**To appear in *Software Engineering Research and Applications*,
C. V. Ramamoorthy, R. Y. Lee, and K. W. Lee (eds.),
Lecture Notes in Computer Science, vol. 3026, Springer, Berlin (2004)**

An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology ⋆, ⋆⋆, ⋆⋆⋆

A. Cain [1], T. Y. Chen [1], D. Grant [1], Pak-Lok Poon [2], Sau-Fun Tang [1, 3], and T. H. Tse [4]

[1] School of Information Technology
Swinburne University of Technology
Hawthorn 3122, Australia
[2] School of Accounting and Finance
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
e-mail: afplpoon@inet.polyu.edu.hk
phone: (+852) 2766 7072 fax: (+852) 2356 9550
[3] Department of Finance and Decision Sciences
Hong Kong Baptist University
Kowloon Tong, Kowloon, Hong Kong
[4] Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong

Abstract. Grochtmann and Grimm have developed the classification-tree method (CTM) to facilitate software testers in generating test cases from functional specifications. While the method is very useful, it is hindered by the lack of a systematic tree construction algorithm. This problem has been alleviated by Chen et al. via their "integrated" classification-tree methodology (ICTM). In this paper, we describe and discuss a prototype system ADDICT that is built on ICTM.

Keywords Automatic test case generation, black box testing, category-partition method, choice relation framework, classification-tree method, software testing

⋆⋆ A preliminary version of this paper was presented at the 1st ACIS International Conference on Software Engineering Research and Applications (SERA '03) [3].

⋆⋆⋆ This work is supported in part by a grant of the Research Grants Council of Hong Kong (Project No. HKU 7029/01E), a research and conference grant of The University of Hong Kong, and a grant from the Australian Research Council (ARC Discovery Project: DP 0345147).

1 Introduction

The generation of test cases is an important aspect in software testing because it affects the scope and, hence, the quality of the process [1, 7]. This importance has inspired researchers to develop various test case generation methods. Among these researchers, Grochtmann and Grimm [8] have developed the *classification-tree method* (CTM). The major concept of the method is to generate test cases via the construction of classification trees (which we shall denote by $\mathcal{T}$s). Although the concept of CTM is promising, this method has a major weakness — the absence of a systematic tree construction algorithm. As a result, users of CTM are left with a loosely defined task of constructing $\mathcal{T}$s. For complicated specifications, this construction task could be difficult and hence prone to human errors. If a $\mathcal{T}$ is wrongly constructed, the quality of the resulting test cases generated from it will be adversely affected.

The problem of the absence of a tree construction algorithm is alleviated by Chen et al. via their integrated classification-tree methodology (ICTM) [5]. With this methodology, software testers can construct $\mathcal{T}$s by using a systematic tree construction algorithm. In this paper, we discuss the development and functionality of a prototype system ADDICT (which stands for **A**utomate**D** test **D**ata generation using the **I**ntegrated **C**lassification-**T**ree methodology) built upon ICTM.

The rest of the paper is organized as follows. Section 2 outlines the major concept of ICTM [5]. Section 3 describes in detail the hardware and software platforms on which ADDICT is built, the various system features of ADDICT, and the major contribution of ADDICT. Section 4 discusses our planned extension to ADDICT. Section 5 describe other work related to CTM and ICTM. Finally, Sect. 6 summarizes and concludes the paper.

2 Overview of the Integrated Classification-Tree Methodology (ICTM)

Basically, ICTM [5] helps testers generate test cases from specifications via the construction of $\mathcal{T}$s. The tree construction task is supported by a predefined algorithm. ICTM consists of the following steps:

(1) Decompose the specification into several *functional units* $\mathcal{U}$s that can be tested independently. For each $\mathcal{U}$ selected for testing, repeat steps (2) to (7) below.

(2) Identify classifications and their associated classes for the selected $\mathcal{U}$. *Classifications* are different criteria for partitioning the input domain of the selected $\mathcal{U}$, whereas *classes* are disjoint subsets of values for each classification. For every classification $[X]$, its associated classes should partition the possible values of $[X]$ completely.[5] The grouping of certain values in a single class $|X:x|$ indicates the belief that a test case with any value in $|X:x|$ is essentially as good as one with any other value in that class [11].

[5] In this paper, classifications are enclosed by square brackets [ ] while classes are enclosed by vertical bars | |. Furthermore, the notation $|X:x|$ denotes class x in classification X.

(3) Construct a *classification-hierarchy table* $\mathcal{H}_{\mathcal{U}}$ for $\mathcal{U}$, which captures the hierarchical relation for each pair of classifications.

(4) Construct a *classification tree* $\mathcal{T}_{\mathcal{U}}$ from $\mathcal{H}_{\mathcal{U}}$.

(5) Construct a *combination table* from $\mathcal{T}_{\mathcal{U}}$. Various combinations of classes can then be selected from the table according to a set of predefined selection rules. Each of these combinations of classes is called a *potential test frame B*.

(6) Check all *B*'s against $\mathcal{U}$, with a view to identifying whether they are complete or incomplete. Given a *B*, if a standalone input to $\mathcal{U}$ can be formed by selecting one element from every class in *B*, then *B* is a *complete test frame* (denoted by B^c). Otherwise, *B* is *incomplete*. Incomplete test frames are not useful to testing and, hence, should be discarded before testing commences.

