Title	A choice relation framework for supporting category-partition test case generation
Author(s)	Chen, TY; Poon, PL; Tse, TH
Citation	Ieee Transactions On Software Engineering, 2003, v. 29 n. 7, p. 577-593
Issued Date	2003
URL	http://hdl.handle.net/10722/43665
Rights	Creative Commons: Attribution 3.0 Hong Kong License

A Choice Relation Framework for Supporting Category-Partition Test Case Generation

T.Y. Chen, Pak-Lok Poon, *Member*, *IEEE*, and T.H. Tse, *Senior Member*, *IEEE*

Abstract—We describe in this paper a choice relation framework for supporting category-partition test case generation. We capture the constraints among various values (or ranges of values) of the parameters and environment conditions identified from the specification, known formally as choices. We express these constraints in terms of relations among choices and combinations of choices, known formally as test frames. We propose a theoretical backbone and techniques for consistency checks and automatic deductions of relations. Based on the theory, algorithms have been developed for generating test frames from the relations. These test frames can then be used as the basis for generating test cases. Our algorithms take into consideration the resource constraints specified by software testers, thus maintaining the effectiveness of the test frames (and hence test cases) generated.

Index Terms—Category-partition testing, choice relation framework, choice relation table, specification-based testing, test case construction, test frame.

1 Introduction

According to various authors [10], [14], [18], software testing is a labor-intensive and expensive process, which may account for 50 percent of the total project cost. In order to improve on its effectiveness, testing should be well planned and organized. In particular, the construction of test cases is an important aspect because it affects the scope and, hence, the quality of the process [2], [8], [11]. This inspired various researchers to develop test case construction methods.

Among them, Ostrand and Balcer [17] have developed the *category-partition method* (CPM). A *category* is defined as "a major property or characteristic of a parameter or an environment condition." An example is the "Account Balance" in a typical accounting application. Such categories can easily be identified from the functional specification of a system. Each category is partitioned into a set of *choices*, which represent "all the different kinds of values that are possible for the category." Examples are "Account Balance $\geq$ 0" and "Account Balance $<$ 0." Then, all valid combinations of choices are generated as *test frames*. Invalid combinations of choices are suppressed via various constraints. Finally, test cases are constructed from the generated test frames. Following up on the work of Ostrand and Balcer, several studies on CPM were conducted. For instance, Amla and Ammann [1] and Ammann and Offutt [2] studied the viability of applying the method to Z specifications. Offutt and Irvine [16] investigated the fault-detection effectiveness of CPM when applied to object-oriented programs.

Our study of CPM reveals the following problems:

1. All the constraints among choices must be defined manually. This can be ineffective and prone to human errors in real-life situations where there are many such constraints.
2. There is no precise mechanism for checking for consistency among constraints. This may affect the correctness and completeness of the test frames generated.
3. The generator for processing the test specification is meant to be run repeatedly, with additional constraints being imposed in each round, thereby reducing the number of test frames generated, until the software tester can afford to run the test cases generated from test frames [17]. Such an approach can be avoided if resource constraints are considered during, rather than after, the test frame generation process.

To address these problems, we propose a choice relation framework to support CPM. Our framework includes the following features:

- a more rigorous approach for representing different types of constraints among individual choices,
- consistency checks of specified constraints among choices,
- automatic deductions of new constraints among choices whenever possible, and
- a more effective test frame construction process.

The rest of this paper is structured as follows: Section 2 outlines the major steps of CPM. Section 3 is the core of the paper, proposing a choice relation framework for CPM. Section 4 describes the work related to category-partition testing. Finally, Section 5 concludes the paper.

- *T.Y. Chen is with the School of Information Technology, Swinburne University of Technology, Hawthorn 3122, Australia. E-mail: tychen@it.swin.edu.au.*
- *P.-L. Poon is with the Department of Accountancy, The Hong Kong Polytechnic University, Hung Hom, Hong Kong. E-mail: acplpoon@inet.polyu.edu.hk.*
- *T.H. Tse is with the Department of Computer Science and Information Systems, The University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: tse@csis.hku.hk.*

Manuscript received 10 Aug. 2001; revised 9 July 2002; accepted 18 Feb. 2003.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 114748.

2 Category-Partition Method

CPM is a specification-based testing technique developed by Ostrand and Balcer [17]. It helps software testers create test cases by refining the functional specification of a program into test specifications. It identifies the elements that influence the functions of the program and generates test cases by methodically varying these elements over all values of interest. The method consists of the following steps:

1. Decompose the functional specification into functional units that can be tested independently.
2. Identify the parameters (the explicit inputs to a functional unit) and environment conditions (the state of the system at the time of execution) that affect the execution behavior of the function.
3. Find categories (major properties or characteristics) of information that characterize each parameter and environment condition.
4. Partition each category into choices, which include all the different kinds of values that are possible for that category.
5. Determine the constraints among the choices of different categories. For example, one choice may require that another is absent or has a particular value.
6. Write the test specification (which is a list of categories, choices, and constraints in a predefined format) using the test specification language TSL.
7. Use a generator to produce test frames from the test specification. Each generated test frame is a set of choices such that each category contributes no more than one choice.
8. For each generated test frame, create a test case by selecting a single element from each choice in that test frame.

3 Choice Relation Framework for Category-Partition Testing

Motivated by problems 1 to 3 of CPM as suggested in Section 1, we propose a choice relation framework to support the method. Basically, our framework helps construct test cases from functional specifications via the notion of a choice relation table. The intuition of this table is to capture the constraints imposed on the choices by the specification. These constraints are expressed as relations between pairs of choices. They are essential information for the automatic generation of test frames.

Our approach consists of the following major steps:

1. Decompose the functional specification into functional units that can be tested separately.
2. For every functional unit, identify its parameters and environment conditions and, hence, define the categories and their associated choices.
3. Construct a choice relation table $\mathcal{T}$ for each functional unit.
4. For each $\mathcal{T}$, construct the corresponding choice priority table $\mathcal{P}$, which captures the relative priorities for the use of the choices in generating test frames.
5. From each $\mathcal{T}$ and the corresponding $\mathcal{P}$, construct the set of test frames.
6. Create a test case from each generated test frame.

Steps 1, 2, and 6 above are identical to Steps 1, 2-4, and 8, respectively, of CPM described in Section 2. We shall therefore concentrate on Steps 3, 4, and 5 in our discussions in Sections 3.1, 3.2, and 3.3, respectively.

3.1 Construction of the Choice Relation Table

As mentioned above, the choice relation table $\mathcal{T}$ is intended to capture the constraints imposed by the specification on the choices. To construct $\mathcal{T}$, we need to determine the relation between each pair of choices. This is explained in the following sections.

3.1.1 Determination of Relations among Choices

The steps prior to the construction of $\mathcal{T}$ correspond to Steps 1-4 of CPM mentioned in Section 2 and, hence, detailed explanations are not repeated here. Instead, we shall simply illustrate the concepts of categories and choices through an example.

Example 1 (Loan Example). Suppose a software tester is given the following specification:

Develop a program *loan* for use by ABC Bank to process applications by its customers for personal loans, based on their employment and credit card details. In order to evaluate an application, the program will accept the following details from the applicant. The evaluation criteria are not specified here.

- Employment Status: Either "Employed" or "Unemployed."
- Type of Employment (if the applicant is working): Either "Self-Employed" or "Employed by Others."
- Type of Job (if the applicant is working): Either "Permanent" or "Temporary."
- Monthly Salary S (if the applicant is working): Either "$\$0 < S \leq \$2,000$," "$\$2,000 < S \leq \$3,000$," or "$S > \$3,000$."
- Type of Applicant: Either "Cardholder" or "Non-Cardholder."
- Type of Credit Card (if the applicant is a cardholder): Either "Gold" or "Classic."
- Credit Limit (applicable only to a classic card): Either "$2,000" or "$3,000."

It should be noted that there is no credit limit for a gold card.

Suppose the software tester decides that *loan* can be tested as a whole and thus further break down into smaller functional units is not required. Additionally, suppose the categories and their associated choices for *loan* are simply defined based on the above input details. For example, "Employment Status" is defined as a category. Its two associated choices are "Employment Status = Employed" and "Employment Status = Unemployed." When there is no ambiguity, we can

simply refer to these two choices as "Employed" and "Unemployed."[1,2]

Before we proceed further, let us define the concepts of test frames, valid choices, and relations among choices.

Definition 1 (Test Frames and their Completeness). *A **test frame** B is a set of choices. A test frame B is said to be **complete** if, whenever a single element is selected from each choice in B, a standalone input will be formed. Otherwise, it is said to be **incomplete**.*

Definition 2 (Set of Complete Test Frames Related to a Choice). *Let TF denote the set of all complete test frames. Given any choice x, we define the **set of complete test frames related to** x as $TF(x) = \{B \in TF : x \in B\}$. A choice x is **valid** if and only if $TF(x)$ is nonempty.*

Different types of relations are possible for a given pair of valid choices, as illustrated in the following example:

Example 2 (Relation between Two Choices). Consider the specification of the program *loan* in Example 1. *TF*("Classic") contains all the complete test frames containing the choice "Classic" in the category "Type of Credit Card." The relation between "Classic" and any other choice x can be one of three types:

- $x \in B$ for any $B \in TF(\text{"Classic"})$. An example of x is the choice "Cardholder" under the category "Type of Applicant."
- $x \in B$ for some, but not all, $B \in TF(\text{"Classic"})$. Consider the following two complete test frames B_1 and $B_2 \in TF(\text{"Classic"})$:

  $$B_1 = \{\text{Unemployed}, \text{Cardholder}, \text{Classic}, \$2,000\},$$
  $$B_2 = \{\text{Unemployed}, \text{Cardholder}, \text{Classic}, \$3,000\}.$$

  Suppose $x =$ "$\$2,000$." It is obvious from B_1 and B_2 that x appears in some, but not all, $B \in TF(\text{"Classic"})$.
- $x \notin B$ for any $B \in TF(\text{"Classic"})$. An example of x is the choice "Non-Cardholder" under the category "Type of Applicant." Another example of x is the choice "Gold" under the category "Type of Credit Card."

Because of the importance of the above distinction, we define the three corresponding types formally as follows:

Definition 3 (Relation between Two Choices). *Given any valid choice x, its **relation** with another valid choice y (denoted by $x \mapsto y$) is defined in terms of one of three **relational operators** as follows:*

1. *x is **fully embedded** in y (denoted by $x \sqsubset y$) if and only if $TF(x) \subseteq TF(y)$.*
2. *x is **partially embedded** in y (denoted by $x \boxsubset y$) if and only if $TF(x) \not\subseteq TF(y)$ and $TF(x) \cap TF(y) \neq \emptyset$.*
3. *x is **not embedded** in y (denoted by $x \not\sqsubset y$) if and only if $TF(x) \cap TF(y) = \emptyset$.*

The choice relations "full embedding" and "nonembedding" in the above definition have straightforward meanings in ordinary logic and, hence, the motivation behind them is fairly obvious. On the other hand, the motivation behind the choice relation "partial embedding" merits some discussion.

Example 3 (Motivation behind the Partial Embedding Relation). Consider the following simple specification in a typical credit card system:

> If Total Transaction Amount > \$1,000,
> then add 200 bonus points.
> If Average Transaction Amount > \$100,
> then add 50 bonus points.

1. According to Definition 3, the two choices "Total Transaction Amount > \$1,000" and "Average Transaction Amount > \$100" are partially embedded in each other. There is no logical relationship between them. However, this specification is important to the user and useful to the implementer.
2. The notion of partial embedding will be useful for testing against problematic implementations such as the following:

> If Total Transaction Amount > \$1,000,
> then add 200 bonus points
> ***else*** if Average Transaction Amount > \$100,
> then add 50 bonus points.

