depicts this consolidation process.

   (b) For every pair of paths of the forms $X_i - y_i^k - X_j$ and $X_i - y_i^k - X_k$, if there exist sub-trees with paths $X_j - y_j^f - X_{k_1}$, $X_{k_1} - y_{k_1}^g - X_{k_2}$, ..., $X_{k_n} - y_{k_n}^z - X_k$, then delete X_k in the sub-tree with root X_i.

   (c) For every pair of sub-trees with roots X_i and X_j such that if X_j appears as a terminal node in the sub-tree with root X_i, then the root X_j and the terminal node X_j are merged together as shown in Figure 3.

3. Identification of stand alone classifications

   For every X_i such that X_i does not appear in any sub-trees produced after step (2), form a sub-tree with X_i as its only node.

4. Construction of the classification tree

   Define a general root node (denoted by a small circle and represents the entire input domain of

the test) and attach all the sub-trees produced in steps (2) and (3) to this root node. This results in a tree with classifications as its terminal nodes. Then, for each classification in the tree, complete all its child classes.

4. An example

Having discussed the methodology of the classification-hierarchy table, an example will be used to illustrate the procedures for:

- constructing the classification-hierarchy table from a given set of classifications and their corresponding classes; and
- constructing the classification tree from the classification-hierarchy table.

Instead of using the previously mentioned `COUNT` function which is too simple to demonstrate all procedures of our methodology, a more complicated sales order processing (`SALES`) function is used. Following is the specification of the `SALES` function:

1. Input the customer number.
2. Check the customer number against the Customer Master File. If the customer number does not exist, reject the transaction. If the customer number exists and its status is deactivated, go to step (3). If the customer number exists and its status is activated, go to step (4).
3. Check whether the transaction is specially approved by management. If yes, accept the transaction without further checking. If no, reject the transaction.
4. Check the existence of the credit limit. If yes, go to step (5) otherwise reject the transaction.
5. If credit limit exists, confirm whether it is suspended or not. If not, go to step (6) otherwise reject the transaction.
6. If credit limit is not suspended, compare the credit limit and the invoice amount. If the credit limit is not less than invoice amount, accept the transaction; otherwise reject the transaction.

Suppose the classifications and their corresponding classes shown in Table 3 are identified.

Classifications	**Corresponding or Associated Classes**
Customer No. Exists	No, Yes and Deactivated, Yes and Activated
Credit Limit Exists#	Yes, No
Credit Limit Suspended@	Yes, No
Amount of Credit Limit@	$< 0, = 0, > 0$
Credit Limit > Invoice Amount&	$<, =, >$
Invoice Amount	$< 0, = 0, > 0$
Management Approval*	Yes, No

\# Assume the existence of an activated Customer Number.

@ Assume the existence of an activated Customer Number and Credit Limit.

& Assume the existence of an activated Customer Number and Credit Limit. Also, the Credit Limit is not suspended.

* Assume the existence of a deactivated Customer Number.

Table 3. Classifications and Classes for the `SALES` Function

Let t_{ij} denote the element at the ith row and the jth column in Table 4 which is the corresponding classification-hierarchy table.

	Customer No. Exists (No, Yes and Deactivated, Yes and Activated)	**Credit Limit Exists#** (Yes, No)	**Credit Limit Suspended@** (Yes, No)	**Amount of Credit Limit@** (<0, =0, >0)	**Credit Limit > Invoice Amount&** (<, = , >)	**Invoice Amount** (<0, =0, >0)	**Management Approval*** (Yes, No)
Customer No. Exists (No, Yes and Deactivated, Yes and Activated)	⊗	⇒	⇒	⇒	⇒	⊗	⇒
Credit Limit Exists# (Yes, No)	⊗	⊗	⇒	⇒	⇒	⊗	~
Credit Limit Suspended@ (Yes, No)	⊗	⊗	⊗	⊗	⇒	⊗	~
Amount of Credit Limit@ (<0, =0, >0)	⊗	⊗	⊗	⊗	⊗	⊗	~
Credit Limit > Invoice Amount& (<, =, >)	⊗	⊗	⊗	⊗	⊗	⊗	~
Invoice Amount (<0, =0, >0)	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Management Approval* (Yes, No)	⊗	~	~	~	~	⊗	⊗

Assume the existence of an activated Customer Number.

