To appear in *International Journal of Software Engineering and Knowledge Engineering*

An Integrated Classification-Tree Methodology for Test Case Generation *†

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, Hong Kong
acplpoon@inet.polyu.edu.hk

T.H. Tse §¶
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Abstract

This paper describes an integrated methodology for the construction of test cases from functional specifications using the classification-tree method. It is an integration of our extensions to the classification-hierarchy table, the classification tree construction algorithm, and the classification tree restructuring technique. Based on the methodology, a prototype system ADDICT*, which stands for **A**utomate**D** test **D**ata generation system using the **I**ntegrated **C**lassification-**T**ree method, has been built.*

Keywords: *Black Box Testing, Classification-Hierarchy Table, Classification Tree, Test Data Generator, Test Case Selection*

† This research is supported in part by the Hong Kong Research Grants Council.

‡ Part of the work was carried out when Chen was with the Vocational Training Council, Hong Kong.

§ **Contact author.**

¶ Part of the work was carried out when Tse was on leave at the Vocational Training Council, Hong Kong.

1 Introduction

Software testing is the most commonly used technique to reveal the presence (albeit not the absence) of faults in software. Even if testing does not reveal any fault, it still provides more confidence in the correctness of the software [3, 16, 21]. In relation to this, Goel [11] and Musa [17, 18] proposed to guide software testing by means of an operational profile, which is a quantitative characterization of how the software will be used. In this way, the most frequently used functions of the software will receive the most testing and hence the reliability level will be maximized within the testing constraints such as budget and time. In addition, Iyer and Lee [15] pointed out that the operational profile should be generated when the software is operational rather than in its development phase, so as to further improve on the reliability. We should note, however, that undetected faults may exist even after extensive testing has been performed. In order to maximize the chance of uncovering faults, testing should be well planned, organized, and exercised.

A critical component of testing is the construction of *test cases*, since this has a direct impact on the scope and therefore the comprehensiveness of testing [2, 4, 10, 12, 14, 20]. The importance of test case construction motivated Ostrand and Balcer to develop the *category-partition method* [4, 20] in order to assist software testers to construct test cases effectively from the *functional specifications* (referred to as the "specifications" in this paper). Numerous studies have been performed by other researchers [1, 2, 12, 19] based on Ostrand and Balcer's work. Among these, Grochtmann and Grimm [12, 13] extended the concept of the category-partition method, resulting in their *classification-tree method*. This method helps the identification of test cases via the construction of *classification trees*. However, their tree construction method is rather ad hoc. This results in the variation of the classification trees constructed from one software tester to the next, according to his/her personal experience and expertise.

This problem was later alleviated by Chen and Poon via the notion of the *classification-hierarchy table* [8]. The table helps construct classification trees by capturing the hierarchical relation for each pair of classifications. They also identified some properties of the *hierarchical operators* used for the construction of the classification-hierarchy table [8]. Subsequently, they observed that there is an effectiveness aspect associated with a classification tree [7]. They argued that the quality of a classification tree should be determined by its effectiveness in identifying the set of *legitimate test cases*. Based on this observation, they defined a metric to measure the effectiveness, and proposed a tree restructuring technique to improve on the quality.

We propose that Chen and Poon's classification-tree construction and restructuring methods can be further improved using an integrated approach. The following are the major features of the new approach:

(a) We have developed techniques for consistency checking and automatic deduction of hierarchical operators. With these, the correctness and efficiency of constructing the classification-hierarchy table can be improved.

(b) We have enhanced the classification-hierarchy table in order to build an effective test data generation system.

(c) We have designed an algorithm that integrates the tree construction process with the tree restructuring process. As a result, the effectiveness of the classification trees can be improved without resorting to a separate restructuring process as in the earlier approach [7].

This paper presents the integrated classification-tree methodology and illustrates its feasibility through a prototype system ADDICT, which stands for **A**utomate**D** test **D**ata generation system using the **I**ntegrated **C**lassification-**T**ree method.

The integrated classification-tree methodology is a black-box testing technique. It refers to program testing based on software specifications whereas a white-box technique refers to that based on information from the source code of the developed systems. It has been shown [5] that neither black-box techniques nor white-box techniques are sufficient for comprehensive software testing. Readers interested in white-box techniques may like to consult our other papers such as [5].

The rest of this paper is structured as follows. Section 2 gives an overview of the classification-hierarchy table, the tree construction algorithm, and the tree restructuring technique. Section 3 discusses our integrated methodology for the construction of classification trees by extending and incorporating the classification-hierarchy table, the tree construction algorithm, and the tree restructuring technique developed by Chen and Poon [7, 8]. Section 4 describes the major features of a prototype system ADDICT. Finally, Section 5 concludes the whole paper.

2 Previous Work on Classification Trees

2.1 Grochtmann and Grimm

By extending Ostrand and Balcer's category-partition method [4, 20], Grochtmann and Grimm [12, 13] developed the classification-tree method in order to assist software testers to construct test cases from specifications via the construction of classification trees. They define *classifications* as the different criteria for partitioning the input domain of the program to be tested, and *classes* as the disjointed subsets of values for each classification. A classification tree organizes the classifications and classes into a hierarchical structure according to the specification. Consider, for example, a program that calculates the sum of the square roots of two real numbers M and N. The numbers M and N are two possible classifications and each of them has three possible classes "< 0", "$= 0$", and "> 0". Consider another example where a program calculates the square root of the sum of two real numbers M and N. We can have one classification "$M+N$" with three possible classes "< 0", "$= 0$", and "> 0".

Although Grochtmann and Grimm's classification-tree method is effective for the construction of test cases from classification trees, the construction of the trees themselves may be difficult, since it is only based on ad hoc techniques.

2.2 Chen and Poon

The problem in the original classification-tree method triggered Chen and Poon [8] to develop a methodology for the construction of classification trees from the given sets of classifications and their associated classes via the notion of a classification-hierarchy table.

The intuition of the classification-hierarchy table is to capture the hierarchical relation for each pair of classifications. Suppose there are w classifications. The dimension of the classification-hierarchy table is $w \times w$.

Classifications are defined formally as sets of associated classes. Thus, classifications are denoted by letters in upper case, and classes by letters in lower case. For example, let X be a classification and x and x'

be the associated classes. Then we write $X = \{x, x'\}$. When the classification X takes the class x, we write $class(X) = x$.

Given a pair of classifications X and Y, their hierarchical relation (denoted by $X \mapsto Y$) was defined by Chen and Poon in terms of one of the hierarchical operators "$\Rightarrow$", "$\sim$", and "$\otimes$" as follows:

(1) X is said to be an *ancestor* of Y, denoted by $X \Rightarrow Y$, if and only if the following conditions are satisfied:

  (a) There exist some $x \in X$ and $y \in Y$ such that $class(X) = x$ and $class(Y) = y$ are part of a legitimate input.

  (b) There exists some $x' \in X$ such that, for any $y \in Y$, we cannot have $class(X) = x'$ and $class(Y) = y$ in any legitimate input.

(2) X is said to be *incompatible with* Y, denoted by $X \sim Y$, if and only if for any $x \in X$ and $y \in Y$, we cannot have $class(X) = x$ and $class(Y) = y$ in any legitimate input.

(3) X is said to have *other relations with* Y, denoted by $X \otimes Y$, if and only if X is neither an ancestor of Y nor incompatible with it.

Since the conditions for "$\Rightarrow$", "$\sim$", and "$\otimes$" are exhaustive and mutually exclusive, the hierarchical operator for $X \mapsto Y$ is well defined. It should be noted that the hierarchical operator for $X \mapsto X$ is "$\otimes$".

Some properties of the hierarchical operators are as follows:

- **Property 1:** If $X \Rightarrow Y$, then $Y \otimes X$.
- **Property 2:** If $X \sim Y$, then $Y \sim X$.
- **Property 3:** If $X \otimes Y$, then $Y \Rightarrow X$ or $Y \otimes X$.

The proofs of these properties are straightforward.[1]

After determining the hierarchical relations for all pairs of classifications, the classification tree can be constructed using Chen and Poon's tree construction algorithm. The algorithm comprises the following steps:

(1) Construct subtrees using the *parent-child* or *ancestor-descendent* hierarchical relation. For the parent-child relation, a classification is directly placed under one or more classes of another classification. For the ancestor-descendent relation, a classification is indirectly placed under one or more classes of another classification.

(2) Rearrange the related subtrees formed in step (1).

(3) Construct subtrees for stand alone classifications.

(4) Integrate all the subtrees formed in steps (2) and (3) to produce the final classification tree.

Please refer to [8] for details.

[1] These proofs assume a constraint in Grochtmann and Grimm's classification-tree method. See Section 3.2 for details.

Example 1
Suppose a software tester is given the following specification of a program *arith-sum*:

(1) *arith-sum* has nine input variables A, B, C, D, E, F, G, H, and I.

(2) H has three possible values (denoted by h_1, h_2, and h_3), whereas each of the remaining variables has two possible values (denoted, for example, by a_1 and a_2 for A).

(3) The input domain of *arith-sum* may contain any combination of possible values from some of these variables, except the following:

- (i) (A is a_2) and (B is b_1 or b_2)
- (ii) (A is a_2) and (C is c_1 or c_2)
- (iii) (A is a_2) and (D is d_1 or d_2)
- (iv) (A is a_1) and (E is e_1 or e_2)
- (v) (B is b_2) and (C is c_1 or c_2)
- (vi) (B is b_2) and (D is d_1 or d_2)
- (vii) (B is b_1 or b_2) and (E is e_1 or e_2)
- ($viii$) (C is c_2) and (D is d_1 or d_2)
- (ix) (C is c_1 or c_2) and (E is e_1 or e_2)
- (x) (C is c_1 or c_2) and (F is f_2)
- (xi) (C is c_2) and (G is g_2)
- (xii) (C is c_1 or c_2) and (H is h_1, h_2, or h_3)
- ($xiii$) (D is d_1 or d_2) and (E is e_1 or e_2)
- (xiv) (D is d_1 or d_2) and (F is f_2)
- (xv) (D is d_1 or d_2) and (H is h_1, h_2, or h_3)
- (xvi) (E is e_1 or e_2) and (G is g_1 or g_2)
- ($xvii$) (F is f_2) and (G is g_1 or g_2)
- ($xviii$) (F is f_1) and (H is h_1, h_2, or h_3)
- (xix) (G is g_1 or g_2) and (H is h_1, h_2, or h_3)

(4) *arith-sum* calculates the arithmetic sum of those variables entered.

Suppose we simply define the classifications as the input variables and the associated classes as the possible values. For example, A is taken as a classification with a_1 and a_2 as its two associated classes. Then Table 1 shows the classification-hierarchy table for *arith-sum*.