(7) From each B^c, construct a test case by selecting one element from each class in B^c.

In step (6) above, two potential reasons for a *B* to be incomplete are: (a) *B* contains insufficient classes to form a standalone input to $\mathcal{U}$, and/or (b) the combination of classes in *B* contradicts $\mathcal{U}$. Readers may refer to [5] for details. Nevertheless, when we describe the various system features of ADDICT in Sect. 3.2, we shall elaborate on the above steps with examples.

3 ADDICT: A Prototype System for Automated Test Case Generation

3.1 Technical Details

We have carefully considered the hardware and software platforms on which ADDICT should be built. To improve its applicability, we have implemented ADDICT on the standard PC platform with Microsoft Windows as the operating system. ADDICT is written in Delphi, which is a Pascal-based object-oriented programming language. We have applied object-oriented techniques when designing and coding ADDICT. For examples, a classification $[X]$ is an aggregation of classes $|X{:}\,x_1|$, $|X{:}\,x_2|$, ..., $|X{:}\,x_k|$, and $[X]$ is related to other classifications through hierarchical relationships. In general, ADDICT does not impose a maximum limit on the number of classifications and classes, as long as the available memory in the PC can support them. The current version of ADDICT supports steps (2) to (5) of ICTM outlined in Sect. 2 above.

3.2 Functionality of ADDICT

We use a commercial specification, denoted by PURCHASE, to explain each step of ICTM mentioned in Sect. 2 and to describe the various features of ADDICT. Part of the specification is listed below:

Part of the Specification PURCHASE **for the Program** $\mathcal{P}_{\texttt{PURCHASE}}$**:**

XYZ is an international bank that issues credit cards to approved customers. ... For each purchase, $\mathcal{P}_{\texttt{PURCHASE}}$ shall accept the transaction details together with the various

information of the credit card. Thereafter, validation of these details is performed in order to determine whether the purchase should be approved. The following are the various inputs to $\mathcal{P}_{\texttt{PURCHASE}}$:

(a) Details of Credit Cards:
- **Class of Credit Card:** Either "Gold" or "Classic".
- **Credit Limit of Credit Card:** For gold credit cards, the credit limit is either \$5 000 or \$6 000. For classic credit cards, the credit limit is either \$2 000 or \$3 000.
- ...

(b) Details of Purchase:
- **Current Purchase Amount:** It can be any amount greater than \$0.00.
- ...

Step (1) of ICTM (Decomposition of Specification):

The first step is to decompose PURCHASE into a number of independent functional units $\mathcal{U}$s. In our case, because of the simplicity of PURCHASE, no decomposition is needed. In other words, the entire specification can be treated as one functional unit denoted by $\mathcal{U}_{\texttt{PURCHASE}}$.

Step (2) of ICTM (Identification of Classifications and Classes):

From $\mathcal{U}_{\texttt{PURCHASE}}$, the tester identifies 9 classifications and 22 classes. The number of classes contained in a classification ranges from 2 to 5. The following lists four examples of these classifications together with their associated classes:

(a) Classification [Class of Credit Card], with |Class of Credit Card: Gold| and |Class of Credit Card: Classic| as its two associated classes.

(b) Classification [Credit Limit of Gold Card], with |Credit Limit of Gold Card: \$5,000| and |Credit Limit of Gold Card: \$6,000| as its two associated classes.

(c) Classification [Credit Limit of Classic Card], with |Credit Limit of Classic Card: \$2,000| and |Credit Limit of Classic Card: \$3,000| as its two associated classes.

(d) Classification [Current Purchase Amount (*PA*)], with |Current Purchase Amount: $PA \leq$ \$2 000.00|, |Current Purchase Amount: \$2 000.00 $< PA \leq$ \$3 000.00|, |Current Purchase Amount: \$3 000.00 $< PA \leq$ \$5 000.00|, |Current Purchase Amount: \$5 000.00 $< PA \leq$ \$6 000.00|, and |Current Purchase Amount: $PA >$ \$6 000.00| as its five associated classes.

It can be seen from (d) above that a class can be defined for a range of values and, hence, although all the classes in a classification $[X]$ should cover the input domain relevant to $[X]$, the number of classes in $[X]$ is not necessarily large.

Consider Fig. 1 which depicts an input screen provided by ADDICT for entering the full and short names of classifications and classes. In this figure, the tester has defined classification [Credit Limit of Gold Card] in the upper-left box, and class |Credit Limit of Gold Card: \$5 000| in the bottom-right box. Additionally, the tester is adding class |Credit Limit of Gold Card: \$6 000| through the upper-right box.

