To appear in *Proceedings of the 5th International Conference on Quality Software* (*QSIC 2005*),
IEEE Computer Society Press, Los Alamitos, California (2005)

# Identification of Categories and Choices in Activity Diagrams [*] [†]

T. Y. Chen
*Swinburne University of Technology*
*tchen@ict.swin.edu.au*

Pak-Lok Poon
*The Hong Kong Polytechnic University*
*afplpoon@inet.polyu.edu.hk*

Sau-Fun Tang
*Swinburne University of Technology*
*sftang@hkbu.edu.hk*

T. H. Tse [‡]
*The University of Hong Kong*
*thtse@cs.hku.hk*

## Abstract

*The choice relation framework* (CHOC'LATE) *provides a systematic skeleton for constructing test cases from specifications. An early stage of the framework is to identify a set of categories and choices from the specification, which is not a trivial task when this document is largely informal and complex. Despite the difficulty, the identification task is very important because the quality of the identified categories and choices will affect the comprehensiveness of the test cases and, hence, the chance of revealing software faults. This paper alleviates the problem by introducing a technique for identifying categories and choices from the activity diagrams in the specification. This technique also helps determine the relations between some pair of choices in the choice relation table — an essential step of* CHOC'LATE *for the subsequent generation of test cases.*

**Keywords:** *Activity diagrams, category-partition method, choice relation framework, classification-tree method, specification-based testing, test frame*

## 1. Introduction

Program testing encompasses a range of tasks in that sequence: (a) establishing test objectives, (b) generating a *test suite* (the set of test cases used for testing), (c) executing the program with every test case in the generated test suite, and (d) examining the test results. Among these tasks, test suite generation is very important [9]. This is because the comprehensiveness of the test suite will affect the scope and, hence, the effectiveness of testing.

In general, there are two approaches for test suite generation: the *white-box* and *black-box* approaches. The white-box approach generates a test suite according to the information derived from the source code of the program under test. White-box testing typically requires the coverage of certain aspects of the program structures. Control flow testing [13], data flow testing [5], and domain testing [12] are some well-known examples. On the other hand, the black-box approach generates a test suite without the knowledge of the internal structure of the program. In most cases, the generation process is based on a specification [1] that exists in a spectrum of forms. At one extreme of the spectrum is the completely *informal* specification primarily written in natural language. At the other extreme is the completely *formal* specification written in a mathematical notation. In



[†] This research is supported in part by grants of the Research Grants Council of Hong Kong (Project Nos. PolyU 5177/04E and HKU 7029/01E), an Australian Research Council Discovery Grant (Project No. DP 0345147), and a grant of The University of Hong Kong.

[‡] All correspondence should be addressed to T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: thtse@cs.hku.hk.

[1] Some researchers and practitioners also use the term "model" to refer to this abstract description of the software under development.

general, the format of a specification may lie somewhere between these two extremes.

Because of the rigorous nature, formal specifications are relatively easier for test suite generation. These specifications, however, are not as popular as they should be, mainly because more software developers are not familiar with the mathematical concepts involved and find the techniques difficult to understand and use. Thus, some software developers turn to generating test suites from informal specifications.

Among the various test suite generation methods that can be applied to informal specifications, the category-partition method (CPM) [10] has received much attention. Later, Chen et al. [4] have developed a **CHO**i**C**e re**LAT**ion fram**E**work, abbreviated as CHOC'LATE, to improve on the effectiveness of CPM.

In CHOC'LATE (as well as CPM), an early step is to identify a set of categories and choices from the specification. A *category* is defined as "a major property or characteristic of a parameter or an environment condition of the software that affects its execution behavior".[2] An example is the category [GPA Score ($S$)] in an undergraduate's award classification system. The possible values associated with each category are then 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 [10]. Examples of choices are |GPA Score ($S$): $0.0 \leq S < 2.0$|, |GPA Score ($S$): $2.0 \leq S < 2.5$|, |GPA Score ($S$): $2.5 \leq S < 3.0$|, |GPA Score ($S$): $3.0 \leq S < 3.5$|, and |GPA Score ($S$): $3.5 \leq S \leq 4.0$|.[3] These choices are defined as such because they determine whether a student is eligible to graduate, and if yes, what level of award (for example, first-class honor) a student will obtain. Note that a choice is considered as a set of its possible values. For example, |GPA Score ($S$): $0.0 \leq S < 2.0$| = $\{0.0,\ 0.1,\ 0.2,\ \ldots,\ 1.8,\ 1.9\}$. Additionally, given a category $[X]$, all its associated choices together should cover the entire input domain relevant to $[X]$.

After identifying all the major categories and choices, the software tester has to construct a choice relation table, which captures the constraint between every pair of choices. These constraints allow all valid combinations of choices to be generated as "test frames" and at the same time invalid combinations are suppressed as far as possible. An example of these constraints is that the choice |GPA Score ($S$): $3.5 \leq S \leq 4.0$| cannot be combined with the choice |Number of Years of Study: $\leq 1$| of the category [Number of Years of Study] to form part of any test frame. This is because only one year of study or less is insufficient for a student to attain this range of GPA score. Finally, test cases are generated from valid test frames.