@ Assume the existence of an activated Customer Number and Credit Limit.

& Assume the existence of an activated Customer Number and Credit Limit. Also, the Credit Limit is not suspended.

* Assume the existence of a deactivated Customer Number.

Table 4. Classification-Hierarchy Table for the SALES Function

For t_{12}, the hierarchical operator is "$\Rightarrow$" because:

- when the class of the classification "Customer No. Exists" is "Yes and Activated", all classes ("Yes" and "No") of the classification "Credit Limit Exists" are feasible; and
- when the classes of the classification "Customer No. Exists" are "No" and "Yes and Deactivated", none of the classes ("Yes" and "No") of the classification "Credit Limit Exists" are feasible.

Using similar arguments, t_{13}, t_{14}, t_{15}, t_{17}, t_{23}, t_{24}, t_{25} and t_{35} also have the hierarchical operator "$\Rightarrow$".

For t_{72}, the hierarchical operator is "$\sim$" because when the classification "Management Approval" is either "Yes" or "No", none of the classes ("Yes" and "No") for the classification "Credit Limit Exists" are feasible (as "Management Approval" assumes the customer number exists and is deactivated while "Credit Limit Exists" assumes the customer number exists and is activated). Similar argument applies to t_{27}, t_{37}, t_{47}, t_{57}, t_{73}, t_{74} and t_{75}.

Thus, all remaining entries have the hierarchical operator "$\otimes$".

Now, let us illustrate how to construct the classification tree using the Tree Construction Algorithm mentioned in Section 3:

1. Construction of sub-trees for $X_i \Rightarrow X_j$

   As pointed out above, the table elements with the hierarchical operator "$\Rightarrow$" are t_{12}, t_{13}, t_{14}, t_{15}, t_{17}, t_{23}, t_{24}, t_{25} and t_{35}. Since t_{12} is "Customer No. Exists" $\Rightarrow$ "Credit Limit Exists", its corresponding sub-tree has "Customer No. Exists" as its root and "Yes and Activated" as its child class; and "Credit Limit Exists" as the child classification of "Yes and Activated".

   Sub-trees for t_{13}, t_{14}, t_{15}, t_{17}, t_{23}, t_{24}, t_{25} and t_{35} can be formed in a similar way. Some of these sub-trees are depicted in Figure 4.

2. Consolidation of sub-trees

   In this case, all the sub-trees produced in step (1) are eventually consolidated to form one single new sub-tree (Figures 5, 6 and 7 refer).

3. Identification of stand alone classifications

   In Table 4, only the classification "Invoice Amount" does not appear in any sub-trees produced in step (2). Therefore, "Invoice Amount" should form a sub-tree with itself as the only node.

4. Construction of the classification tree

   All the sub-trees produced in steps (2) and (3) are connected to a general root node. Also, all the possible classes for each classification are incorporated to complete the classification tree (Figure 8 refers).

5. Conclusions and future work

The Classification-Tree Method developed by Grochtmann and Grimm [4] provides a systematic approach for the test engineers to generate test cases from the functional specification. However, their approach of constructing the classification tree is an ad hoc one. As a result, the classification trees constructed may vary from individual test engineers as the construction process relies largely on the expertise and experience of the test engineers to assemble the classifications and corresponding classes together.

Our investigation aims at enhancing their method by providing a methodology to construct the classification tree in a systematic way. This objective is achieved via the use of the Classification-Hierarchy Table which is defined to capture the hierarchical relationships between all pairs of classifications. This table forms the basis for our algorithm to construct classification tree.

A prototype is now being developed to automate the construction of the classification-hierarchy table and the corresponding classification tree. Once this prototype is available, then we can measure the effectiveness of our methodology.