Let t_{ij} denote the element at the ith row and the jth column of Table 1. The hierarchical operator for t_{12} is "$\Rightarrow$" because

	A	B	C	D	E	F	G	H	I
A	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
B	$\otimes$	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\sim$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
C	$\otimes$	$\otimes$	$\otimes$	$\Rightarrow$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
D	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
E	$\otimes$	$\sim$	$\sim$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$
F	$\otimes$	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\otimes$	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\otimes$
G	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
H	$\otimes$	$\otimes$	$\sim$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$
I	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$

Table 1: Classification-Hierarchy Table for *arith-sum*

- any legitimate input that contains $class(A) = a_1$ also contains $class(B) = b_1$ or b_2, and
- any legitimate input that contains $class(A) = a_2$ does not contain $class(B) = b_1$ or b_2.

By similar reasoning, the hierarchical operator "$\Rightarrow$" is also applicable to t_{13}, t_{14}, t_{15}, t_{23}, t_{24}, t_{34}, t_{63}, t_{64}, t_{67}, and t_{68}.

The hierarchical operator for t_{25} is "$\sim$" because any legitimate input that contains $class(B) = b_1$ or b_2 cannot contain $class(E) = e_1$ or e_2. By similar arguments, the hierarchical operator "$\sim$" is also applicable to t_{35}, t_{38}, t_{45}, t_{48}, t_{52}, t_{53}, t_{54}, t_{57}, t_{75}, t_{78}, t_{83}, t_{84}, and t_{87}.

Obviously, the hierarchical operator of all the remaining elements is "$\otimes$".

From Table 1, the corresponding classification tree (denoted by $\mathcal{T}_{arith-sum}$ in Figure 2.2) can then be produced using Chen and Poon's tree construction algorithm. ■

2.3 Test Case Construction Technique and Effectiveness of Classification Trees

A small circle at the top of a classification tree, as shown in Figure 2.2, is the *general root node*. It represents the whole input domain. The classifications directly under the general root node, such as A, F, and I in Figure 2.2, are called the *top-level classifications*. In general, a classification X may have a number of classes x directly under it. X is known as the *parent classification* and x is known as a *child class*. In Figure 2.2, for example, A is the parent classification of a_1 and a_2, whereas a_1 and a_2 are the child classes of A. Similarly, a class x may have a number of classifications Y ($\neq X$) directly under it. Then x is known as the *parent class* and Y is known as a *child classification*. In Figure 2.2, for example, f_1 is the parent class of C and G, whereas C and G are the child classifications of f_1. From the classification tree, test cases can be expressed in the *test case table* using the following steps:

(1) Draw the grids of the test case table under the classification tree. The columns of the table correspond to the terminal nodes of the classification tree. The rows correspond to potential test cases.

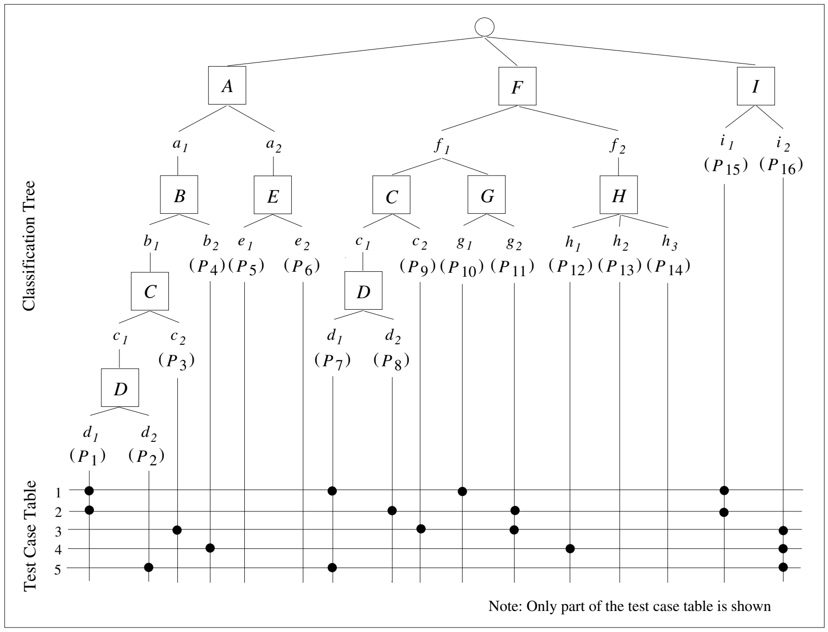


Figure 1: $\mathcal{T}_{arith-sum}$ and Part of its Test Case Table

(2) Construct a test case in the test case table by selecting a combination of classes in the classification tree as follows:

(a) Select one and only one child class of each top-level classification.

(b) For every child classification of each selected class, recursively select one and only one child class.

A test case constructed in this manner is known as a *potential test case*. For example, row 3 of the test case table in Figure 2.2 represents a potential test case for which $class(A) = a_1$, $class(B) = b_1$, $class(C) = c_2$, $class(F) = f_1$, $class(G) = g_2$, and $class(I) = i_2$.

Each path from the general root node of the classification tree to a terminal node is known as a *feasible path*. Given n terminal nodes in a classification tree, we will use P_i (where $1 \leq i \leq n$) to denote a feasible path. For example, P_3 in Figure 2.2 denotes the feasible path A—a_1—B—b_1—C—c_2. Obviously, every potential test case constructed from the classification tree corresponds to a set of feasible paths. For instance, the potential test case in row 3 corresponds to the set of feasible paths $\{P_3, P_9, P_{11}, P_{16}\}$.

Occasionally, some constraints among the classifications may not be reflected by a classification tree. Hence, all the potential test cases expressed in the test case table have to be checked against the specification, in order to identify and remove those not complying with the specification. The potential test cases removed due to such inconsistencies are referred to as *illegitimate test cases*, whereas those remaining after the removal process are referred to as *legitimate test cases*. For example, rows 3 and 5 of the test case table in Figure 2.2 represent two illegitimate test cases, since the former contains both c_2 and g_2 (which contradicts constraint $(3)(xi)$ of the specification), and the latter contains both d_1 and d_2 (which should be disjointed).

Only part of the test case table for *arith-sum* is shown in Figure 2.2. The complete test case table produces a total of 108 potential test cases. Out of these, 80 are found to be illegitimate after checking with the specification. Hence, only 28 legitimate test cases should remain for subsequent testing.

Since the ultimate purpose for the construction of a classification tree $\mathcal{T}$ is to generate a set of legitimate test cases, Chen and Poon [7] argued that the quality of $\mathcal{T}$ is closely related to the effectiveness in identifying such test cases. Given N_l legitimate test cases and N_p potential test cases, they define the *effectiveness metric* as

$$E_{\mathcal{T}} = \frac{N_l}{N_p}$$

For example, $E_{\mathcal{T}_{arith-sum}}$ is calculated to be $\frac{28}{108} = 0.26$. Obviously, a small value of $E_{\mathcal{T}}$ is undesirable since more effort is required to identify all the illegitimate test cases. Furthermore, the manual process of identifying illegitimate test cases is more prone to human errors when N_p is large. This may in turn affect the comprehensiveness (and hence the quality) of testing if some legitimate test cases are somehow mistakenly classified as illegitimate and hence not being used in testing.

2.4 Classification Tree Restructuring Technique

It would not be sufficient just to know the value of the effectiveness metric $E_{\mathcal{T}}$. The key idea is to improve on $\mathcal{T}$ whenever possible. In [7], Chen and Poon observed that a major reason for the occurrence of illegitimate test cases is the duplication of subtrees (or classifications) under different top-level classifications in $\mathcal{T}$. In $\mathcal{T}_{arith-sum}$ of Figure 2.2, for instance, the classification D is duplicated under the top-level classifications A and F. As a result, both the disjointed classes d_1 and d_2 may be selected in a single test case, leading to a contradiction and hence illegitimacy.

Based on this observation, Chen and Poon [7] developed a tree restructuring technique for the reduction of illegitimate test cases. This is done by suppressing the occurrence of duplicated subtrees under different top-level classifications. However, their tree restructuring technique may sometimes introduce incompatible classes, thereby converting some legitimate test cases into illegitimate ones. After the restructuring process, therefore, a reformatting procedure has to be performed to convert these illegitimate test cases into legitimate ones. Readers may refer to [7] for details.

3 An Integrated Classification-Tree Methodology

3.1 An Overview

In view of the significance of the above techniques, we propose an integrated methodology that supports (a) the construction of the classification-hierarchy table, (b) the construction of the classification tree, (c) the restructuring of the classification tree, and (d) the construction of the set of potential test cases. Some of these techniques have been extended or combined to allow for a full integration. A prototype system has been developed.

Basically, our integrated classification-tree methodology consists of the following three phases:

(1) Construction of Classification-Hierarchy Table

A constraint of the classification-tree method is that the parent-child or ancestor-descendent hierarchical relation must be anti-symmetric for any pair of classifications. In order to facilitate the detection of symmetric parent-child or ancestor-descendent hierarchical relations between any pair of classifications, we propose to refine the original set of hierarchical operators in [8].

Obviously, the effort of defining all the $w \times w$ hierarchical relations is substantial, especially when w is large. In order to improve on the correctness and efficiency of constructing the classification-hierarchy table, we have developed techniques for (a) the consistency checking of known hierarchical relations and (b) the automatic deduction of new hierarchical relations from the known ones. In additional, Chen and Poon's classification-hierarchy table [8] has been enhanced to make the subsequent tree construction phase more efficient.

(2) Construction of Classification Tree

A suboptimization process for the effectiveness metric $E_{\mathcal{T}}$ has been built into the construction of the classification tree, so that the quality of the latter can be improved. To accomplish this, Chen and Poon's tree construction algorithm [8] has been extended substantially.

(3) Construction of Potential Test Cases

Because of the suboptimization process in step (2), there is no need for tree restructuring as proposed in [7]. As a result, the production of potential test cases from the classification tree has become a straightforward process. Hence, minor details for this part of the methodology will not be presented in this paper.

3.2 Construction of Classification-Hierarchy Table

A constraint of Grochtmann and Grimm's classification-tree method [12, 13], even though it is not discussed in the paper, is that the parent-child or ancestor-descendent hierarchical relation must be *anti-symmetric* for any pair of classifications. Otherwise a classification tree cannot be constructed. In other words, $X \Rightarrow Y$ must imply $Y \not\Rightarrow X$. Software testers may need to redefine the original set of classifications and classes in order to meet this constraint while preserving the requirements of the target system. The following example illustrates this point.

Example 2

Suppose we are given two classifications M and N, where M is associated with two classes m_1 and m_2, and N is associated with another two classes n_1 and n_2. Suppose, further, that there are only three legitimate inputs:

(a) $class(M) = m_1$

(b) $class(N) = n_1$

(c) $class(M) = m_2$ and $class(N) = n_2$

Obviously, M or N or both must be top-level classifications. In this situation, only three types of classification trees are possible:

- **Type (1):** Classification trees having both M and N as their top-level classifications.
- **Type (2):** Classification trees having M as their only top-level classification, and N as child classification(s) of M.
- **Type (3):** Classification trees having N as their only top-level classification, and M as child classification(s) of N.