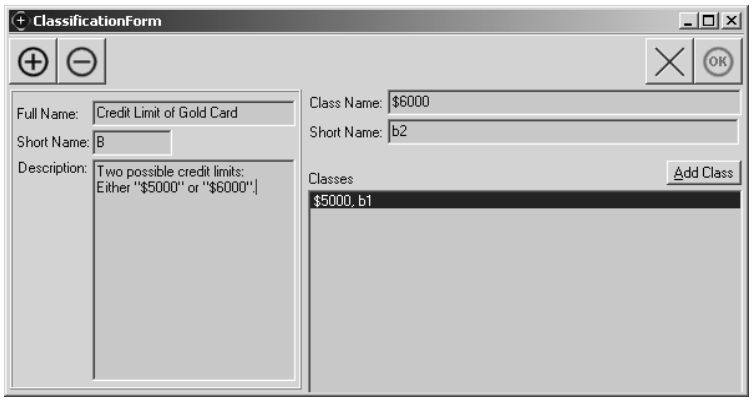


Fig. 1. Input screen for classifications and classes

Step (3) of ICTM (Construction of Classification-Hierarchy Table):

After entering all the classifications and classes into ADDICT, the next step is to construct the classification-hierarchy table $\mathcal{H}_{\texttt{PURCHASE}}$ for PURCHASE. Here, the tester defines the hierarchical relation for each pair of classifications $[X]$ and $[Y]$ (denoted by $[X] \rightarrow [Y]$). There are four possible types of hierarchical relation as follows:

- $[X]$ is a ***loose ancestor*** of $[Y]$ (denoted by $[X] \Leftrightarrow [Y]$),
- $[X]$ is a ***strict ancestor*** of $[Y]$ (denoted by $[X] \Rightarrow [Y]$),
- $[X]$ is ***incompatible with*** $[Y]$ (denoted by $[X] \sim [Y]$), and
- $[X]$ has ***other relations*** with $[Y]$ (denoted by $[X] \otimes [Y]$).

In the above, the symbols "$\Leftrightarrow$", "$\Rightarrow$", "$\sim$", and "$\otimes$" are known as *hierarchical operators*. Readers may refer to [5] for details, especially the conditions in determining the correct hierarchical relation for $[X] \rightarrow [Y]$. Note that the conditions associated with each of the above hierarchical relations are mutually exclusive and exhaustive and, hence, $[X] \rightarrow [Y]$ is well defined. These hierarchical relations will determine the relative position of $[X]$ and $[Y]$ in $\mathcal{T}$. For example, $[X] \Rightarrow [Y]$ corresponds to the situation where $[X]$ will appear as either a parent or an ancestor of $[Y]$ in $\mathcal{T}$.[6]

Figure 2 depicts an input screen to capture the constraints of [Credit Limit of Gold Card] on [Credit Limit of Classic Card]. These captured constraints will be used by ADDICT to determine the appropriate hierarchical operator for [Credit Limit of Gold Card] $\rightarrow$ [Credit Limit of Classic Card]. In the input screen, the tester indicates that |Credit Limit of Gold Card: \$5 000| cannot be combined with any class (that is, |Credit Limit of Classic Card: \$2 000| and |Credit Limit of Classic Card: \$3 000|) in [Credit Limit of Classic Card] to form part of any complete test frame. By clicking the "Make

[6] For the *parent-child* relation, a classification is "directly" placed under one or more classes of another classification. For the *ancestor-descendant* relation, a classification is "indirectly" placed under one or more classes of another classification.

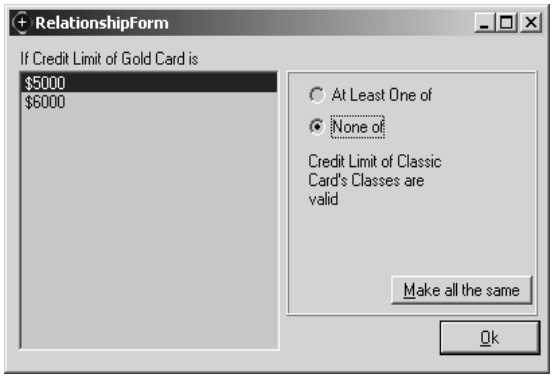


Fig. 2. Input screen for constraints between a pair of classifications

`all the same`" button near the bottom-right part of the input screen, the tester also indicates that the constraint of |Credit Limit of Gold Card: $6 000| on all the classes in [Credit Limit of Classic Card] is the same as that of |Credit Limit of Gold Card: $5 000| on all the classes in [Credit Limit of Classic Card]. This saves the effort in defining all constraints individually. Based on the entered constraints in Fig. 2, ADDICT will automatically assign the hierarchical operator "$\sim$" to [Credit Limit of Gold Card] $\rightarrow$ [Credit Limit of Classic Card]. In short, ADDICT will determine and assign the appropriate hierarchical operator to $[X] \rightarrow [Y]$, based on the captured constraints of $[X]$ on $[Y]$.

Figure 3 depicts the completed $\mathcal{H}_{\texttt{PURCHASE}}$ with every element in it contains a hierarchical operator and corresponds to the hierarchical relation between a pair of classifications.[7] We use t_{ij} to denote the element in the ith row and the jth column of $\mathcal{H}_{\mathcal{U}}$. Consider, for example, t_{23} in $\mathcal{H}_{\texttt{PURCHASE}}$. It contains the hierarchical operator "$\sim$", and corresponds to [Credit Limit of Gold Card] $\sim$ [Credit Limit of Classic Card]. Note that the background color of all unassigned elements in $\mathcal{H}_{\texttt{PURCHASE}}$ is initially set to "blue". Once the constraints corresponding to an element t_{ij} have been entered and a hierarchical operator has been assigned to it, the background color of that element will change to "white".

