<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>DESSERT: A Divide-and-conquer methodology for identifying categories, choices, and choice relations for test case generation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Chen, TY; Poon, PL; Tang, SF; Tse, TH</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>IEEE Transactions On Software Engineering, 2012, v. 38 n. 4, p. 794-809</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2012</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/140787">http://hdl.handle.net/10722/140787</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>IEEE Transactions on Software Engineering. Copyright © IEEE.; ©2012 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.</td>
</tr>
</tbody>
</table>
DESSERT: a DividE-and-conquer methodology for identifying categoryS, choiceS, and choiceE Relations for Test case generation

Tsong Yueh Chen, Member, IEEE, Pak-Lok Poon, Member, IEEE, Sau-Fun Tang, Member, IEEE, and T.H. Tse, Senior Member, IEEE

Abstract—This paper extends the CHOiCe reLATion framEwork, abbreviated as CHOCLATE, which assists software testers in the application of category/choice methods to testing. CHOCLATE assumes that the tester is able to construct a single choice relation table from the entire specification; this table then forms the basis for test case generation using the associated algorithms. This assumption, however, may not hold true when the specification is complex and contains many specification components. For such a specification, the tester may construct a preliminary choice relation table from each specification component, and then consolidate all the preliminary tables into a final table to be processed by CHOCLATE for test case generation. However, it is often difficult to merge these preliminary tables because such merging may give rise to inconsistencies among choice relations or overlaps among choices. To alleviate this problem, we introduce a DividE-and-conquer methodology for identifying categoryS, choiceS, and choiceE Relations for Test case generation, abbreviated as DESSERT. The theoretical framework and the associated algorithms are discussed. To demonstrate the viability and effectiveness of our methodology, we describe case studies using the specifications of three real-life commercial software systems.

Index Terms—Black-box testing, category-partition method, choice relation framework, choice relation table, software testing, test case generation.

1 INTRODUCTION

The black-box approach is a mainstream category of techniques for test case generation [3], [12], where test cases are constructed according to information derived from the specification without requiring knowledge of any implementation details. In software development, user and systems requirements are established before implementation, and hence, the specification should exist prior to program coding. The black-box approach is useful because test cases can be generated before coding has been completed. This facilitates development phases being performed in parallel, thus allowing time for preparing more thorough test plans and yet shortening the duration of the whole development process. Another merit is that it can be applied to test off-the-shelf software packages, where the source code is normally not available from vendors. These reasons make black-box testing very popular in the commercial sector.

Our investigation is built on the CHOiCe reLATion framEwork [7], [16], abbreviated as CHOCLATE, which supports category/choice methods in black-box testing. CHOCLATE assumes that a single choice relation table can be constructed from the specification in its entirety. This table captures choices and choice constraints and is the basis for test case generation using the associated algorithms provided by CHOCLATE. The assumption, however, may not hold true when the specification is complex and contains many specification components, such as narrative descriptions, use cases, and class diagrams. For such a specification, the tester may construct a preliminary choice relation table from each specification component individually, and then consolidate these preliminary tables into a final table to be processed by CHOCLATE for test case generation. These preliminary tables are often difficult to merge because such merging may give rise to inconsistencies among choice relations or overlaps among choices. To alleviate this problem, we introduce a DividE-and-conquer methodology for identifying categoryS, choiceS, and choiceE Relations for Test case generation, abbreviated as DESSERT.

Section 2 of this paper gives the motivation of our study by presenting a major problem in CHOCLATE [7], [16] that may hinder effective and wider application. Section 3 introduces important concepts of CHOCLATE that are essential for understanding DESSERT. Section 4 presents an overview of DESSERT. Section 5 discusses the key step of our methodology—the consolidation of preliminary choice relation tables—and describes part of our case studies involving a real-life commercial specification. Section 6 continues to discuss other parts of the case studies involving two additional commercial specifications. The
aim is to demonstrate the viability and effectiveness of DESSERT. Finally, Section 7 concludes the paper.

## 2 Motivation of Study

### 2.1 Overview of Choice Relation Framework

CHOC’LATE [7], [16] provides a systematic skeleton for constructing test cases from specifications using the category-partition approach [1], [15]. It identifies instances that influence the functions of a software system and generates test cases by systematically varying these instances over all values of interest. It generates test cases in three steps: 1) identify choices to partition the input domain (that is, the set of all possible inputs) of the software under test, 2) based on the constraints among choices, select valid combinations of choices so that each combination contains sufficient choices for test case generation, and 3) construct test cases from these valid choice combinations.

Several other black-box test case generation methods, such as the classification-tree method [6], [10], [11], [17], n-parameter-order [13], [18], domain testing [2], equivalence partitioning [14], and the avoid and replace methods [9], also largely follow the above three steps for test case generation. We will refer to them collectively as other category/choice methods in this paper.

The following example illustrates these three steps:

**Example 1 (Choice Relation Framework).** Consider an undergraduate degree classification system AWARD, which accepts the details for each student from an input file $F$. These details include the student ID, the number of years of study, the cumulative number of credits, and the grade point average (GPA). AWARD will then determine and advise the user whether a student is eligible to graduate. The minimum requirements for graduation are three years of study, 120 cumulative credits, and a GPA of 2.0. (Because of a restriction on the maximum number of courses students can enroll in each semester, it is impossible for students to attain 120 or more cumulative credits in less than three years of study.) If a student is eligible to graduate, AWARD will further determine the level of award that the student will obtain, such as a first-class honor.

**Step 1.** Categories and choices are identified from the specification of AWARD. A category is defined as a major property or characteristic of a parameter or an environment condition of the software system that affects its execution behavior. Parameters are explicit inputs to a system supplied by either the user or another system/program, whereas environment conditions are the states of a system at the time of its execution. The possible values associated with each category are partitioned into distinct subsets known as choices, with the assumption that all values in the same choice are similar either in their effect on the system’s behavior, or in the type of output they produce. Table 1 depicts the possible categories and their associated choices for AWARD. The category “Status of $F$” is defined with respect to an environment condition of AWARD, whereas the remaining four categories are defined with respect to parameters of AWARD.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Associated Choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status of $F$</td>
<td>Status of $F_{\text{undefined}}$, Status of $F_{\text{defined but empty}}$, Status of $F_{\text{defined and nonempty}}$</td>
</tr>
<tr>
<td>Student ID</td>
<td>Student ID $\leq$ digit number, Student ID $&gt; \text{digit number}$</td>
</tr>
<tr>
<td>Number of Years of Study</td>
<td>Number of Years of Study $&lt; 3$, Number of Years of Study $\geq 3$</td>
</tr>
<tr>
<td>Cumulative Number of Credits</td>
<td>Cumulative Number of Credits $&lt; 120$, Cumulative Number of Credits $\geq 120$</td>
</tr>
<tr>
<td>GPA</td>
<td>GPA $[0,0,2.0]$, GPA $[2.0,2.5]$, GPA $[2.5,3.0]$, GPA $[3.0,3.5]$, GPA $[3.5,4.0]$</td>
</tr>
</tbody>
</table>

Consider the valid choice combination $B_1$ first. It contains “Status of $F_{\text{defined but empty}}$” only because of the constraint that “Status of $F_{\text{defined but empty}}$” cannot be combined with any choice in categories “Student ID,” “Number of Years of Study,” “Cumulative Number of Credits,” and “GPA.” This constraint is based on an obvious rationale that when $F$ is empty, student details are not present. $B_1$ is useful for testing how AWARD behaves in the exceptional circumstances when nobody enrolls in a particular program. Now, consider the valid choice combination $B_2$. The choices “Student ID $\leq$ digit number,” “Number of Years of Study $\geq 3$,” “Cumulative Number of Credits $< 120$,” and “GPA $[0,0,2.0]$” require the coexistence of the choice “Status of $F_{\text{defined but empty}}$” only because of the constraint that “Status of $F_{\text{defined but empty}}$” cannot be combined with any choice in categories “Student ID,” “Number of Years of Study,” “Cumulative Number of Credits,” and “GPA.” This constraint is based on an obvious rationale that when $F$ is empty, student details are not present. $B_1$ is useful for testing how AWARD behaves in the exceptional circumstances when nobody enrolls in a particular program.
of $F_{\text{defined and nonempty}}$ to form a valid choice combination to be used in step 3 for test case generation. Specific student details, such as the number of years of study, can be obtained only when $F$ is defined and is nonempty.

**Step 3.** A test case is formed from every valid choice combination $B$ generated in step 2 by randomly selecting and combining an instance from each choice in $B$. Thus, a test case is a set of instances of the choices that forms a stand-alone input. Consider, for instance, the choice combination $B_3$ in step 2. A test case $tc = \{\text{Status of } F_{\text{defined and nonempty}}, \text{Student ID}_{7\text{-digit number}}, \text{Number of Years of Study} \geq 3, \text{Cumulative Number of Credits} \geq 98, \text{GPA} = 1.7\}$ can be formed. Here, the values "$3241750,$ "$3," "$98," and "$1.7$" are randomly selected from the relevant choices.

### 2.2 A Major Problem

We note that steps 1 and 2 of CHOC’LATE are very important. In step 1, the comprehensiveness of the identified categories and choices will affect the effectiveness of the set of test cases generated in step 3 for fault detection [4]. Suppose, for instance, that the software tester fails to identify a valid choice $x$. Then, any choice combination containing $x$ will not be generated. Consequently, any software fault associated with $x$ may not be detected. In step 2, the correctness of the defined choice constraints is also critical for the comprehensiveness of the generated test cases [8]. Any incorrectly defined choice constraint may result in the omission of some valid choice combinations. This in turn causes some test situations to be missed.