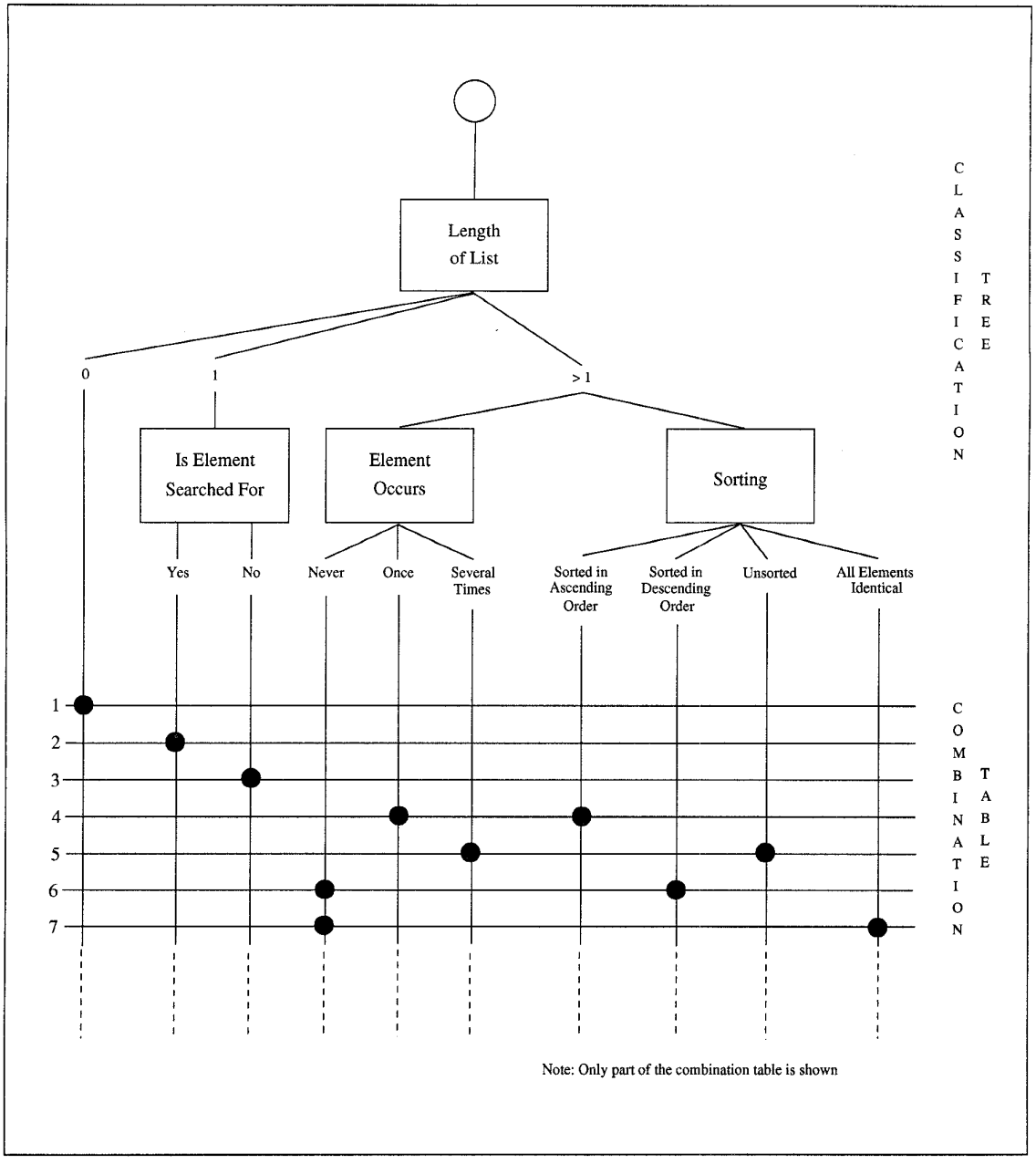

Figure 1. The Classification Tree and the Combination Table for the `COUNT` Function