Since the three types of choice relations in Definition 3 are exhaustive and mutually exclusive, $x \mapsto y$ can be uniquely determined. It should be noted that, immediately from the definition, the relational operator for $x \mapsto x$ is "$\sqsubset$" and the relational operator for $x \mapsto y$ is "$\not\sqsubset$" if x and y are two different choices in the same category.

Example 4 (Choice Relation Table). Consider the *loan* example again. The choice relation table $\mathcal{T}_{loan}$ is constructed and depicted in Fig. 1. Let w be the total number of choices. Let $t(i,j)$ denote the element at the ith row and jth column of $\mathcal{T}_{loan}$, $i, j = 1, 2, \ldots, w$. For $t(12, 14)$ (Self-Employed $\mapsto$ Employed), the relational operator is "$\sqsubset$" because "Self-Employed" always requires "Employment Status" to be "Employed." For $t(15, 10)$ (Unemployed $\mapsto$ Permanent), the relational operator is "$\not\sqsubset$" because "Unemployed" and "Permanent" are mutually exclusive, as the latter requires "Employment Status" to be "Employed." For $t(5, 15)$ (Cardholder $\mapsto$ Unemployed), the relational operator is obviously "$\boxsubset$." The relational operator for the remaining elements in Fig. 1 can be determined in a similar way.

Readers may note that the choice relations defined in Definition 3 focus on the constraint between a pair of choices. In situations where the relationships among three or more input variables have to be considered at the same time, we can define a single category that involves these

1. One may argue that we need a further category "Status of Customer Master File" to reflect the environment condition, with three associated choices "File Does Not Exist," "File is Empty," and "File is Not Empty." For simplicity of illustration, however, we shall only concentrate on the categories and choices defined for input parameters in this example.

2. It should be noted that a choice can be defined for a range of values. For example, "$\$0 < S \leq \$2,000$," "$\$2,000 < S \leq \$3,000$," and "$S > \$3,000$" are three possible choices for the category "Monthly Salary (S)."

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		\$2000	\$3000	Gold	Classic	Cardholder	Non-Cardholder	\$0 < S ⩽ \$2000	\$2000 < S ⩽ \$3000	S > \$3000	Permanent	Temporary	Self-Employed	Employed by Others	Employed	Unemployed
1	\$2000	⊏	⋤	⋤	⊏	⊏	⋤	P	P	P	P	P	P	P	P	P
2	\$3000	⋤	⊏	⋤	⊏	⊏	⋤	P	P	P	P	P	P	P	P	P
3	Gold	⋤	⋤	⊏	⋤	⊏	⋤	P	P	P	P	P	P	P	P	P
4	Classic	P	P	⋤	⊏	⊏	⋤	P	P	P	P	P	P	P	P	P
5	Cardholder	P	P	P	P	⊏	⋤	P	P	P	P	P	P	P	P	P
6	Non-Cardholder	⋤	⋤	⋤	⋤	⋤	⊏	P	P	P	P	P	P	P	P	P
7	\$0 < S ⩽ \$2000	P	P	P	P	P	P	⊏	⋤	⋤	P	P	P	P	⊏	⋤
8	\$2000 < S ⩽ \$3000	P	P	P	P	P	P	⋤	⊏	⋤	P	P	P	P	⊏	⋤
9	S > \$3000	P	P	P	P	P	P	⋤	⋤	⊏	P	P	P	P	⊏	⋤
10	Permanent	P	P	P	P	P	P	P	P	P	⊏	⋤	P	P	⊏	⋤
11	Temporary	P	P	P	P	P	P	P	P	P	⋤	⊏	P	P	⊏	⋤
12	Self-Employed	P	P	P	P	P	P	P	P	P	P	P	⊏	⋤	⊏	⋤
13	Employed by Others	P	P	P	P	P	P	P	P	P	P	P	⋤	⊏	⊏	⋤
14	Employed	P	P	P	P	P	P	P	P	P	P	P	P	P	⊏	⋤
15	Unemployed	P	P	P	P	P	P	⋤	⋤	⋤	⋤	⋤	⋤	⋤	⋤	⊏

Fig. 1. Choice relation table for the loan example.

input variables. Consider, for instance, the relationship $(A + B + C = 10)$, where A, B, and C are input variables. In this case, we can define a single category "$A + B + C$" with two associated choices "$= 10$" and "$\neq 10$."

It would obviously be quite inefficient and error-prone if all the relational operators in the choice relation table had to be defined manually, especially when the number of choices is large. To solve the problem, we have developed techniques for consistency checks and automatic deductions of choice relations. These techniques are rendered possible by a set of properties of the relational operators described in the next section.

3.1.2 Properties of Relations among Choices

Some useful properties for the three relational operators are stated in the following propositions and corollaries. Readers may refer to Appendix E for the proofs of these propositions and corollaries.

Proposition 1 (Symmetry of the Nonembedding of Choices). *For any pair of valid choices x and y, $x \not\sqsupset y$ if and only if $y \not\sqsupset x$.*

Immediately from Proposition 1, we have the following corollary:

Corollary 1 (Reverse of Full and Partial Embedding of Choices). *Let x and y be valid choices. 1) If $x \sqsubset y$, then $y \sqsubset x$ or $y \sqsubseteq x$. 2) If $x \sqsubseteq y$, then $y \sqsubset x$ or $y \sqsubseteq x$.*

It should be noted that the results in Corollary 1 cannot be narrowed down any further. This is further discussed in Appendix E of the paper.

Proposition 2 (Full Embedding of Choices). *Let x, y, and z be valid choices. 1) If $x \sqsubset y$ and $y \sqsubset z$, then $x \sqsubset z$. 2) If $x \sqsubset y$ and $x \sqsubset z$, then $y \sqsubset z$ or $y \sqsubseteq z$.*

Proposition 3 (Full Embedding and Nonembedding of Choices). *Let x, y, and z be valid choices. 1) If $x \sqsubset y$ and $y \not\sqsupset z$, then $x \not\sqsupset z$. 2) If $x \sqsubset y$ and $x \not\sqsupset z$, then $y \sqsubseteq z$ or $y \not\sqsupset z$.*

Immediately from Propositions 1 and 3, we have the following corollary:

Corollary 2 (Full Embedding and Nonembedding of Choices). *Let x, y, and z be valid choices. 1) If $x \sqsubset z$ and $y \not\sqsupset z$, then $x \not\sqsupset y$. 2) If $y \sqsubset z$ and $x \not\sqsupset y$, then $z \sqsubseteq x$ or $z \not\sqsupset x$.*

Proposition 4 (Full and Partial Embedding of Choices). *Let x, y, and z be valid choices. 1) If $x \sqsubset y$ and $x \sqsubseteq z$, then $y \sqsubseteq z$. 2) If $x \sqsubset z$ and $y \sqsubseteq z$, then $y \sqsubseteq x$ or $y \not\sqsupset x$. 3) If $y \sqsubset z$ and $x \sqsubseteq y$, then $z \sqsubset x$ or $z \sqsubseteq x$.*

Proposition 5 (Partial Embedding and Nonembedding of Choices). *Let x, y, and z be valid choices. 1) If $x \sqsubseteq y$ and $y \not\sqsupset z$, then $x \sqsubseteq z$ or $x \not\sqsupset z$. 2) If $x \sqsubseteq y$ and $x \not\sqsupset z$, then $y \sqsubseteq z$ or $y \not\sqsupset z$.*

Immediately from Propositions 1 and 5, we have the following corollary:

Corollary 3 (Partial Embedding and Nonembedding of Choices). *Let x, y, and z be valid choices. 1) If $x \sqsubseteq z$ and $y \not\sqsupset z$, then $x \sqsubseteq y$ or $x \not\sqsupset y$. 2) If $y \sqsubseteq z$ and $x \not\sqsupset y$, then $z \sqsubseteq x$ or $z \not\sqsupset x$.*

Each of the above propositions and corollaries provides a certain scope of consistency checks for the relations among choices. For example, we know that $x \sqsubseteq y$ and $y \not\sqsupset x$ cannot coexist or else it would contradict Proposition 1. However, not all incorrectly defined relations can be identified as inconsistencies. For example, suppose $x \sqsubset y$ and $y \sqsubset x$ are correct but somehow mistakenly defined as $x \sqsubset y$ and $y \sqsubseteq x$. This mistake is not inherently inconsistent.

3.1.3 System Deduction Rules for Choice Relations

Consider again Propositions 1, 2.1, 3.1, and 4.1, and Corollary 2.1. The "then" parts of these propositions and corollary contain definite relations, which provide a basis for automatic deductions. Thus, when appropriate relations have been defined, other relations can be deduced. For example, once we know that $x \sqsubset y$ and $y \sqsubset z$, we can conclude from Proposition 2.1 that $x \sqsubset z$.

The effectiveness of automatic deductions, however, varies with the chronological order of defining the relations. The following example illustrates this point.

Example 5 (Chronological Order of Defining Choice Relations). Consider the choices "$2,000," "Classic," and "Permanent" in Fig. 1 for the *loan* example. Suppose the relations ($\$2,000 \mapsto \text{Classic}$), ($\$2,000 \mapsto \text{Permanent}$), and ($\text{Classic} \mapsto \text{Permanent}$) have yet to be defined. If ($\$2,000 \sqsubset \text{Classic}$) and ($\$2,000 \boxsubset \text{Permanent}$) are manually defined first, then ($\text{Classic} \boxsubset \text{Permanent}$) can be deduced using Proposition 4.1. On the other hand, if ($\$2,000 \sqsubset \text{Classic}$) and ($\text{Classic} \boxsubset \text{Permanent}$) are manually defined first, then ($\$2,000 \boxsubset \text{Permanent}$) must still be manually defined since it cannot be deduced from any of the propositions or the corollary.

In view of Example 5, we shall propose a heuristic approach in Section 3.1.4 to determine the chronological order of defining relations in order to improve on the effectiveness of automatic deductions. First, however, we discuss some important system deduction rules that form the basis of our heuristic approach.

System Deduction Rule (1). Given $x \sqsubset y$, we should next define $y \mapsto z$, $z \mapsto y$, $x \mapsto z$, and $z \mapsto x$ if they have not yet been defined or deduced.

If $y \sqsubset z$, then we can deduce $x \sqsubset z$ by applying Proposition 2.1. If $y \not\sqsubset z$, then we can deduce $x \not\sqsubset z$ by applying Proposition 3.1. If $z \not\sqsubset y$, then we can deduce $x \not\sqsubset z$ by applying Corollary 2.1. If $x \boxsubset z$, then we can deduce $y \boxsubset z$ by applying Proposition 4.1. If $z \sqsubset x$, then we can deduce $z \sqsubset y$ by applying Proposition 2.1.

System Deduction Rule (2). Given $x \boxsubset y$, we should next define $x \mapsto z$ if it is not yet defined or deduced.

If $x \sqsubset z$, then we can deduce $z \boxsubset y$ by applying Proposition 4.1.

System Deduction Rule (3). Given $x \not\sqsubset y$, we should next define $z \mapsto x$ and $z \mapsto y$ if they have not yet been defined or deduced.

If $z \sqsubset x$, then we can deduce $z \not\sqsubset y$ by applying Proposition 3.1. If $z \sqsubset y$, then we can deduce $z \not\sqsubset x$ by applying Corollary 2.1.

The above system deduction rules provide the basis of our heuristic approach for the automatic identification of the next relation to be defined.