Inspired by this observation, we have conducted a close examination of CHOC’LATE (as well as other category/choice methods in testing), focusing particularly on steps 1 and 2. We find that CHOC’LATE, like other category/choice methods, is not explicitly developed for large and complex specifications. It assumes that identifying categories, choices, and choice constraints can be done in one single round for the entire specification. This assumption is not always true. Software testers often find the identification task for the entire specification difficult if the document is large and complex, expressed in many different styles and formats, or contains a large variety of components such as narrative descriptions, use cases, class diagrams, state machines, activity diagrams, and data flow diagrams.

To alleviate the problem, we propose a systematic methodology, referred to as DESSERT, to support steps 1 and 2 of CHOC’LATE. An appealing feature is that the identification process focuses on one specification component at a time and, hence, greatly eases the difficulties of identification associated with the entire specification. Grounded on a sound theoretical framework, DESSERT provides algorithms for consolidating preliminary choice relation tables (constructed from individual specification components) into a final table to be processed by CHOC’LATE for test case generation.

### 3 Preliminaries

We first introduce the important concepts [7], [16] that are essential for understanding DESSERT.

**Definition 1 (Test Frame and Its Completeness).** A test frame $B$ is a set of choices. $B$ is complete if, whenever a single instance is selected from every choice in $B$, a stand-alone input is formed. Otherwise, $B$ is incomplete.

The notion of test frames is, in fact, a formal treatment of choice combinations. Technically speaking, the input domain is partitioned into nonempty disjoint subsets that correspond to complete test frames. These test frames then form the basis for test case generation.

**Example 2 (Test Frame and Its Completeness).** Refer to Example 1. $B_1$ and $B_2$ are complete test frames. Consider the test frame

$$B_3 = \{\text{Status of } F_{\text{defined and nonempty}}, \text{Student ID}_{7\text{-digit number}}, \text{Number of Years of Study} \geq 3, \text{Cumulative Number of Credits} \geq 120\}.$$ 

$B_3$ is incomplete because we need additional information about GPA in order to generate a test case for AWARD.

CHOC’LATE provides predefined algorithms to generate a set of complete test frames and to construct test cases from these complete test frames. Among these test frames, we are interested in identifying those that share a common choice. Thus, we have the following definition:

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

**Example 3 (Set of Complete Test Frames Related to a Choice).** If we exhaustively list all the complete test frames for AWARD in Example 1, a total of 18 complete test frames can be found. First, consider the choice “Status of $F_{\text{undefined}}$.” $TF(\text{Status of } F_{\text{undefined}})$ is simply $\{\{\text{Status of } F_{\text{undefined}}\}\}$. Now, consider the choice “Number of Years of Study $< 3$.” We find that $TF(\text{Number of Years of Study} < 3)$ contains five complete test frames. For example, one of these complete test frames is

$$\{\text{Status of } F_{\text{defined and nonempty}}, \text{Student ID}_{7\text{-digit number}}, \text{Number of Years of Study} < 3, \text{Cumulative Number of Credits} < 120, \text{GPA}[0.0, 2.0]\}.$$ 

The above concept of $TF(x)$ is used to define the validity of a choice (Definition 3) and the relation between two choices (Definition 4).

**Definition 3 (Validity of a Choice).** Any choice $x$ is valid if $TF(x) \neq \emptyset$. Otherwise, it is invalid.

Obviously, a choice is meaningless or inappropriate if it is not related to a nonempty subset of the input domain.

**Example 4 (Validity of a Choice).** Refer to Example 3. Since $TF(\text{Number of Years of Study} < 3) \neq \emptyset$, “Number of Years of Study $< 3$” is a valid choice.

For the rest of this paper, valid choices are simply referred to as “choices.”
Refer to Definitions 1 and 2. CHOC’LATE generates valid combinations of choices as complete test frames by considering the constraints between pairs of choices [7], [16]. These constraints are captured in a choice relation table, denoted by $T$. Given $k$ choices, the dimension of $T$ is $k \times k$. A constraint between any two choices is formalized through the following concept:

**Definition 4 (Choice Relation between Two Choices).**

Given any choice $x$, its relation with another choice $y$ (denoted by $x \rightarrow y$) is defined as follows: 1) $x$ is **fully embedded** in $y$ (denoted by $x \sqsubset y$) if and only if every complete test frame that contains $x$ also contains $y$; 2) $x$ is **partially embedded** in $y$ (denoted by $x \sqsubseteq y$) if and only if there are complete test frame(s) that contain both $x$ and $y$ while there are also complete test frame(s) that contain $x$ but not $y$; and 3) $x$ is **not embedded** in $y$ (denoted by $x \not\sqsubseteq y$) if and only if there is no complete test frame that contains both $x$ and $y$.

In other words, 1) $x \sqsubset y$ if and only if $\text{TF}(x) \subseteq \text{TF}(y)$, 2) $x \sqsubseteq y$ if and only if $\text{TF}(x) \cap \text{TF}(y) \neq \emptyset$ and $\text{TF}(x) \subseteq \text{TF}(y)$, and 3) $x \not\sqsubseteq y$ if and only if $\text{TF}(x) \cap \text{TF}(y) = \emptyset$. Fig. 1 illustrates these relationships using Venn Diagrams.

Throughout the whole paper, we will use the “$\subseteq$” symbol to denote a **subset** relation and the “$\subset$” symbol to denote a proper subset relation. Also, in Definition 4, the symbols “$\subset$”, “$\sqsubseteq$”, “$\not\subseteq$”, and “$\not\sqsubseteq$” are called relational operators. Since the three types of choice relations are exhaustive and mutually exclusive, $x \sqsubset y$ can be uniquely determined. In addition, immediately from Definition 4, for any category $P$, the relational operator for $P_x \rightarrow P_y$ is “$\subset$” and that for $P_x \rightarrow P_y$ is “$\not\subset$” if $P_x \neq P_y$, since any pair of distinct choices $P_x$ and $P_y$ should be disjoint if defined properly.

**Example 5 (Choice Relation between Two Choices).** Refer to Example 1. We have (Number of Years of Study $\leq 3$) $\sqsubseteq$ (Status of $F_{\text{defined and nonempty}}$), indicating that every complete test frame containing “Number of Years of Study $\leq 3$” must also contain “Status of $F_{\text{defined and nonempty}}$.” The rationale is that $F$ must be defined and nonempty, from which the information on the number of years of study by the student can be obtained. An example of a partial embedding relation is (Status of $F_{\text{defined and nonempty}}$) $\not\subseteq$ (Number of Years of Study $\leq 3$). Any complete test frame containing “Status of $F_{\text{defined and nonempty}}$” may or may not contain “Number of Years of Study $\leq 3$”, because a complete test frame containing “Status of $F_{\text{defined and nonempty}}$” may contain “Number of Years of Study $\leq 3$” instead of “Number of Years of Study $\leq 3$”. Finally, an example of a non-embedding relation is (Number of Years of Study $\leq 3$) $\not\subset$ (Cumulative Number of Credits $\geq 120$). As stated in the specification of AWARD, a student cannot attain a minimum of 120 cumulative credits in less than three years of study.

The correctness of choice relations directly affects the comprehensiveness of the generated complete test frames. However, it is tedious and error prone to manually define all choice relations. Hence, Chen et al. [7] have identified various properties of these relations to form the basis for automatic deductions and consistency checking. We only list two of these properties here for illustration: (Property 1) Given any choices $x$, $y$, and $z$, if $x \sqsubset y$ and $y \not\sqsubseteq z$, then $x \not\sqsubseteq z$. (Property 2) Given any choices $x$, $y$, and $z$, if $x \sqsubseteq z$ and $y \not\sqsubseteq z$, then $y \not\subseteq x$ or $y \not\sqsubseteq x$.

The “then” part of Property 1 consists of a definite relation and, hence, provides a basis for automatic deduction of choice relations. More specifically, if $x \sqsubset y$ and $y \not\sqsubseteq z$ are manually defined by the tester, $x \not\sqsubseteq z$ can be automatically deduced without human intervention. As for Property 2, the “then” part contains two possible relations. Although this property cannot be used for automatic deductions, it nevertheless allows the tester to check the consistency of the relations among choices. For example, the tester knows that when $x \sqsubseteq z$ and $y \not\subseteq z$, we cannot have $y \sqsubset x$, or else it will contradict Property 2.

### 4 Overall Approach of Our Identification Methodology: DESSERT

To alleviate the problem of applying CHOC’LATE to complex specifications, DESSERT uses the following three-step approach to constructing a choice relation table $T$:

1. decompose the entire specification $S$ into several components $C_1, C_2, \ldots, C_n$ (where $n \geq 1$), with each $C_i$ ($i = 1, 2, \ldots, n$) modeling part of the behavior of the software under test, 2) construct a preliminary choice relation table $\tau_i$ from each $C_i$, and 3) consolidate $\tau_1, \tau_2, \ldots, \tau_n$ into a single $T$. Fig. 2 outlines the three steps of DESSERT. This “divide-and-conquer” approach is particularly useful when the software tester finds $S$ to be too large and complex for one single round of identifying categories, choices, and choice relations.

Strategies for supporting step 1 of DESSERT have been well discussed in the literature to decompose a specification into components for testing (based, for instance, on the functionality of individual systems). Also, much work [5], [6], [7], [10], [11], [17] has been done to support the identification of categories, choices, and choice constraints with the assumption that the proposed technique is applied to the entire specification in one go. Although the assumption may not work for large and complex specifications, such techniques...
are still effective in identifying categories, choices, and choice constraints during the construction of preliminary choice relation tables from specification components (where the tester can consider each component as a small specification). Because of this, we will focus only on step 3 of DESSERT in the rest of the paper.