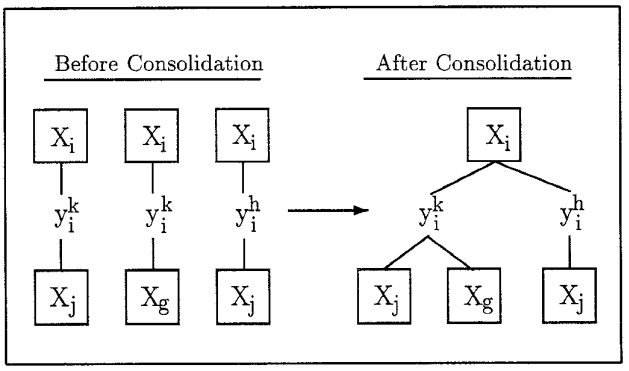

Figure 2. Consolidation of Sub-Trees for Step (2a)

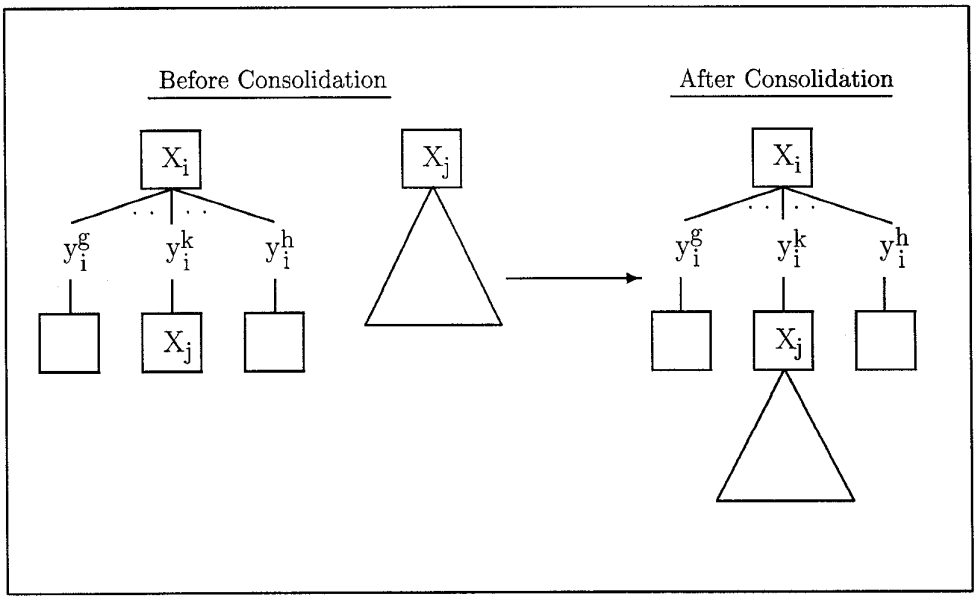

Figure 3. Consolidation of Sub-Trees for Step (2c)

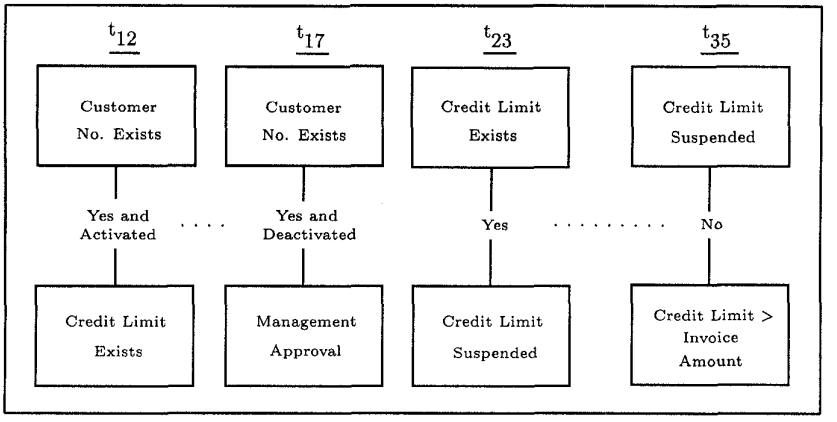

Figure 4. Some of the Sub-Trees Produced in Step (1)