3.1.4 Table Construction

As explained in Section 3.1.2, an incorrectly defined choice relation may not lead immediately to any inconsistency highlighted by Propositions 1-5 and Corollaries 1-3. Other choice relations may have been defined manually before the incorrect choice relation is identified. By this time, additional choice relations may have already been deduced automatically by the system, some of which are erroneously based on the original incorrect relation. It is therefore desirable to provide a function to the tester so that an incorrectly defined relation and the erroneously deduced relations can be corrected during the construction of the choice relation table. This correction function will be discussed later in this section. Before that, we must first explore the possible types of elements in the choice relation table $\mathcal{T}$.

Each element $t(i, j)$ in $\mathcal{T}$ can be classified into one of the following three types:

- A *defined element* if it is manually defined.
- A *deduced element* if it is automatically deduced.
- A *yet-to-be-defined element* if it has not been defined or deduced.

An element $t(m, n)$ in $\mathcal{T}$ is said to be an *ancestor* of another element $t(i, j)$ if the former has been used to deduce the latter, either directly or indirectly. When the user identifies a defined element for correction, the system will also check (and amend if necessary) all the elements that have been deduced from it. When the system amends a deduced element, it will also check (and amend if necessary) all the relevant ancestors that have been defined. Ancestor information is therefore vital to the correction of incorrectly defined or deduced elements in $\mathcal{T}$. We use an element relation table with parent linked lists to capture this information. Besides, this table will also serve as a guide for automatically identifying the next relation that should be manually defined, as explained in Section 3.1.3 and further elaborated below.

Given w choices, the dimension of $\mathcal{T}$ is $w \times w$. A corresponding *element relation table* $\mathcal{E}$ has the same dimension. Each element of $\mathcal{E}$, denoted by $e(i, j)$, consists of the following four fields:

- ***Type:*** It contains an integer value of -1, 0, or 1, indicating that the corresponding $t(i, j)$ is a defined, yet-to-be-defined, or deduced element, respectively.
- ***Parent-Pointer:*** An element $t(m, n)$ in $\mathcal{T}$ is called a *parent* of another element $t(i, j)$ if the former is an immediate ancestor of the latter. The set of all parents of $t(i, j)$ is represented by a *parent linked list* $PL_{i,j}$. If $t(i, j)$ is a deduced element and the corresponding pair of choices belong to different categories, then the parent-pointer contains the header address of the parent linked list. Otherwise, the parent-pointer is set to a null value.
- ***Deduction-Pointer:*** If $i = j$ or the type is 0, then the deduction-pointer contains a null value. Otherwise, it contains the header address of a *deduction linked list* $DL_{i,j}$. Each node of $DL_{i,j}$ contains a yet-to-be-defined element in $\mathcal{T}$ which, according to the three system deduction rules in Section 3.1.3, should be a candidate for the next manual definition. Obviously, $DL_{i,j}$ may be empty.
- ***Counter:*** This field is used to determine the relative chance of deducing some other relations if we know the relational operator for the corresponding $t(i, j)$. It

TABLE 1
Contents of $e(i,j)$ of Element Relation Table

Given $t(i,j)$	Contents of $e(i,j)$			
	Type	Parent-Pointer	Deduction-Pointer	Counter
Defined	-1	Null	Header address of $DL_{i,j}$	-1
Yet-to-be-defined	0	Null	Null	≥ 0
Deduced	1	Header address of $PL_{i,j}$ if the corresponding pair of choices belong to different categories; null otherwise	Header address of $DL_{i,j}$ if $i \neq j$; null otherwise	-1

contains an integer value ≥ -1. Intuitively, a higher value in this field indicates a greater chance of deducing other relations when we know the relational operator. It should be noted that the value of this field is -1 if the corresponding $t(i,j)$ has been defined or deduced, and this field may be updated once $\mathcal{T}$ is updated.

Table 1 summarizes the possible values for the above four fields.

Appendix A shows an algorithm *build_table* for the construction of $\mathcal{T}$. It incorporates the features for automatic deductions and consistency checks. The main philosophy is to perform automatic deduction for every yet-to-be-defined $t(i,j)$ as far as possible and to perform consistency checks whenever a $t(i,j)$ is manually defined. We note that:

1. The procedure *correct_operator*() will not only correct the erroneous elements selected by the user, but also all the deduced elements resulting from these erroneous elements. As a result, the user need only continue constructing the choice relation table from that point, rather than repeat the entire table construction process.
2. For every incorrectly defined choice relation, the number of executions required to correct this relation as well as other related relations is in the order of w^4 in the worst case, where w is the total number of choices. Thus, if m choice relations have been defined incorrectly, then the number of executions is in the order of mw^4.

The algorithm also automatically identifies the next $t(i,j)$ to be manually defined, as guided by the system deduction rules explained in Section 3.1.3. It would be useful to outline the main idea behind this feature. Each element $t(i,j)$ in $\mathcal{T}$ is associated with a counter at $e(i,j)$ of $\mathcal{E}$, which provides an estimate of the number of other relations that can be deduced. The higher the value of the counter at $e(i,j)$, the greater will be the chance of the corresponding $t(i,j)$ to be selected for the next manual definition so that the chance of deducing other relations will be increased. This is illustrated by the following example:

Example 6 (System Deduction Rules). Assume that we are constructing the choice relation table for the *loan* example, as shown in Fig. 1. Suppose that the choice relation $(\$2,000 \not\sqsupseteq \text{Gold})$ has just been defined. According to System Deduction Rule 3 in Section 3.1.3, we should next define $(z \mapsto \$2,000)$ and $(z \mapsto \text{Gold})$, where z is a choice such as "Permanent." Suppose further that $(z \mapsto \$2,000)$ is yet-to-be-defined and $(z \mapsto \text{Gold})$ has been defined or deduced. Then, the counter value for $(z \mapsto \$2,000)$ will be increased by one, but not that for $(z \mapsto \text{Gold})$. This will effectively increase the chance of $(z \mapsto \$2,000)$ being selected for the next manual definition.

We have built a prototype system implementing the algorithm *build_table*, in which previously presented techniques for consistency checks and automatic deductions have been incorporated. Fig. 2 shows the input screen for defining the relation between a pair of distinct choices. It also provides users with the option of defining group constraints by means of a single manual definition (see Step 5d of the procedure *build_table*()). This will further reduce the number of manual definitions required. When users select the option of group constraint definitions in Fig. 2, for instance, the relational operator "$\not\sqsupseteq$" will not only be assigned to $(\text{Gold} \mapsto \$2,000)$, but also to $(\text{Gold} \mapsto \$3,000)$. Fig. 3 depicts a system screen that alerts users about detected inconsistencies among relations and allows them to choose the erroneous relations to be removed. "Full-EmbedIn," "PartEmbedIn," and "NotEmbedIn" in the figure represent the relational operators "$\sqsubset$," "$\sqsubseteq$," and "$\not\sqsupseteq$," respectively.

3.1.5 Empirical Studies

We have conducted empirical studies to evaluate the effectiveness of our table construction technique and to compare our approach with the original CPM. Our studies involve four real-life commercial specifications:

- The specification $\mathcal{S}_{register}$ is for the inventory registration module of an inventory management system

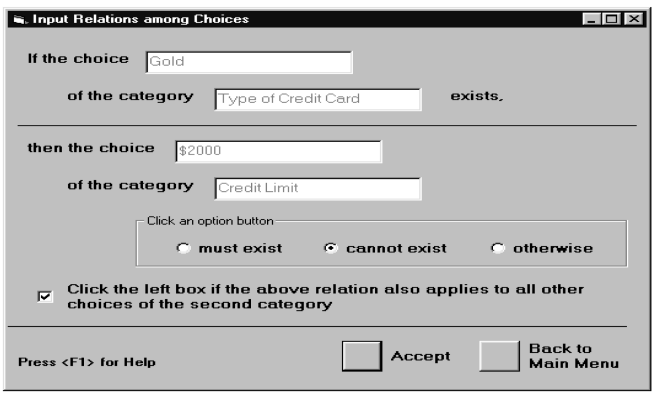

Fig. 2. Input screen for the relation between a pair of distinct choices.

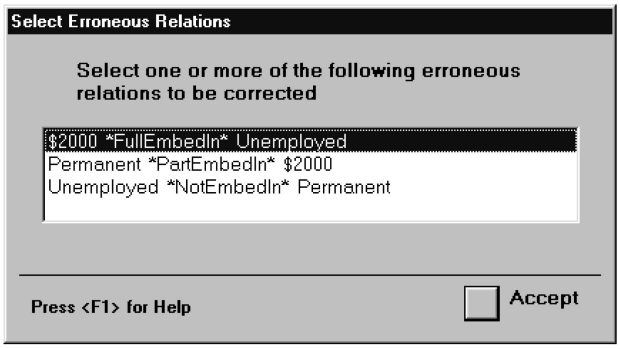

Fig. 3. Input screen to trigger the correction of erroneous relations.

used by a group of public hospitals. The main functions of the module are to record inventory details, to capture parent-child relationships among the inventory items, and to generate bar-code labels.

- $\mathcal{S}_{purchase}$ is for a purchase-order generation module used by the same group of public hospitals. The module allows purchasing officers to add procurement information to replenishment requests sent from various hospital units and generates purchase orders automatically.
- $\mathcal{S}_{inquiry}$ is for an online telephone inquiry system used in a large telecom company. The system handles more than 60,000 inquiries a day and supports various modes of inquiries such as incomplete name searches, alternative name searches, and complex phonetic searches.
- $\mathcal{S}_{meal}$ is for the meal scheduling module of a meal ordering system used by an international airline catering company. The main function is to determine the quantity for every type of meal to be prepared and loaded onto the aircrafts served by the company.

Empirical Study 1: Effectiveness of Table Construction

1. ***Effectiveness of Automatic Deductions and Group Constraint Definitions.*** First, we identified the categories and choices for the four specifications described above in the usual manner. For the categories and choices for each specification, we randomly generated five different initial sequences of choices. It should be noted that the actual sequence of manually defined choice relations depends on the initial sequence of choices and the deduction heuristics described in Sections 3.1.3 and 3.1.4.