Obviously, the quality of the identified categories and choices will eventually affect the comprehensiveness of the test suite and, hence, the effectiveness of revealing software faults. If, for example, a valid choice is missing, then any fault associated with this choice may not be detected. We observe, however, that there is no systematic methodology for identifying categories and choices for informal specifications. As a result, this identification process is often performed in an ad hoc manner and, hence, the quality of the resulting test suite may be in question. This problem inspires us to develop a systematic identification methodology for informal specifications. The methodology will also help determine the constraints between some pairs of choices in the choice relation table, from which test frames are generated.

The rest of this paper is organized as follows. Section 2 briefly discusses some previous work on the identification of categories and choices from specifications, and explains how such work relates to our identification methodology. Section 3 first outlines the major concept of activity diagrams, followed by some important concepts and definitions such as different types of choice relation and problematic category and choice, and then our identification technique in detail. Finally, Section 4 concludes the paper.

## 2. Previous work on category and choice identification

Using commercial specifications primarily of informal nature, Chen et al. [3] have conducted some empirical studies on the "ad hoc" identification of categories and choices. The primary objective of the studies is to investigate the common mistakes made by software testers in such an identification approach. Results of the studies have shown that missing categories and choices, and various types of problematic category and choice are likely to occur during an ad hoc identification process. Readers may refer to [2, 3] for details. The results have confirmed the great demand for a systematic identification methodology for specifications which are largely informal in nature. As an interim solution, Chen et al. [3] have developed a checklist to help software testers avoid and detect such mistakes.

In addition, Grochtmann and Grimm [6] have attempted to use artificial intelligence techniques to generate categories and choices automatically, but without

[2] Parameters and environment conditions of the software are collectively known as *factors* in this paper.

[3] In this paper, categories are enclosed by square brackets [] and choices are enclosed by vertical bars | |. Additionally, the notation |$X$ : $x$| denotes a choice |$x$| in the category $[X]$.

much success.[4] Eventually, Grochtmann and Grimm have concluded that identifying categories and choices is a "creative" process that probably can never be done entirely automatically. (We concur with their conclusion about the difficulty in fully automating the identification process.) Grochtmann and Grimm have then shifted their attention to the identification process based on formal specifications. Other researchers, such as Amla and Ammann [1] and Hierons et al. [7], have also conducted some work in that direction. Our work discussed in this paper, however, takes a different direction because we aim to develop an identification methodology for informal specifications.

## 3. Our identification methodology

Unlike formal specifications which are written in rigorous specification languages such as Z and Boolean predicates [8, 11], informal specifications are often expressed in many different styles and formats, and contain a large variety of components. Examples of these specification components are narrative descriptions, data flow diagrams, entity-relationship diagrams, system flowcharts, decision tables, use cases, activity diagrams, and statechart diagrams.

In view of the various possible combinations of these components in an informal specification, we adopt the following approach when developing our identification methodology:

(1) To determine the major and common components that exist in most informal specifications.

(2) To develop an identification technique for each major and common specification component.

This approach will make our identification methodology applicable to a large variety of informal specifications.

Limited by the space of this paper, here we focus only on the identification of categories and choices based on activity diagrams, which are a common component in a UML (Unified Modeling Language) specification.

### 3.1. Overview of activity diagrams

In general, the activity diagram, denoted by $\mathcal{D}$, supplements the use case (almost an essential specification component), by providing a graphical representation of the flow of interaction within a specific scenario. In its basic form, a $\mathcal{D}$ is a simple and intuitive illustration of what happens in a control flow (or workflow), what activities can be done in parallel, and whether there are alternative paths through the control flow. As such, $\mathcal{D}$'s can be used to model everything from a high-level business workflow that involves many different use cases, to the details of an individual use case, all the way down to the specific details of an individual method.

Often, a complete control flow description in a $\mathcal{D}$ will have a basic flow, and one or more alternative flows. This control flow has a structure that can be defined textually using statements such as IF-THEN-ELSE. For a simple control flow with a simple structure these textual definitions may be fairly sufficient, but in the situation of more complex structures, a $\mathcal{D}$ helps us clarify and make more apparent what the control flow is.

A $\mathcal{D}$ starts with a solid circle, representing the *initial activity*. An arrow shows the control flow of activities, which are represented by rounded rectangles labeled for the activities performed. An asterisk on an arrow indicates that the control flow is iterated. The end of a control flow is indicated by a "bull's eye", known as a *final activity*. A single path of execution through a $\mathcal{D}$ is called a *thread*.

A diamond represents a *decision point*. Conditions for each arrow out of a decision point (known as an *alternative thread*) are enclosed in brackets, and they are called *guard conditions*. The diamond icon can also be used to show whether the alternative threads merge again. A solid thick bar is called a *synchronization bar*. Multiple arrows out of a synchronization bar indicate activities that can be performed in parallel. Multiple arrows into a synchronization bar indicate activities that must all be completed before the next process can begin. Refer to Figure 1 for a sample activity diagram $\mathcal{D}_{\mathsf{award}}$.