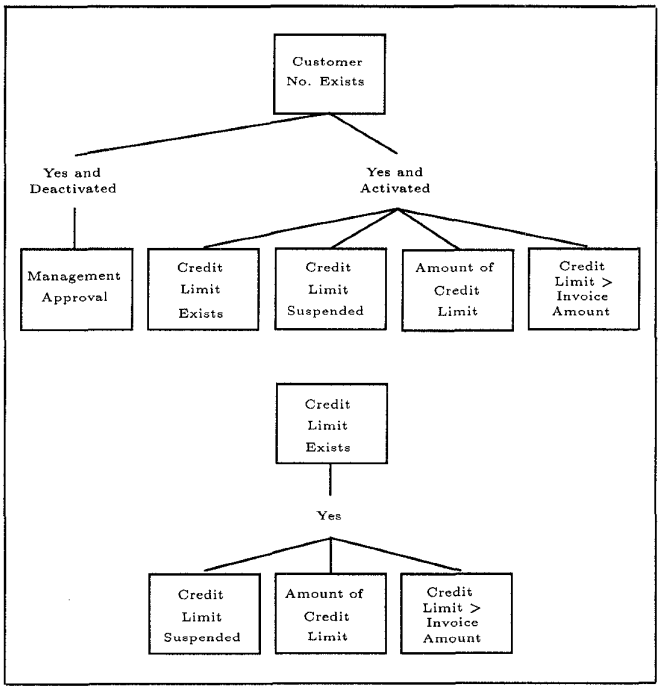

Figure 5. Consolidation of Sub-Trees for Step (2a)

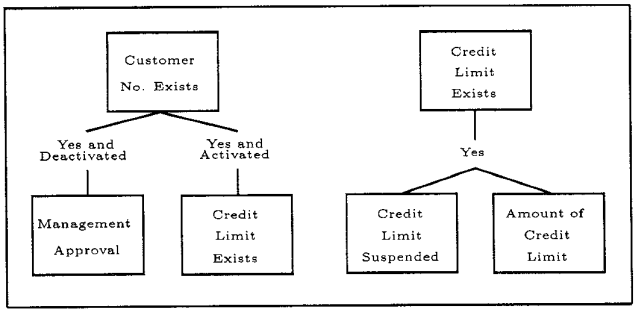

Figure 6. Consolidation of Sub-Trees for Step (2b)

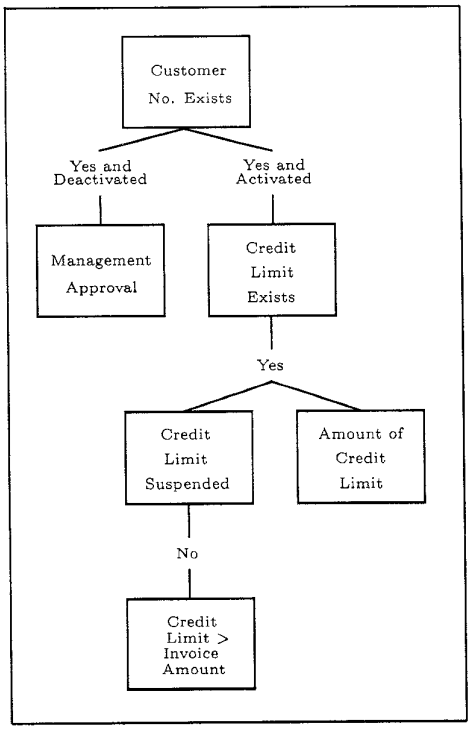

Figure 7. Consolidation of Sub-Trees for Step (2c)