With regard to the construction of $\mathcal{H}_{\mathcal{U}}$, the following features provided by ADDICT are worth mentioning:

(a) A constraint of ICTM is that the parent-child or ancestor-descendant hierarchical relation must be *anti-symmetric* for any pair of classifications. Otherwise a $\mathcal{T}$ cannot be constructed. In other words, $[X] \Rightarrow [Y]$ must imply $[X] \not\Rightarrow [Y]$. Software

[7] Note that, short names instead of full names for the classifications (both names are entered via the input screen as depicted in Fig. 1) are displayed as row and column headings in $\mathcal{H}_{\texttt{PURCHASE}}$. The idea is to fit the entire $\mathcal{H}_{\texttt{PURCHASE}}$ into the screen. In the situation where $\mathcal{H}_{\texttt{PURCHASE}}$ is too large (because of too many classifications) that exceeds the size of the screen, then vertical and horizontal scroll bars can be used to view different parts of $\mathcal{H}_{\texttt{PURCHASE}}$.

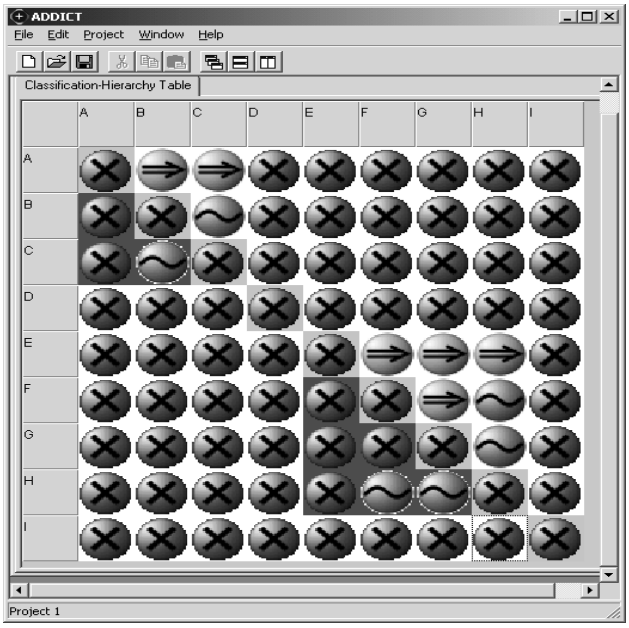


Fig. 3. Classification-hierarchy table $\mathcal{H}_{\texttt{PURCHASE}}$ for PURCHASE

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 (see [5] for details).

Regarding this issue, ICTM helps testers identify such unwarranted situations by means of the hierarchical operator "$\Leftrightarrow$". Whenever $[X] \Leftrightarrow [Y]$ is being defined, we know that a symmetric parent-child or ancestor-descendant hierarchical relation occurs between $[X]$ and $[Y]$. In this case, testers will be alerted to redefine $[X]$ and $[Y]$ (and their associated classes) so as to prevent a loop in $\mathcal{T}$.

Consider t_{12} and t_{21} in $\mathcal{H}_{\texttt{PURCHASE}}$ of Fig. 3. They correspond to ([Class of Credit Card] $\Rightarrow$ [Credit Limit of Gold Card]) and ([Credit Limit of Gold Card] $\otimes$ [Class of Credit Card]), respectively. Suppose, during the process of entering the constraints between these two classifications, the tester has made a mistake and eventually caused ADDICT to assign the hierarchical operator "$\Leftrightarrow$" to both t_{12} and t_{21}. Accordingly, the background color of t_{12} and t_{21} will change from "white" to "red", thus alerts the tester that symmetric parent-child or ancestor-descendant hierarchical relations occur. Note that, in this case, the unwarranted situation happens to occur because of an input error; symmetric parent-child or ancestor-descendant hierarchical relations in fact do not exist in $\mathcal{U}_{\texttt{PURCHASE}}$. In some other cases, however, this occurrence may result from correct inputs because symmetric parent-child or ancestor-descendant hierarchical relations do exist between some pairs of classifications identified from $\mathcal{U}$.

(b) In [5], Chen et al. have identified three properties of the hierarchical relations 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 $[Y] \otimes [X]$.

Using these properties, ADDICT provides a certain degree of automatic deduction and consistency check during the construction of $\mathcal{H}_{\mathcal{U}}$. Examples are given as below:

(i) **Automatic deduction:** Consider Fig. 2 again. This input screen is used to enter the constraints of each class in [Credit Limit of Gold Card] on [Credit Limit of Classic Card]. The entered constraints cause ADDICT to assign the hierarchical operator "$\sim$" to [Credit Limit of Gold Card] $\rightarrow$ [Credit Limit of Classic Card]. Later, without automatic deduction, the tester is required to enter the constraints of each class in [Credit Limit of Classic Card] on [Credit Limit of Gold Card] via another input screen similar to Fig. 2, if such constraints have not yet been entered. Now, by using Prop. 2, this requirement no longer exists because ADDICT will automatically deduce the hierarchical operator for [Credit Limit of Classic Card] $\rightarrow$ [Credit Limit of Gold Card] to be "$\sim$". Accordingly, the background color for t_{32} (which corresponds to [Credit Limit of Classic Card] $\sim$ [Credit Limit of Gold Card]) will change from "blue" to "green" to inform the tester that the hierarchical operator for t_{32} is automatically deduced (note that the background color for all the table elements whose hierarchical operators are manually defined is "white"). Besides Prop. 2, ADDICT will also provide automatic deduction based on Prop. 1. In fact, with the feature of automatic deduction, only about three-quarters of the hierarchical relations in $\mathcal{H}_{\texttt{PURCHASE}}$ have to be manually defined.