None of these types of classification trees, however, can generate all the valid combinations of classes above, because:

- Combinations (a) and (b) cannot be constructed from Type (1).
- Combination (b) cannot be constructed from Type (2).
- Combination (a) cannot be constructed from Type (3).

The root cause of this problem is that the parent-child hierarchical relations between M and N are symmetric. In other words, M is a parent classification of N while N is also a parent classification of M, hence resulting in a loop rather than a tree structure. M and N (and their associated classes) should be redefined before a classification tree can be constructed correctly. ■

The hierarchical operators introduced by Chen and Poon [8] and described in Section 2.2 simply assume that symmetric parent-child or ancestor-descendent hierarchical relations do not exist for any pair of classifications. We would like, however, to help software testers identify such unwarranted situations, thereby providing them with an opportunity to review and improve on the classifications and classes. This can be achieved by refining the hierarchical operators as follows:

(1) We define X to be a *loose ancestor* of Y, denoted by $X \Leftrightarrow Y$, if and only if the following conditions are satisfied:

- (a) There exist some $x \in X$ and $y \in Y$ such that $class(X) = x$ and $class(Y) = y$ are part of a legitimate input.
- (b) There exists some $x' \in X$ such that, for any $y' \in Y$, we cannot have $class(X) = x'$ and $class(Y) = y'$ in any legitimate input.
- (c) There exists some $y'' \in Y$ such that, for any $x'' \in X$, we cannot have $class(X) = x''$ and $class(Y) = y''$ in any legitimate input.

(2) We define X to be a *strict ancestor* (or simply an *ancestor*) of Y, denoted by $X \Rightarrow Y$, if and only if the following conditions are satisfied:

- (a) There exist some $x \in X$ and $y \in Y$ such that $class(X) = x$ and $class(Y) = y$ are part of a legitimate input.
- (b) There exists some $x' \in X$ such that, for any $y' \in Y$, we cannot have $class(X) = x'$ and $class(Y) = y'$ in any legitimate input.

Given	Deduced or Defined		
	$Y \Rightarrow X$	$Y \sim X$	$Y \otimes X$
$X \Rightarrow Y$	Not applicable	Not applicable	**Deduced**
$X \sim Y$	Not applicable	**Deduced**	Not applicable
$X \otimes Y$	Can be defined	**Error**	Can be defined

Table 2: Deduction and Constraints on $Y \mapsto X$ from a Given $X \mapsto Y$

(c) There does not exist any $y'' \in Y$ such that, for any $x'' \in X$, we cannot have $class(X) = x''$ and $class(Y) = y''$ in any legitimate input.

(3) We define X to be *incompatible with* Y, denoted by $X \sim Y$, if and only if for any $x \in X$ and $y \in Y$, we cannot have $class(X) = x$ and $class(Y) = y$ in any legitimate input.

(4) We define X to have *other relations with* Y, denoted by $X \otimes Y$, if and only if X is neither a loose ancestor nor a strict ancestor of Y, and is not incompatible with it.

Given the refined hierarchical relations, whenever $X \Leftrightarrow Y$ is being defined, we know that a symmetric parent-child or ancestor-descendent hierarchical relation occurs between two classifications X and Y. Software testers should be alerted to redefine X and Y (and their associated classes) so as to prevent a loop in the classification tree.

We accept that the hierarchical operators "$\Rightarrow$" and "$\otimes$" defined here are different from those defined by Chen and Poon, particularly for the situation where $X \Leftrightarrow Y$ for some classifications X and Y. However, our algorithm for constructing the classification-hierarchy table will ensure that $X \Leftrightarrow Y$ does not occur for any pair of classifications. As a result, the "$\Rightarrow$" and "$\otimes$" operators here are equivalent to those of Chen and Poon. Hence, the three properties presented in Section 2.2 are still applicable. This explains why we prefer to reuse the symbols for the three operators of Chen and Poon despite the slight semantic difference.

Using these three properties, it may be possible to deduce some hierarchical relations. For example, if we know that $X \Rightarrow Y$, then $Y \otimes X$ can be deduced automatically. In this way, not all the $w \times w$ hierarchical relations have to be independently defined.

From these properties, we have constructed Table 3.2 showing the validity of various combinations of $X \mapsto Y$ and $Y \mapsto X$. We find that $X \otimes Y$ and $Y \sim X$ should not coexist in the classification-hierarchy table. Thus, errors of the hierarchical operators in the classification-hierarchy table can be identified.

Each element t_{ij} in the classification-hierarchy table is classified as a *defined element* if it is manually defined, or a *deduced element* if it is automatically deduced. An element M is said to be a *parent element* of another element N if N is deduced from M. The *element-type table* is used to indicate whether a particular element t_{ij} is defined or deduced, so as to ease the removal of inconsistent hierarchical operators from the classification-hierarchy table.

Basically, the element-type table has the same dimension (namely $w \times w$) as its corresponding classification-hierarchy table. Each element e_{ij} (where $1 \leq i, j \leq w$) may take a value of "-1", "0", or "1", indicating whether the corresponding element t_{ij} is defined, unassigned, or deduced, respectively.

The principles of our algorithm for constructing the classification-hierarchy table are:

- To perform automatic deduction instead of manual definition for each unassigned t_{ij} whenever possible.
- To perform consistency checking after every manual definition of t_{ij}.

The following is the algorithm *build_table* for constructing the classification-hierarchy table, in which the techniques for (a) identifying any symmetry in the parent-child or ancestor-descendent hierarchical relations, (b) consistency checking ($check\_operator(\,)$), and (c) automatic deduction ($deduce\_operator(\,)$) are incorporated:

Algorithm *build_table* for Building the Classification-Hierarchy Table

```
procedure  build_table( );
  foreach  e_ij  do                /*  initialize the element-type table  */
     e_ij := "0";
     end_foreach;
  foreach  t_ii  do                /*  define diagonal elements  */
     t_ii := "⊗";
     e_ii := "−1";
     end_foreach;
  while  number of unassigned t_ij's > 0  do
     input  action_flg;            /*  users should set action_flg to 1 if normal processing is required,
                                       or to −1 to delete an incorrect hierarchical relation  */
     if  action_flg = 1  then      /*  normal processing  */
        symmetry_flg := 0;
        define_next_element(symmetry_flg);
        if  symmetry_flg = 1  then      /*  symmetric parent-child or ancestor-descendent
                                            hierarchical relations are detected  */
           stop;
           end_if;
     else                          /*  users wish to delete an incorrect hierarchical relation  */
        input  R;                  /*  R is a stack containing one or more elements to be removed
                                       from the classification-hierarchy table  */
        remove_operator(R);
        deduce_operator( );
        end_if;
     end_while;

  /*  The following steps convert every "⇒" (representing both the parent-child and
      ancestor-descendent hierarchical relations) in the classification-hierarchy table into
      either ">" (for the parent-child relation) or "≫" (for the ancestor-descendent
      relation)  */
```

```
foreach t_ik = "⇒" do
    if there exist t_ij = ("⇒" or "≫") and t_jk = ("⇒" or "≫") then
        t_ik := "≫";
        end_if;
    end_foreach;
foreach t_ij = "⇒" do
    t_ij := ">";
    end_foreach;
```

```
procedure define_next_element(symmetry_flg);

    define next unassigned t_ij;          /* manual definition */
    if t_ij = "⇔" then
        output (i, j), t_ij;              /* alert users about elements with symmetric parent-child
                                             or ancestor-descendent hierarchical relations */
        symmetry_flg := 1;
    else
        e_ij := "−1";
        chk_flg := 0;                     /* chk_flg will be set to 1 if an inconsistency is detected */
        check_operator(i, j, chk_flg);
        if chk_flg = 1 then
            output (i, j), t_ij, (j,i), t_ji;  /* alert users about inconsistent elements */
            input R;                      /* if t_ij, t_ji, or both are to be removed, R is a stack
                                             containing (i, j), (j,i), or both, respectively */
            remove_operator(R);
            end_if;
        deduce_operator( );               /* automatic deduction is only possible after at least one
                                             t_ij has been defined */
        end_if;
```

```
procedure check_operator(i, j, chk_flg);

    /* Consistency checking of a pair t_ij and t_ji based on Table 2 */

    if t_ij = "∼" and t_ji = "⊗" then
        chk_flg := 1;
        end_if;
```

```
procedure remove_operator(R);

    while R ≠ empty do
        pop (i, j) from R;
        begin_case
            case e_ij = "−1":             /* t_ij is a defined element */
                e_ij := "0";
```

```
            if e_ji = "1" then          /* t_ji is deduced from t_ij and hence should also be removed */
               e_ji := "0";
               end_if;
         case e_ij = "1":               /* t_ij is deduced from t_ji, which is a defined element.
                                           Both should be removed */
            e_ij := "0";
            e_ji := "0";
         end_case;
      end_while;

procedure deduce_operator( );

   /* Property 3 mentioned in Section 2.2 is not used because, given X ⊗ Y, it is not
      deterministic whether Y ⇒ X or Y ⊗ X */

   foreach defined t_ij do
      begin_case
         case t_ij = "⇒":               /* X ⇒ Y */
            if e_ji = "0" then
               t_ji := "⊗";             /* use Property 1 */
               e_ji := "1";
               end_if;
         case t_ij = "∼":               /* X ∼ Y */
            if e_ji = "0" then
               t_ji := "∼";             /* use Property 2 */
               e_ji := "1";
               end_if;
         end_case;
      end_foreach;
```

In the above algorithm, it should be noted that:

(a) Whenever the hierarchical operator "⇔" is being defined in *define_next_element*( ) for any pair of classifications, the execution of *build_table* will be stopped and software testers will be asked to redefine the classifications and classes.

(b) The removal of inconsistent hierarchical relations operates in an interactive mode so that software testers are able to select one or more inconsistent elements whose hierarchical operators are to be removed. For example, suppose $X \otimes Y$ and $Y \sim X$ are detected as inconsistencies according to the constraints in Table 3.2. Software testers can decide whether (i, j), (j, i), or both of them are to be included in R (which is subsequently passed to *remove_operator*( ) as a parameter). In the situation where a fully automated mode is required, the removal mechanism can be modified to achieve the same purpose by storing the inconsistent elements (that is, the elements included in the second **output** statement of *define_next_element*( )) in R without any user intervention.

	A	B	C	D	E	F	G	H	I
A	$\otimes$	$\Rightarrow$							
B	$\otimes$	$\otimes$							
C			$\otimes$						
D				$\otimes$					
E					$\otimes$				
F						$\otimes$			
G							$\otimes$		
H								$\otimes$	
I									$\otimes$

Table 3: Classification-Hierarchy Table after Step (5) of Example 3

(c) There may be a situation where the hierarchical operator of a t_{ij} is incorrectly defined but not detected as an inconsistency in $check\_operator()$. For example, suppose $X \sim Y$ and $Y \sim X$ are correct but somehow incorrectly defined as $X \otimes Y$ and $Y \otimes X$. These mistakes would not be detected as inconsistencies via Table 3.2, because $X \otimes Y$ and $Y \otimes X$ are a possible combination. By setting $action\_flg$ to -1 manually, however, software testers are allowed to initiate the execution of $remove\_operator()$ for the deletion of such incorrect hierarchical relations.