5 CONSOLIDATION OF PRELIMINARY CHOICE RELATION TABLES

5.1 Terminology of DESSERT

In addition to the important concepts described in Section 3, we need the concept of overlapping choices [4] and the new concepts of header and trailer choices for understanding our consolidation technique for preliminary choice relation tables. The concept of overlapping choices is introduced to address the scenario that two distinct choices of the same category identified in two different specification components have common elements, which violates the basic requirement that choices of the same category must correspond to nonempty disjoint subsets of the input domain.

Definition 5 (Overlapping Choices). Given a category \( P_x \), two distinct choices \( P_x \) and \( P_y \) are said to be overlapping if and only if \( P_x \cap P_y \neq \emptyset \). In this case, \( P_x \) is a category with overlapping choices.

Example 6 (Overlapping Choices). Refer to Example 1. Suppose the category “Number of Years of Study” is now identified with two associated choices “Number of Years of Study < 3” and “Number of Years of Study \( \geq 3 \)”. In this case, the two choices are overlapping because the instance (Number of Years of Study = 3) exists in both choices. Furthermore, “Number of Years of Study” is a category with overlapping choices.

Readers are reminded that in all our previous discussions before the introduction of Definition 5 (including the automatic deductions and consistency checking of choice relations provided by CHOC’LATE [7]), choices are assumed to be nonoverlapping. Overlapping choices may occur when we consolidate preliminary choice relation tables together into a new table, which will be explained in Section 5.2 below.

Before defining header and trailer categories/choices, we need to introduce the following notation:

1. \( S \) denotes the entire specification with \( n \) components \( C_i \) (\( i = 1, 2, \ldots, n \)).
2. \( x \mapsto y \) denotes the choice relation between \( x \) and \( y \) with respect to the entire specification \( S \).
3. \( D \) denotes a nonempty subset of \( S \).
4. \( x \mapsto_D y \) denotes the choice relation between \( x \) and \( y \) with respect to \( D \). In particular, when \( D = S \), then \( x \mapsto_D y \) becomes \( x \mapsto y \).

We can then define header and trailer categories/choices.

Definition 6 (Header and Trailer Categories/Choices in a Choice Relation). Given any choice relation \( P_x \mapsto_D Q_y \), we refer to \( P \) and \( Q \) as the header category and trailer category and \( P_x \) and \( Q_y \) as the header choice and trailer choice.

Example 7 (Header and Trailer Categories/Choices in a Choice Relation). Consider the choice relation (Number of Years of Study \( < 3 \cap \) (Status of \( F \) defined and nonempty)) in Example 5. “Number of Years of Study,” “Status of \( F \), “Number of Years of Study \( \geq 3 \)”, and “Status of \( F \) defined and nonempty” are the header category, trailer category, header choice, and trailer choice, respectively, of this relation.

5.2 Problems to be Solved by DESSERT

Step 3 of DESSERT (see Section 4 above) is complicated because of the following problems:

1. **Problem of different choice relations for the same pair of choices.** A specification component alone may carry incomplete information about a particular choice and its associated choice relations. Let us consider an example. Suppose that two choices \( x \) and \( y \) always coexist in an input with respect to the specification component \( C_1 \) but never occur together in any input with respect to another component \( C_2 \). One tester may conclude that \( x \subseteq C_{(1)} \) and \( y \subseteq C_{(1)} \) by considering \( C_1 \) alone, while another tester may conclude that \( x \nsubseteq C_{(1)} \) and \( y \nsubseteq C_{(1)} \) from \( C_2 \) alone. In fact, \( C_1 \) and \( C_2 \) together suggest that the choice relations should be \( x \subseteq_{(1, C_1)} y \) and \( y \subseteq_{(1, C_1)} x \).

2. **Problem of overlapping choices.** Constructing preliminary choice relation tables separately from individual specification components may result in the occurrence of overlapping choices across different preliminary choice relation tables. Such overlapping choices can only be detected when considering different specification components simultaneously.

3. **Problem of different choice relations and overlapping choices.** When problems 1 and 2 occur together, the situation will become more complicated. We illustrate this situation with an example. Consider two distinct categories \( P \) and \( Q \) with the following properties: 3.1) Choices \( P_x \) and \( Q_a \) are identified from \( C_1 \). The choice relations \( P_x \subseteq_{(C_1)} Q_a \) and \( Q_a \subseteq_{(C_1)} P_x \) are then defined. 3.2) Choices \( P_y \) and \( Q_a \) are identified from \( C_2 \). The choice relations \( P_y \nsubseteq_{(C_1)} Q_a \) and \( Q_a \nsubseteq_{(C_1)} P_y \) are then defined.
3.3) $P_x \neq P_y$ and $P_x \cap P_y \neq \emptyset$. Here, problems 3.1 and 3.2 correspond to problem 1, and problem 3.3 corresponds to problem 2.

Let $P_x = P_y \cap P_z$. Based on $C_1$ alone, we can deduce that $P_x \subseteq (Q_a \cap C_1) P_x$ and $(Q_a \cap C_1) P_z$ or $Q_a \cap C_1 P_z$ because $P_x \subseteq P_z$. On the other hand, based on $C_2$ alone, we can deduce that $P_x \subseteq Q_a \cap C_2 P_x$. Hence, we have different choice relations between $P_x$ and $Q_x$ based on different specification components. In such circumstances, we need to redefine $P_x$ and $P_y$ by considering $C_1$ and $C_2$ together. The redefinition will render the previously determined choice relations $P_x \subseteq Q_a \cap C_1$, $Q_a \subseteq C_1$, $P_z$, $P_x \subseteq Q_a \cap C_2$, and $Q_a \subseteq C_2$ useless. Thus, the initial effort spent on defining the original choices and choice relations will be wasted. Note that problem 3.1 above will become even more complicated if $Q_a$ in problem 3.2 is replaced by $Q_b$ such that $Q_b$ overlaps with $Q_a$ in problem 3.1.

Obviously, the presence of problematic choices or choice relations may also indicate that the full specification is inconsistent. However, similarly to most other black-box testing techniques, our DESSERT methodology assumes that the specification is correct when it is used as the basis for test case generation.

The following example describes part of our first study using the specification of a commercial software system. Its aims are to illustrate steps 1 and 2 of DESSERT as well as to demonstrate the possible occurrence of the above problems in these two steps in a real-life setting.

Example 8 (Processing Visitor Requests: Part 1). Our first study involved the specification $S_{VISIT}$ of a Web-based visitor administration system VISIT, a real-life commercial software system now in use in an international airline, which is simply referred to as AIR in this paper. The main purposes of VISIT are to provide systematic and efficient registration, authorization, access control, and reporting of visitor activities at AIR.

To register an anticipated visitor to AIR, the staff member concerned makes a request in VISIT. Information such as the particulars of the staff member and the visitor, as well as visiting details, is entered as part of the request. If the visit

1. does not occur on a weekend or a public holiday,
2. is within office hours,
3. spans only a day or less, and
4. involves an access area within the default zone,

the request will go into the receptionist’s log. Otherwise, the request is considered exceptional and will go into the endorsement log, awaiting the approval of AIR Security. A request will also go into the endorsement log if the visitor is blacklisted in VISIT. If this happens, AIR Security can waive (or reject) the visitor request. After AIR Security has approved an exceptional request or waived a request involving a blacklisted visitor, the request will be moved from the endorsement log to the receptionist’s log. Later, when the visitor arrives at any reception counter, the operator will search the receptionist’s log for the appropriate visitor request record. If found, it will be edited by the operator before issuing a visitor access card.

The specification $S_{VISIT}$ contains various components such as narrative descriptions of the system, state machines, activity diagrams, data flow diagrams, and sample input and output screens. Hence, $S_{VISIT}$ lends itself to being a very good specification for our first study. Our study mainly focused on the function “Process Visitor Requests,” which is a core feature of VISIT. We recruited a volunteer for our study, referred to as Participant A. He has a postgraduate degree in IT and several years of commercial experience in software development.

We found one activity diagram (denoted by $AD_{REQUEST}$) in $S_{VISIT}$ related to the processing of visitor requests. We gave Participant A a copy of $AD_{REQUEST}$ and a one-page executive summary of $S_{VISIT}$ (instead of the entire specification), and asked him to construct from $AD_{REQUEST}$ a preliminary choice relation table (denoted by $\tau_{AD_{REQUEST}}$) using existing identification techniques such as the construction algorithm provided in [4]. The executive summary served mainly as a means to provide an overview of VISIT. This arrangement ensured that $\tau_{AD_{REQUEST}}$ could be constructed without the need for information from other specification components. As a further precaution, we explicitly asked Participant A not to refer to the executive summary when constructing $\tau_{AD_{REQUEST}}$ from $AD_{REQUEST}$. Our subsequent checking of $\tau_{AD_{REQUEST}}$ confirmed that this was indeed the case.

A close examination of $\tau_{AD_{REQUEST}}$ revealed that Participant A defined five categories, each associated with two choices. An example of these categories is “Type of Visitor” with “Type of Visitor normal” and “Type of Visitor blacklisted” as its two associated choices. To complete $\tau_{AD_{REQUEST}}$, Participant A determined 100 ($= (5 \times 2)^3$) choice relations.