(ii) **Consistency Checking:** Consider t_{12} and t_{21} in $\mathcal{H}_{\texttt{PURCHASE}}$ in Fig. 3 again, which correspond to ([Class of Credit Card] $\Rightarrow$ [Credit Limit of Gold Card]) and ([Credit Limit of Gold Card] $\otimes$ [Class of Credit Card]), respectively. Suppose,

- The constraints for t_{21} are entered before that for t_{12}.
- The constraints for t_{21} are entered correctly, causing ADDICT to assign the hierarchical operator "$\otimes$" to t_{21}.
- Thereafter, the tester has made a mistake during the entry of the constraints for t_{12}, causing ADDICT to incorrectly assign the hierarchical operator "$\sim$" to t_{12}.

This mistake is undesirable because incorrect hierarchical relations will eventually result in the generation of incomplete test frames, or the omission of some complete test frames. Regarding this problem, ADDICT provides a consistency check for the defined hierarchical relations. In fact, the incorrect hierarchical operator "$\sim$" for t_{12} will be detected as an inconsistency by ADDICT with reference to Prop(s). (2) and (3) mentioned above. This is because the combination of ([Class of Credit Card] $\sim$ [Credit Limit of Gold Card]) and ([Credit Limit of Gold Card] $\otimes$ [Class of Credit Card]) contradicts these two properties. Accordingly, the background of t_{12} and t_{21} in $\mathcal{H}_{\texttt{PURCHASE}}$ will change from

"white" to "red" to alert the tester to take correction actions. An alert message box will also be displayed automatically by ADDICT to inform the tester about the inconsistency (see Fig. 4).

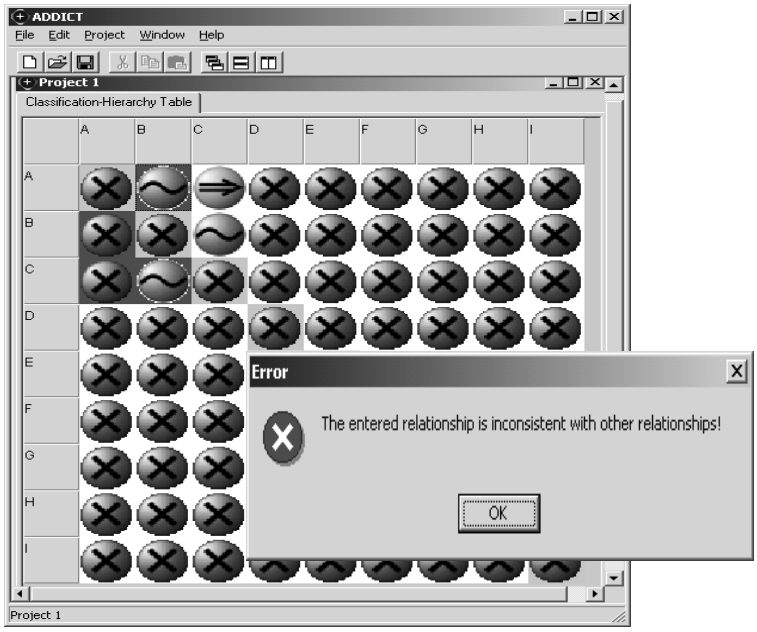


Fig. 4. Message box to alert users about inconsistent hierarchical relations

In summary, ADDICT adopts the following principles in order to improve on the effectiveness and efficiency of table construction:

- 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}.

Step (4) of ICTM (Construction of Classification Tree):

Based on the completed $\mathcal{H}_{\texttt{PURCHASE}}$ in Fig. 3, ADDICT will automatically construct the corresponding classification tree $\mathcal{T}_{\texttt{PURCHASE}}$ (see Fig. 5), based on a predefined tree construction algorithm provided in [5]. Similarly to $\mathcal{H}_{\texttt{PURCHASE}}$, short names are used for the classifications and classes in displaying $\mathcal{T}_{\texttt{PURCHASE}}$, and vertical and horizontal scroll bars can be used to view different parts of $\mathcal{T}_{\texttt{PURCHASE}}$ if the tree is too large to fit into the screen.

In step (5) of ICTM described in Sect. 2, potential test frames B's are generated by selecting combinations of classes from the combination table of $\mathcal{T}$, based on certain selection rules. Thereafter, the combination of classes in every B has to be checked against $\mathcal{U}$, with a view to classifying B either as a complete test frame B^c or as an