Intuitively, the decision points and guard conditions in a $\mathcal{D}$ indicate where and how the software system behaves differently. This characteristic makes a $\mathcal{D}$ a very useful source of deriving information to identify categories and choices, which are then processed by CHOC'LATE for test case generation.

### 3.2. Background concepts and definitions

First, we present a few important concepts and definitions, introduced in [3], and a new concept "complete thread", which are essential for understanding the identification technique in this paper.

**Definition 1 (Complete and Incomplete Test Frames)** *A **test frame** B is a set of choices. B is **complete** if, whenever a single element is selected from every choice in B, a test case is formed. Otherwise, B is **incomplete**.* ■

[4] In [6], categories and choices are known as "classifications" and "classes", respectively.

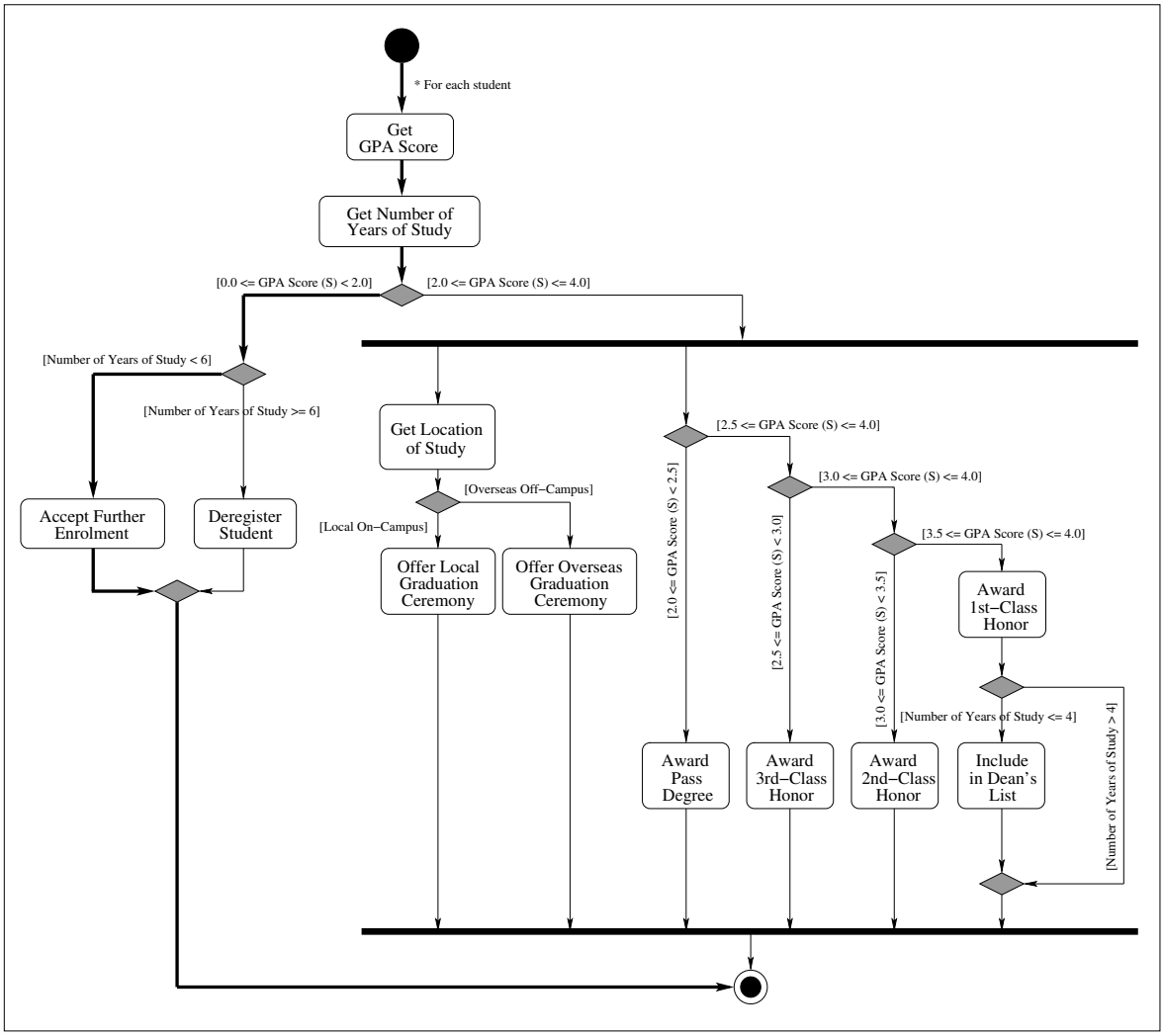


**Figure 1. An activity diagram $\mathcal{D}_{\text{award}}$**

**Example 1 (Complete and Incomplete Test Frames)** Consider a program $P$ that reads an input file $F$ containing two integers $m$ and $n$, and outputs the value of $(1/\sqrt{m-n})$. Here, [Status of $F$] and $[m-n]$ are two possible categories identified with respect to an environment condition and a parameter, respectively, that affect the execution behavior of $P$. The category [Status of $F$] has three associated choices: |Status of $F$: Does Not Exist|, |Status of $F$: Exists but Empty|, and |Status of $F$: Exists and Non-Empty|. On the other hand, the category $[m-n]$ has three associated choices: $|m-n:<0|$, $|m-n:=0|$, and $|m-n:>0|$. These three choices correspond to an undefined result of $(1/\sqrt{m-n})$ involving taking the square root of a negative number, an undefined result involving division by zero, and a well-defined result, respectively. With these two categories and their associated choices, there are altogether five complete test frames as follows:

- $B_1 = \{|\text{Status of } F\text{: Does Not Exist}|\}$,
- $B_2 = \{|\text{Status of } F\text{: Exists but Empty}|\}$,
- $B_3 = \{|\text{Status of } F\text{: Exists and Non-Empty}|, |m-n: < 0|\}$,
- $B_4 = \{|\text{Status of } F\text{: Exists and Non-Empty}|, |m-n: = 0|\}$, and
- $B_5 = \{|\text{Status of } F\text{: Exists and Non-Empty}|, |m-n: > 0|\}$.

Consider $B_5$. A possible test case generated from it is (Status of $F$ = Exists and Non-Empty, $m-n=5$). On the other hand, $\{|\text{Status of } F\text{: Exists and Non-Empty}|\}$

is an incomplete test frame because we need additional information about the value of $(m-n)$ in order to generate a test case for $P$. ■

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

**Example 2 (Set of Complete Test Frames Related to a Choice)** Refer to program $P$ in Example 1. Here, $TF(|\text{Status of } F\text{: Does Not Exist}|) = \{B_1\}$, whereas $TF(|\text{Status of } F\text{: Exists and Non-Empty}|) = \{B_3, B_4, B_5\}$. Furthermore, both |Status of $F$: Does Not Exist| and |Status of $F$: Exists and Non-Empty| are valid choices, because $TF(|\text{Status of } F\text{: Does Not Exist}|)$ and $TF(|\text{Status of } F\text{: Exists and Non-Empty}|)$ are nonempty. ■

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

(1) $|X : x|$ *is **fully embedded** in* $|Y : y|$ *(denoted by* $|X : x| \sqsubset |Y : y|$*) if and only if* $TF(|X : x|) \subseteq TF(|Y : y|)$.

(2) $|X : x|$ *is **partially embedded** in* $|Y : y|$ *(denoted by* $|X : x| \boxminus |Y : y|$*) if and only if* $TF(|X : x|) \not\subseteq TF(|Y : y|)$ *and* $TF(|X : x|) \cap TF(|Y : y|) \neq \emptyset$.

(3) $|X : x|$ *is **not embedded** in* $|Y : y|$ *(denoted by* $|X : x| \not\sqsubset |Y : y|$*) if and only if* $TF(|X : x|) \cap TF(|Y : y|) = \emptyset$. ■

Because the three types of choice relations in Definition 3 are exhaustive and mutually exclusive, $|X : x| \mapsto |Y : y|$ can be uniquely determined. Additionally, immediately from the definition, the relational operator for $|X : x| \mapsto |X : x|$ is "$\sqsubset$" and the relational operator for $|X : x_1| \mapsto |X : x_2|$ is "$\not\sqsubset$" if $|X : x_1| \neq |X : x_2|$.

**Example 3 (Relation between Two Choices)** Refer to Example 1. The following lists three pairs of choices and their corresponding choice relations:

(1) $|m - n := 0| \sqsubset$ |Status of $F$: Exists and Non-Empty|: This is because, for every complete test frame $B$ containing $|m-n := 0|$ (that is, $B_4$), $B$ also contains |Status of $F$: Exists and Non-Empty|.

(2) |Status of $F$: Exists and Non-Empty| $\boxminus$ $|m-n :< 0|$: This is because,

- For some complete test frame $B$ containing |Status of $F$: Exists and Non-Empty| (that is, $B_3$), $B$ also contains $|m-n :< 0|$, and
- For some complete test frame $B'$ containing |Status of $F$: Exists and Non-Empty| (that is, $B_4$ and $B_5$), $B'$ does not contain $|m-n :< 0|$.

(3) |Status of $F$: Exists but Empty| $\not\sqsubset$ $|m-n := 0|$: This is because,

- For every complete test frame $B$ containing |Status of $F$: Exists but Empty| (that is, $B_2$), $B$ does not contain $|m-n := 0|$, and
- For every complete test frame $B'$ containing $|m-n := 0|$ (that is, $B_4$), $B'$ does not contain |Status of $F$: Exists but Empty|. ■

In CHOC'LATE [4], after identifying categories and their associated choices from the specification, the next step is to construct a *choice relation table* $\mathcal{T}$, which captures the relation between every pair of choices. Note that a choice relation $|X : x| \mapsto |Y : y|$ essentially corresponds to a constraint between the choices $|X : x|$ and $|Y : y|$. These choice relations then form the basis for the subsequent generation of complete test frames using the algorithms in CHOC'LATE. Readers may refer to [4] for more details.

In their studies [3], Chen et al. have observed some common mistakes made by software testers when categories and choices are identified from informal specifications in an "ad hoc" manner. Two examples of these mistakes are given and expressed in Definitions 4 and 5 below.