   Table 2 shows the effectiveness of automatic deductions and group constraint definitions in the algorithm *build_table*. On average, about 41.9 percent of the choice relations were automatically deduced. This automatic deduction feature was not available in the original CPM, in which all the constraints among choices had to be defined manually. In addition, because of group constraint definitions, an extra 28.0 percent of choice relations need not be defined individually. As a result, the amount of human effort was significantly reduced—only about 30.1 percent of the total number of choice relations had to be specified manually.
2. ***Effectiveness of Consistency Checks.*** At the conclusion of the 20 trial runs, we experienced seven erroneous choice relations, as shown in the rightmost column of Table 2. All these problematic cases were defined at an intermediate stage of the table construction process. Each error was detected by the consistency checking mechanism almost

TABLE 2
Empirical Study 1: Effectiveness of Table Construction

Specification	Trial Number	% of Choice Relations Defined Manually	% of Choice Relations Deduced Automatically	% of Choice Relations Generated Automatically from Group Constraint Definitions	No. of Inconsistent Choice Relations Detected
$\mathcal{S}_{register}$	1	21.4	43.6	35.0	1 (detected immediately)
	2	25.0	37.3	37.7	0
	3	24.8	37.2	38.0	0
	4	22.5	42.1	35.4	2 (detected immediately)
	5	24.4	37.9	37.7	1 (detected immediately)
$\mathcal{S}_{purchase}$	1	21.0	64.8	14.2	1 (detected immediately)
	2	21.6	63.0	15.4	0
	3	21.9	63.0	15.1	0
	4	21.3	64.5	14.2	0
	5	21.6	64.2	14.2	0
$\mathcal{S}_{inquiry}$	1	31.0	39.3	29.7	0
	2	31.9	41.0	27.0	0
	3	31.8	40.8	27.4	1 (detected immediately)
	4	31.0	39.3	29.7	0
	5	32.3	39.9	27.8	1 (detected after next manual definition)
$\mathcal{S}_{meal}$	1	43.5	24.9	31.7	0
	2	45.4	18.9	35.7	0
	3	43.5	24.9	31.7	0
	4	43.5	24.9	31.7	0
	5	43.0	26.0	30.9	0
Average		**30.1**	**41.9**	**28.0**	**N/A**

TABLE 3
Empirical Study 2: Comparison with the Original CPM

Specification	Person-A			Person-B		
	Sequence	No. of Incorrect Definitions when Constructing TSL Specification	No. of Incorrect Definitions when Constructing CRT	Sequence	No. of Incorrect Definitions when Constructing TSL Specification	No. of Incorrect Definitions when Constructing CRT
$\mathcal{S}_{register}$	C-T	7	0	T-C	3	1 (detected immediately)
$\mathcal{S}_{purchase}$	C-T	4	1 (detected immediately)	T-C	0	1 (detected immediately)
$\mathcal{S}_{inquiry}$	T-C	0	3 (detected immediately)	C-T	3	4 (two detected immediately, the other two detected after the next manual definition)
$\mathcal{S}_{meal}$	T-C	2	3 (detected immediately)	C-T	0	2 (detected immediately)

LEGEND **Sequence T-C:** TSL specification constructed first, followed by choice relation table (CRT).
Sequence C-T: CRT constructed first, followed by TSL specification.

immediately after the manual definition, rather than after a series of other manual definitions. A plausible explanation is that real-life specifications involve a fairly large number of categories and choices and, hence, a large number of choice relations. The chance of erroneously defining a choice relation at an intermediate stage, and yet slipping through the consistency checking mechanism, is very slim.

The immediate detection of inconsistencies caused by erroneously defined choice relations also means that the number of related choice relations to be corrected by the procedure $correct\_operator()$ of the algorithm $build\_table$ can be kept to a minimum. This is because, whenever an incorrect choice relation is defined, it will be detected and, hence, retracted immediately before it is erroneously used for deducing other choice relations.

As mentioned earlier, the number of executions required to correct m incorrectly defined choice relations is $O(mw^4)$ in the worst case. Our actual experience shows, however, that the amount of work due to corrections is much less than that portrayed by this formula because:

- Only six of the 20 trial runs involved erroneous manual definitions. Each erroneous case involved only one or two incorrectly defined choice relations.
- In the six trial runs that involved erroneous manual definitions, the incorrect choice relations were detected almost immediately before they were used for deducing other relations. Hence, the number of steps involved in removing an error was much less than $O(w^4)$.

By virtue of the consistency checking mechanism, all the choice relations are verified before the construction of test frames, as described in Section 3.3 below. This proves to be very handy. For the original CPM, on the other hand, users have to check manually for incorrectly defined constraints against test specifications. If some incorrect constraints are detected only after the execution of the associated generator, certain effort will be wasted.

Empirical Study 2: Comparison with the Original CPM. It would also be useful to further evaluate the effectiveness of consistency checks in the choice relation framework by comparing our approach with the original CPM. The study involves two subjects, to be referred to as Person-A and Person-B. Both of them have postgraduate qualifications in computer science or information technology and have about 10 years of working experience in industry. Neither of the subjects has been involved with the development of the original CPM or our choice relation framework.

Before the study started, both subjects were given the relevant sections of this paper as well as the literature relating to the original CPM [4], [17] for self-study. We then gave them a sample specification and asked them to construct a TSL specification in the original CPM and a choice relation table using the prototype system. The exercise was followed by a thorough discussion of the results. The idea was to familiarize them with CPM and our framework.

After the initial training, Person-A first constructed the choice relation tables followed by the TSL specifications for the specifications $\mathcal{S}_{register}$ and $\mathcal{S}_{purchase}$. He also constructed the TSL specifications followed by the choice relation tables for the specifications $\mathcal{S}_{inquiry}$ and $\mathcal{S}_{meal}$. On the other hand, Person-B first constructed the TSL specifications followed by the choice relation tables for $\mathcal{S}_{register}$ and $\mathcal{S}_{purchase}$. She also constructed the choice relation tables followed by the TSL specifications for $\mathcal{S}_{inquiry}$ and $\mathcal{S}_{meal}$.

Table 3 summarizes the results of the study. It shows that the subjects have included erroneous definitions in both the TSL specifications and the choice relation tables. We have two observations:

1. The number of error cases for TSL specifications (19) was not substantially different from that of the choice relation tables (15). The number of errors varied little independent of whether the TSL specifications or the choice relation tables were constructed first.
2. All 15 error cases in the choice relation tables were corrected during table construction with the help of the consistency checking and correction mechanisms. About 86.7 percent of these errors were detected immediately. The rest were detected after the next manual definition. As a result, all the

definitions were correct at the conclusion of the table construction processes. This feature was not available in the TSL specifications. None of the 19 errors were detected by the subjects themselves. We note that, when erroneous constraint definitions are left undetected, the resulting number of erroneous test frames is usually many times that of the incorrect definitions.

3.2 Construction of the Choice Priority Table

In most real-life situations, resource constraints are imposed on the software tester. Hence, not all complete test frames may be used for generating test cases for a program. A possible approach is to define the relative priorities for the choices based on the software tester's expertise and experience in the application domain. In this way, the choices with higher priorities can first be used to generate test frames, thus respecting both the resource constraints and the relative importance of the choices.

Users are requested to define the following parameters after the choice relation table $\mathcal{T}$ has been constructed:

1. **Preferred Maximum Number of Test Frames $\overline{M}$.** Software testers are required to define a preferred maximum number of test frames $\overline{M}$ that they are willing to handle. The word "preferred" implies that $\overline{M}$ is not absolute, as the limit may be overwritten by the minimally achievable priority level $\underline{m}$ in 3.
2. **Relative Priority of Every Choice.** Given w choices, a choice priority table $\mathcal{P}$ with a dimension of $w \times 2$ is constructed, capturing the relative priority of every choice. Let $p(i,1)$ and $p(i,2)$ denote the first and second elements, respectively, of the ith row of $\mathcal{P}$. Basically, $p(i,1)$ contains a valid choice x_i and $p(i,2)$ contains a positive integer r_i. The smaller the value of r_i, the higher will be the priority of the corresponding x_i for inclusion as part of a test frame. Our framework assumes that the smallest value of r_i is 1.
3. **Minimally Achievable Priority Level $\underline{m}$.** The definition of a minimally achievable priority level $\underline{m}$ allows the software tester to ensure that those x_is with $r_i \leq \underline{m}$ will always be selected for inclusion as part of a test frame, independent of whether the number of generated test frames exceeds $\overline{M}$ or not. Our framework assumes that $\underline{m} \geq 0$. In the situation where $\overline{M}$ should not be waived by $\underline{m}$, the software tester should set $\underline{m}$ to zero. In this way, $\overline{M}$ becomes the *absolute* maximum number of generated test frames.

In the above, the value of $\overline{M}$ is largely dependent on the testing resources. The more the available resources, the higher should $\overline{M}$ be defined. As pointed out in [12], [13], [15], 1) it would be far more effective to have an idea of the kinds of faults that are most probable or most damaging and then to construct test cases that are likely to reveal these significant faults and 2) this fault-guessing process depends largely on the software tester's expertise and experience. Smaller values of r_i, representing higher priorities, should be assigned to those crucial choices x_i that are likely to reveal the significant faults. Furthermore, $\underline{m}$ should not be assigned a value smaller than any of these r_i.

TABLE 4
Choice Priority Table for Loan Example

Category	Choice x_i	Priority r_i
Employment Status	Employed	1
Employment Status	Unemployed	1
Type of Employment	Self-Employed	2
Type of Employment	Employed by Others	3
Type of Job	Permanent	4
...	...	...

Example 7. Consider the program *loan* in Example 1. Suppose the software tester defines $\overline{M}$ to be 10 and assigns the relative priority for all the choices, as illustrated partially in Table 4. Now, suppose $\underline{m}$ is set to 3. In this situation, the choices "Employed," "Unemployed," "Self-Employed," and "Employed by Others" will always be used for the construction of test frames, regardless of whether the number of generated test frames exceeds 10 or not.

3.3 Construction of Test Frames

3.3.1 Test Frames and Their Relations

According to Definition 1, a test frame consists of a group of choices. Furthermore, a test frame B is complete if and only if, whenever a single element is selected from every choice in B, the result will constitute a standalone input.

Consider, for instance, the following test frame for *loan* in Example 1:

$$B' = \{\text{Type of Applicant} = \text{Cardholder}, \text{Type of Credit Card} = \text{Classic}, \text{Credit Limit} = \$2,000\}.$$

B' is incomplete because additional details such as "Employment Status = Employed" must be supplied before we have sufficient information to generate a test case for execution.

Although B' is incomplete, it may be a subset of a complete test frame, such as the following:

$$\begin{aligned} B = \{&\text{Type of Applicant} = \text{Cardholder},\\ &\text{Type of Credit Card} = \text{Classic}, \text{Credit Limit} = \$2,000,\\ &\text{Employment Status} = \text{Employed},\\ &\text{Type of Employment} = \text{Employed by Others},\\ &\text{Type of Job} = \text{Permanent},\\ &\text{Monthly Salary } (S) = \$0 < S \leq \$2,000\}. \end{aligned}$$

Definition 4 (Set of Complete Test Frames Related to a Given Test Frame). *Let TF denote the set of all complete test frames. Given any test frame B', we define the* ***set of complete test frames related to*** *B' as the set $TF(B') = \{B \in TF : B' \subseteq B\}$. A test frame B' is* ***valid*** *if and only if $TF(B')$ is nonempty.*

It follows immediately from Definition 4 that a complete test frame must be valid.

Like the treatment of choices, the relations between valid test frames can also be classified into three types:

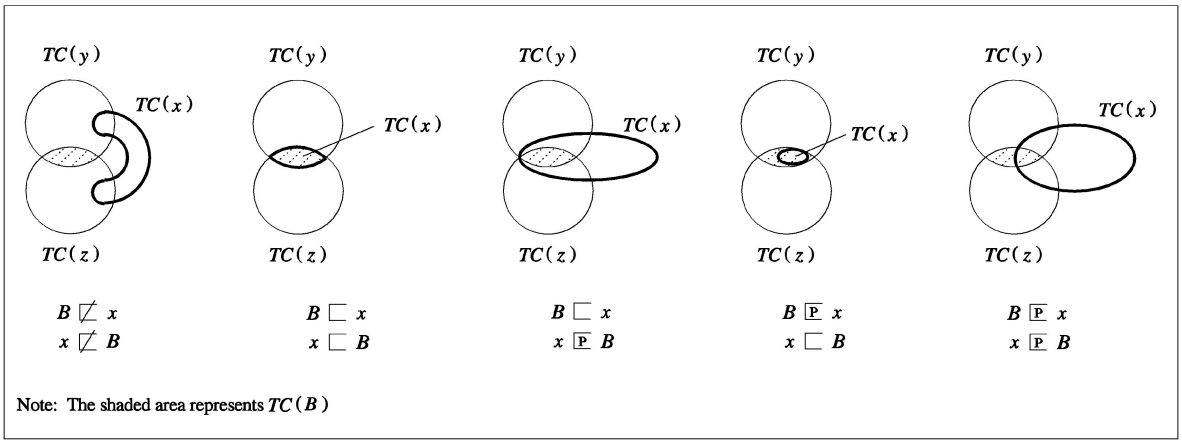

Fig. 4. Some relations between x and $B = \{y, z\}$ when $y \sqsubseteq x$ and $z \sqsubseteq x$.