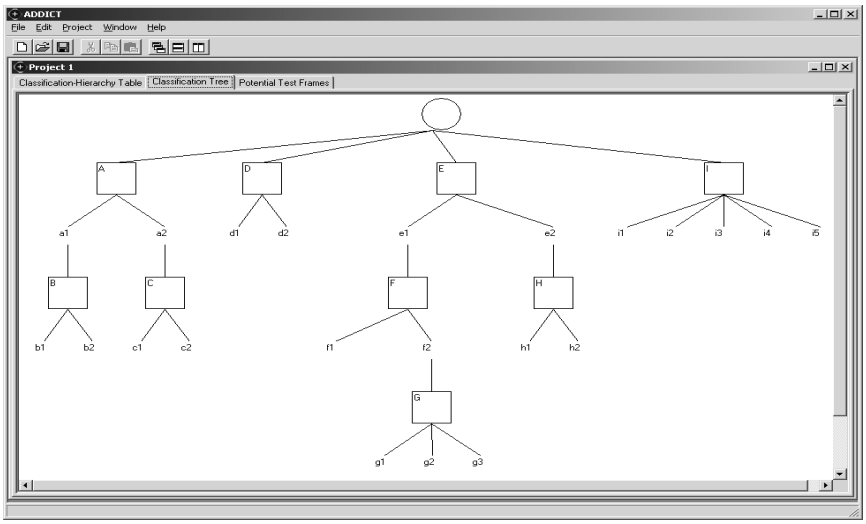


Fig. 5. Classification tree $\mathcal{T}_{\texttt{PURCHASE}}$ for PURCHASE

incomplete test frame. The reason for checking is because, occasionally, a $\mathcal{T}$ may not be able to capture all the constraints and relationships among classifications identified from $\mathcal{U}$. This problem results in the selection of some incomplete test frames from the combination table of $\mathcal{T}$.

Let N_B and N_{B^c} denote the total number of B's and (B^c)'s, respectively, selected from the combination table of $\mathcal{T}$. In [5], Chen et al. define an effectiveness metric $E_{\mathcal{T}}$ for any $\mathcal{T}$ as:

$$E_{\mathcal{T}} = \frac{N_{B^c}}{N_B} \tag{1}$$

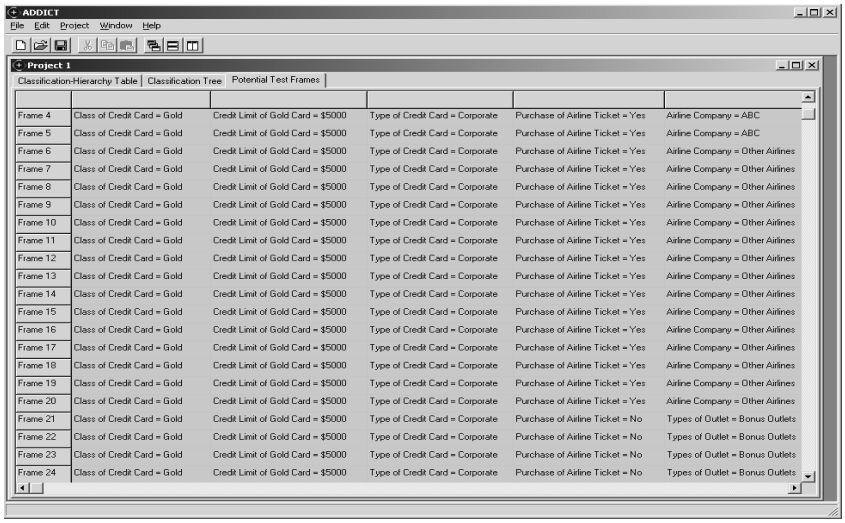
$E_{\mathcal{T}}$ is defined as such based on the argument that $\mathcal{T}$ is merely a means to construct (B^c)'s for testing. The ideal situation is that all B's are complete (that is, $N_B = N_{B^c}$) and, hence, $E_{\mathcal{T}} = 1$. Obviously, a small value of $E_{\mathcal{T}}$ is undesirable since more effort is required to identify all the incomplete test frames. Furthermore, this manual identification process is prone to human errors, especially when N_B is large. If some (B^c)'s are somehow mistakenly classified as incomplete and hence not being used, the comprehensiveness of testing will be adversely affected.

Chen et al. [5] observe that a major reason for a small value of $E_{\mathcal{T}}$ is the occurrence of duplicated subtrees in $\mathcal{T}$. To deal with this problem, they develop a classification tree restructuring technique to suppress the occurrence of duplicated subtrees in $\mathcal{T}$. This restructuring technique is part of their integrated classification tree construction algorithm. Two important properties of this restructuring technique are: (i) to reduce the value of N_B by pruning some duplicated subtrees from $\mathcal{T}$, and (ii) to retain all the (B^c)'s and, hence, N_{B^c} remains unchanged. Because of these two properties, the value of the effectiveness metric $E_{\mathcal{T}}$ can be increased. Readers may refer to [5] for details.

In ADDICT, the construction of the resulting $\mathcal{T}$ is performed on an incremental basis — classifications and classes are firstly assembled together to form subtrees, which in turn are joined together to form the resulting $\mathcal{T}$. During the tree construction process, ADDICT will automatically detect the occurrence of duplicated subtrees. If duplicated subtrees do exist, ADDICT will apply the tree restructuring technique by Chen et al., in order to increase the value of $E_{\mathcal{T}}$ of the resulting $\mathcal{T}$. Note that the tree construction process, that incorporates the restructuring technique, is performed by ADDICT in a fully automatic manner without human intervention.