Example 3 below illustrates how to apply *build_table* for (a) constructing the classification-hierarchy table and the element-type table and (b) removing the incorrectly defined or deduced elements from the classification-hierarchy table.

Example 3

Refer to the specification of *arith-sum* in Example 1.

(1) Every e_{ij} is initialized to "0".

(2) The hierarchical operator of every t_{ii} (that is, $A \mapsto A$, $B \mapsto B$, ..., $I \mapsto I$) is set to "$\otimes$" and the corresponding e_{ii} to "-1".

(3) t_{12} ($A \mapsto B$) is the next unassigned element whose hierarchical operator is to be defined. The software tester defines it to be $A \Rightarrow B$ according to the specification. Consequently, e_{12} is set to "-1".

(4) Consistency checking is performed for t_{12} and t_{21}. There is obviously no inconsistency because t_{21} is unassigned.

(5) Once $A \Rightarrow B$ is defined, the hierarchical operator for t_{21} ($B \mapsto A$) is deduced as "$\otimes$". Then e_{21} is set to "1" accordingly.

Tables 3 and 3 show the classification-hierarchy table and the element-type table, respectively, after this step.

(6) The construction process is continued from left to right and top to bottom, except those elements that are deduced automatically. Suppose t_{49} ($D \mapsto I$) is the next unassigned element. The software tester

	A	B	C	D	E	F	G	H	I
A	-1	-1	0	0	0	0	0	0	0
B	1	-1	0	0	0	0	0	0	0
C	0	0	-1	0	0	0	0	0	0
D	0	0	0	-1	0	0	0	0	0
E	0	0	0	0	-1	0	0	0	0
F	0	0	0	0	0	-1	0	0	0
G	0	0	0	0	0	0	-1	0	0
H	0	0	0	0	0	0	0	-1	0
I	0	0	0	0	0	0	0	0	-1

Table 4: Element-Type Table after Step (5) of Example 3

	A	B	C	D	E	F	G	H	I
A	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
B	$\otimes$	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
C	$\otimes$	$\otimes$	$\otimes$	$\Rightarrow$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
D	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
E	$\otimes$		$\sim$	$\sim$	$\otimes$				
F						$\otimes$			
G							$\otimes$		
H			$\sim$	$\sim$				$\otimes$	
I									$\otimes$

Table 5: Classification-Hierarchy Table after Step (6) of Example 3

defines it to be $D \otimes I$ according to the specification. Since t_{94} $(I \mapsto D)$ has not yet been defined, the system cannot detect any inconsistency between t_{49} and t_{94}, nor deduce any hierarchical operator for t_{94}.

Table 3 shows the classification-hierarchy table after this step and Table 3 shows the element-type table.

(7) The next element to consider is t_{52} $(E \mapsto B)$. Suppose that, instead of setting $B \sim E$ according to the specification, the software tester has defined t_{25} as $B \otimes E$ by mistake. As a result, the system has not deduced any hierarchical operator immediately from $B \otimes E$. In such a case, the software tester defines $E \sim B$ according to the specification. Thus, e_{52} is set to "-1".

(8) Consistency checking is performed for t_{52} and t_{25}. At this point, an inconsistency is detected because $E \sim B$ and $B \otimes E$ should not coexist according to Table 3.2. The software tester is informed of this inconsistency via the second **output** statement in $define\_next\_element()$.

(9) Suppose the software tester realizes that t_{25} $(B \otimes E)$ has been incorrectly defined and hence should be removed. Thus, $(2,5)$ is entered into R through the "**input** R" statement in $define\_next\_element()$, to be passed to $remove\_operator()$ as a parameter.

	A	B	C	D	E	F	G	H	I
A	−1	−1	−1	−1	−1	−1	−1	−1	−1
B	1	−1	−1	−1	−1	−1	−1	−1	−1
C	1	1	−1	−1	−1	−1	−1	−1	−1
D	1	1	1	−1	−1	−1	−1	−1	−1
E	1	0	1	1	−1	0	0	0	0
F	0	0	0	0	0	−1	0	0	0
G	0	0	0	0	0	0	−1	0	0
H	0	0	1	1	0	0	0	−1	0
I	0	0	0	0	0	0	0	0	−1

Table 6: Element-Type Table after Step (6) of Example 3

	A	B	C	D	E	F	G	H	I
A	$\otimes$	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
B	$\otimes$	$\otimes$	$\Rightarrow$	$\Rightarrow$		$\otimes$	$\otimes$	$\otimes$	$\otimes$
C	$\otimes$	$\otimes$	$\otimes$	$\Rightarrow$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
D	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
E	$\otimes$	$\sim$	$\sim$	$\sim$	$\otimes$				
F						$\otimes$			
G							$\otimes$		
H			$\sim$	$\sim$				$\otimes$	
I									$\otimes$

Table 7: Classification-Hierarchy Table after Step (10) of Example 3

(10) In *remove_operator*( ), the following steps are performed:

- (a) Pop $(2,5)$ from R.
- (b) Since e_{25} is "-1", t_{25} is a defined element, and so e_{25} is set to "0". This is equivalent to deleting the hierarchical operator "$\otimes$" from t_{25}. Since e_{52} is "-1", t_{52} is also a defined element, and hence t_{52} remains unchanged.

Tables 3 and 3 show the classification-hierarchy table and the element-type table, respectively, after this step.

(11) From $E \sim B$, we deduce that $B \sim E$ and set e_{25} to "1".

(12) We continue the construction process until hierarchical operators have been assigned for all the t_{ij}'s. The classification-hierarchy table after this step is depicted in Table 1.

(13) For t_{12} $(A \Rightarrow B)$ and t_{13} $(A \Rightarrow C)$ in Table 1, since t_{23} is defined with the hierarchical relation $B \Rightarrow C$, the hierarchical operator for t_{13} is changed from "$\Rightarrow$" to the symbol "$\gg$" to indicate that it is an ancestor-descendent relation. Similarly, the hierarchical operators for t_{14}, t_{24}, and t_{64} are also changed from "$\Rightarrow$" to "$\gg$". This leaves the hierarchical operator "$\Rightarrow$" for t_{12}, t_{15}, t_{23}, t_{34}, t_{63}, t_{67}, and t_{68} unchanged.

	A	B	C	D	E	F	G	H	I
A	-1	-1	-1	-1	-1	-1	-1	-1	-1
B	1	-1	-1	-1	0	-1	-1	-1	-1
C	1	1	-1	-1	-1	-1	-1	-1	-1
D	1	1	1	-1	-1	-1	-1	-1	-1
E	1	-1	1	1	-1	0	0	0	0
F	0	0	0	0	0	-1	0	0	0
G	0	0	0	0	0	0	-1	0	0
H	0	0	1	1	0	0	0	-1	0
I	0	0	0	0	0	0	0	0	-1

Table 8: Element-Type Table after Step (10) of Example 3

	A	B	C	D	E	F	G	H	I
A	$\otimes$	$>$	$\gg$	$\gg$	$>$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
B	$\otimes$	$\otimes$	$>$	$\gg$	$\sim$	$\otimes$	$\otimes$	$\otimes$	$\otimes$
C	$\otimes$	$\otimes$	$\otimes$	$>$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
D	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
E	$\otimes$	$\sim$	$\sim$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$
F	$\otimes$	$\otimes$	$>$	$\gg$	$\otimes$	$\otimes$	$>$	$>$	$\otimes$
G	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$
H	$\otimes$	$\otimes$	$\sim$	$\sim$	$\otimes$	$\otimes$	$\sim$	$\otimes$	$\otimes$
I	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$

Table 9: Classification-Hierarchy Table after Step (13) of Example 3

We then change the hierarchical operators for the remaining elements from "$\Rightarrow$" to the symbol "$>$" to indicate that they are parent-child relations. In this way, we can distinguish between parent-child and ancestor-descendent relations, thus avoiding the redundant construction and pruning of subtrees as presented in [7].

The final classification-hierarchy table is shown in Table 3.2. ■

3.3 Construction of Classification Tree

From the classification-hierarchy table, the corresponding classification tree can be constructed using the following algorithm:

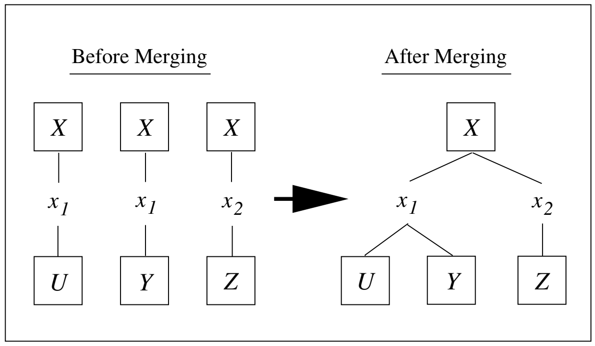


Figure 2: An Example of Step $(2)(a)$ of *build_tree*

Algorithm *build_tree* for Constructing the Classification Tree

(1) Construction of Subtrees with Parent-Child Hierarchical Relation

Form subtrees for all the t_{ij}'s with the parent-child operator "$>$", thus:

Suppose $X > Y$. Select all the classes $x \in X$ such that any legitimate input containing $class(X) = x$ also contains $class(Y) = y$ for some $y \in Y$. For each of these classes x, form a subtree with X as the root, x as the only child class of X, and Y as the only child classification of x.

(2) Merging of Related Subtrees

(a) Merge together all the subtrees from step (1) having X as their roots, to form a new subtree whose root is still X. The child classes of X in the new subtree are all the classes of the merged subtrees without duplications (see Figure 3.3).

(b) For each pair of subtrees with roots X and Y, if Y appears as a terminal node of the subtree with root X, combine the two subtrees to form a new one by replacing the terminal node Y with the subtree with root Y (see Figure 3.3).

(c) For all the subtrees remaining after step $(2)(b)$, add all the child classes for each classification.

(3) Pruning of Duplicated Subtrees

Let τ_i (where $i \geq 1$) denote a subtree formed in step $(2)(c)$ above and $S^X_{\tau_i}$ denote a subtree within τ_i, with the classification X as its root. In order to distinguish between the two kinds of subtrees, we will refer to τ_i as a *top-level subtree*. It should be noted that there may be more than one subtree $S^X_{\tau_i}$ with identical classifications and classes *within* a given top-level subtree τ_i. Let τ'_i denote the top-level subtree after pruning all the identical $S^X_{\tau_i}$'s from τ_i, and $N(\tau_i)$ denote the total number of combinations of classes for τ_i.