Definition 5 (Relation between Two Test Frames). *Given any pair of valid test frames B_1 and B_2,*

1. *B_1 is **fully embedded** in B_2 (denoted by $B_1 \sqsubset B_2$) if and only if $TF(B_1) \subseteq TF(B_2)$.*
2. *B_1 is **partially embedded** in B_2 (denoted by $B_1 \sqsubseteq B_2$) if and only if $TF(B_1) \not\subseteq TF(B_2)$ and $TF(B_1) \cap TF(B_2) \neq \emptyset$.*
3. *B_1 is **not embedded** in B_2 (denoted by $B_1 \not\sqsubset B_2$) if and only if $TF(B_1) \cap TF(B_2) = \emptyset$.*

3.3.2 Properties of Relations among Test Frames

In light of Definition 5, we can extend Propositions 1-5 and Corollaries 1-3. For example, Proposition 1 and Corollary 1 can be extended into the following *dual* versions:

Dual Proposition 1 (Symmetry of the Nonembedding of Test Frames). *For any valid test frames B_1 and B_2, $B_1 \not\sqsubset B_2$ if and only if $B_2 \not\sqsubset B_1$.*

Dual Corollary 1 (Reverse of Full and Partial Embedding of Test Frames). *Let B_1 and B_2 be valid test frames. 1) If $B_1 \sqsubset B_2$, then $B_2 \sqsubset B_1$ or $B_2 \sqsubseteq B_1$. 2) If $B_1 \sqsubseteq B_2$, then $B_2 \sqsubset B_1$ or $B_2 \sqsubseteq B_1$.*

Propositions 2-5 and Corollaries 2-3 can be extended in a similar fashion. Readers may refer to Appendix D for a full list of dual propositions and corollaries.

Proposition 6 (Generalization Property). *Given any valid choice x, $TF(\{x\}) = TF(x)$.*

Because of Proposition 6, a choice x and the test frame $\{x\}$ will be used interchangeably in this paper.

3.3.3 Deduction of Relations among Test Frames

Two very important special cases of $B_1 \mapsto B_2$ are $B \mapsto x$ and $x \mapsto B$. Propositions on these special cases are practically useful for automatic deductions of relations. They will be discussed in this section.

Given any valid choice x and any valid test frame B, we can only have three scenarios: 1) There exists some $y \in B$ such that $y \not\sqsubset x$, 2) there exists some $y \in B$ such that $y \sqsubset x$, and 3) $y \sqsubseteq x$ for every $y \in B$. Proposition 7 below shows that these three scenarios are not only exhaustive but also mutually exclusive. Before we present Proposition 7, we need the following lemma.

Lemma 1 (Full Embedding Lemma). *Given any valid test frames B_1 and B_2, if $B_1 \subseteq B_2$, then $B_2 \sqsubset B_1$.*

Proposition 7 (Mutual Exclusion of Fully Embedded and Nonembedded Choices). *Let x be a valid choice and B be a valid test frame. We cannot have any y and $z \in B$ such that $y \sqsubset x$ and $z \not\sqsubset x$.*

The following proposition shows how we can uniquely determine $B \mapsto x$ and $x \mapsto B$ when there exists some $y \in B$ such that $y \not\sqsubset x$.

Proposition 8 (Test Frame Containing Nonembedded Choice). *Let x be a valid choice and B be a valid test frame. If there exists some valid choice $y \in B$ such that $y \not\sqsubset x$, then $B \not\sqsubset x$ and $x \not\sqsubset B$.*

The next proposition shows how we can uniquely determine $B \mapsto x$ and $x \mapsto B$ when there exists some $y \in B$ such that $y \sqsubset x$.

Proposition 9 (Test Frame Containing Fully Embedded Choice). *Let x be a valid choice and B be a valid test frame. If there exists some valid choice $y \in B$ such that $y \sqsubset x$, then the following will hold: 1) $B \sqsubset x$. 2) If there exists some $z \in B$ such that $x \sqsubseteq z$, then $x \sqsubseteq B$; otherwise, $x \sqsubset B$.*

The following example shows that we cannot uniquely determine $B \mapsto x$ and $x \mapsto B$ when $y \sqsubseteq x$ for every $y \in B$.

Example 8 (Test Frame Containing Partially Embedded Choices Only). Let x be a valid choice and $B = \{y, z\}$ be a valid test frame. Suppose $y \sqsubseteq x$ and $z \sqsubseteq x$. In this case, $B \mapsto x$ and $x \mapsto B$ may take one of the forms depicted in Fig. 4. It can be seen from the figure that the relational operator for $B \mapsto x$ and $x \mapsto B$ may be "$\sqsubset$," "$\sqsubseteq$," or "$\not\sqsubset$."

In spite of the above limitation, the next proposition shows that $x \mapsto B$ can still be determined in some circumstances.

Proposition 10 (Test Frame Containing Fully Embedding Choices Only). *Let x be a valid choice and B be a valid test frame. We have $x \sqsubset B$ if and only if $x \sqsubset y$ for every $y \in B$.*

In summary, except when $y \sqsubseteq x$ for every $y \in B$, both the relational operators for $B \mapsto x$ and $x \mapsto B$ can be uniquely determined using Propositions 8 and 9. Even in the partial embedding case, $x \mapsto B$ can still be determined using Proposition 10 under certain circumstances. Thus, a manual definition of relational operators is not required in all these cases. This proves to be very handy in test frame construction, which will be discussed in Section 3.3.4.

3.3.4 Test Frame Construction

Our approach for the construction of test frames is *incremental*, consisting of the following two steps:

1. For every unprocessed choice x, combine it with each test frame B to form one or more test frames B_{new}s. Continue with this process until the resource constraints in terms of $\overline{M}$ and $\underline{m}$ have been exceeded.
   a. The selection of x depends on its relative priority so that choices with higher priorities are selected first for combining with existing Bs to form B_{new}s.
   b. The relational operators for $B \mapsto x$ and $x \mapsto B$ determine how to generate B_{new}s. Hence, these operators are indispensable in test frame construction. They are determined automatically using Propositions 8 and 9 except when $y \sqsubseteq x$ for every $y \in B$. In the latter case, $x \mapsto B$ can be determined using Proposition 10 under certain circumstances. Furthermore, when B consists only of a single choice y, $B \mapsto x$ and $x \mapsto B$ are effectively the same as $y \mapsto x$ and $x \mapsto y$ because of Proposition 6. Hence, they can be found directly from the choice relation table.
2. For every incomplete test frame B_{new} generated in Step 1, extend it into a complete test frame.

Before we present our construction algorithm for test frames, we have to discuss the construction rules first. These rules help us to generate new test frames in an incremental manner.

Given a valid choice x and a valid test frame B, it follows from Dual Proposition 1 and Dual Corollary 1 that the only possible relations between x and B are

1. $B \not\sqsupset x$ and $x \not\sqsupset B$,
2. $B \sqsubset x$ and $x \sqsubset B$,
3. $B \sqsubset x$ and $x \sqsubseteq B$,
4. $B \sqsubseteq x$ and $x \sqsubset B$, and
5. $B \sqsubseteq x$ and $x \sqsubseteq B$.

If we know the relations between x and B, we can use the following rules to construct new test frames:

Construction Rule 1. If $B \not\sqsupset x$ and $x \not\sqsupset B$, then any complete test frame containing x will not contain B and any complete test frame containing B will not contain x. To generate all the complete test frames for this situation, we need to retain the original test frame B and construct a new test frame $\{x\}$.

Construction Rule 2. If $B \sqsubset x$ and $x \sqsubset B$, then any complete test frame containing B will contain x and vice versa. To reflect the situation, we replace B by a new test frame $B \cup \{x\}$.

Construction Rule 3. If $B \sqsubset x$ and $x \sqsubseteq B$, then any complete test frame containing B will contain x but not vice versa. To reflect the situation, we replace B by new test frames $\{x\}$ and $B \cup \{x\}$.

Construction Rule 4. If $B \sqsubseteq x$ and $x \sqsubset B$, then any complete test frame containing x will contain B but not vice versa. To reflect the situation, we need to retain the original test frame B and construct a new test frame $B \cup \{x\}$.

Construction Rule 5. If $B \sqsubseteq x$ and $x \sqsubseteq B$, then complete test frames containing B may or may not overlap with complete test frames containing x. To generate all the complete test frames for this situation, we retain the original test frame B and construct new test frames $\{x\}$ and $B \cup \{x\}$.

Let x be a valid choice and B be a valid test frame. If there exists some valid choice $y \in B$ such that $y \not\sqsupset x$, then by Proposition 8 and Construction Rule 1, we need to retain the original test frame B and construct a new test frame $\{x\}$. If there exists some valid choice $y \in B$ such that $y \sqsubset x$, then, by Proposition 9 and Construction Rules 2 and 3, we should replace B by a new test frame $B \cup \{x\}$ and add a new test frame $\{x\}$ if appropriate. If $x \sqsubset y$ and $y \sqsubseteq x$ for every $y \in B$, then, by Proposition 10, Construction Rule 2 or 4 should apply. Since every test frame generated by Construction Rule 2 will also be generated by Construction Rule 4 but not vice versa, we recommend applying Construction Rule 4 in order to play it safe and avoid possible omissions. Thus, we shall retain the original test frame B and construct a new test frame $B \cup \{x\}$.

For the case where $y \sqsubseteq x$ for every $y \in B$ and no other information is known, we recommend applying Construction Rule 5 so as to play it safe and avoid possible omissions since every test frame generated by any of Construction Rules 1 to 4 will also be generated by Construction Rule 5 but not vice versa. Thus, the original test frame B will be retained and new test frames $\{x\}$ and $B \cup \{x\}$ will be constructed. By Proposition 7, the above scenarios are mutually exclusive and, hence, the recommended procedure is well-defined.

Appendix B shows the algorithm $build\_test\_frame$ for the incremental construction of test frames. As we can see from this algorithm, the number of executions is in the order of $c \times w \times \overline{M}$ in the worst case, where c is the total number of categories, w is the total number of choices, and $\overline{M}$ is the absolute maximum number of generated test frames.

On the completion of $build\_test\_frame$, all the test frames generated are stored in $\mathcal{K}$. Furthermore, because of Step 2.2.b.v in $build\_test\_frame()$, all the generated test frames are distinct.

3.3.5 Test Frame Extension

In the algorithm $build\_test\_frame$, a preferred maximum number of test frames $\overline{M}$ and a minimally achievable priority level $\underline{m}$ are used by the software tester to specify the resource constraints. As a result, some of the choices

may remain unprocessed or partially processed when the algorithm is terminated because of resource limitations. In this way, some test frames in $\mathcal{K}$ may not be complete. Obviously, these incomplete test frames should be extended further.

Given any incomplete test frame B, we would like to extend it to include valid choices x that remain unprocessed or partially processed, as long as B and x do not mutually exclude each other. The extension rule can easily be formulated using the relation $B \mapsto x$, as follows:

Extension Rule. When B is fully or partially embedded in x, extend B into $B \cup \{x\}$. When B is not embedded in x, do not change B.

We observe that the number of test frames will remain unchanged when the extension rule is applied. Hence, we can preserve the constraint on the number of generated test frames as imposed by $\overline{M}$ and $\underline{m}$.