**Definition 4 (Missing Choice)** *Given a category $[X]$, and all the associated valid choices $|X : x_1|$, $|X : x_2|$, ..., $|X : x_n|$ in $[X]$, if there exist some other valid choice $|X : x|$ yet to be identified and some value $v \in |X : x|$ such that $v \notin |X : x_i|$, for every $1 \leq i \leq n$, then $|X : x|$ is a **missing choice**. In this case, we also say that $[X]$ is a **category with a missing choice**.* ■

**Example 4 (Missing Choice)** Refer to Example 1. Suppose the category [Status of $F$] is identified with only two associated choices, namely |Status of $F$: Does Not Exist| and |Status of $F$: Exists but Empty|, as if |Status of $F$: Exists and Non-Empty| has not been identified. In this case, |Status of $F$: Exists and Non-Empty| is a missing choice. Accordingly, [Status of $F$] is a category with a missing choice. ■

**Definition 5 (Overlapping Choices)** *Given a category $[X]$, two distinct valid choices $|X : x_1|$ and $|X : x_2|$ are said to be **overlapping** if $|X : x_1| \cap |X : x_2| \neq \emptyset$. In this case, $[X]$ is a **category with overlapping choices**.* ■

**Example 5 (Overlapping Choices)** Refer to Example 1. Suppose the category $[m-n]$ is now identified with three associated choices: $|m-n:<0|$, $|m-n:=0|$, and $|m-n:\geq 0|$. In this case, $|m-n:=0|$ and $|m-n:\geq 0|$ are overlapping choices because the element $(m-n=0)$ exists in both choices. Furthermore, $[m-n]$ is a category with overlapping choices. ■

**Definition 6 (Complete Thread)** *In a $\mathcal{D}$, a "single" path of execution is called a **thread**, and it is said to be **complete** if and only if it starts with the initial activity and ends with a final activity.* ■

**Example 6 (Complete Thread)** Refer to $\mathcal{D}_{\mathsf{award}}$ in Figure 1. The leftmost path indicated by a dark line represents a thread. Because this thread starts with the initial activity and ends with the final activity, it is also a complete thread. ■

### 3.3. Category and choice identification in activity diagrams

Having introduced the above concepts and definitions, we are now ready to present an algorithm for identifying categories and choices in activity diagrams. The algorithm also provides some information for the subsequent determination of choice relations.

---

**An Algorithm for Identifying Categories and Choices in Activity Diagrams:**

Given an activity diagram $\mathcal{D}$ which contains one or more guard conditions denoted by $gc_i$'s (where $i \geq 1$), with each $gc_i$ contains one or more subconditions, denoted by $sc_{(i,j)}$'s (where $j \geq 1$), which are separated from the others by the logical operators "AND" or "OR":

(1) Let:

- $\odot$ denote any arithmetic operator ("$+$", "$-$", "$\times$", and "$\div$"),
- $\sim$ denote any arithmetic relational operator ("$=$", "$\neq$", "$<$", "$>$", "$\leq$", and "$\geq$") [5],
- $\mathcal{V}$ denote $(v_1 \odot v_2 \odot \cdots \odot v_m)$; $v_i$ is any variable in $\mathcal{V}$ where $1 \leq i \leq m$, and
- $F(sc_{(i,j)})$ denote the factor(s) associated with $sc_{(i,j)}$.

Repeat this step (1) for every $sc_{(i,j)}$ in every $gc_i$ in $\mathcal{D}$:

If $sc_{(i,j)}$ contains only one single arithmetic relational operator, then:

(a) If $sc_{(i,j)}$ is not in the form "$\mathcal{V} \sim c$" (where $c$ is a constant), then re-express the subcondition in this format.

(b) Define $[\mathcal{V}]$ as a category if it does not exist. [6]

(c) Define $|\mathcal{V} :\sim c|$ as a choice in $[\mathcal{V}]$ if this choice does not exist.

else:

(d) Define the category $[F(sc_{(i,j)})]$ if it does not exist.

(e) Define the entire $sc_{(i,j)}$ as a choice in $[F(sc_{(i,j)})]$ if this choice does not exist.

(2) Repeat steps (2)(a) and (2)(b) below until there do not exist any overlapping choices $|X : x_i|$ and $|X : x_j|$ (note that $|X : x_i| \neq |X : x_j|$ because of steps (1)(c) and (1)(e) above):

(a) If $|X : x_i| \subsetneq |X : x_j|$, then:

(i) Delete $|X : x_j|$.

(ii) Define the choice $(|X : x_j| \setminus |X : x_i|)$ if it does not exist.

(b) If $(|X : x_i| \not\subset |X : x_j|)$ and $(|X : x_j| \not\subset |X : x_i|)$, then:

(i) Delete $|X : x_i|$ and $|X : x_j|$.

(ii) Define the following choices if they do not exist:

- $|X : x_i| \setminus |X : x_j|$
- $|X : x_j| \setminus |X : x_i|$
- $|X : x_i| \cap |X : x_j|$

(3) Let $E([X])$ denote the set of all possible elements associated with the category $[X]$, and $|X : x_1|$, $|X : x_2|, \ldots, |X : x_n|$ (where $n \geq 1$) denote all choices in $[X]$ identified after step (2). For every category $[X]$ with missing choices, define a new choice $|X : x|$ such that $E([X]) = |X : x| \cup |X : x_1| \cup |X : x_2| \cup \cdots \cup |X : x_n|$.