Suppose there are two or more top-level subtrees $\tau_1,\ \tau_2,\ \ldots,\ \tau_n$ containing duplicated subtrees $S^X_{\tau_1}$, $S^X_{\tau_2}$, …, $S^X_{\tau_n}$, respectively. Select a top-level subtree τ_k (where $1 \leq k \leq n$) such that, if we prune

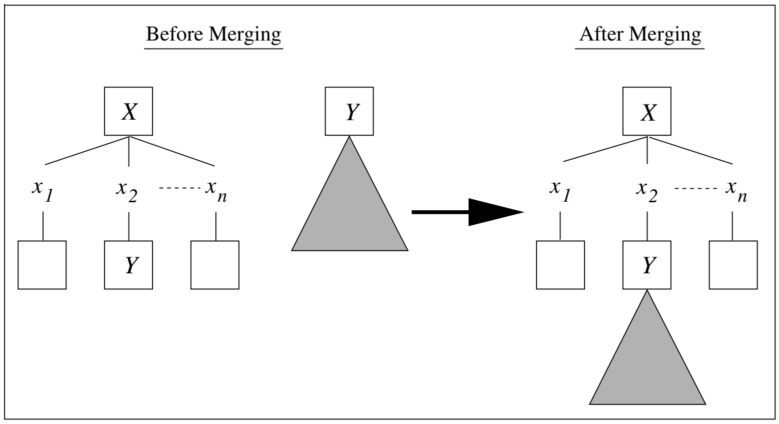


Figure 3: An Example of Step $(2)(b)$ of *build_tree*

all the subtrees $S^X_{\tau_1}$, $S^X_{\tau_2}$, …, $S^X_{\tau_{k-1}}$, $S^X_{\tau_{k+1}}$, …, $S^X_{\tau_n}$ from $\tau_1,\ \tau_2,\ \ldots,\ \tau_{k-1},\ \tau_{k+1},\ \ldots,\ \tau_n$, respectively, it yields the smallest value of

$$\left(\prod_{j=1}^{k-1} N(\tau'_j)\right) \times N(\tau_k) \times \left(\prod_{j=k+1}^{n} N(\tau'_j)\right)$$

Replace the top-level subtrees $\tau_1,\ \tau_2,\ \ldots,\ \tau_{k-1},\ \tau_{k+1},\ \ldots,\ \tau_n$ by $\tau'_1,\ \tau'_2,\ \ldots,\ \tau'_{k-1},\ \tau'_{k+1},\ \ldots,\ \tau'_n$, respectively, but leave the selected top-level subtree τ_k unchanged.

Repeat this step until there are no duplicated subtrees $S^X_{\tau_j}$ and $S^X_{\tau_k}$ across any pair of distinct top-level subtrees τ_j and τ_k. Note, however, that $S^X_{\tau_k}$ is allowed to occur more than once *within* a top-level subtree τ_k.

(4) Identification of Stand Alone Classifications

For every X that does not appear in any remaining top-level subtree, form a top-level subtree with X as the root and add all its child classes.

(5) Integration of All Subtrees

Use the general root node (denoted by a small circle) to link up all the top-level subtrees formed in steps (3) and (4).

The algorithm *calculate_combination* is based on the formulae from [7], as listed in Appendix 1. It can be used for the computation of $N(\tau_k)$ and $N(\tau'_j)$ in step (3) of *build_tree*.

The intuition of step (3) of *build_tree* is to prevent the occurrence of duplicated subtrees under different top-level classifications (corresponding to the top-level subtrees τ_i's of step (3)). Otherwise these duplicated subtrees would lead to the occurrence of illegitimate test cases, resulting in a smaller $E_{\mathcal{T}}$. This may be

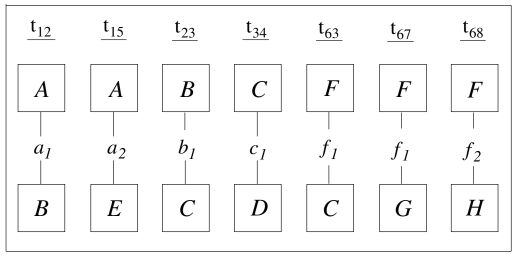


Figure 4: Subtrees Formed in Step (1) of Example 4

prevented by pruning the duplicated subtrees from all but one τ_i. In this process, the identification of the duplicated subtrees to be pruned is guided by minimizing the value of

$$\left(\prod_{j=1}^{k-1} N(\tau'_j)\right) \times N(\tau_k) \times \left(\prod_{j=k+1}^{n} N(\tau'_j)\right)$$

The intuition is to reduce the value of N_p (and hence improve on the value of $E_{\mathcal{T}}$) of the final classification tree.

Our integrated classification-tree algorithm differs from that of Chen and Poon [8] in the following:

- Subtrees with parent-child or ancestor-descendent hierarchical relations are formed in step (1) of Chen and Poon's tree construction algorithm, whereas only those with parent-child hierarchical relations are formed in step (1) of our algorithm. Thus, the redundant processes of forming and subsequent pruning of those subtrees with ancestor-descendent hierarchical relations (as in Chen and Poon's algorithm) do not exist in our algorithm.

- Our approach to the construction of classification trees is guided by the effectiveness metric $E_{\mathcal{T}}$ (and hence subsequent tree restructuring is not required), whereas Chen and Poon's approach considers the effectiveness aspect only after the construction phase.

Now, let us illustrate how to construct a classification tree in Example 4 and how to construct the set of legitimate test cases from that classification tree in Exampl 5.

Example 4
Refer to the classification-hierarchy table for *arith-sum* in Table 3.2.

(1) Construction of Subtrees with Parent-Child Hierarchical Relation

In Table 3.2, the elements with the hierarchical operator ">" are t_{12}, t_{15}, t_{23}, t_{34}, t_{63}, t_{67}, and t_{68}. For t_{12}, when $class(A) = a_1$, $class(B) = b_1$ or b_2. Hence, a subtree is formed with A as its root, a_1 as A's unique child class, and B as a_1's unique child classification. Subtrees for t_{15}, t_{23}, t_{34}, t_{63}, t_{67}, and t_{68} are formed in a similar way. Figure 4 depicts all the subtrees formed in this step.

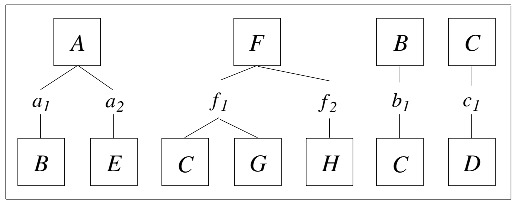


Figure 5: Subtrees Formed in Step $(2)(a)$ of Example 4

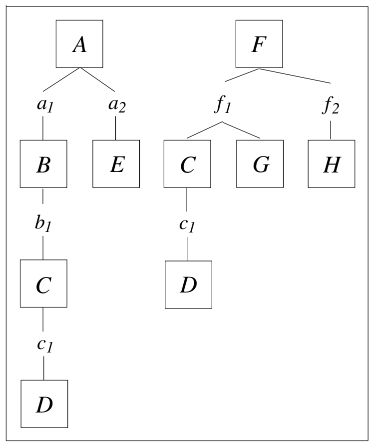


Figure 6: Subtrees Formed in Step $(2)(b)$ of Example 4

(2) Merging of Related Subtrees

(a) Since both the subtrees corresponding to t_{12} and t_{15} have A as their roots, they are merged together to form a single subtree with root A. Similarly, the subtrees corresponding to t_{63}, t_{67}, and t_{68} are merged together to form another single subtree with root F. It should be noted that, although f_1 appears in both the subtrees corresponding to t_{63} and t_{67} before the merging process, f_1 appears only once in the newly formed subtree after the merging process. Figure 4 depicts the two newly formed subtrees after the merging processes, together with the subtrees corresponding to t_{23} and t_{34}, which are left intact throughout the merging processes.

(b) Let τ_X denote the subtree with classification X as its root. The subtrees in Figure 4 are merged as follows:

(i) Combine τ_F and τ_C to form a new subtree by replacing the terminal node C of τ_F with τ_C.

(ii) Combine τ_B and τ_C to form a new subtree (denoted by τ'_B) by replacing the terminal node C of τ_B with τ_C.

(iii) Combine τ_A and the newly formed τ'_B to form a new subtree by replacing the terminal node B of τ_A with τ'_B.

Figure 4 depicts the resultant subtrees after these merging processes.

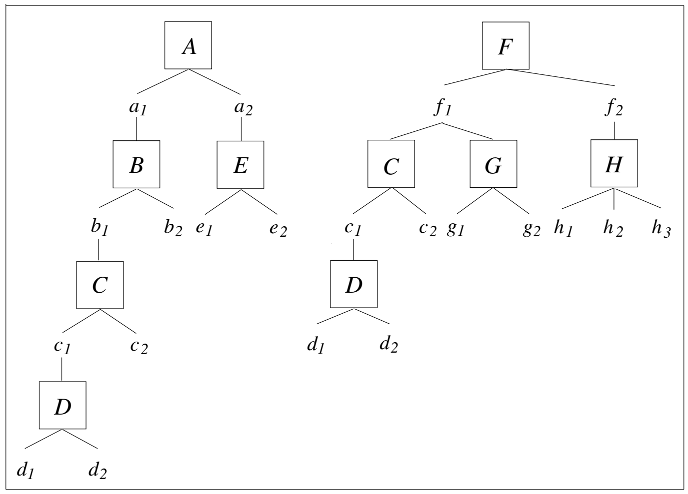


Figure 7: Subtrees Formed in Step $(2)(c)$ of Example 4

(c) The two subtrees in Figure 4 are the only ones remaining after step $(2)(b)$. After all the classes of every classification in these two subtrees have been added, the resultant subtrees are depicted in Figure 4.

(3) Pruning of Duplicated Subtrees

Let τ_1 and τ_2 denote the two top-level subtrees in Figure 4 with A and F as their roots, respectively. They contain the duplicated subtrees $S^C_{\tau_1}$ and $S^C_{\tau_2}$, respectively. Figure 4 depicts the resultant top-level subtree τ'_1 formed after pruning $S^C_{\tau_1}$ from τ_1, and the resultant top-level subtree τ'_2 after pruning $S^C_{\tau_2}$ from τ_2. Using Eq. (1) and Eq. (2) in Appendix 1, $N(\tau_2)$ can be calculated as follows:

$$N(S^D_{\tau_2}) = 2$$
$$N(S^{c_1}_{\tau_2}) = N(S^D_{\tau_2}) = 2$$
$$N(S^C_{\tau_2}) = 1 + N(S^{c_1}_{\tau_2}) = 1 + 2 = 3$$
$$N(S^G_{\tau_2}) = 2$$
$$N(S^{f_1}_{\tau_2}) = N(S^C_{\tau_2}) \times N(S^G_{\tau_2}) = 3 \times 2 = 6$$
$$N(S^H_{\tau_2}) = 3$$
$$N(S^{f_2}_{\tau_2}) = N(S^H_{\tau_2}) = 3$$
$$N(\tau_2) = N(S^F_{\tau_2}) = N(S^{f_1}_{\tau_2}) + N(S^{f_2}_{\tau_2}) = 6 + 3 = 9$$

Similarly, $N(\tau_1)$, $N(\tau'_1)$, and $N(\tau'_2)$ are calculated to be 6, 4, and 5, respectively. Hence, $N(\tau_1) \times N(\tau'_2) = 30$ and $N(\tau'_1) \times N(\tau_2) = 36$. Since $N(\tau_1) \times N(\tau'_2) < N(\tau'_1) \times N(\tau_2)$, τ_2 should be the top-level subtree chosen for pruning.