To determine $B \mapsto x$ for the extension rule, we note the following:

1. If B consists only of a single choice, we can determine $B \mapsto x$ from the choice relation table.
2. Otherwise, if there exists some valid choice $y \in B$ such that $y \not\sqsupset x$, then, by Proposition 8, we have $B \not\sqsupset x$.
3. Otherwise, if there exists some valid choice $y \in B$ such that $y \sqsubset x$, then, by Proposition 9, we have $B \sqsubset x$.
4. Otherwise, if $x \sqsubset y$ for every valid choice $y \in B$, then, by Proposition 10, we have $x \sqsubset B$. Hence, by Dual Corollary 1.1, we must have $B \sqsubset x$ or $B \not\sqsubset x$.

For those cases not covered above, users will be requested to define $B \mapsto x$ manually. Based on the above logic, we have developed an algorithm called $extend\_test\_frame$ (as shown in Appendix C), which extends every incomplete test frame generated by $build\_test\_frame$. For this algorithm, the number of executions involved is on the order of $c \times w \times N$ in the worst case, where c is the total number of categories, w is the total number of choices, and N is the total number of generated test frames.

Once we have a set of complete test frames, the generation of test cases is straightforward. Given any complete test frame B, we randomly select a single element from each choice contained in B. The set of elements thus selected will constitute a test case corresponding to B. Consider, for example, the following complete test frame generated by the algorithm $extend\_test\_frame$:

$$
\begin{aligned}
B_1 = \{&\text{Type of Applicant} = \text{Cardholder},\\
&\text{Type of Credit Card} = \text{Classic}, \text{Credit Limit} = \$2,000,\\
&\text{Employment Status} = \text{Employed},\\
&\text{Type of Employment} = \text{Employed by Others},\\
&\text{Type of Job} = \text{Permanent},\\
&\text{Monthly Salary } (S) = \$0 < S \leq \$2,000\}.
\end{aligned}
$$

The following may be selected as a test case corresponding to B_1:

$$
\begin{aligned}
&\text{Type of Applicant} = \text{Cardholder},\\
&\text{Type of CreditCard} = \text{Classic}, \text{Credit Limit} = \$2,000,\\
&\text{Employment Status} = \text{Employed},\\
&\text{Type of Employment} = \text{Employed by Others},\\
&\text{Type of Job} = \text{Permanent}, \text{Monthly Salary } (S) = \$1,358.
\end{aligned}
$$

3.3.6 Merits of Test Frame Construction

Our choice relation framework supports CPM mainly in 1) consistency checks and automatic deductions of choice relations and 2) the automatic but constrained generation of test frames. The effectiveness of consistency checks, automatic deductions, and group constraint definitions has been discussed in Section 3.1.5. The approaches in constraining the total number of generated test frames can be compared as follows:

- In the original CPM, special annotations [`error`] and [`single`] can be attached to a choice x so that a complete test frame containing only x will be generated [4], [17]. The [`error`] annotation is designed to test a particular value that will cause an exception or other error state. It is assumed that any call of the function with this particular value in the annotated parameter or environment condition will result in the same error. On the other hand, the [`single`] annotation is intended to describe special, unusual, or redundant conditions that do not need to be combined with other possible choices. The main purpose of using these two annotations is to reduce the number of complete test frames generated.

  Given a special choice x, this objective can also be achieved through our framework by defining all the choice relations $x \mapsto y$ as $x \not\sqsupset y$ for any choice $x \neq y$. In this case, x will not be combined with any other choices to form complete test frames. Instead, a single complete test frame containing only x will be generated.
- In the original CPM, the number of generated test frames can only be reduced by means of incorporating additional constraints among choices. As a result, the tester does not have direct control of the exact number of test frames generated. After all the constraints have been taken into consideration, further reduction will not be possible even if the number of generated test frames is still too large for the available testing resources. On the contrary, our framework provides a means of further reducing the number of test frames after all the choice relations (that is, constraints) have been considered. This is achieved using $\overline{M}$, $\underline{m}$, and the relative priorities of individual choices, as explained in Section 3.2.

4 Related Work

It would be worth reviewing other work related to the original CPM and our choice relation framework:

1. Amla and Ammann [1] suggest that CPM is applicable to natural-language functional specifications, which may be incomplete and unstructured. Software testers will need undue effort to define testing requirements, thus hampering the effectiveness of

the method. On the other hand, they argue that testing requirements are, to a large extent, already captured in formal specifications. They analyze the feasibility of applying CPM to Z specifications and verify that testing requirements can be derived from formal specifications more easily.

2. Following up on the above study, Ammann and Offutt [2] define a minimal coverage criterion, called the *base-choice-coverage criterion*, for category-partition testing. They develop a procedure for converting Z specifications into test specifications that satisfy this criterion and introduce a method to produce test scripts from the test specifications.
3. Using the notion of test templates, Stocks and Carrington [19] develop a unified, flexible, and formal framework for specification-based testing. Their framework provides not only a formal model of tests and test suites, but also a method for applying the model in testing. In this way, test suites can be constructed in a concise and formal manner. They also investigate several application areas of the framework, including test oracles, refinement, and regression testing.
4. Zeil and Wild [20] observe that test descriptions generated by testing criteria are, effectively, sets of constraints that define test cases. Solutions to a set of constraints correspond to actual test data.[3] There may, however, be more that one solution and a common practice is to choose one of them arbitrarily. Zeil and Wild argue that some solutions may have a higher probability of revealing failures. Hence, they suggest a refinement process to reduce the solution set with the aim of identifying test data with a higher failure-revealing capability. Refinement is achieved by imposing further constraints incrementally.
5. Offutt and Irvine [16] investigate the effectiveness of fault detection in object-oriented programs using test cases generated by CPM. Common types of faults in C++ programs are identified. Such faults are inserted into two programs. Test cases are then generated using CPM with a view to uncovering these seeded faults. Their results show that these test cases help identify almost all the faults, except those involving memory management. They propose that C++ programs can be tested effectively by combining CPM with a tool for detecting memory management faults. They further conclude that traditional testing techniques, such as CPM, are also effective for testing object-oriented programs and, hence, software developers do not need new testing methods in the object-oriented paradigm.
6. Grochtmann and Grimm [11] propose a *classification-tree method* to help construct test cases from functional specifications. In this method, a classification tree, which is in the form of a hierarchical structure, organizes classifications and classes at alternate levels.[4] The basic approach of the classification-tree method is very similiar to that of CPM, namely, to build a model of the constraints in the input domain with a view to generating all the valid test frames while suppressing invalid ones as far as possible. Chen et al. [7] further study how to improve on the tree structure to facilitate the construction of test frames.
7. Previous work has also been done on *test case prioritization*. For example, Avritzer and Weyuker [3] develop load testing strategies to generate test suites to check the resource allocation behavior of software systems according to operational profiles. Elbaum et al. [9] study version-specific test case prioritization techniques in regression testing, with a view to improving the rate of fault detection.

The major difference between approaches 1-3 and ours is that the former are based on formal specifications. In our project, rather than formalizing the specification language, we attempt to improve on CPM by proposing a rigorous and systematic framework. On the other hand, both approach 3 and ours provide a formal framework to systematically define test suites for specification-based testing.

In 3 and 4, the notion of refinement has been used to derive test cases. Our framework can also support the generation of test cases via refinement, such as by splitting choices or imposing additional constraints on choices. It will be interesting to investigate, as future research, how the concept of refinement can be used to enhance the choice relation framework so as to further facilitate the construction of test cases from test frames.

With regard to 5, we note that discussions on the testing of object-oriented software are beyond the scope of the current paper. Readers may refer to [5], [6] for our perspective on this topic.

CPM and 6 differ in how invalid test frames are modeled and suppressed. CPM achieves this by capturing constraints in textual form, whereas 6 represents constraints in the form of tree structures. It has been found that the latter approach may not be applicable to every scenario.

Finally, the work highlighted in 7 studies the generation of test suites to cover states in proportion to their use [3], or the prioritization of existing tests in regression testing [9]. On the other hand, our work addresses prioritization from the perspective of the specification-based CPM.

5 Conclusion

In this paper, we have developed a choice relation framework for supporting category-partition test case generation. The major merits of the framework are:

1. We capture the constraints among choices in a rigorous and systematic manner via the introduction of various relations.
2. We improve on the effectiveness and efficiency of complete test frame construction by means of consistency checks and automatic deductions of relations.
3. We provide a means of removing only the incorrectly defined relations and any related ones, thereby saving the effort of repeating the entire construction process for the choice relation table.
4. We provide a direct way to control the maximum number of generated test frames.

3. Note that the term "test case" used by Zeil and Wild corresponds to "test frame" in our choice relation framework, while their term "test data" corresponds to "test case" in our terminology.

4. Classifications and classes in the classification-tree method correspond to categories and choices in CPM, respectively.

5. We enable the software tester to specify the relative priorities for choices that are used for the subsequent formation of complete test frames.

We have applied our approach to real-life situations and reported on the effectiveness of consistency checks and automatic deductions of choice relations.

APPENDIX A

ALGORITHM FOR CONSTRUCTING THE CHOICE RELATION TABLE

Procedure $build\_table(\mathcal{T})$

1. **Initialization of Buffer of Manually Defined Relations BR and List of Previously Defined and Deduced Relations LR**
   a. Initialize BR as an empty linked list. It will be used for storing temporarily the choice relations defined manually by users.
   b. Initialize LR as an empty linked list. Each element of LR is a linked list that captures the contents of the choice relation table $\mathcal{T}$ immediately before the system carries out a correction of erroneous choice relations.
2. **Initialization of Element Relation Table $\mathcal{E}$**
3. **Initialization of Diagonal Elements**
4. **Initialization of Elements for Different Choices of the Same Category.** For every element $t(i,j)$ in $\mathcal{T}$ such that $i \neq j$ and the corresponding pair of choices belong to the same category, initialize $t(i,j)$ and update the relevant entries and deduction linked lists in $\mathcal{E}$ using the procedure $update\_deduction\_details(i,j)$.
5. **Updating of Choice Relation Table $\mathcal{T}$.** Repeat the following until all the elements $t(i,j)$ in $\mathcal{T}$ have been defined or deduced:
   a. Choose an element $t(i,j)$ with the *largest* counter value of $e(i,j)$. If there is more than one such $t(i,j)$, then arbitrarily choose one of them.
      - If the relational operator for $t(i,j)$ appears in the buffer BR, then move it to $\mathcal{T}$.
      - Otherwise, prompt the user to define the relational operator for $t(i,j)$ and store it in $\mathcal{T}$.
   b. Set the type and the counter of $e(i,j)$ to -1.
   c. Update the relevant counters and deduction linked lists in $\mathcal{E}$ using the procedure
      $$update\_deduction\_details(i,j).$$
   d. Suppose $t(i,j)$ in Step 5.a corresponds to the choice relation $x \mapsto y$, where y $(\neq x)$ is under the category Y. If, for every choice y' $(\neq y)$ under Y, the choice relation $x \mapsto y'$ is not found in $\mathcal{T}$ and BR, then confirm with the user whether the relational operator for $t(i,j)$ should also be applied to all such $x \mapsto y'$. If so, store the relational operators for all $x \mapsto y'$ into the buffer BR.
   e. Perform consistency checks for all the defined and deduced elements in $\mathcal{T}$ using the propositions and corollaries of Section 3.1.2. If no inconsistency is detected, then proceed to Step 5f. Otherwise, check whether the combination of defined and deduced elements in $\mathcal{T}$ exists in the linked list LR. If so, alert the users that they have encountered this problem before and prompt them to define another relational operator for $t(i,j)$. Otherwise, perform the following:
      - Append a copy of the contents of $\mathcal{T}$ to the linked list LR.
      - For any set of inconsistent elements $S = \{t(m_1,n_1), t(m_2,n_2), \ldots, t(m_k,n_k)\}$ in $\mathcal{T}$, alert the users about the following:
        i. every element in S that is manually defined,
        ii. every element in S that is automatically initialized in Step 4 of this procedure, and
        iii. the manually defined ancestor(s) of every element in S that is automatically deduced in Step 5f of this procedure.