In relation to visitor request processing, we also found one data flow diagram, one state machine, and one section of narrative description in $S_{VISIT}$. These are denoted by $DFD_{REQUEST}$, $SM_{REQUEST}$, and $ND_{REQUEST}$, respectively, in this paper. We repeated the study of $AD_{REQUEST}$ for each of these specification components to produce three more preliminary choice relation tables $\tau_{DFD_{REQUEST}}$, $\tau_{SM_{REQUEST}}$, and $\tau_{ND_{REQUEST}}$. The total numbers of categories (choices) defined from $AD_{REQUEST}$, $DFD_{REQUEST}$, $SM_{REQUEST}$, and $ND_{REQUEST}$ were 5 (10), 3 (9), 1 (2), and 12 (30), respectively. We noted that some of these categories and choices defined independently from different components were identical. After tallying, we found 12 categories and 32 choices that were distinct. We also observed that none of the individual components allowed Participant A to define all the categories and choices completely. This observation is consistent with our earlier argument that an individual specification component may only carry partial information about a choice and its associated choice relations.

Among the four preliminary choice relation tables, we found 44 pairs of choice relations that exhibit problem 1 as mentioned above. Examples of such pairs of choice relations are (Duration of Visit > 1 day) $\forall \{AD_{REQUEST}\}$ (Dates of Visit outside weekends and public holidays)
Fig. 3. Excerpt from the activity diagram $A_{\text{REQUEST}}$.

and (Duration of Visit $>$ 1 day) $\{\text{ND}_{\text{REQUEST}}\}$ (Dates of Visit outside weekends and public holidays). Consider the first choice relation. Its relational operator is “$\land$” because no thread (an execution path in an activity diagram) in $A_{\text{REQUEST}}$ is associated with the two guard conditions “$>$ 1 day” and “outside weekends and public holidays.” (See Fig. 3 for an excerpt from the activity diagram $A_{\text{REQUEST}}$ for illustration.) Now consider the second choice relation. According to $N_{\text{REQUEST}}$, the duration and the dates of visit are entered into VISIT as separate inputs. Furthermore, for a visitor request that spans more than a day, it may or may not include weekends and public holidays. This explains why the relational operator for the second choice relation is “$\lor$”.

We also found four pairs of overlapping choices (problem 2). For each of these pairs, the two overlapping choices were defined from different specification components and were therefore not detected by Participant A in the earlier stages of the study. The following explains how these overlapping choices occurred:

1. In $A_{\text{REQUEST}}$, Participant A found one decision point associated with two guard conditions “within office hours” and “outside office hours.” With respect to these two guard conditions, Participant A defined the category “Period of Visit” with “Period of Visit within office hours” and “Period of Visit outside office hours” as its associated choices. On the other hand, $N_{\text{REQUEST}}$ stated that the starting and ending times of visit were to be entered into VISIT as separate inputs. This information caused Participant A to define the category “Period of Visit” with three associated choices, namely “Period of Visit within office hours,” “Period of Visit partially outside office hours,” and “Period of Visit completely outside office hours.” Since

\[
\begin{align*}
\text{(Period of Visit\_{outside office hours})} \\
= \text{(Period of Visit\_{partially outside office hours})} \\
\cup \text{(Period of Visit\_{completely outside office hours})},
\end{align*}
\]

we have two pairs of overlapping choices.

2. $S_{\text{REQUEST}}$ indicated two states of a previously issued visitor access card, namely, “returned” and “not yet returned.” This information resulted in the definition of the category “Previous Access Card” with “Previous Access Card returned” and “Previous Access Card not yet returned” as its two associated choices. On the other hand, $N_{\text{REQUEST}}$ stated that VISIT would process the visitor request for a repeated visitor differently, depending on the return status of the previously issued access card: a) returned on time, b) returned late, and c) not yet returned. This information resulted in the definition of three choices “Previous Access Card returned on time,” “Previous Access Card returned late,” and “Previous Access Card not yet returned” for the category “Previous Access Card.” Since

\[
\text{(Previous Access Card\_{returned on time})} \\
\cup \text{(Previous Access Card\_{returned late})},
\]

there are two pairs of overlapping choices.

Altogether, we found 246 choice relations that involved overlapping choices. The numbers of choice relations associated with overlapping choices “Period of Visit\_{outside office hours},” “Period of Visit\_{partially outside office hours},” “Period of Visit\_{completely outside office hours},” “Previous Access Card\_{returned},” “Previous Access Card\_{returned on time},” and “Previous Access Card\_{returned late}” were 19, 59, 59, 3, 59, and 59, respectively.2

Because of the above problems, $T_{\text{REQUEST}}$, $T_{\text{DESSERT}}$, $T_{\text{SM\_REQUEST}}$, and $T_{\text{ND\_REQUEST}}$ could not be directly consolidated into a final choice relation table. Thus, the enormous effort spent by Participant A in constructing the four preliminary choice relation tables was wasted. Furthermore, the task of redefining the choices and their relations (by considering all the specification components together in one go) in order to get rid of the above problems was not easy to manage without the support of systematic methodologies.

In view of the possible occurrence of the above problems in steps 1 and 2 of DESSERT, step 3 of DESSERT is decomposed into two substeps, namely, step 3.a that deals with problem 1 and step 3.b that deals with problems 2 and 3, as discussed below.

5.3 Step 3.a of DESSERT

To alleviate problem 1 highlighted in Section 5.2, we have formulated Proposition 1 below, which is a simple, elegant, and yet useful result. Given any pair of choices with their relations identified separately from two distinct sets $D_1$ and $D_2$ of specification components, the main purpose of the proposition is to automatically deduce the choice relation with respect to both $D_1$ and $D_2$ without any manual definition process. For the rest of this paper, an automatically deduced choice relation will simply be referred to as a deduced choice relation, whereas a manually defined choice relation will simply be known as a defined choice relation.

Proposition 1 (Choice Relations in Different Sets of Specification Components). Let 1) $P$ and $Q$ be distinct
categories, 2) \( P_x \) and \( Q_x \) be choices, and 3) \( D_1 \) and \( D_2 \) be different sets of specification components. If the relational operators for \( P_x \rightarrow D, Q_x \) and \( P_x \rightarrow D, Q_y \) are identical, then \( P_x \rightarrow (D_1 \cup D_2) Q_x \) has the same relational operator as \( P_x \rightarrow D_1, Q_x \), and \( P_x \rightarrow D_2, Q_x \). Otherwise, the relational operator for \( P_x \rightarrow D_1 \cup D_2, Q_x \) is “\( \neq \)”. The proofs of all the propositions in this paper are given in the Appendix, which can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2011.69.

We next present our integration algorithm for merging two or more preliminary choice relation tables according to Proposition 1 (see, in particular, step 2.a1). We have three assumptions behind the algorithm: 1) Every preliminary choice relation table \( \tau_i \) involves at least two distinct categories because the integration of preliminary choice relation tables is only meaningful when every such table contains choice relations whose header and trailer categories are different. 2) Before integration starts, all the categories, choices, and choice relations in every \( \tau_i \) have been properly determined with respect to \( C_i \), corresponding to \( \tau_i \). 3) Overlapping choices do not exist within an individual preliminary choice relation table.

**Algorithm integration to Merge Preliminary Choice Relation Tables**

Suppose \( \tau_1, \tau_2, \ldots, \tau_n \) (where \( n \geq 2 \)) are the preliminary choice relation tables to be merged. Let \( S = \{ C_1, C_2, \ldots, C_n \} \) be the set of all specification components such that \( C_i \) (\( i = 1, 2, \ldots, n \)) correspond to the preliminary choice relation tables \( \tau_i \) (\( i = 1, 2, \ldots, n \)). Let \( D_j \) and \( D_l \) be any nonempty subsets of \( S \). We will use a linked list \( L \) to capture the result of merging \( \tau_1, \tau_2, \ldots, \tau_n \). Each element \( L_k \) of the linked list \( L \) (where \( k \geq 1 \)) points to an associated nonempty linked list \( LL_k \). Each \( LL_k \) is used to store the choice relations determined with respect to the nonempty subset \( D_k \) of \( S \). Each choice relation is stored as a pair of choices and their relational operator.

1. **Initialization of Linked List** /"Process \( \tau_1 \)"

   Initialize \( L \) as an empty linked list. For every choice relation \( P_x \rightarrow (C_1) Q_a \) in \( \tau_1 \) (where \( P \) and \( Q \) are **distinct** categories), store it in \( LL_1 \) associated with \( L_1 \) in \( L \).

2. **Integration of Preliminary Choice Relation Tables**

   /"Process \( \tau_2 \) to \( \tau_n \)"

   Incrementally integrate the choice relations in \( \tau_2, \tau_3, \ldots, \tau_n \) into those relations already stored in the linked lists associated with \( L \) (if applicable) by repeating the following steps for every \( \tau_i \) (\( i = 2, 3, \ldots, n \)):

   a. For every unprocessed choice relation \( P_x \rightarrow (C_i) Q_a \) (where \( P \) and \( Q \) are **distinct** categories):

      a1. If there exists some \( L_j \) in \( L \) pointing to an associated linked list \( LL_j \) that contains a choice relation \( P_x \rightarrow D_1, Q_a \), then: (i) Use Proposition 1 to deduce \( P_x \rightarrow D, Q_a \) from \( P_x \rightarrow D_1, Q_a \); and \( P_x \rightarrow D_2, Q_a \), where \( D_1 = \{ C_i \} \cup D_1 \); (ii) Delete \( P_x \rightarrow D_1, Q_a \) from \( LL_j \) associated with \( L_i \) in \( L \). (iii) Store \( P_x \rightarrow D_1, Q_a \) in \( LL_i \) associated with \( L_i \) in \( L \).

   a2. Otherwise, store \( P_x \rightarrow (C_i) Q_a \) in \( LL_i \) associated with \( L_i \) in \( L \).

   b. For every empty \( LL_k \), delete \( L_k \) from \( L \).