Step (5) of ICTM (Construction of Combination Table and Selection of Potential Test Frames):

With $\mathcal{T}_{\texttt{PURCHASE}}$ in Fig. 5, the next step is to construct the corresponding combination table, from which *B*'s can be selected. This step is rather straightforward by following some selection rules given in [5], which will not be repeated here. Same as $\mathcal{T}_{\texttt{PURCHASE}}$, the construction of the combination table and the selection of *B*'s are done by ADDICT automatically. In our case, a total of 240 *B*'s will be selected from the combination table of $\mathcal{T}_{\texttt{PURCHASE}}$ by ADDICT. Figure 6 shows a partial list of *B*'s constructed by ADDICT.



Frame 4	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = ABC
Frame 5	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = ABC
Frame 6	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 7	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 8	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 9	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 10	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 11	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 12	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 13	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 14	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 15	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 16	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 17	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 18	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 19	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 20	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = Yes	Airline Company = Other Airlines
Frame 21	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = No	Types of Outlet = Bonus Outlets
Frame 22	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = No	Types of Outlet = Bonus Outlets
Frame 23	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = No	Types of Outlet = Bonus Outlets
Frame 24	Class of Credit Card = Gold	Credit Limit of Gold Card = $5000	Type of Credit Card = Corporate	Purchase of Airline Ticket = No	Types of Outlet = Bonus Outlets

Fig. 6. Partial list of potential test frames for PURCHASE

Step (6) of ICTM (Differentiation between Complete and Incomplete Test Frames):

As discussed in step (4) above, the tester has to check all the *B*'s with $\mathcal{U}_{\texttt{PURCHASE}}$ to see whether any of them is incomplete. In our case, no incomplete test frame exists and, hence, all the 240 *B*'s are also complete.

Step (7) of ICTM (Construction of Test Cases):

For each of the 240 (B^c)'s, the tester selects an element from each class contained in B^c to form a test case. Consider, for example, the following B_1^c for $\mathcal{U}_{\texttt{PURCHASE}}$ generated by ADDICT:

> $\{|$Class of Credit Card: Gold$|$, $|$Credit Limit of Gold Card: \$6 000$|$, $|$Current Purchase Amount (PA): \$5 000.00 $< PA \leq$ \$6 000.00$|, \ldots\}$

A possible test case for B_1^c is:

> (Class of Credit Card = Gold, Credit Limit of Gold Card = \$6 000, Current Purchase Amount = \$5 123.40, ...)

Obviously, a total of 240 test cases will be constructed in this step for testing $\mathcal{U}_{\texttt{PURCHASE}}$.

3.3 Major Contribution of ADDICT

As mentioned earlier in the paper, the main purpose of ICTM and ADDICT is to provide a systematic method for the construction of $\mathcal{T}$s from specifications. This feature does not exist in CTM. Hence, users of CTM have to construct $\mathcal{T}$s in an ad hoc manner based on their knowledge and experience. This ad hoc approach does not provide assurance on the quality of the constructed $\mathcal{T}$s. If these $\mathcal{T}$s are incorrectly constructed, some (B^c)'s may not be generated and, hence, parts of the system that contain faults may not be tested. In this respect, the contribution of ICTM and ADDICT is obvious. Readers may note that the effectiveness of ICTM and ADDICT is greatly improved via the automatic detection of symmetric parent-child or ancestor-descendant hierarchical relations, the automatic deduction and consistency checking of hierarchical relations, and the automatic detection and removal of duplicated subtrees (in order to improve on the effectiveness metric $\mathcal{E}_{\mathcal{T}}$).

4 Planned Extensions to ADDICT

The current version of ADDICT supports steps (2) to (5) of ICTM outlined in Sect. 2. For step (1), the decomposition of the specification into several $\mathcal{U}$s is not a trivial task that can be easily automated. Similarly, it is difficult, if not impossible, to automate step (6) regarding the differentiation between complete and incomplete test frames.

With regard to step (7), we plan to extend the system features provided by the current version of ADDICT in the following two ways:

(a) After the automatic construction of the combination table and the selection of B's in step (5), the next task is to construct a test case from every B. This task can be performed automatically by ADDICT by arbitrarily selecting one element from every class contained in B. Note that, in this approach, the generated test cases may contradict $\mathcal{U}$ because the B's have not been checked against $\mathcal{U}$ to determine whether they are complete or incomplete. Hence, the tester has to check all the generated test cases against $\mathcal{U}$ to see which of them are useful for testing.

(b) In (a) above, only one test case is generated for each *B*. (Similarly, in the original ICTM, only one test case is generated for each B^c.) In our planned extension to ADDICT, the approach in (a) can be made more flexible so that one or more test cases can be generated for each *B*. This caters for the situation where the tester can afford to test $\mathcal{U}$ with more test cases.

5 Related Work

Finally, we would like to compare CTM and ICTM with other related work, thus allowing readers to have a better grasp of the current state of research and practice:

(1) Ostrand et al. [2, 11] have developed the *category-partition method* (CPM) for generating test cases from specifications. The basic approach of CPM is very similar to that of CTM/ICTM — all of them aim at constructing a model of the constraints in the input domain so that combinations of compatible classes[8] can be generated and combinations of incompatible classes can be suppressed as far as possible. The main difference between CPM and CTM/ICTM is how the constraints among classes are captured. While the former captures the constraints via the notion of a formal test specifications (which is a list of categories[9], choices, and constraints in textural format), the latter capture these constraints by means of a classification tree $\mathcal{T}$. CPM has also been enhanced by Chen et al. [6] via the *choice relation framework* so that the test case generation process can be more systematic.