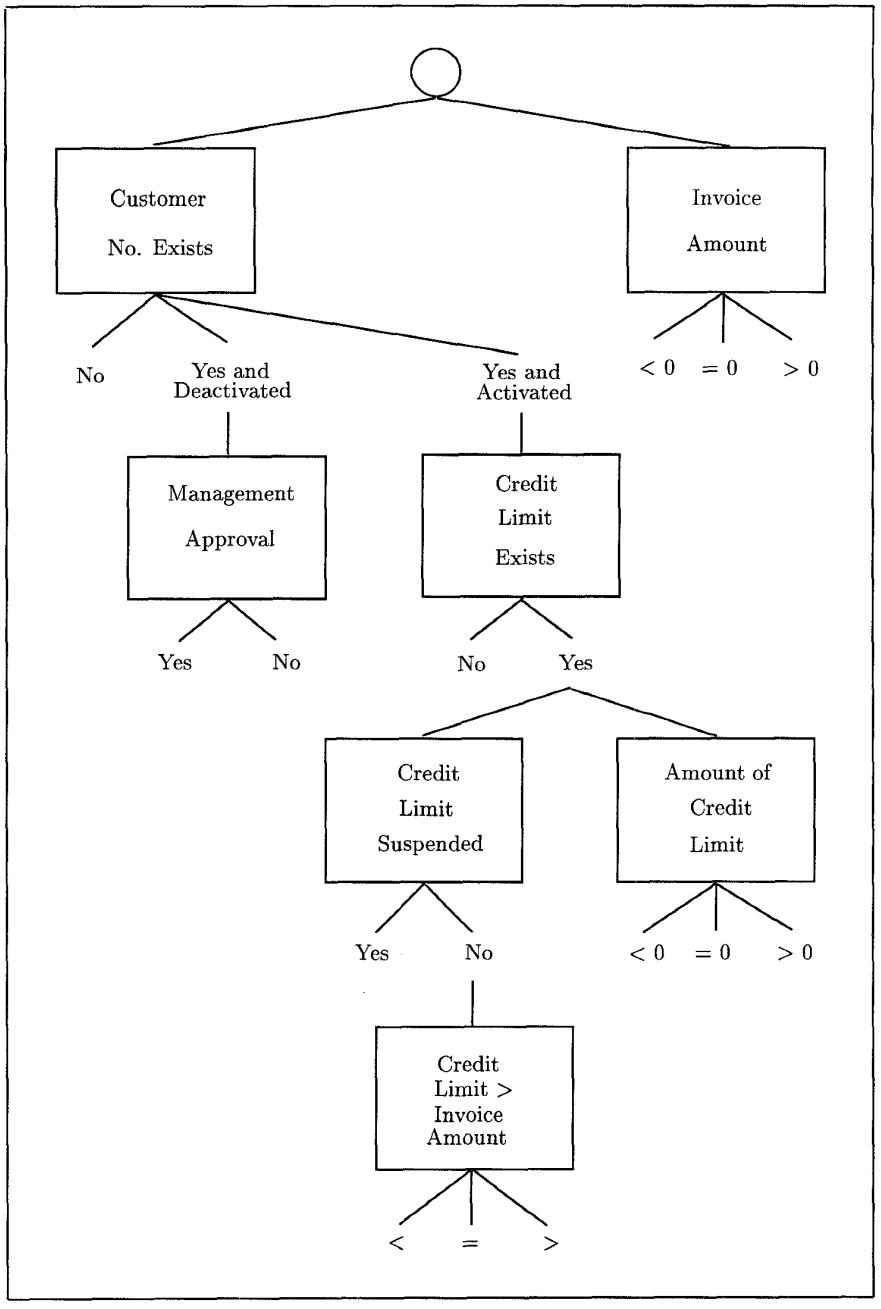

Figure 8. The Classification Tree for the SALES Function

References

[1] P. Ammann and J. Offutt. Using formal methods to derive test frames in Category-Partition Testing. *COMPASS '94: Proceedings of the Ninth Annual Conference on Computer Assurance, Safety, Reliability, Fault Tolerance, Concurrency and Real Time Security*, pages 69–79, 1994.

[2] R. Bache and M. Müllerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, pages 86–92, March 1990.

[3] T. Chusho. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, 13(5):509–517, May 1987.

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

[5] T.J. Ostrand and M.J. Balcer. The Category-Partition Method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.