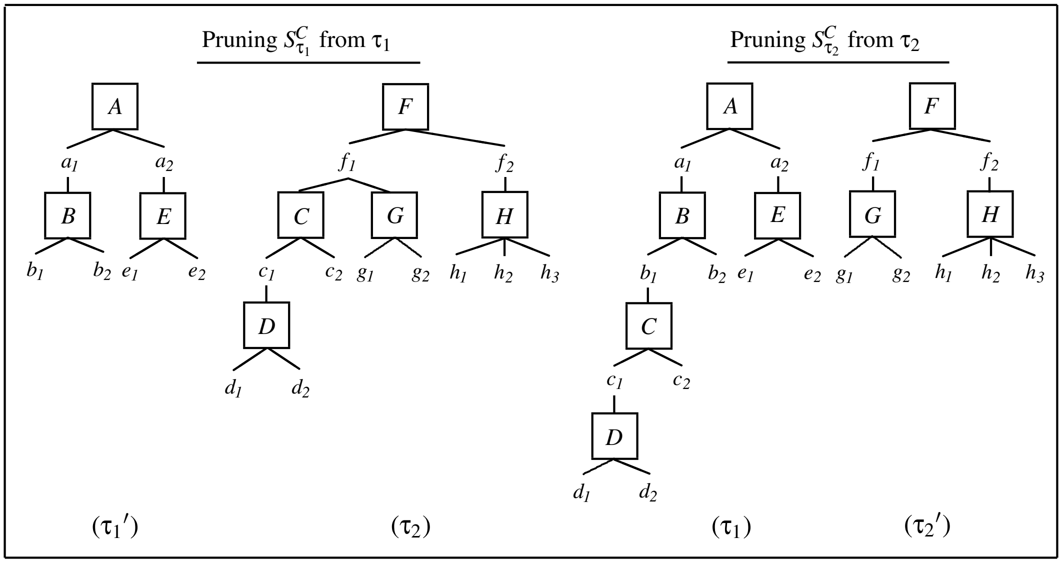


Figure 8: Resultant Subtrees Formed after Pruning $S^C_{\tau_1}$ and $S^C_{\tau_2}$ from τ_1 and τ_2, respectively

(4) Identification of Stand Alone Classifications

Since I is the only classification that does not appear in any top-level subtree remaining after step (3), a top-level subtree with I as its root is formed. Then, all the classes (i_1 and i_2) of I are added to this partially formed top-level subtree to produce a complete τ_I.

(5) Integration of All Subtrees

The two top-level subtrees τ_1 and τ_2' formed in step (3) and the top-level subtree τ_I formed in step (4) are linked to the general root node to form the final classification tree. It is depicted in Figure 3.3 as $\mathcal{T}'_{arith-sum}$. ■

Example 5

From $\mathcal{T}'_{arith-sum}$ in Figure 3.3, a total of 60 potential test cases can be constructed and are shown in Table 5. By checking all these potential test cases against the specification of *arith-sum* in Example 1, the following 32 potential test cases are found to be illegitimate and should therefore be removed:

- The potential test cases 5–10, 15–20, and 25–30 are illegitimate because $class(F) = f_2$ cannot coexist with $class(C) = c_1$ or c_2.
- The potential test cases 31–34, 41–44, and 51–54 are illegitimate because $class(F) = f_1$ must coexist with $(class(C) = c_1$ and $class(D) = d_1)$, $(class(C) = c_1$ and $class(D) = d_2)$, or $class(C) = c_2$.

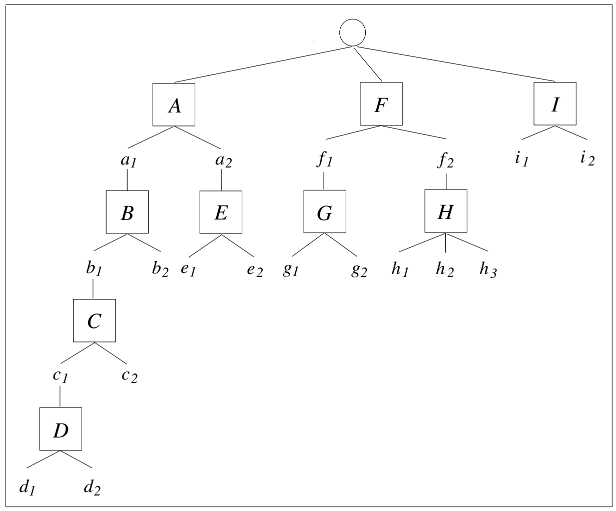


Figure 9: Final Classification Tree ($\mathcal{T}'_{arith-sum}$) for *arith-sum*

No.	Potential Test Cases	No.	Potential Test Cases	No.	Potential Test Cases
1	$a_1, b_1, c_1, d_1, f_1, g_1, i_1$	21	$a_1, b_1, c_2, f_1, g_1, i_1$	41	a_2, e_1, f_1, g_1, i_1
2	$a_1, b_1, c_1, d_1, f_1, g_1, i_2$	22	$a_1, b_1, c_2, f_1, g_1, i_2$	42	a_2, e_1, f_1, g_1, i_2
3	$a_1, b_1, c_1, d_1, f_1, g_2, i_1$	23	$a_1, b_1, c_2, f_1, g_2, i_1$	43	a_2, e_1, f_1, g_2, i_1
4	$a_1, b_1, c_1, d_1, f_1, g_2, i_2$	24	$a_1, b_1, c_2, f_1, g_2, i_2$	44	a_2, e_1, f_1, g_2, i_2
5	$a_1, b_1, c_1, d_1, f_2, h_1, i_1$	25	$a_1, b_1, c_2, f_2, h_1, i_1$	45	a_2, e_1, f_2, h_1, i_1
6	$a_1, b_1, c_1, d_1, f_2, h_1, i_2$	26	$a_1, b_1, c_2, f_2, h_1, i_2$	46	a_2, e_1, f_2, h_1, i_2
7	$a_1, b_1, c_1, d_1, f_2, h_2, i_1$	27	$a_1, b_1, c_2, f_2, h_2, i_1$	47	a_2, e_1, f_2, h_2, i_1
8	$a_1, b_1, c_1, d_1, f_2, h_2, i_2$	28	$a_1, b_1, c_2, f_2, h_2, i_2$	48	a_2, e_1, f_2, h_2, i_2
9	$a_1, b_1, c_1, d_1, f_2, h_3, i_1$	29	$a_1, b_1, c_2, f_2, h_3, i_1$	49	a_2, e_1, f_2, h_3, i_1
10	$a_1, b_1, c_1, d_1, f_2, h_3, i_2$	30	$a_1, b_1, c_2, f_2, h_3, i_2$	50	a_2, e_1, f_2, h_3, i_2
11	$a_1, b_1, c_1, d_2, f_1, g_1, i_1$	31	a_1, b_2, f_1, g_1, i_1	51	a_2, e_2, f_1, g_1, i_1
12	$a_1, b_1, c_1, d_2, f_1, g_1, i_2$	32	a_1, b_2, f_1, g_1, i_2	52	a_2, e_2, f_1, g_1, i_2
13	$a_1, b_1, c_1, d_2, f_1, g_2, i_1$	33	a_1, b_2, f_1, g_2, i_1	53	a_2, e_2, f_1, g_2, i_1
14	$a_1, b_1, c_1, d_2, f_1, g_2, i_2$	34	a_1, b_2, f_1, g_2, i_2	54	a_2, e_2, f_1, g_2, i_2
15	$a_1, b_1, c_1, d_2, f_2, h_1, i_1$	35	a_1, b_2, f_2, h_1, i_1	55	a_2, e_2, f_2, h_1, i_1
16	$a_1, b_1, c_1, d_2, f_2, h_1, i_2$	36	a_1, b_2, f_2, h_1, i_2	56	a_2, e_2, f_2, h_1, i_2
17	$a_1, b_1, c_1, d_2, f_2, h_2, i_1$	37	a_1, b_2, f_2, h_2, i_1	57	a_2, e_2, f_2, h_2, i_1
18	$a_1, b_1, c_1, d_2, f_2, h_2, i_2$	38	a_1, b_2, f_2, h_2, i_2	58	a_2, e_2, f_2, h_2, i_2
19	$a_1, b_1, c_1, d_2, f_2, h_3, i_1$	39	a_1, b_2, f_2, h_3, i_1	59	a_2, e_2, f_2, h_3, i_1
20	$a_1, b_1, c_1, d_2, f_2, h_3, i_2$	40	a_1, b_2, f_2, h_3, i_2	60	a_2, e_2, f_2, h_3, i_2

Table 10: All the Potential Test Cases Constructed from $\mathcal{T}'_{arith-sum}$

- The potential test cases 23 and 24 are illegitimate because $class(C) = c_2$ cannot coexist $class(G) = g_2$.

Thus, 28 legitimate test cases remain after the removal of the above illegitimate test cases. They are 1–4, 11–14, 21–22, 35–40, 45–50, and 55–60 in Table 5. ■

In Example 5, it can be seen that the total number of potential test cases (namely 60) constructed from $\mathcal{T}'_{arith-sum}$ in Figure 3.3 using the integrated approach is significantly smaller than the number (namely 108) constructed from $\mathcal{T}_{arith-sum}$ in Figure 2.2. Also, all the legitimate test cases can be constructed directly from $\mathcal{T}'_{arith-sum}$ without the need for further reformatting the relevant potential test cases (as required by Chen and Poon's restructuring technique).

The usefulness of our integrated approach has been verified in a credit-card approval system [9] and an integrated hospital system [6]. The results of these applications are very encouraging.

In fact, our integrated approach has the following two important properties:

(a) Preservation Property

Let $\mathcal{T}$ be a classification tree and $\mathcal{T}'$ be the new tree after pruning a set of duplicated subtrees in step (3) of *build_tree*. All the legitimate test cases identified from $\mathcal{T}$ can also be identified from $\mathcal{T}'$.

(b) Convergence Property

Let N_p be the number of potential test cases in $\mathcal{T}$ and N'_p be that in $\mathcal{T}'$. Then $N'_p \leq N_p$.

Readers may refer to Appendix 2 for the proofs.

4 Major Features of Prototype System ADDICT

Based on the integrated classification-tree methodology presented above, a prototype system (ADDICT) has been built for the construction of test cases from specifications. ADDICT has been developed using the object-oriented, event-driven Visual Basic in order to provide a better human-machine interface. Essentially, ADDICT has the following main functions:

(1) Define or remove classifications and their associated classes.