        Then, prompt the user to select the erroneous ones from i and iii above.
      - Correct the selected elements using the procedure $correct\_operator(\,)$.

      Repeat Step e until no inconsistency is detected.
   f. Whenever possible, perform automatic deductions for yet-to-be-defined elements, using Propositions 1, 2.1, 3.1, and 4.1, and Corollary 2.1. For every element $t(p,q)$ whose relational operator has been deduced:
      - Set the type of $e(p,q)$ to 1 and the corresponding counter to -1.
      - Update the relevant counters and deduction linked lists in $\mathcal{E}$ using the procedure $update\_deduction\_details(p,q)$.
      - Initialize $PL_{p,q}$ as an empty linked list, and store its header address in the parent-pointer of $e(p,q)$.
      - Append each parent element of $t(p,q)$ to $PL_{p,q}$.

Procedure $update\_deduction\_details(i,j)$

1. Initialize $DL_{i,j}$ as an empty linked list and store its header address in the deduction-pointer of $e(i,j)$.
2. Identify all the elements $t(m,n)$ of $\mathcal{T}$ such that the system deduction rules in Section 3.1.3 can be applied. For every such $t(m,n)$:
   a. If the counter in the corresponding $e(m,n)$ is smaller than the largest integer value supported by the system, then increase it by one and
   b. Append a node "(m,n)" to $DL_{i,j}$.
3. For every defined or deduced element $t(p,q)$ in $\mathcal{T}$, delete the node "(i,j)," if any, from the corresponding $DL_{p,q}$.

Procedure $correct\_operator(m, n)$

1. Delete the following from $\mathcal{T}$:
   a. the manually defined element $t(m, n)$ and
   b. all the deduced elements, if any, that have $t(m, n)$ as an ancestor.

   Note that the ancestor information of an element can be obtained from its parent linked list.

   To delete an element $t(p, q)$ from $\mathcal{T}$:
   i. Set the type of the corresponding $e(p, q)$ to 0 and its parent-pointer to null.
   ii. Set the counter of the corresponding $e(p, q)$ to the largest integer value supported by the system. This ensures that $t(p, q)$ has the highest priority for the next manual definition.
   iii. For every element $t(x, y)$ of $\mathcal{T}$ corresponding to an element (x, y) of the deduction linked list $DL_{p,q}$, decrease the counter of $e(x, y)$ by one.
   iv. Set the deduction-pointer of $e(p, q)$ to null.
2. Prompt the user to define the relational operator for $t(m, n)$ and store it in the buffer BR.

Appendix B

Algorithm for Generating Test Frames

Procedure $build\_test\_frame(\mathcal{T}, \mathcal{P}, \mathcal{K})$. In this procedure, a linked list $\mathcal{K}$ is used to store all the generated test frames. In terms of data structures, each element of the linked list $\mathcal{K}$ points to a linked list whose elements are the choices of a test frame.

1. **Initialization**
   1.1. Input the preferred maximum number of complete test frames $\overline{M}$ and the minimally achievable priority level $\underline{m}$.
   1.2. Suppose r denotes the relative priority of a choice in the choice priority table $\mathcal{P}$. Set the current priority level (denoted by L) to the minimum r defined.
   1.3. Initialize $\mathcal{K}$ as an empty linked list.
2. **Construction of Possible Test Frames**. Let $N(\mathcal{K})$ denote the total number of test frames stored in $\mathcal{K}$. While (there exists any unprocessed choice) and $(L \leq \underline{m}$ or $N(\mathcal{K}) < \overline{M})$, do the following:
   2.1. Select an unprocessed choice x with priority level L.
   2.2. If $N(\mathcal{K}) = 0$, then store the test frame $\{x\}$ into $\mathcal{K}$. Otherwise, perform the following:
      a. Initialize $\mathcal{K}_{new}$ as an empty linked list. This linked list has the same structure as $\mathcal{K}$.
      b. Let $N(\mathcal{K}_{new})$ denote the total number of test frames stored in $\mathcal{K}_{new}$. While $N(\mathcal{K}) > 0$ and $(L \leq \underline{m}$ or $(N(\mathcal{K}) + N(\mathcal{K}_{new}) < \overline{M}))$, do the following:
         i. Select any test frame B from $\mathcal{K}$.
         ii. Determine $B \mapsto x$ and/or $x \mapsto B$ using Propositions 8, 9, and 10. Then, generate new test frames for $\mathcal{K}_{new}$ according to the recommendations in Section 3.3.4.
         iii. Store all the newly generated test frames in $\mathcal{K}_{new}$. If, according to the construction rule, B should be retained, then store it into $\mathcal{K}_{new}$ also.
         iv. Delete B from $\mathcal{K}$.
         v. Remove all but one duplicated test frame from $\mathcal{K}_{new}$.
      c. For every B_{new} in $\mathcal{K}_{new}$, move it to $\mathcal{K}$ until $N(\mathcal{K}_{new}) = 0$ or $(L > \underline{m}$ and $N(\mathcal{K}) \geq \overline{M})$.
   2.3. Set L to the smallest relative priority of all the unprocessed choices, if any.

Appendix C

Algorithm for Extending Incomplete Test Frames

Procedure $extend\_test\_frame(\mathcal{T}, \mathcal{P}, \mathcal{K})$. For every unprocessed or partially processed choice x in the choice priority table $\mathcal{P}$, repeat the following for all the test frames B in $\mathcal{K}$:

1. If B consists only of a single choice, determine $B \mapsto x$ from the choice relation table. Extend B into $B \cup \{x\}$ if $B \sqsubset x$ or $B \sqsubseteq x$.
2. Otherwise, if there exists some valid choice $y \in B$ such that $y \not\sqsupset x$, then take no action.
3. Otherwise, if there exists some valid choice $y \in B$ such that $y \sqsubset x$, then extend B into $B \cup \{x\}$.
4. Otherwise, if $x \sqsubset y$ for every valid choice $y \in B$, then extend B into $B \cup \{x\}$.
5. Otherwise, prompt the user for $B \mapsto x$. Extend B into $B \cup \{x\}$ if $B \sqsubset x$ or $B \sqsubseteq x$.

Appendix D

Dual Propositions and Corollaries

Dual Proposition 1 (Symmetry of the Nonembedding of Test Frames). *For any valid test frames B_1 and B_2, $B_1 \not\sqsupset B_2$ if and only if $B_2 \not\sqsupset B_1$.*

Dual Corollary 1 (Reverse of Full and Partial Embedding of Test Frames). *Let B_1 and B_2 be valid test frames. 1) If $B_1 \sqsubset B_2$, then $B_2 \sqsubset B_1$ or $B_2 \sqsubseteq B_1$. 2) If $B_1 \sqsubseteq B_2$, then $B_2 \sqsubset B_1$ or $B_2 \sqsubseteq B_1$.*

Dual Proposition 2 (Full Embedding of Test Frames). *Let B_1, B_2, and B_3 be valid test frames. 1) If $B_1 \sqsubset B_2$ and $B_2 \sqsubset B_3$, then $B_1 \sqsubset B_3$. 2) If $B_1 \sqsubset B_2$ and $B_1 \sqsubset B_3$, then $B_2 \sqsubset B_3$ or $B_2 \sqsubseteq B_3$.*

Dual Proposition 3 (Full Embedding and Nonembedding of Test Frames). *Let B_1, B_2, and B_3 be valid test frames. 1) If $B_1 \sqsubset B_2$ and $B_2 \not\sqsupset B_3$, then $B_1 \not\sqsupset B_3$. 2) If $B_1 \sqsubset B_2$ and $B_1 \not\sqsupset B_3$, then $B_2 \sqsubseteq B_3$ or $B_2 \not\sqsupset B_3$.*

Dual Corollary 2 (Full Embedding and Nonembedding of Test Frames). *Let B_1, B_2, and B_3 be valid test frames. 1) If $B_1 \sqsubset B_3$ and $B_2 \not\sqsupset B_3$, then $B_1 \not\sqsupset B_2$. 2) If $B_2 \sqsubset B_3$ and $B_1 \not\sqsupset B_2$, then $B_3 \sqsubseteq B_1$ or $B_3 \not\sqsupset B_1$.*

Dual Proposition 4 (Full and Partial Embedding of Test Frames). *Let B_1, B_2, and B_3 be valid test frames. 1) If $B_1 \sqsubset B_2$ and $B_1 \sqsubseteq B_3$, then $B_2 \sqsubseteq B_3$. 2) If $B_1 \sqsubset B_3$ and $B_2 \sqsubseteq B_3$,*

then $B_2 \sqsubseteq B_1$ or $B_2 \not\sqsubset B_1$. 3) If $B_2 \sqsubset B_3$ and $B_1 \sqsubseteq B_2$, then $B_3 \sqsubset B_1$ or $B_3 \sqsubseteq B_1$.

Dual Proposition 5 (Partial Embedding and Nonembedding of Test Frames). *Let B_1, B_2, and B_3 be valid test frames. 1) If $B_1 \sqsubseteq B_2$ and $B_2 \not\sqsubset B_3$, then $B_1 \sqsubseteq B_3$ or $B_1 \not\sqsubset B_3$. 2) If $B_1 \sqsubseteq B_2$ and $B_1 \not\sqsubset B_3$, then $B_2 \sqsubseteq B_3$ or $B_2 \not\sqsubset B_3$.*

Dual Corollary 3 (Partial Embedding and Nonembedding of Test Frames). *Let B_1, B_2, and B_3 be valid test frames. 1) If $B_1 \sqsubseteq B_3$ and $B_2 \not\sqsubset B_3$, then $B_1 \sqsubseteq B_2$ or $B_1 \not\sqsubset B_2$. 2) If $B_2 \sqsubseteq B_3$ and $B_1 \not\sqsubset B_2$, then $B_3 \sqsubseteq B_1$ or $B_3 \not\sqsubset B_1$.*

APPENDIX E

PROOFS AND DISCUSSIONS OF LEMMA, PROPOSITIONS, AND COROLLARY

Proof of Proposition 1 (Symmetry of the Nonembedding of Choices). This proposition follows directly from the definition of "$\not\sqsubset$." □

Discussions on Corollary (Reverse of Full and Partial Embedding of Choices). Given $x \sqsubset y$, the set of possible complete test frames may be divided into the following three disjoint partitions:

- $P(x \wedge y)$: Its complete test frames must contain x, y, and possibly other choices.
- $P(y \wedge \neg x)$: Its complete test frames must contain y and possibly some other choices, but not x.
- $P(\neg x \wedge \neg y)$: Its complete test frames contain neither x nor y.

Partition $P(x \wedge y)$ cannot be empty because x is a valid choice and $x \sqsubset y$. On the other hand, $P(y \wedge \neg x)$ or $P(\neg x \wedge \neg y)$ may be empty. If $P(y \wedge \neg x)$ is empty, we have $y \sqsubset x$. Otherwise, we have $y \sqsubseteq x$.