In the above algorithm, for each element \( L_k \) of \( L \), the corresponding \( LL_k \) stores the choice relations determined with respect to the nonempty subset \( D_k \) of \( S \). There are no overlapping choices within the same \( LL_k \). However, \( P_x \rightarrow D_1, Q_a \) and \( P_y \rightarrow D_2, Q_b \) (where \( j \neq i \)) may have overlapping choices. After applying integration, a) the choice relation involving \( P_x \) as the header choice and \( Q_a \) as the trailer choice is unique with respect to \( L \), that is, there is a unique \( L_k \) such that the corresponding \( LL_k \) contains this choice relation, and b) \( L \) has at most \( \sum n_i - 1 \) elements, where \( n \) is the number of specification components in \( S \), because each element of \( L \) corresponds to an nonempty subset of \( S \). Step 1 involves a one-off initialization and step 2 is iterated \( n - 1 \) times. Suppose \( r \) is the total number of choice relations across **all** the preliminary choice relation tables. With each execution of step 2, every element of \( \tau_i \) will be processed once with respect to all the relations stored in the associated linked lists of the current \( L \). Thus, the worst-case complexities of step 2 and the algorithm are of the order \( r^2 \) and \( nr^2 \), respectively.

In steps 1 and 2 of integration, any choice relation whose header and trailer choices belong to the **same** category does not need to be stored in any linked list associated with \( L \). This is because, given any category \( P \) and nonempty subset \( D \) of \( S \), by Definition 4, the relational operator for \( P_x \rightarrow D \) \( P_y \) and \( P_z \rightarrow D \) \( P_y \) (where \( P_x \) and \( P_y \) are distinct and non-overlapping choices) must be “\( = \)” and “\( \neq \)”, respectively. Thus, such choice relations can be automatically deduced in the next algorithm, refinement, to be introduced in Section 5.4 later.

**Example 9 (Processing Visitor Requests: Part 2)**. Let us continue from Example 8. When Participant A simulated the integration algorithm to merge the four preliminary choice relation tables, he encountered an unanticipated problem. Only one category was defined from \( SM_{\text{REQUEST}} \). This phenomenon contradicted an assumption in the algorithm that every preliminary choice relation table involves two or more categories. To solve this problem, we asked Participant A to consider \( DFD_{\text{REQUEST}} \) and \( SM_{\text{REQUEST}} \) together as one single \( C \) (denoted by \( M_{\text{REQUEST}} \)). Using this approach, four categories and 11 choices were defined from \( M_{\text{REQUEST}} \). Subsequently, Participant A constructed the corresponding preliminary choice relation table, denoted by \( \tau_{M_{\text{REQUEST}}} \), by defining the relation between every pair of choices. At this stage, three preliminary choice relation tables, namely, \( \tau_{AD_{\text{REQUEST}}} \), \( \tau_{M_{\text{REQUEST}}} \), and \( \tau_{ND_{\text{REQUEST}}} \), remained and their respective dimensions were \( 10 \times 10 \), \( 11 \times 11 \), and \( 30 \times 30 \).

Participant A then applied the integration algorithm to consolidate the three preliminary choice relation tables. During the consolidation process, he found 44 pairs of choice relations to which Proposition 1 could be applied. One such pair of choice relations is

\[
(\text{Duration of Visit} \geq 1 \text{ day}) \ P_\{AD_{\text{REQUEST}}\} \\
(\text{Dates of Visit outside weekends and public holidays})
\]
The application of Proposition 1 to this pair of choice relations in step 2.1 of the algorithm results in

\[
\text{(Duration of Visit}_{\text{non-weekend and public holidays}}) \subset \text{(Duration of Visit}_{\text{non-weekend and public holidays}})
\]

The focus on problem 2 and consider a hypothetical scenario as shown in Table 2. Three distinct categories \( P, Q, \) and \( R \) and their associated choices are identified from \( C_1 \). The relations among these choices are determined and captured in \( \tau_1 \) as shown in Table 2. Three distinct categories \( P, Q, \) and \( W \) and their associated choices are identified from \( C_2 \). The relations among these choices are determined and captured in \( \tau_2 \), as shown in Table 3. Overlapping choices do not exist within \( \tau_1 \) and \( \tau_2 \) individually.

Suppose \( Q_b \) in \( \tau_1 \) overlaps with \( Q_c \) and \( Q_d \) in \( \tau_2 \) such that \( Q_b = Q_c \cup Q_d \). Software testers may not be aware of these overlapping choices when constructing \( \tau_1 \) and \( \tau_2 \) separately because \( Q_b \) exists only in \( \tau_1 \) but not \( \tau_2 \), whereas \( Q_c \) and \( Q_d \) exist only in \( \tau_2 \) but not \( \tau_1 \). Note that, in this hypothetical case, only one category (namely, \( Q \)) involves overlapping choices. Without doubt, the case will become more complicated if \( P \) also contains overlapping choices.

To solve the problem, a straightforward approach is to replace \( Q_b \) in \( \tau_1 \) by \( Q_c \) and \( Q_d \) in \( \tau_2 \) after the replacement. We do not, however, recommend such an approach because software testers need to put extra effort in defining the new replacement choice relations, which would mean that the previous effort spent on determining numerous choice relations in \( \tau_1 \) is wasted. (About 31 percent of the choice relations are affected in this example.) It will be desirable if there is a refinement mechanism that will automatically deduce the new replacement choices and their choice relations as far as possible.

Note that this overlapping problem, involving \( Q_b, Q_c, \) and \( Q_d, \) is only related to problem 2. The case will become further complicated (corresponding to problem 3) if, after refining \( Q_b \) into \( Q_c \) and \( Q_d \) in \( \tau_1 \), some choice relations involving \( Q_c \) and \( Q_d \) defined from \( C_1 \) are different from their counterpart choice relations defined from \( C_2 \) (corresponding to problem 1).

With this need in mind, we have developed a refinement mechanism for overlapping choices and their choice relations. An appealing feature of the refinement technique is the incorporation of both the original version (introduced in [7]) and our extended version of the mechanisms for the automatic deductions and consistency checking of choice relations. While the original version can only be applied to nonoverlapping choices such as Properties 1 and 2 in Section 3, our extended version can be used for overlapping choices.

Before we present our refinement algorithm, we first introduce the following two propositions, which serve as the basis. See also Fig. 2 for their purposes and uses.

As can be seen in Examples 8 and 10, the overlap of choices is a core problem when merging preliminary choice relation tables into one final table. Consider the overlap of header choices first. Proposition 2 below is developed to refine choice relations having overlapping header choices. It aims to a) deduce new nonoverlapping header choices to replace the overlapping ones, and b) deduce, as far as possible, the choice relations for these newly deduced choices.
Next, we will explain how to use Proposition 2 to resolve the problem of overlapping header choices, such as $P_x$ and $P_y$ defined in $D_1$ and $D_2$, respectively. Suppose $P_x \notin P_y$. The first step is to decompose $P_x$ such that $P_x = P_x \cap P_y$, $P_x = P_x \cap P_y \cap P_y$, and $P_x = P_x \setminus P_y$. Obviously, $P_y$ does not overlap with $P_y$ or $P_x$. Proposition 2 is then used to determine the new choice relations involving $P_x$ and $P_y$ in the context of $D_1$, which replace $P_x \leftrightarrow D_1 Q_a$ involving the overlapping header choice $P_x$. As a result, $P_x$ can be replaced by $P_y$ and $P_x$.

We have two possible scenarios: $P_y$ may or may not be a subset of $P_x$. If $P_y \subset P_x$, then 1) $P_x$ is just $P_y$, and 2) we need to apply Proposition 1 to deduce the choice relation involving $P_y$ in the context of $D_1 \cup D_2$ if necessary. (When $Q = R$ and $Q_a = R_a$, Proposition 1 can be applied to $P_x \leftrightarrow D_1 R_a$ and the newly determined $P_x \leftrightarrow D_1 Q_a$). Otherwise, there exists some $P_y = P_y \setminus P_x \neq 0$. The next step is to decompose $P_x$ such that $P_y = P_y \cup P_y \cup P_y$. Thereafter, the pair of overlapping header choices $P_x$ and $P_y$ are replaced by new header choices $P_x$, $P_x$, and $P_y$, which do not overlap with one another. Proposition 2 is then applied again to determine the new choice relations involving $P_x$ and $P_y$ in the context of $D_2$. Note that there may be two new choice relations involving $P_x$, one in the context of $D_1$ and one in the context of $D_2$. Proposition 1 can then be used to deduce the choice relation involving $P_x$ in the context of $D_1 \cup D_2$ if necessary.

Similarly, the case of overlapping trailer choices can be resolved by means of the following proposition, which is a dual of Proposition 2.