(4) Initialize a choice relation table $\mathcal{T}$ by assigning a "null" value to every $|X : x| \mapsto |Y : y|$ in $\mathcal{T}$.

(5) For every $|X : x_i| \mapsto |X : x_i|$ in $\mathcal{T}$, assign the (choice) relational operator "$\sqsubset$" to it.

(6) For every $|X : x_i| \mapsto |X : x_j|$ in $\mathcal{T}$ such that $|X : x_i| \neq |X : x_j|$, assign the (choice) relational operator "$\not\sqsubset$" to it.

[5] Do not confuse the arithmetic relational operators mentioned here with the (choice) relational operators ("$\sqsubset$", "$\sqsubseteq$", and "$\not\sqsubset$") introduced in Definition 3.

[6] Note that the same category may be associated with different $sc_{(i,j)}$'s in the same or different $gc_i$'s.

(7) Let $sc(|X : x|)$ denote the subcondition corresponding to the choice $|X : x|$. For every pair of $|X : x| \mapsto |Y : y|$ in $\mathcal{T}$ such that:

- $[X] \neq [Y]$,
- both $sc(|X : x|)$ and $sc(|Y : y|)$ appear in $\mathcal{D}$, and
- both $sc(|X : x|)$ and $sc(|Y : y|)$ are not associated with any parallel threads in $\mathcal{D}$,

then, use the following rules to determine the relevant choice relation for $|X : x| \mapsto |Y : y|$:

(a) Assign the (choice) relational operator "$\sqsubset$" to $|X : x| \mapsto |Y : y|$ if, for every complete thread $t$ associated with $sc(|X : x|)$, $t$ is also associated with $sc(|Y : y|)$.

(b) Assign the (choice) relational operator "$\mathbb{r}$" to $|X : x| \mapsto |Y : y|$ if:

(i) there exists some complete thread $t$ associated with $sc(|X : x|)$ such that, $t$ is also associated with $sc(|Y : y|)$; and

(ii) there exists some complete thread $t'$ associated with $sc(|X : x|)$ such that, $t'$ is not associated with $sc(|Y : y|)$.

(c) Assign the (choice) relational operator "$\not\sqsubset$" to $|X : x| \mapsto |Y : y|$ if:

(i) for every complete thread $t$ associated with $sc(|X : x|)$, $t$ is not associated with $sc(|Y : y|)$; and

(ii) for every complete thread $t'$ associated with $sc(|Y : y|)$, $t'$ is not associated with $sc(|X : x|)$.

---

There are two important characteristics in the above algorithm:

- It helps identify a set of categories and choices based on the guard conditions (and their subconditions) appear in $\mathcal{D}$. Intuitively, a guard condition *gc* corresponds to a particular execution behavior of the software system and, hence, some categories and choices should be identified with respect to *gc*.

- It will not only identify a set of categories and choices from $\mathcal{D}$, but will also determine the choice relations for some pairs of $|X : x_i| \mapsto |Y : y_j|$ in $\mathcal{T}$. Obviously, for the remaining pairs of $|X : x_i| \mapsto |Y : y_j|$ in $\mathcal{T}$, the software tester has to define their choice relations based on the tester's own expertise and judgment.

Certain steps in the above identification algorithm warrant additional explanations and discussions:

- Consider step (1)(a). Suppose from $\mathcal{D}$ we found the subcondition "$(u+v-2) > (x-y+5)$", where *u*, *v*, *x*, and *y* are variables. Here, we need to re-express this subcondition as "$(u+v-x+y) > 7$", so that the category $[u+v-x+y]$ and its associated choice $|u+v-x+y :> 7|$ can be identified. This approach reduces the chance where two subconditions with different syntactic structures of the same semantic meaning (for example, "$(u+v-2) > (x-y+5)$" and "$(u+y-1) > (x-v+6)$") result in the definition of different categories and choices.

  In some situations where $sc_{(i,j)}$ contains only one single arithmetic relational operator with two or more variables and this subcondition cannot be expressed in the form "$\mathcal{V} \sim c$", such as "$(u,v > 5)$" (this form is obviously nonstandard), then $sc_{(i,j)}$ should first be decomposed into two or more subconditions in standard forms, such as "$(u > 5)$" and "$(v > 5)$", before each of these decomposed subconditions is processed by steps (1)(b) and (1)(c) for category and choice identification.

- Consider steps (1)(d) and (1)(e) and refer to Figure 1. An example of a subcondition with no arithmetic relational operator is "Local On-Campus". Here, in this example, "Local On-Campus" is also a guard condition by itself. Now, look at the guard condition "$3.5 \leq$ GPA Score ($S$) $\leq 4.0$ in Figure 1. This guard condition is also a subcondition, which has two arithmetic relational operators. Consider another example "$u < v < x \leq y$". This complex hypothetical subcondition has four variables and three arithmetic relational operators.