Now, suppose $x \sqsubseteq y$. In this case, the set of possible complete test frames may be divided into the following four disjoint partitions:

- $P(x \wedge y)$: Its complete test frames must contain x, y, and possibly other choices.
- $P(x \wedge \neg y)$: Its complete test frames must contain x and possibly some other choices, but not y.
- $P(y \wedge \neg x)$: Its complete test frames must contain y and possibly some other choices, but not x.
- $P(\neg x \wedge \neg y)$: Its complete test frames contain neither x nor y.

Partitions $P(x \wedge y)$ and $P(x \wedge \neg y)$ cannot be empty because x is a valid choice and $x \sqsubseteq y$. On the other hand, $P(y \wedge \neg x)$ or $P(\neg x \wedge \neg y)$ may be empty. If $P(y \wedge \neg x)$ is empty, we have $y \sqsubset x$. Otherwise, we have $y \sqsubseteq x$.

Proof of Proposition 2 (Full Embedding of Choices).

1. The proof follows directly from the definition of "$\sqsubset$."
2. Suppose $x \sqsubset y$ and $x \sqsubset z$. By Definition 3, $TF(x) \subseteq TF(y)$ and $TF(x) \subseteq TF(z)$. Hence, any complete test frame $B \in TF(x)$ is also in $TF(y)$ and $TF(z)$. Since x is valid, $TF(x)$ is nonempty. Thus, $TF(y) \cap TF(z) \neq \emptyset$, which means that $y \not\sqsubset z$ cannot be true. In other words, $y \sqsubset z$ or $y \sqsubseteq z$. □

Proof of Proposition 3 (Full Embedding and Nonembedding of Choices).

1. The proof follows directly from the definition of "$\sqsubset$" and "$\not\sqsubset$."
2. Suppose $x \sqsubset y$ and $x \not\sqsubset z$. Let us assume $y \sqsubset z$. It would follow from Proposition 2.1 that $x \sqsubset z$, which would contradict $x \not\sqsubset z$. Hence, we must have $y \sqsubseteq z$ or $y \not\sqsubset z$. □

Proof of Proposition 4 (Full and Partial Embedding of Choices).

1. Suppose $x \sqsubset y$ and $x \sqsubseteq z$. If we assumed $y \sqsubset z$, then it would follow from Proposition 2.1 that $x \sqsubset z$, which would contradict $x \sqsubseteq z$. On the other hand, if we assumed $y \not\sqsubset z$, then it would follow from Proposition 3.1 that $x \not\sqsubset z$, which would also contradict $x \sqsubseteq z$. Hence, we must have $y \sqsubseteq z$.
2. Suppose $x \sqsubset z$ and $y \sqsubseteq z$. Let us assume $y \sqsubset x$. It would follow from Proposition 2.1 that $y \sqsubset z$, which would contradict $y \sqsubseteq z$. Hence, we must have $y \sqsubseteq x$ or $y \not\sqsubset x$.
3. Suppose $x \sqsubseteq y$ and $y \sqsubset z$. Let us assume $z \not\sqsubset x$. By Proposition 3.1, we would have $y \not\sqsubset x$. It would follow from Proposition 1 that $x \not\sqsubset y$, which would contradict $x \sqsubseteq y$. Hence, we must have $z \sqsubset x$ or $z \sqsubseteq x$. □

Proof of Proposition 5 (Partial Embedding and Nonembedding of Choices).

1. Suppose $x \sqsubseteq y$ and $y \not\sqsubset z$. Let us assume $x \sqsubset z$. It would follow from Corollary 2.1 that $x \not\sqsubset y$, which would contradict $x \sqsubseteq y$. Hence, we must have $x \sqsubseteq z$ or $x \not\sqsubset z$.
2. Suppose $x \sqsubseteq y$ and $x \not\sqsubset z$. Let us assume $y \sqsubset z$. It would follow from Corollary 2.1 that $y \not\sqsubset x$, which would contradict $x \sqsubseteq y$ according to Proposition 1. Hence, we must have $y \sqsubseteq z$ or $y \not\sqsubset z$. □

The proofs of Dual Propositions 1 to 5 are similar to those of Propositions 1 to 5.

Proof of Proposition 6 (Generalization Property). Given any valid choice x, for any $B \in TF$, $B \in TF(\{x\}) \Leftrightarrow \{x\} \subseteq B \Leftrightarrow x \in B \Leftrightarrow B \in TF(x)$. □

Proof of Lemma 1 (Full Embedding Lemma). Suppose $B_1 \subseteq B_2$. For any $B \in TF(B_2)$, by Definition 4, we must have $B \in TF(B_1)$. By Definition 5, therefore, we have $B_2 \sqsubset B_1$. □

Proof of Proposition 7 (Mutual Exclusion of Fully Embedded and NonEmbedded Choices). Assume there exists a valid choice $y \in B$ such that $y \sqsubset x$. By Lemma 1, we have $B \sqsubset y$. Hence, by Dual Proposition 2.1, we have $B \sqsubset x$. Assume there exists a valid choice $z \in B$ such that $z \not\sqsubset x$. By Lemma 1, we have $B \sqsubset z$. Hence, by Dual Proposition 3.1, we have $B \not\sqsubset x$. Thus, we have a contradiction. □

Proof of Proposition 8 (Test Frame Containing Nonembedded Choice). Since $y \in B$, by Lemma 1, we have $B \sqsubset y$. Hence, by Dual Proposition 3.1, if $y \not\sqsubset x$, then $B \not\sqsubset x$. By Dual Proposition 1, we can also conclude that $x \not\sqsubset B$. □

Proof of Proposition 9 (Test Frame Containing Fully Embedded Choice).

1. Since $y \in B$, by Lemma 1, we have $B \sqsubset y$. Hence, by Dual Proposition 2.1, if $y \sqsubset x$, then $B \sqsubset x$.
2. Suppose there exists some $z \in B$ such that $x \sqsubseteq z$. By Lemma 1, we have $B \sqsubset z$. Hence, by Dual Proposition 4.2, we have $x \sqsubseteq B$ or $x \not\sqsupset B$. Now, according to part 1 of this proposition, we have $B \sqsubset x$. Hence, by Dual Proposition 1, we cannot have $x \not\sqsupset B$. Thus, we must have $x \sqsubseteq B$.

   Suppose there is no $z \in B$ such that $x \sqsubseteq z$. Since $B \sqsubset x$, by Dual Corollary 1.1, we can only have $x \sqsubset B$ or $x \sqsubseteq B$. Assume that $x \sqsubseteq B$. From Definition 5.2, we would have $TF(x) \not\subseteq TF(B)$. By Definitions 2 and 4, there would exist some $B' \in TF$ such that $x \in B'$ and $B \not\subseteq B'$. In other words, there would exist some $B' \in TF$ and $z \in B$ such that $x \in B'$ and $z \notin B'$. This would contradict the nonexistence of $z \in B$ such that $x \sqsubseteq z$. Hence, we must have $x \sqsubset B$. □

Proof of Proposition 10 (Test Frame Containing Fully Embedding Choices Only). The result follows directly from Definitions 4 and 5. □

ACKNOWLEDGMENTS

This research is supported in part by grants of the Research Grants Council of Hong Kong (Project Nos. HKU 7029/01E and CityU 1048/01E) and a discovery grant of the Australian Research Council (Project No. DP0345147).

REFERENCES

[1] N. Amla and P. Ammann, "Using Z Specifications in Category-Partition Testing," *Systems Integrity, Software Safety, and Process Security: Building the Right System Right: Proc. Seventh Ann. IEEE Conf. Computer Assurance (COMPASS '92)*, pp. 3-10, 1992.

[2] 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: Proc. Ninth Ann. Conf. Computer Assurance (COMPASS '94)*, pp. 69-79, 1994.

[3] A. Avritzer and E.J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Trans. Software Eng.*, vol. 21, no. 9, pp. 705-716, Sept. 1995.

[4] M.J. Balcer, W.M. Hasling, and T.J. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications," *Proc. Third ACM Ann. Symp. Software Testing, Analysis, and Verification (TAV '89)*, pp. 210-218, 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 Trans. Software Eng. and Methodology*, vol. 7, no. 3, pp. 250-295, 1998.

[6] H.Y. Chen, T.H. Tse, and T.Y. Chen, "TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels," *ACM Trans. Software Eng. and Methodology*, vol. 10, no. 1, pp. 56-109, 2001.

[7] T.Y. Chen, P.L. Poon, and T.H. Tse, "Classification-Tree Restructuring Methodologies: A New Perspective," *IEE Proc.: Software*, vol. 149, no. 2, pp. 65-74, 2002.

[8] T. Chusho, "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing," *IEEE Trans. Software Eng.*, vol. 13, no. 5, pp. 509-517, May 1987.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 159-182, Feb. 2002.

[10] R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 1, pp. 63-86, 1996.

[11] M. Grochtmann and K. Grimm, "Classification Trees for Partition Testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63-82, 1993.

[12] P. Jorgensen, *Software Testing: A Craftsman's Approach.* Boca Raton, Fla.: CRC Press, 2001.

[13] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software.* New York: Wiley, 1999.

[14] B. Korel, "Automated Test Data Generation for Programs with Procedures," *Proc. 1996 Int'l Symp. Software Testing and Analysis (ISSTA '96)*, pp. 209-215, 1996.

[15] G.J. Myers, *Software Reliability: Principles and Practices.* New York: Wiley, 1976.

[16] A.J. Offutt and A. Irvine, "Testing Object-Oriented Software Using the Category-Partition Method," *Proc. 17th Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS 17)*, pp. 293-304, 1995.

[17] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM*, vol. 31, no. 6, pp. 676-686, 1988.

[18] J. Sanders and E. Curran, *Software Quality: A Framework for Success in Software Development and Support.* Wokingham, U.K.: Addison-Wesley, 1994.

[19] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Trans. Software Eng.*, vol. 22, no. 11, pp. 777-793, Nov. 1996.

[20] S.J. Zeil and C. Wild, "A Knowledge Base for Software Test Refinement," *Proc. Eighth Knowledge-Based Software Eng. Conf.*, pp. 50-57, 1993.

T.Y. Chen received the BSc and MPhil degrees from The University of Hong Kong, MSc degree and DIC from the Imperial College of Science and Technology, and the PhD degree from the University of Melbourne. He is currently the Professor of Software Engineering in the School of Information Technology, Swinburne University of Technology, Australia. His research interests include software testing, debugging, software maintenance, and software design.

Pak-Lok Poon received the Master of Business (information technology) degree from the Royal Melbourne Institute of Technology and the PhD degree in software engineering from the University of Melbourne. He is currently an assistant professor in the Department of Accountancy at The Hong Kong Polytechnic University, where he teaches courses on information systems. He is on the editorial committee of the *Information Systems Control Journal*. His research interests include software testing, requirements inspection, computer audit and control, electronic commerce, and computers in education. He is a member of the IEEE and the IEEE Computer Society.

T.H. Tse received the PhD degree from the University of London and was a visiting fellow at the University of Oxford. He is currently an associate professor of computer science at The University of Hong Kong. He was the program chair of COMPSAC 2001, a program cochair of APAQS 2000, the steering committee chair of APAQS 2001, and a standing committee member of COMPSAC 2002. He is the steering committee chair of QSIC 2003 and a standing committee member of COMPSAC 2003. Dr. Tse is a fellow of the British Computer Society, a fellow of the Institute for the Management of Information Systems, a fellow of the Institute of Mathematics and Its Applications, a fellow of the Hong Kong Institution of Engineers, a senior member of the IEEE, and a member of the IEEE Computer Society. He was decorated with an MBE by the Queen.