**Proposition 3 (Refinement of Overlapping Trailer Choices).** Let $P$, $Q$, and $R$ be categories and $P_x$, $R_y$, $Q_a$, and $Q_b$ be choices such that 1) $P \neq Q$ and $R \neq Q$, 2) $P_x \leftrightarrow D_1 Q_a$ and $R_y \leftrightarrow D_2 Q_b$ for two distinct sets $D_1$ and $D_2$ of specification components, and 3) $Q_a$ and $Q_b$ are distinct and overlapping and, hence, $Q_a \cap Q_b \neq 0$ and $(Q_a \notin Q_b$ or $Q_b \notin Q_a$). Without loss of generality, suppose $Q_a \notin Q_b$. Let $Q_c = Q_a \cap Q_b$ and $Q_{ad} = Q_a \setminus Q_b$. We have a) If $P_x \subset D_1 Q_a$, then $(P_x \subset D_1 Q_a$ and $P_x \cup D_1 Q_a$, $(P_x \setminus D_1 Q_a$ and $P_x \setminus D_1 Q_a$), or $(P_x \setminus D_1 Q_a$ and $P_x \setminus D_1 Q_a$). b) If $P_x \setminus D_1 Q_a$, then $(P_x \setminus D_1 Q_a$ and $P_x \setminus D_1 Q_a$, $(P_x \setminus D_1 Q_a$ and $P_x \setminus D_1 Q_a$), or $(P_x \setminus D_1 Q_a$ and $P_x \setminus D_1 Q_a$). c) If $P_x \setminus D_1 Q_a$, then $P_x \setminus D_1 Q_a$, or $P_x \setminus D_1 Q_a$.

Following the same argument for applying Propositions 1 and 2 to resolve the problem of overlapping trailer choices, Propositions 1 and 3 can be similarly applied to resolve the problem of overlapping header choices.

To explain how to apply Propositions 2 and 3 (and possibly Proposition 1) iteratively to refine overlapping choices and their relations, consider a pair of choice relations involving overlapping choices. Let $D_1$ and $D_2$ be two different sets of specification components. Suppose we identify a pair of choices $P_x$ and $Q_a$ (where $P \neq Q$) from $D_1$ and define their relation $P_x \leftrightarrow D_1 Q_a$, and identify another pair of choices $R_y$ and $Q_b$ (where $R \neq W$) from $D_2$ and define their relation $R_y \leftrightarrow D_2 W_b$. Table 4 lists all the possible scenarios of $P_x \leftrightarrow D_1 Q_a$ and $R_y \leftrightarrow D_2 W_b$ that involve overlapping choices. The last column shows the proposition(s) to be applied for each scenario.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Overlapping Situation</th>
<th>Propositions to Apply</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, P_y \subseteq P_x$</td>
</tr>
<tr>
<td>2</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, P_y \subseteq P_x$</td>
</tr>
<tr>
<td>3</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$Q_a \cap Q_b \neq \emptyset, Q_a \subseteq Q_b, Q_b \subseteq Q_a$</td>
</tr>
<tr>
<td>4</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$Q_a \subseteq Q_b$</td>
</tr>
<tr>
<td>5</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, P_y \subseteq P_x, Q_a \cap Q_b = \emptyset$</td>
</tr>
<tr>
<td>6</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, P_y \subseteq P_x, Q_a \cap Q_b = \emptyset$</td>
</tr>
<tr>
<td>7</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$P_x \cap P_y = \emptyset, Q_a \cap Q_b \neq \emptyset, Q_a \subseteq Q_b, Q_b \subseteq Q_a$</td>
</tr>
<tr>
<td>8</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_a$</td>
<td>$P_x \cap P_y = \emptyset, Q_a \subseteq Q_b$</td>
</tr>
<tr>
<td>9</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, P_y \subseteq P_x, Q_a \cap Q_b \neq \emptyset, Q_a \subseteq Q_b, Q_b \subseteq Q_a$</td>
</tr>
<tr>
<td>10</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, P_y \subseteq P_x, Q_a \subseteq Q_b$</td>
</tr>
<tr>
<td>11</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$</td>
<td>$P_x \cap P_y \neq \emptyset, P_x \subseteq P_y, Q_a \cap Q_b \neq \emptyset, Q_a \subseteq Q_b, Q_b \subseteq Q_a$</td>
</tr>
<tr>
<td>12</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$</td>
<td>$P_x \cap P_y \neq \emptyset, Q_a \subseteq Q_b \text{ or } Q_b \subseteq Q_a$</td>
</tr>
<tr>
<td>13</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 W_b$ &amp; $N/A$ &amp; $Q_a \cap Q_b \neq \emptyset, Q_a \subseteq Q_b, Q_b \subseteq Q_a$</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$ &amp; $N/A$ &amp; $P_x \subset P_y$</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$ &amp; $N/A$ &amp; $Q_a \subset Q_b$</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>$P_x \leftrightarrow D_1 Q_a, P_y \leftrightarrow D_2 Q_b$ &amp; $N/A$ &amp; $Q_a \subset Q_b$</td>
<td>3</td>
</tr>
</tbody>
</table>

* $Q$ and $W$ are distinct categories

$P$ and $R$ are distinct categories
Similarly, if \( Q = W \), \( Q_a \) may be identical to \( W_b \) or may overlap with it. Given \( P_x \rightarrow_{D^1} Q_a \) and \( R_y \rightarrow_{D^1} W_b \) with overlapping choices, we have three cases:

1. \( P_x \) and \( R_y \) are in the same category but \( Q_a \) and \( W_b \) are in two different categories (that is, \( P = R \) and \( Q \neq W \)). In this case, \( P_x \) and \( R_y \rightarrow_{(P_y)} \) must be overlapping. There are two possible types of overlapping for \( P_x \) and \( R_y \rightarrow_{(P_y)} \), corresponding to scenarios 13 and 14.

2. \( Q_a \) and \( W_b \) are in the same category but \( P_x \) and \( R_y \) are in two different categories (that is, \( Q = W \) and \( P \neq R \)). In this case, \( Q_a \) and \( W_b \rightarrow_{(P_x)} \) must be overlapping. There are two possible types of overlapping for \( Q_a \) and \( W_b \rightarrow_{(P_x)} \), corresponding to scenarios 15 and 16.

3. \( P_x \) and \( R_y \) are in one category while \( Q_a \) and \( W_b \) are in another category (that is, \( P = R \) and \( Q = W \)). We have the following two cases:

3.1. Either \( P_x = R_y \) or \( Q_a = W_b \). Note that \( (P_x = R_y \) and \( Q_a = W_b) \) is not possible. When \( P_x = R_y \), \( Q_a \) and \( W_b \) must be overlapping. There are two possible types of overlapping, corresponding to scenarios 3 and 4. Similarly, when \( Q_a = W_b \), scenarios 1 and 2 apply.

3.2. \( P_x \neq R_y \) and \( Q_a \neq W_b \). We have the following three subcases: i) \( P_x \) and \( R_y \) are overlapping while \( Q_a \) and \( W_b \) are not. Since there are two possible types of overlapping for \( P_x \) and \( R_y \), we have scenarios 5 and 6. ii) \( Q_a \) and \( W_b \) are overlapping while \( P_x \) and \( R_y \) are not. Since there are two possible types of overlapping for \( Q_a \) and \( W_b \), we have scenarios 7 and 8. iii) \( P_x \) and \( R_y \) are overlapping, and so are \( Q_a \) and \( W_b \). Let us consider \( P_x \) and \( R_y \) first. We have two possible types of overlapping: \( (P_x \cap R_y \neq \emptyset, P_x \not\subseteq R_y, \text{and} R_y \not\subseteq P_x) \) and \( (P_x \not\subset R_y) \). In the context of a specified relation between \( P_x \) and \( R_y \), we need to consider all three possible overlapping situations for \( Q_a \) and \( W_b \). When \( (P_x \cap R_y \neq \emptyset, P_x \not\subseteq R_y, \text{and} R_y \not\subseteq P_x) \), we have scenarios 9 and 10. Scenario 9 covers the situation \((Q_a \subseteq W_b \neq \emptyset, Q_a \not\subseteq W_b, \text{and} W_b \not\subseteq Q_a)\) while scenario 10 covers the remaining two situations \((Q_a \subseteq W_b \neq \emptyset, Q_a \not\subseteq W_b, \text{and} W_b \not\subseteq Q_a)\) because \((W_b \subseteq Q_a)\) can be grouped under the same scenario as \((Q_a \subseteq W_b)\). When \( P_x \not\subset R_y \), we have scenarios 11 and 12. Scenario 11 covers the situation \((Q_a \not\subseteq W_b \neq \emptyset, Q_a \not\subseteq W_b, \text{and} W_b \not\subseteq Q_a)\), and scenario 12 covers the remaining two situations \((Q_a \subseteq W_b \neq \emptyset, Q_a \not\subseteq W_b, \text{and} W_b \not\subseteq Q_a)\).

We have chosen scenario 9, which is one of the most difficult cases, for illustration below. Other scenarios can be handled similarly.

**Scenario 9.** Suppose \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^2} Q_b \) such that 1) \( P \) and \( Q \) are distinct categories; 2) \( P_y \not\subseteq Q_a \); 3) \( Q_a \not\subseteq Q_b \); 4) \( Q_a \not\subseteq Q_b \); and 5) \( Q_b \not\subseteq Q_a \). Let \( P_x = P_y \subseteq P_y \). We have two overlapping header choices and overlapping trailer choices. We need to deduce or define new choice relations for \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^1} Q_b \) as follows:

1. The aim of this step is to replace choice relations with an overlapping header choice \( P_x \) or \( P_y \) by new relations with a nonoverlapping header choice \( P_x \) or \( P_y \). We apply Proposition 2 to determine the following new choice relations:
   \[ P_x \rightarrow_{D^1} Q_a \] and \( P_y \rightarrow_{D^1} Q_b \), which replace \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^1} Q_b \).