  Now, let us consider the guard conditions (or subconditions) "Local On-Campus" and "Overseas Off-Campus" in Figure 1. They are associated with the same decision point and, hence, the same factor (that is, "Location of Study"). According to steps (1)(d) and (1)(e) of the algorithm, we should define [Location of Study] as a category and |Location of Study: Local On-Campus| and |Location of Study: Overseas Off-Campus| as its associated choices, if they do not exist. One may argue that, in this example, we should instead define the entire subconditions as categories with "Yes" as their associated choices. That is, we should define the category [Local On-Campus] with an associated choice |Local On-Campus: Yes|, and the category [Overseas Off-Campus] with an associated choice |Overseas Off-Campus: Yes|. In this

approach, however, we have two *different* categories and is therefore counter-intuitive.

- Consider step (7). For any choice $|X : x|$ in $\mathcal{T}$, it may be directly defined from a $sc_{(i,j)}$ in a $gc_i$ in step (1)(c) or (1)(e), or may be generated in step (2)(a)(ii), (2)(b)(ii), or (3). Thus, some $|X : x|$'s in $\mathcal{T}$ may not have their corresponding $sc(|X : x|)$'s in $\mathcal{D}$.

  Also, in step (7), the rules for determining the choice relation for $|X : x| \mapsto |Y : y|$ is based on the rationale that, if $sc(|X : x|)$ and $sc(|Y : y|)$ appear in the same complete thread $t$ in $\mathcal{D}$, then we must combine $|X : x|$ and $|Y : y|$ together to form part of some complete test frames, from which test cases can be generated to traverse $t$ for the purpose of testing.

Let us use $\mathcal{D}_{\mathsf{award}}$ in Figure 1 to illustrate how to apply our identification algorithm:

(1) Consider, for example, the guard conditions (or subconditions) "$0.0 \leq$ GPA Score ($S$) $< 2.0$" and "$2.0 \leq$ GPA Score ($S$) $\leq 4.0$" associated with the "top" decision point (the one that is near the initial activity). Each of these subconditions has two arithmetic relational operators. The factor associated with these two subconditions is "GPA Score ($S$)". Accordingly, we define the category [GPA Score ($S$)]. Moreover, this category should be defined with two associated choices |GPA Score ($S$): $0.0 \leq S < 2.0$| and |GPA Score ($S$): $2.0 \leq S \leq 4.0$| (see steps (1)(d) and (1)(e) of the algorithm). Additional categories and choices should be defined for other guard conditions in a similar way. Table 1 shows the three categories and their associated choices defined after this step.

(2) In Table 1, [GPA Score ($S$)] is a category with overlapping choices, such as |GPA Score ($S$): $2.0 \leq S < 2.5$| and |GPA Score ($S$): $2.0 \leq S \leq 4.0$|. Because |GPA Score ($S$): $2.0 \leq S < 2.5$| $\subsetneq$ |GPA Score ($S$): $2.0 \leq S \leq 4.0$|, we delete |GPA Score ($S$): $2.0 \leq S \leq 4.0$|. Note that the definition of the choice |GPA Score ($S$): $2.5 \leq S \leq 4.0$| is not needed because it already exists. We repeat this step in a similar way until overlapping choices no longer exist. Table 2 shows the resultant categories and choices upon the completion of this step.

(3) Missing choices are not found in all the three categories [GPA Score ($S$)], [Number of Years of Study], and [Location of Study]. Hence, no action is taken in this step.

(4) There are altogether 10 choices in Table 2. For every table element in $\mathcal{T}_{\mathsf{award}}$ (each corresponds to a choice relation for a pair of choices), we assign a null value to it.

(5) Consider, for example, |GPA Score ($S$): $0.0 \leq S < 2.0$| $\mapsto$ |GPA Score ($S$): $0.0 \leq S < 2.0$| in $\mathcal{T}_{\mathsf{award}}$. We assign the (choice) relational operator "$\sqsubset$" to it. Similarly, we assign the same (choice) relational operator to every other pair of identical choices in $\mathcal{T}_{\mathsf{award}}$.

(6) Now, consider, for example, |Number of Years of Study: $\leq 4$| $\mapsto$ |Number of Years of Study: $> 4$ and $< 6$|, which corresponds to a pair of distinct choices in the same category. We assign the (choice) relational operator "$\not\sqsubset$" to it. The rationale is that, no more than one choice can be selected from each category to form part of any complete test frame. Similarly, we assign the same (choice) relational operator to every other pair of distinct choices of the same category in $\mathcal{T}_{\mathsf{award}}$.

(7) Consider the choices |GPA Score ($S$): $0.0 \leq S < 2.0$| and |Number of Years of Study: $\geq 6$| in Table 2. Since:

  (a) these two choices belong to different categories,

  (b) both subconditions "$0.0 \leq$ GPA Score ($S$) $< 2.0$" and "Number of Years of Study $\geq 6$" appear in $\mathcal{D}_{\mathsf{award}}$ and are not associated with any parallel thread, and

  (c) only one (but not all) complete thread in $\mathcal{D}_{\mathsf{award}}$ associated with the subcondition "$0.0 \leq$ GPA Score ($S$) $< 2.0$" is also associated with the subcondition "Number of Years of Study $\geq 6$",

  we assign the (choice) relational operator "$\sqsubsetneq$" to |GPA Score ($S$): $0.0 \leq S < 2.0$| $\mapsto$ |Number of Years of Study: $\geq 6$| in $\mathcal{T}_{\mathsf{award}}$. Some other choice relations in $\mathcal{T}_{\mathsf{award}}$ can be determined similarly.