(2) Singh et al. [12] have developed a technique to generate test cases from Z specifications by combining CTM with disjunctive normal forms (DNFs). In this technique, "high-level" test cases are first generated from the Z specification via the construction of a $\mathcal{T}$. These high-level test cases are then refined by generating a DNF for them. Also working on Z, Hierons et al. [9] have introduced an approach that extracts predicates from a Z specification and constructs a $\mathcal{T}$ from these predicates, thus showing how the construction of a $\mathcal{T}$ can be semi-automated based on a formal specification.

Readers may note that the work described in [9] and [12] mainly focuses on the application of CTM to Z specifications, whereas our work in this paper is more general, in the sense that our prototype system ADDICT does not impose any limitation on the type of specification, as long as a set of classifications and classes can be identified.

(3) In [10], Lehmann and Wegener have described a classification-tree editor (CTE) known as CTE XL (eXtended Logics). This editor is used to solve some weaknesses they have identified in CTM. An example of such weaknesses is that CTM does not provide a feature to specify logical dependencies between classes. Hence, when selecting potential test frames from the combination table, software testers have to take care of the logical compatibility of the classes themselves.

The objective of our work is quite different from that of [10]. Our prototype ADDICT helps testers construct $\mathcal{T}$s from specifications. On the other hand, CTE

[8] "Classes" in CTM/ICTM are known as "choices" in CPM.

[9] "Classifications" in CTM/ICTM are known as "categories" in CPM.

XL is mainly a classification-tree "editor" rather than a "generator", and requires testers to construct a $\mathcal{T}$ by themselves, usually in an ad hoc manner.

(4) At the initial stage of CTM/ICTM, software testers have to identify a set of classifications and classes. Owing to the absence of a systematic identification technique, this identification process is currently performed in an ad hoc approach. Chen et al. [4] argue that this approach does not provide reasonable assurance on the quality of the identified classifications and classes, and hence on the quality of the resulting test cases. They have performed case studies to find out the common mistakes made by software testers when they identify classifications and classes from specifications in such an ad hoc approach.

6 Summary and Conclusion

In this paper, we have introduced ICTM and outlined its major steps. This is followed by discussions of the technical details and system features of ADDICT, particularly the automatic detection of symmetric parent-child or ancestor-descendant hierarchical relations, the automatic deduction and consistency checking of hierarchical relations, and the automatic detection and removal of duplicated subtrees. We have also highlighted the major contribution of ADDICT and two possible areas for extending the current version of ADDICT in order to make it more useful. We believe that ICTM is a viable method for generating test cases from specifications, especially with the support of appropriate automated tools such as ADDICT. We plan to perform case studies to further investigate the contributions of ADDICT, especially with respect to the application of the system to the testing of real-life software.

References

1. P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. *Safety, Reliability, Fault Tolerance, Concurrency, and Real Time Security: Proceedings of the 9th Annual IEEE Conference on Computer Assurance* (*COMPASS '94*), pages 69–79. Los Alamitos, California: IEEE Computer Society Press, 1994.
2. M. J. Balcer, W. M. Hasling, and T. J. Ostrand. Automatic generation of test scripts from formal test specifications. *Proceedings of the 3rd ACM Annual Symposium on Software Testing, Analysis, and Verification* (*TAV '89*), pages 210–218. New York: ACM Press, 1989.
3. A. Cain, T. Y. Chen, D. Grant, P.-L. Poon, S.-F. Tang, and T. H. Tse. ADDICT: a prototype system for automated test data generation using the integrated classification-tree methodology. *Proceedings of the 1st ACIS International Conference on Software Engineering Research and Applications* (*SERA '03*), pages 76–81. Mt. Pleasant, Michigan: International Association for Computer and Information Science, 2003.
4. T. Y. Chen, P.-L. Poon, S.-F. Tang, and T. H. Tse. An experimental analysis of the identification of categories and choices from specifications. *Proceedings of the 3rd ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (*SNPD 2003*), pages 99–106. Mt. Pleasant, Michigan: International Association for Computer and Information Science, 2002.
5. T. Y. Chen, P.-L. Poon, and T. H. Tse. An integrated classification-tree methodology for test case generation. *International Journal of Software Engineering and Knowledge Engineering*, 10(6):647–679, 2000.

6. T. Y. Chen, P.-L. Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. *IEEE Transactions on Software Engineering*, 29(7):577–593, 2003.
7. 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, 1987.
8. M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.
9. R. M. Hierons, M. Harman, and H. Singh. Automatically generating information from a Z specification to support the classification tree method. Volume 2651 of *Lectures Notes in Computer Science*, pages 388–407. Berlin, Heidelberg: Springer-Verlag, 2003.
10. E. Lehmann and J. Wegener. Test case design by means of the CTE XL. *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review* (*EuroSTAR 2000*), 2000.
11. 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, 1988.
12. H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. *Proceedings of the 1st IEEE International Conference on Formal Engineering Methods* (*ICFEM '97*), pages 81–90. Los Alamitos, California: IEEE Computer Society Press, 1997.