(2) Define or remove the *influence* of one classification on the others (where influence is defined as the effect of the occurrence of each class of a classification on the feasibility of the classes of another classification).

(3) Construct the classification-hierarchy table by:

   (a) Defining the hierarchical relation for some pairs of distinct classifications based on the influences entered in step (2).

   (b) Checking the existence of symmetric parent-child or ancestor-descendent hierarchical relations for any pair of classifications.

   (c) Performing the consistency checking of the defined hierarchical relations.

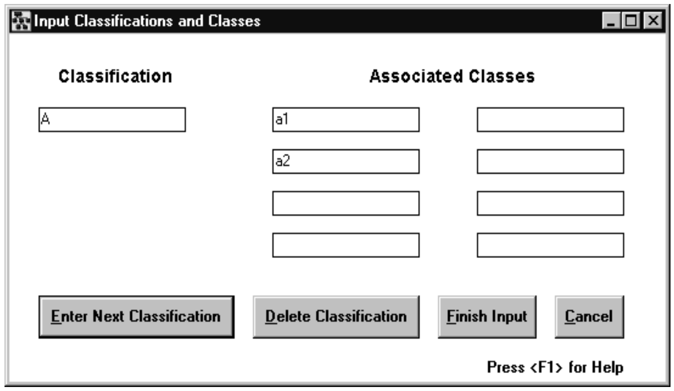


Figure 10: Input Screen for Classifications and Associated Classes

(d) Performing the automatic deduction of new hierarchical relations (if possible).

(4) Construct the classification tree $\mathcal{T}$ from the classification-hierarchy table. In order to reduce the number of illegitimate test cases resulting from the duplication of subtrees under different top-level classifications, this construction process is guided by the effectiveness metric $E_{\mathcal{T}}$.

(5) Construct the set of potential test cases from $\mathcal{T}$.

Let us use the specification of *arith-sum* in Example 1 to illustrate the functions of ADDICT.

Example 6

(1) First, all the classifications and their associated classes for *arith-sum* have to be entered into ADDICT. Figure 6 depicts the input screen through which A and its associated classes (a_1 and a_2) are entered.

(2) For some pairs of distinct classifications X and Y, the influence of every class of X on the classes of Y has to be entered. For example, in Figure 6, the influence of each class (b_1 and b_2) of B on the classes of E is being entered. It can be seen that when $class(B) = b_1$ or b_2, $class(E) \neq e_1$ and e_2. This will trigger ADDICT to automatically assign the hierarchical operator "$\sim$" to $B \mapsto E$. ADDICT will also automatically deduce the hierarchical operator for $E \mapsto B$ to be "$\sim$" (after performing consistency checks and detecting no inconsistencies).

Similarly, the influence of each class of B on the classes of C can also be entered in two dialogue windows, indicating that (i) when $class(B) = b_1$, $class(C) = c_1$ or c_2, (ii) when $class(B) = b_2$, $class(C) \neq c_1$ and c_2, and (iii) when $class(C) = c_1$ or c_2, $class(B) = b_1$ or b_2. Such a combination will trigger ADDICT to automatically assign the hierarchical operator "$\Rightarrow$" to $B \mapsto C$.

(3) Occasionally, incorrect influences may accidentally be entered into ADDICT, and incorrect hierarchical relations may be defined or deduced as a result. These incorrect influences or hierarchical relations may be subsequently detected either by the consistency checking mechanism of the prototype system or by the software testers themselves. In such cases, ADDICT allows the software testers to remove

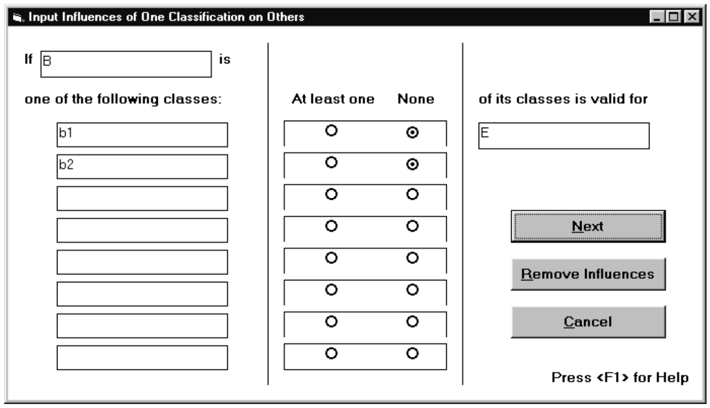


Figure 11: Input Screen for the Influences of One Classification on Others

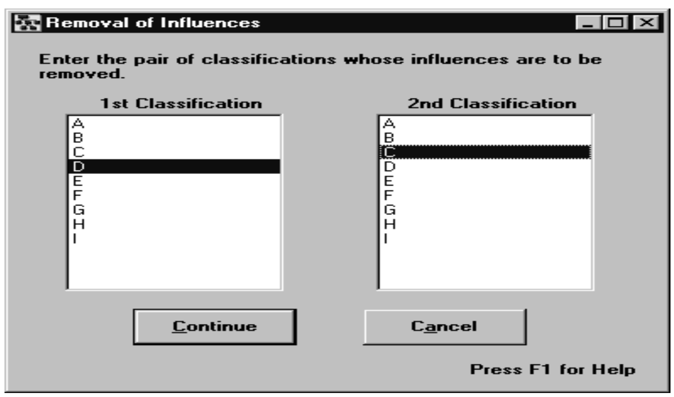


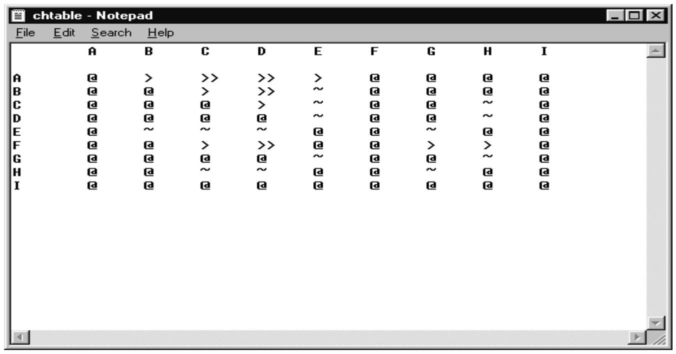
Figure 12: Input Screen for Selecting the Pair of Classifications whose Influences are to be Removed

these incorrect influences or hierarchical relations. During the process, the prototype system will automatically identify and remove any other influences or hierarchical relations that may be affected.

For example, in Figure 6, the software tester wants to remove the influence of D on C. By referring to the element-type table (internally maintained by ADDICT), the prototype system will also remove the influences of C on D (as $D \otimes C$ is deduced from $C \Rightarrow D$).

(4) After hierarchical operators have been assigned to all the t_{ij}'s, all the "$\Rightarrow$"s are converted into "$\gg$"s or "$>$"s. See Figure 6. Note that the symbol "@" in Figure 6 corresponds to the hierarchical operator "$\otimes$" used throughout this paper.

(5) From the classification-hierarchy table, ADDICT will automatically construct the corresponding classification tree (see Figure 4), from which the set of potential test cases is constructed. Some of the potential test cases for *arith-sum* constructed by ADDICT are shown in Figure 4. ■



chtable - Notepad

	A	B	C	D	E	F	G	H	I
A	@	>	>>	>>	>	@	@	@	@
B	@	@	>	>>	~	@	@	@	@
C	@	@	@	>	~	@	@	~	@
D	@	@	@	@	~	@	@	~	@
E	@	~	~	~	@	@	~	@	@
F	@	@	>	>>	@	@	>	>	@
G	@	@	@	@	~	@	@	~	@
H	@	@	~	~	@	@	~	@	@
I	@	@	@	@	@	@	@	@	@

Figure 13: Classification-Hierarchy Table for *arith-sum*

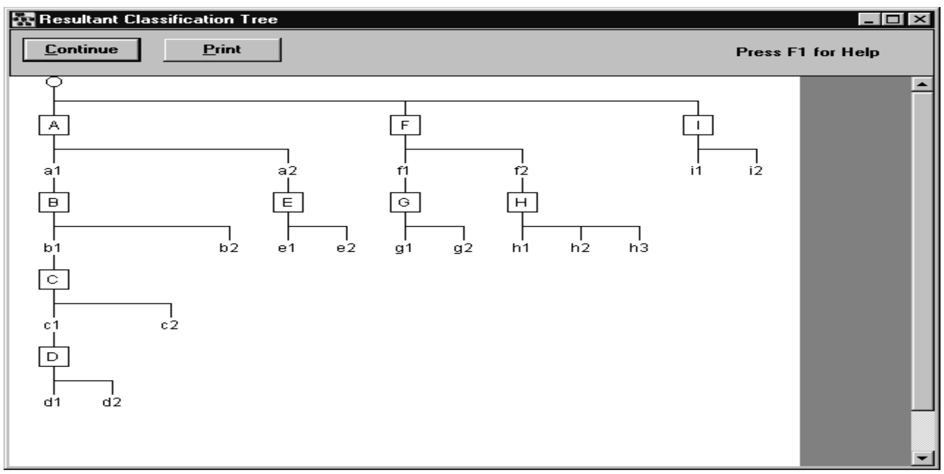
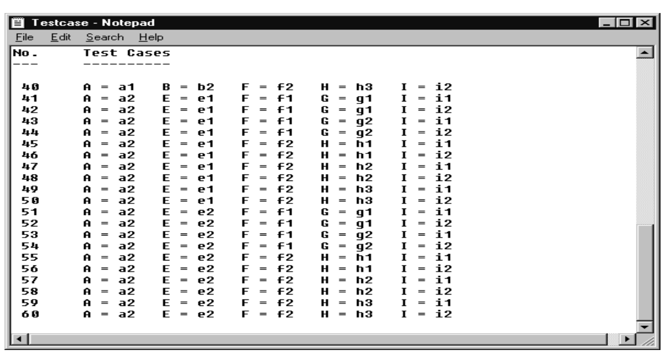


Figure 14: Classification Tree for *arith-sum* Constructed by ADDICT



Testcase - Notepad

No.	Test Cases				
40	A = a1	B = b2	F = f2	H = h3	I = i2
41	A = a2	E = e1	F = f1	G = g1	I = i1
42	A = a2	E = e1	F = f1	G = g1	I = i2
43	A = a2	E = e1	F = f1	G = g2	I = i1
44	A = a2	E = e1	F = f1	G = g2	I = i2
45	A = a2	E = e1	F = f2	H = h1	I = i1
46	A = a2	E = e1	F = f2	H = h1	I = i2
47	A = a2	E = e1	F = f2	H = h2	I = i1
48	A = a2	E = e1	F = f2	H = h2	I = i2
49	A = a2	E = e1	F = f2	H = h3	I = i1
50	A = a2	E = e1	F = f2	H = h3	I = i2
51	A = a2	E = e2	F = f1	G = g1	I = i1
52	A = a2	E = e2	F = f1	G = g1	I = i2
53	A = a2	E = e2	F = f1	G = g2	I = i1
54	A = a2	E = e2	F = f1	G = g2	I = i2
55	A = a2	E = e2	F = f2	H = h1	I = i1
56	A = a2	E = e2	F = f2	H = h1	I = i2
57	A = a2	E = e2	F = f2	H = h2	I = i1
58	A = a2	E = e2	F = f2	H = h2	I = i2
59	A = a2	E = e2	F = f2	H = h3	I = i1
60	A = a2	E = e2	F = f2	H = h3	I = i2

Figure 15: Some of the Potential Test Cases for *arith-sum* Constructed by ADDICT

5 Conclusion

The original classification-tree method proposed by Grochtmann and Grimm [12, 13] provided a means for software testers to construct test cases from specifications via the construction of classification trees. From the notion of classification-hierarchy tables, Chen and Poon [8] provided a methodology for the construction of classification trees, from the given sets of classifications and their associated classes. Using classification trees, the construction of their sets of potential test cases is relatively straightforward.