2. This step aims to replace the choice relations determined in step 1, which involve a nonoverlapping header choice \( P_x \) or \( P_y \) and an overlapping trailer choice \( Q_a \) or \( Q_b \), by new relations with a nonoverlapping trailer choice \( Q_a \) or \( Q_b \). Here, we apply Proposition 3 to determine the following new choice relations:
   a. \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^1} Q_b \), which replace \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^1} Q_b \).
   b. \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^1} Q_b \), which replace \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^1} Q_b \).
   c. \( P_x \rightarrow_{D^2} Q_a \) and \( P_y \rightarrow_{D^2} Q_b \), which replace \( P_x \rightarrow_{D^2} Q_a \) and \( P_y \rightarrow_{D^2} Q_b \).
   d. \( P_x \rightarrow_{D^2} Q_a \) and \( P_y \rightarrow_{D^2} Q_b \), which replace \( P_x \rightarrow_{D^2} Q_a \) and \( P_y \rightarrow_{D^2} Q_b \).

3. This step applies Proposition 1 to deduce a new choice relation \( P_x \rightarrow_{D^1} Q_a \), and use it to replace \( P_x \rightarrow_{D^1} Q_a \) and \( P_y \rightarrow_{D^2} Q_a \) determined in steps 2.c and 2.d.

The above illustration shows that new choice relations, involving more “fine grain” choices, can be automatically deduced as far as possible by applying Propositions 1, 2, and 3. Even in some situations where automatic deductions are not possible, automatic consistency checking of manually defined choice relations can be provided. For example, in step 1a of scenario 9 involving the application of Proposition 2b when \( P_x \not\subseteq Q_a \), we know that the combination \( P_x \subseteq Q_a \) and \( P_y \subseteq Q_a \) is not possible. The features of automatic deductions and consistency checking greatly contribute to the effectiveness of determining choice relations.  

So far, we have illustrated how to iteratively apply Propositions 1, 2, and 3 to refine a pair of choice relations having overlapping choices. We now extend our refinement mechanism to handle more than two choice relations. In the
following algorithm, steps 2.b and 2.c refine choice relations with overlapping header choices and overlapping trailer choices, respectively, while step 3 stores the choice relations after refinement in the final choice relation table \( T \) and apply the construction algorithm provided by CHOC’LATE [7], [16] to complete the construction of \( T \).

Algorithm refinement to Refine Choice Relations Having Overlapping Choices

We follow the notation used in the integration algorithm. Given a linked list \( L \) with \( m \) elements \( L_1, L_2, \ldots, L_m \) (where \( m \geq 1 \)) and given \( m \) associated nonempty linked lists \( LL_1, LL_2, \ldots, LL_m \) (which are the output results of the integration algorithm for storing choice relations, each \( LL_i \) containing all the choice relations determined with respect to a nonempty subset \( D_i \) of \( S \)), perform the following steps:

1. Initialiation of Selection of Choice Relations

   a) Process \( LL_i \)
   
   Initialize \( E \) as an empty set. Then, for every choice relation in \( LL_i \), store it in \( E \).

   b) Each element of a nonempty \( E \) is a choice relation.

2. Refinement of Choice Relations with Overlapping Choices

   For every \( LL_i \) (\( i = 2, 3, \ldots, m \)), incrementally refine the choice relations with overlapping choices by repeating the following steps:

   a. For every choice relation in \( LL_i \), store it in \( E \).

   b. Refining Overlapping Header Choices and Their Choice Relations

      For every pair of choice relations \( P_x \rightarrow_D Q_a \) (where \( P \neq Q \) ) and \( P_y \rightarrow_D R_b \) (where \( P \neq R \) ) in \( E \) such that \( P_x \) overlaps with \( P_y \) and \( P_x \not\subseteq P_y \), do the following:

         b1. Apply Proposition 2 to refine the overlapping header choice \( P_x \) into new nonoverlapping header choices, of which new choice relations will need to be determined. Whenever possible, perform automatic deductions of new choice relations according to Proposition 2. Perform consistency checks for all the new, manually defined choice relations using the proposition. If any inconsistency is detected, alert the users about the problem and prompt them to undo the step immediately. Replace the processed \( P_x \rightarrow_D Q_a \) in \( E \) by the newly determined relations.

         b2. Repeat b1 above on the overlapping header choice \( P_y \), if necessary. “Apply this substep if \( P_y \not\subseteq P_x \)”.

         b3. Because of the newly determined relations in steps 2.b1-b2, \( E \) may contain choice relations with the same pairs of header and trailer choices but determined with respect to different subsets of \( S \). If this happens, apply Proposition 1 to integrate these relations together. Then, replace the processed choice relations by the newly deduced one in \( E \).

   c. Refining Overlapping Trailer Choices and Their Choice Relations

      Perform similar refinement tasks as in step 2.b above to refine overlapping trailer choices and their choice relations. During the refinement process, use Proposition 3 instead of Proposition 2. Also use Proposition 1, if applicable.

3. Construction of Choice Relation Table \( T \)

   a) Initialize \( T \) as an empty table. b) For the choice relations remaining in \( E \) (which do not involve any overlapping choices), store them in \( T \). c) Apply the choice relation table construction algorithm provided by CHOC’LATE [7], [16] (which includes the techniques for automatic deductions and consistency checking of nonoverlapping choices) to determine all the yet-to-be-defined relational operators in \( T \).

As discussed in the paragraph immediately after the integration algorithm, the maximum possible number of linked lists \( LL_i \) associated with \( L \) is \( (2^n - 1) \), where \( n \) is the number of specification components. It should be noted that the maximum number of choice relations stored in the linked lists associated with \( L \) is \( (k^2 - \sum_{j=1}^g |N(P_j)|^2) \), where \( g \) is the total number of categories, \( k \) is the total number of nonoverlapping choices across all categories, and \( N(P_j) \) is the total number of nonoverlapping choices in \( P_j \) after executing the refinement algorithm. Since each associated linked list must contain at least one choice relation, the maximum number of associated linked lists will be the minimum of \( (2^n - 1) \) and \( (k^2 - \sum_{j=1}^g |N(P_j)|^2) \). Step 1 of the refinement algorithm involves a one-off initialization of \( E \). Each of steps 2.b and 2.c involves picking up a choice relation from \( E \) and then comparing it with all the remaining choice relations in \( E \). Thus, the worst-case complexity of steps 2.b and 2.c is of the order \( r^2 \), where \( r \) is the total number of choice relations across all preliminary choice relation tables. Since the number of iterations in step 2 is bounded by the number of linked lists associated with \( L \), the maximum number of iterations of step 2 will not exceed \( \min(2^n - 1, k^2 - \sum_{j=1}^g |N(P_j)|^2) \). Furthermore, as discussed in [7], the computational complexity of step 3 is of the order \( r^2 \). Hence, the worst-case complexity of the algorithm is of the order \( r^2 \min(2^n - 1, k^2 - \sum_{j=1}^g |N(P_j)|^2) \).

Example 11 (Refining Choice Relations with Overlapping Choices). Refer to Example 10 again. Given Tables 2 and 3, after we have applied the integration algorithm and part of the refinement algorithm (after executing step 3.b), the partially constructed choice relation table is shown in Table 5. On close examination of Table 5, we have the following observations:

1. While \( Q_b \) (in Table 2) overlaps with \( Q_c \) and \( Q_d \) (in Table 3) before executing integration, none of the choice relations in Table 5 involves overlapping choices.

2. In Table 5, all the choice relations determined with respect to both \( C_1 \) and \( C_2 \) are automatically deduced in step 2.a1 of integration or in step 2.b or 2.c of

   6. Note that \( k^2 \) is the dimension of the choice relation table \( T \), and \( \sum_{j=1}^g |N(P_j)|^2 \) is the total number of choice relations \( \text{left} \rightarrow \text{right} \) in \( T \), where \( P_x \) and \( P_y \) are distinct and nonoverlapping choices. As explained before, by Definition 4 the relational operator for \( \text{left} \rightarrow \text{right} \) and \( \text{left} \rightarrow \text{left} \) must be “\( \cap \)” and “\( \subseteq \)”, respectively. Hence, these relations need not be stored in the linked lists associated with \( L \) but can be automatically deduced in step 3.c of refinement.
refinement. These choice relations involve categories \( P \) and \( Q \) because these two categories are identified from both \( C_1 \) and \( C_2 \).

3. In Table 5, all the choice relations with \( R_c \) or \( R_f \) as header or trailer choices (but not both) are determined with respect to \( \{ C_1 \} \) only. This is because \( R_c \) and \( R_f \) are not identified from \( C_2 \) initially and, hence, neither of these choices appears as a header or trailer choice in any choice relation in \( \tau_2 \) (shown in Table 3) before the execution of integration. Consequently, steps 2b and 2c of refinement will not deduce any choice relation (with respect to \( \{ C_1, C_2 \} \)) having \( R_c \) or \( R_f \) as its header or trailer choice. Similarly, since \( W_p \) and \( W_q \) are not identified from \( C_1 \) initially, all the choice relations with \( W_p \) or \( W_q \) as header or trailer choices (but not both) are determined with respect to \( \{ C_2 \} \) only.

4. All the choice relations in Table 5 can be considered to be determined with respect to the entire specification \( S \). This can be explained as follows: On completion of step 2 of refinement, all the choice relations determined with respect to \( \{ C_1, C_2 \} \) can be considered to be determined with respect to \( S \), because \( C_1 \) and \( C_2 \) together constitute \( S \). On the other hand, for all the choice relations determined with respect to \( \{ C_1 \} \) or \( \{ C_2 \} \) only, they are effectively the same as those relations determined with respect to \( S \). Consider, for instance, \( P_x \not\subseteq W_p \) in Table 5. As pointed out in observation 3 above, \( C_1 \) does not contain any information leading to the identification of category \( W \) and its associated choices. Thus, when the testers look into the choice relation \( P_x \mapsto W_p \), they can determine \( P_x \not\subseteq W_p \) only if the entire \( S \) is taken into account.