In summary, by using the algorithm, we are able to identify a total of three categories and 10 valid choices for $\mathcal{D}_{\mathsf{award}}$. Additionally, with respect to these categories and choices, there are no missing and overlapping choices. Furthermore, during the identification process, useful information has been derived to help determine the choice relations of some pairs of choices in $\mathcal{T}_{\mathsf{award}}$. This does not only improve the efficiency of completing $\mathcal{T}$, but also reduce the chance of incorrect manual definition of choice relations, resulting in the generation of incomplete test frames by CHOC'LATE. If this happens,

**Table 1. Categories and choices defined after step (1) of the identification algorithm**

| Categories | Associated Choices |
|---|---|
| [GPA Score ($S$)] | \|GPA Score ($S$): $0.0 \leq S < 2.0$\|, \|GPA Score ($S$): $2.0 \leq S < 2.5$\|, \|GPA Score ($S$): $2.0 \leq S \leq 4.0$\|, \|GPA Score ($S$): $2.5 \leq S < 3.0$\|, \|GPA Score ($S$): $2.5 \leq S \leq 4.0$\|, \|GPA Score ($S$): $3.0 \leq S < 3.5$\|, \|GPA Score ($S$): $3.0 \leq S \leq 4.0$\|, \|GPA Score ($S$): $3.5 \leq S \leq 4.0$\| |
| [Number of Years of Study] | \|Number of Years of Study: $\leq 4$\|, \|Number of Years of Study: $> 4$\|, \|Number of Years of Study: $< 6$\|, \|Number of Years of Study: $\geq 6$\| |
| [Location of Study] | \|Location of Study: Local On-Campus\|, \|Location of Study: Overseas Off-Campus\| |

**Table 2. Categories and choices refined after step (2) of the identification algorithm**

| Categories | Associated Choices |
|---|---|
| [GPA Score ($S$)] | \|GPA Score ($S$): $0.0 \leq S < 2.0$\|, \|GPA Score ($S$): $2.0 \leq S < 2.5$\|, \|GPA Score ($S$): $2.5 \leq S < 3.0$\|, \|GPA Score ($S$): $3.0 \leq S < 3.5$\|, \|GPA Score ($S$): $3.5 \leq S \leq 4.0$\| |
| [Number of Years of Study] | \|Number of Years of Study: $\leq 4$\|, \|Number of Years of Study: $> 4$ and $< 6$\|, \|Number of Years of Study: $\geq 6$\| |
| [Location of Study] | \|Location of Study: Local On-Campus\|, \|Location of Study: Overseas Off-Campus\| |

the scope and comprehensiveness of testing will be adversely affected.

## 4. Conclusion

In this paper, we have introduced some fundamental concepts, including complete test frames, valid choices, three different types of choice relation, missing choices, overlapping choices, and complete threads. Thereafter, we have presented our identification technique, through which a set of categories and choices can be systematically identified. An important characteristic of the technique is that it also helps determine the choice relations of some pairs of choices in $\mathcal{T}$. It should be noted that the set of categories and choices identified by applying the technique should be considered as a "preliminary" set; it should be further refined based on the tester's expertise and experience on the problem domain of the software to be tested.

As mentioned earlier, the aim of our identification methodology is to identify the major and common components that exist in most informal specifications, through which a comprehensive set of categories and choices can be systematically identified. Hence, our identification technique, which is based on $\mathcal{D}$'s, should be supplemented by other identification techniques based on other specification components such as state-chart diagrams. Once the development of our entire identification methodology is completed, we will perform case studies or experiments to measure its effectiveness by using some commercial informal specifications.

## References


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

[2] T. Y. Chen and P.-L. Poon. Experience with teaching black-box testing in a computer science/software engineering curriculum. *IEEE Transactions on Education*, 47 (1): 42–50, 2004.

[3] T. Y. Chen, P.-L. Poon, S.-F. Tang, and T. H. Tse. On the identification of categories and choices for specification-based test case generation. *Information and Software Technology*, 46 (13): 887–898, 2004.

[4] T. Y. Chen, P.-L. Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. *IEEE Transactions on Software Engineering*, 29 (7): 577–593, 2003.

[5] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15 (11): 1318–1332, 1989.

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

[7] R. M. Hierons, M. Harman, and H. Singh. Automatically generating information from a Z speci-

fication to support the classification tree method. In *Proceedings of the 3rd International Conference of B and Z Users*, volume 2651 of Lecture Notes in Computer Science, pages 388–407. Springer, Berlin, 2003.

[8] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology*, to appear.

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

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

[11] K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22 (8): 552–562, 1996.

[12] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6 (3): 247–257, 1980.

[13] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, SE-6 (2): 278–286, 1980.