Unfortunately, some potential test cases constructed from the classification trees may be illegitimate, since not all the constraints among the classifications may be reflected by the classification trees. This leaves software testers with a manual task of identifying the legitimate test cases from the potential ones, by validating against the specifications. A smaller set of potential test cases is obviously desirable. Based on this rationale, Chen and Poon [7] defined an effectiveness metric $E_{\mathcal{T}}$ and developed a tree restructuring technique in order to improve on the quality of a classification tree.

In this paper, we introduce an integrated classification-tree methodology by (i) refining the set of hierarchical operators and (ii) enhancing and integrating the classification-hierarchy table, the tree construction algorithm, and the tree restructuring technique. Our integrated methodology incorporates additional features, including:

(a) a means to identify any unwarranted symmetry in the parent-child or ancestor-descendent hierarchical relations among classifications,

(b) a means to check the consistency of the defined hierarchical relations during the construction of the classification-hierarchy table,

(c) a means for the automatic deduction of some hierarchical relations yet to be defined, based on those already defined,

(d) the use of the parent-child relation to improve on the construction of classification trees,

(e) the provision of a new approach, guided by an effectiveness metric, to construct classification trees from the classification-hierarchy table, and

(f) the elimination of the need for test case reformatting during the construction of test cases from classification trees.

A prototype system ADDICT has been built on our integrated methodology for the construction of test cases from specifications. The practical usefulness of our methodology has been verified in [6, 9].

Acknowledgments

We would like to thank M.F. Lau and C.K. Low of the University of Melbourne, W.H. Kwok and I.K. Mak of the University of Hong Kong, and Y.T. Yu of the City University of Hong Kong for their invaluable comments and suggestions. In addition, we are grateful to Y.Y. Fu and C.K. Low of the University of Melbourne for their help in implementing part of ADDICT. Y.Y. Fu was supported in part by a Summer Vacation Scholarship.

References

[1] N. Amla and P. Ammann, Using Z specifications in category-partition testing, in *Systems Integrity, Software Safety, and Process Security: Building the Right System Right: Proceedings of the 7th Annual IEEE Conference on Computer Assurance* (*COMPASS '92*), pp. 3–10, IEEE Computer Society, Los Alamitos, California, 1992.

[2] P. Ammann and A.J. Offutt, Using formal methods to derive test frames in category-partition testing, in *Safety, Reliability, Fault Tolerance, Concurrency, and Real Time Security: Proceedings of the 9th Annual IEEE Conference on Computer Assurance* (*COMPASS '94*), pp. 69–79, IEEE Computer Society, Los Alamitos, California, 1994.

[3] R. Bache and M. Müllerburg, Measures of testability as a basis for quality assurance. *Software Engineering Journal* **5** (3), 1990, 86–92.

[4] M.J. Balcer, W.M. Hasling, and T.J. Ostrand, Automatic generation of test scripts from formal test specifications, in *Proceedings of the 3rd ACM Annual Symposium on Software Testing, Analysis, and Verification* (*TAV '89*), pp. 210–218, ACM Press, New York, 1989.

[5] 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* **7** (3), 1998, 250–295.

[6] T.Y. Chen, W.H. Kwok, and T.H. Tse, Improving the effectiveness of the classification-tree methodology, in *Proceedings of the 4th Annual IASTED International Conference on Software Engineering and Applications* (*SEA 2000*), ACTA Press, Calgary, Canada, pp. 43–48 (2000).

[7] T.Y. Chen and P.L. Poon, Improving the quality of classification trees via restructuring, in *Proceedings of the Asia-Pacific Software Engineering Conference* (*APSEC '96*), pp. 83–92, IEEE Computer Society, Los Alamitos, California, 1996.

[8] T.Y. Chen and P.L. Poon, Construction of classification trees via the classification-hierarchy table. *Information and Software Technology* **39** (13), 1997, 889–896.

[9] T.Y. Chen, P.L. Poon, and S.F. Tang, A systematic method for auditing user acceptance tests. *IS Audit and Control Journal* **5**, 1998, 31–36.

[10] 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), 1987, 509–517.

[11] A.L. Goel, Software reliability models: assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering* **11** (12), 1985, 1411–1423.

[12] M. Grochtmann and K. Grimm, Classification trees for partition testing. *Software Testing, Verification and Reliability* **3** (2), 1993, 63–82.

[13] M. Grochtmann, J. Wegener, and K. Grimm, Test case design using classification trees and the classification-tree editor CTE, in *Proceedings of 8th International Software Quality Week* (*QW '95*), Software Research Institute, San Francisco, California, 1995.

[14] R. Gupta and M.L. Soffa, Employing static information in the generation of test cases. *Software Testing, Verification and Reliability* **3** (1), 1993, 29–48.

[15] R.K. Iyer and I. Lee, Measurement-based analysis of software reliability, in *Handbook of Software Reliability Engineering* (M.R. Lyu, Ed), pp. 303–358, McGraw-Hill, New York, 1996.

[16] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and J.M. Voas, Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering* **18** (1), 1992, 33–44.

[17] J.D. Musa, Operational profiles in software-reliability engineering. *IEEE Software* **10** (2), 1993, 14–32.

[18] J. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin, The operational profile, in *Handbook of Software Reliability Engineering* (M.R. Lyu, Ed), pp. 167–216, McGraw-Hill, New York, 1996.

[19] A.J. Offutt and A. Irvine, Testing object-oriented software using the category-partition method, in *Proceedings of the 17th International Conference on Technology of Object-Oriented Languages and Systems* (*TOOLS 17*) (R.K. Ege, M. Singh, and B. Meyer, Eds), pp. 293–304, Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

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

[21] W.E. Perry, *Effective Methods for Software Testing*, Wiley, New York, 1995.

Appendix 1

The total number of combinations of classes for a classification tree can be calculated using the following algorithm:

Algorithm *calculate_combination* for Calculating the Total Number of Possible Combinations of Classes

In addition to the notation used in step (3) of *build_tree*, let $S^x_{\tau_i}$ be a subtree within the top-level subtree τ_i, with the class x as its root. Let $N(S^X_{\tau_i})$ and $N(S^x_{\tau_i})$ represent the total number of combinations of classes for $S^X_{\tau_i}$ and $S^x_{\tau_i}$, respectively.

- **For the Computation of N_p for the Whole Classification Tree:**

  Given a classification tree $\mathcal{T}$ with h top-level classifications, the number of potential test cases N_p of $\mathcal{T}$ is given by

$$N_p = \prod_{i=1}^{h} N(\tau_i)$$

- **For the Computation of $N(S^X_{\tau_i})$:**

  Suppose X has j_1 non-terminal classes (denoted by x_t, $t = 1, 2, \ldots, j_1$) and j_2 terminal classes. Then,

$$N(S^X_{\tau_i}) = j_2 + \sum_{t=1}^{j_1} N(S^{x_t}_{\tau_i}) \qquad (1)$$

- **For the Computation of $N(S^x_{\tau_i})$:**

  Suppose x has j child classifications denoted by Y_t, $t = 1, 2, \ldots, j$. Then,

$$N(S^x_{\tau_i}) = \prod_{t=1}^{j} N(S^{Y_t}_{\tau_i}) \qquad (2)$$

Appendix 2

The two properties associated with our integrated classification-tree methodology mentioned in Section 3.3 are proved as follows:

Proposition 1 (Preservation Property)
Let $\mathcal{T}$ be a classification tree and $\mathcal{T}'$ be the new tree after pruning a set of duplicated subtrees in step (3) *of build_tree. All the legitimate test cases identified from $\mathcal{T}$ can also be identified from $\mathcal{T}'$.*

Proof
Given m top-level classifications in $\mathcal{T}$, let $\tau_1, \tau_2, \ldots, \tau_m$ be the top-level subtrees.

Obviously, Proposition 1 is valid when (i) $m = 1$, or (ii) $m \geq 2$ but there are no duplicated subtrees across any pair of distinct top-level subtrees. In both cases, $\mathcal{T} \equiv \mathcal{T}'$.

Let us consider the situation where $m \geq 2$ and there are duplicated subtrees across some distinct top-level subtrees. Without loss of generality, suppose the subtrees $S^X_{\tau_1}, S^X_{\tau_2}, \ldots, S^X_{\tau_n}$ are duplicated across the distinct top-level subtrees $\tau_1, \tau_2, \ldots, \tau_n$, respectively. Suppose, further, that after the pruning process according to step (3) of *build_tree*, all these duplicated subtrees are removed except for the subtree(s) $S^X_{\tau_j}$ of one top-level subtree τ_j (where $1 \leq j \leq n$).

We can classify any feasible path in $\mathcal{T}$ into one of the following three cases:

(a) The path goes through $S^X_{\tau_j}$:

This path will remain intact after the pruning process.

(b) The path goes through one of the duplicated subtrees $S^X_{\tau_i}$ (where $1 \leq i \leq n$ and $i \neq j$):

Since the subtree(s) $S^X_{\tau_j}$ in τ_j are left intact after the pruning process, even though all the classifications and classes in $S^X_{\tau_i}$ are subsequently pruned, they can still be found in $S^X_{\tau_j}$. Hence there will be no loss of classifications and classes.

(c) The path does not go through any of the duplicated subtrees:

Obviously, such a path will also remain the same after the pruning process.

Hence, all the legitimate test cases identified from $\mathcal{T}$ can also be identified from $\mathcal{T}'$. ■

Proposition 2 (Convergence Property)
Let $\mathcal{T}$ be a classification tree and $\mathcal{T}'$ be the new tree after pruning a set of duplicated subtrees in step (3) *of build_tree. Let N_p be the number of potential test cases in $\mathcal{T}$ and N'_p be that in $\mathcal{T}'$. Then $N'_p \leq N_p$.*

Proof
It can be seen from *build_tree* that $\mathcal{T}'$ is equivalent to $\mathcal{T}$ with the duplicated $S^X_{\tau_i}$'s pruned from all but one τ_i. Thus, $N'_p \leq N_p$. ■