When step 3.c of refinement commences, by Definition 4, CHOC’LATE will first automatically assign the relational operators “\( \sqsubseteq \)” and “\( \sqsupseteq \)” to every choice relation \( Z_m \mapsto_D Z_n \) and \( Z_m \mapsto_D Z_n \) (where \( (Z = P, Q, R, or W) \) and \( (m, n = z, y, a, c, d, e, f, p, or q) \) respectively, in \( T_1 \)). Eight yet-to-be-defined choice relations, whose header and trailer choices belong to different categories, will remain after this process. All of them involve \( (R_p \ or \ R_f) \) and \( (W_p \ or \ W_q) \) as their header or trailer choices. These choice relations occur because of the fact that category \( R \) and its associated choices are identified from \( C_1 \) (but not \( C_2 \)), whereas category \( W \) and its associated choices are identified from \( C_2 \) (but not \( C_1 \)). Thus, \( \tau_1 \) and \( \tau_2 \) (shown in Tables 2 and 3) do not contain any choice relation \( R_i \mapsto (\{ C_1 \} \ W_j \or W_j \mapsto (\{ C_1 \} \ R_i \) before executing integration, where \( i = e \ or \ f, j = p \ or \ q, \) and \( k = 1 \ or \ 2 \)). All eight of these yet-to-be-defined relations will be determined in step 3.c of refinement by applying the table construction algorithm provided by CHOC’LATE [7], [16]. When defining choice relations, the testers may need additional information from sources other than the specification, such as end users and software designers.

The following example completes our first study involving the visitor administration system \( VISIT \) by presenting the results of applying the refinement algorithm:

**Example 12 (Processing Visitor Requests: Part 3).** Refer to Examples 8 and 9 again. Participant A applied the refinement algorithm to refine the 246 choice relations involving overlapping choices that remained after the execution of the integration algorithm. After step 2 of refinement, no overlapping choice remained. After step 3 of refinement, a choice relation table (denoted by \( T_{REQUEST} \)) was constructed with a dimension of \( 30 \times 30 \) for the function “Process Visitor Requests” of \( VISIT \). \( T_{REQUEST} \), with all of its 900 choice relations completely determined, could now be further processed by CHOC’LATE to generate a set of complete test frames for testing the function “Process Visitor Request.” A summary of the results of our first study is shown in the first line of Table 7.

### 6 Case Studies

We evaluated the effectiveness of DESSERT using three commercial specifications. They include the specification \( S_{VISIT} \) discussed earlier, a specification \( S_{CHECK-IN} \) for a passenger self-service check-in system \( CHECK-IN \), and a specification \( S_{CAR} \) for a company car and expense claim system \( CAR \) in a multi-national trading firm. \( CHECK-IN \) allows selected groups of passengers (such as privileged club members of \( AIR \)) to perform self-service check-in via the Internet or at any kiosks convenient to them. On the other hand, \( CAR \) assists the regional sales directors of the firm in determining the fee to be charged to each sales manager for any excessive mileage in the use of the company car, and in processing
reimbursement requests regarding various kinds of expenses such as airfare, hotel accommodation, meals, and phone calls. We will refer to the studies involving VISIT, CHECK-IN, and CAR as studies 1, 2, and 3, respectively. Studies 2 and 3 were conducted in a manner similar to study 1. Participant A was recruited again to conduct study 3, whereas another participant (also with a postgraduate degree in IT) was recruited for study 2.

Tables 6, 7, and 8 highlight the experimental data and results of these three studies. Table 6 shows that problems 1 and 2 also occurred in CHECK-IN and CAR, just like VISIT. It can be computed from Table 6 that the percentages of choice relations with problem 1 in relation to the total numbers of choice relations in the choice relation tables for VISIT, CHECK-IN, and CAR were 9.8 percent (= 88/900 x 100%), 0.0 percent (= 0/1849 x 100%), and 4.5 percent (= 20/441 x 100%), respectively (see the last column of Table 7). It can also be calculated from Table 6 that the percentages of choice relations with problem 2 in relation to the total numbers of choice relations in the choice relation tables for VISIT, CHECK-IN, and CAR were 27.3 percent (= 246/900 x 100%), 21.3 percent (= 394/1849 x 100%), and 43.5 percent (= 192/441 x 100%), respectively. Furthermore, the table shows that problems 1 and 2 escalated during the table consolidation process. For example, for the three studies, respectively, the numbers of new choice relations with problem 1 have occurred for VISIT and CHECK-IN during the execution of refinement (see the last column of Table 8).

7. Note that new choice relations with problem 1 have occurred for VISIT and CHECK-IN during the execution of refinement (see the fourth column of Table 8).
three studies. Also note the two rightmost columns of the table, which show that there were a total of 112 (= 32 + 40 + 40) deduced choice relations and a total of 228 (= 36 + 152 + 40) manually defined choice relations, that is, on average, 33 percent (= 112/340 × 100%) of the choice relations were automatically deduced by Proposition 2 or 3. Even when manual definitions of choice relations were needed in the refinement algorithm, they were supported by the consistency check mechanism that ensured the correctness of the defined relations whenever appropriate.

There are two limitations in our current studies. First, they only involved two software practitioners and three specifications. It would be better if more human subjects and specifications were involved. We must point out, however, that obtaining large and complex specifications from the industry is difficult because most companies are hesitant to release them for external use. In any case, our studies still provide a convincing demonstration of the effectiveness of DESSERT for such specifications. After all, our work is not an attempt to test hypotheses or causal relationships among variables and, hence, a large number of subjects and specifications is not a must. Second, it would be better if a comparison between DESSERT and other similar methodologies were made. Nevertheless, as pointed out in Section 2.2, we are not aware of any category/choice methods that explicitly address large and complex specifications. Thus, such a comparison is not applicable. This issue, in fact, clearly demonstrates the novelty and contribution of DESSERT.

7 Summary and Conclusion

In this paper, we have introduced a Divide-and-conquer methodology for identifying categories, choice, and choice relations for test case generation, abbreviated as DESSERT. The purpose is to alleviate a major problem of CHOC’LATE (and also several other related methodologies) in generating test cases for large and complex specifications, or more specifically, the difficulty in consolidating preliminary choice relation tables into a final table for subsequent test case generation. The divide-and-conquer approach of DESSERT should appeal to software practitioners because the methodology can be effectively applied to large commercial software systems whose specifications are often complex and contain many different components. We have discussed in detail how to

1. consolidate preliminary choice relation tables constructed from different specification components into a choice relation table $T^*$,
2. correct inconsistent relations for the same pair of choices due to partial information from different specification components,
3. refine choice relations involving overlapping choices defined from different specification components, and
4. apply consistency checks and automatic deductions for choice relations involving overlapping choices in the construction of $T^*$.

The theoretical backbone and techniques underlying these procedures have also been discussed. We have also conducted case studies to evaluate DESSERT using three real-life commercial specifications that contain several different specification components. The results have confirmed that DESSERT provides a systematic approach to construct a $T$ in which all the choices are nonoverlapping and all the choice relations are properly determined. Once a $T$ is constructed, it can then be processed by CHOC’LATE for test case generation. As such, DESSERT contributes to the industry by alleviating the difficulties and improving the effectiveness of testing.

Acknowledgments

The work described in this paper was partially supported by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 123206 and 717308), a Departmental General Research Fund of The Hong Kong Polytechnic University (project no. 1-ZVZ2H), and a grant of the Australian Research Council (ARC DP0880295).

References


Tsong Yueh Chen received the BSc and MPhil degrees from The University of Hong Kong, the MSc degree and DIC from Imperial College London, and the PhD degree from The University of Melbourne. He is currently a chair professor of software engineering and the leader of the Software Analysis and Testing Group at Swinburne University of Technology, Australia. Prior to joining Swinburne, he taught at The University of Hong Kong and The University of Melbourne. His research interests include software testing, debugging, software maintenance, and software design. He is a member of the IEEE.

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 associate professor in the School of Accounting and Finance at The Hong Kong Polytechnic University, where he teaches courses on information systems and systems analysis and design. He served on the editorial board of the Information Systems Control Journal. His research interests include software testing, requirements engineering and inspection, information systems and enterprise systems, information systems audit and control, electronic commerce, and computers in education. He is a member of the IEEE and the ACM.

Sau-Fun Tang received the Master of Business (Information Technology) degree from the Royal Melbourne Institute of Technology and the PhD degree in software engineering from Swinburne University of Technology in Australia. She is currently affiliated with the Software Analysis and Testing Group at Swinburne. She was an instructor in the Department of Finance and Decision Sciences at Hong Kong Baptist University and a lecturer in the School of Accounting and Finance at The Hong Kong Polytechnic University. Her research interests include software testing, management information systems, electronic commerce, and computers in education. She is a member of the IEEE.

T.H. Tse received the PhD degree from the London School of Economics and was a visiting fellow at the University of Oxford. He is a professor in computer science at The University of Hong Kong. His current research interest is in program testing, debugging, and analysis. He is the steering committee chair of QSIC and an editorial board member of Software Testing, Verification and Reliability, the Journal of Systems and Software, Software: Practice and Experience, and the Journal of Universal Computer Science. He 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, and a fellow of the Hong Kong Institution of Engineers. He was awarded an MBE by the Queen of the United Kingdom. He is a senior member of the IEEE.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.